Paragon: Collaborative Speculative Loop Execution on GPU and CPU

Mehrzad Samadi[1], Amir Hormati[2], Janghaeng Lee[1], and Scott Mahlke[1]
[1]Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{mehrzads, jhaeng, mahlke}@umich.edu
[2]Microsoft Research
Microsoft, inc. - Redmond, WA
amir.hormati@microsoft.com

Abstract

The rise of graphics engines as one of the main parallel platforms for general purpose computing has ignited a wide search for better programming support for GPUs. Due to their non-traditional execution model, developing applications for GPUs is usually very challenging, and as a result, these devices are left under-utilized in many commodity systems. Several languages, such as CUDA, have emerged to solve this challenge, but past research has shown that developing applications in these languages is a daunting task because of the tedious performance optimization cycle or inherent algorithmic characteristics of an application, which could make it unsuitable for GPUs. Also, previous approaches of automatically generating optimized parallel code in CUDA for GPUs using complex compilation techniques have failed to utilize GPUs that are present in everyday computing devices such as laptops and mobile systems.

In this work, we take a different approach. Although it is hard to generate optimized code for GPU, it is beneficial to utilize them speculatively rather than leaving them running idle due to their high raw performance capabilities compared to CPUs. To achieve this goal, we propose Paragon: a collaborative static/dynamic compiler platform to speculatively run possibly-data-parallel pieces of sequential applications on GPUs. Paragon utilizes the GPU in an opportunistic way for loops that are categorized as possibly-data-parallel by its loop classification phase. While running the loop speculatively, Paragon monitors the dependencies using a light-weight kernel management unit, and transfers the execution to the CPU in case a conflict is detected. Paragon resumes the execution on the GPU after the dependency is executed sequentially on the CPU. Our experiments show that Paragon achieves up to 12x speedup compared to unsafe CPU execution with 4 threads.

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

General Terms Design, Performance

Keywords Compiler, GPU, Speculation, Optimization

1. Introduction

In recent years, multi-core 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 multi-core CPUs, GPUs, by introducing general purpose programming models such as CUDA, have also presented the programmer 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 [25] and in the pessimistic case, speedups of 2.5x [18] can be 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, efficiently utilizing GPUs is still a daunting challenge. Difficulty in extracting massive data-level parallelism, utilizing non-traditional memory hierarchy, complicated thread scheduling and synchronization semantics, and lack of efficient handling of control instructions are the main complications that arise while porting traditional applications to CUDA. As a result of this complexity, the computational power of graphics engines is often under-utilized or not used at all in most systems.

Although many researchers have proposed new ways to solve this problem, there is still no solution for the average programmer to target GPUs. In most cases, it is difficult to reshape an application for the massively data-parallel execution engines of GPUs. The programmer has two choices to deal with these cases. The first common solution is to redesign the underlying algorithm used in the application and try to manually extract data-parallelism. Then, the program is re-implemented in CUDA. This solution is clearly not suitable for average programmers and in some cases is very complicated to apply. The second solution is to use automated compiler analyses to automatically extract enough data-parallelism from an application to gain some performance benefit from the resulting code on the target GPU. The main problem with this approach is that the compiler analyses used for auto-parallelization are usually too conservative and fragile resulting in very small or no gains.

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 speculative loop execution on GPUs using *Paragon* for general purpose programs written in C/C++. This approach will not utilize the peak performance capabilities of GPUs. However, it enables general purpose 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.

The idea of speculative loop execution is not a new one. This approach has been extensively investigated in both hardware and software (see Section 6) in the context of multi-core CPUs. Paragon's compilation system is the first that we are aware of that explores this idea in the context of the GPUs and CPUs. 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. In this approach, GPU and CPU both start ex-

GPGPU-5, March 3, 2012, London, UK.

ecuting their version of a possibly-parallel for-loop (sequential on the CPU, data-parallel on GPU). The GPU executes the for-loop assuming there is no data-dependency between the iterations and also checks for possible dependency violations and, if detected, waits for the execution of that piece of the for-loop that has the dependency on the CPU. After a certain number of iterations, the GPU resumes the execution of the rest of the loop using the result of the CPU as its starting input. This approach may not be able to fully utilize the GPU but in many cases results in performance gains on general purpose applications running on commodity systems with GPUs.

The Paragon compilation system is divided into two parts: *offline compilation* and *runtime kernel management*. The offline part mainly performs loop classification and generates CUDA code for the runtime system which monitors the loops on the GPU for dependency violations. The runtime system also performs lightweight profiling 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 no data-dependencies, hard-to-prove data-dependencies, and rare-dependencies on the GPU.

In summary the main contributions of this work are:

- A static/dynamic compilation system for hybrid speculative execution on GPU/CPUs
- Light weight runtime conflict detection on GPUs
- Low overhead rollback mechanism by using the concurrency between GPUs and CPUs

The rest of the paper is organized as follows. In Section 2, the CUDA programming model and the basics of GPUs architecture are discussed. Section 3 explains the motivation behind Paragon. Static and runtime compilation phases of Paragon are discussed in Section 4. Experiments are shown in Section 5. Finally, in Section 6, we discuss related works.

2. Background

The CUDA programming model is a multi-threaded SIMD model that enables implementation of general purpose programs on heterogeneous GPU/CPU systems. There are two different device types in CUDA: the host processor and the GPU. 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 which 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 is for performing computations. Data-level parallelism (DLP) and thread-level parallelism (TLP) are handled differently in these systems. DLP is converted into TLP and executed onto the GPU devices, while TLP is handled by executing multiple kernels on different GPU devices launched by the host processor. The threading and memory abstraction of the CUDA model is shown in Figure 1.

The threading abstraction in CUDA consists of three levels of hierarchy. The basic block of work 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 ($\_\_syncthreads()$). However, there is no explicit software or hardware support for synchronization across thread blocks. Synchronization between thread blocks is performed through the global memory of the GPU, and the barriers needed for synchronization are handled by the host processor. Thread blocks communicate by executing separate kernels on the GPU.

Together, these thread blocks combine to form the parallel segments called *grids* where each grid is scheduled onto a GPU one at a time. However, the newer generation of NVIDIA's GPUs, Fermi,

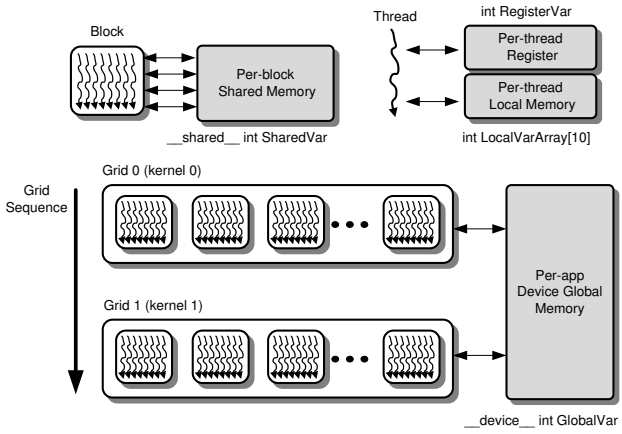

Figure 1: *CUDA/GPU Execution Model*

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 memory abstraction in CUDA consists of multiple levels of hierarchy. The lowest level of memory is *registers*, which are on-chip memories private to a single thread. The next level of memory is *shared memory*, which is an on-chip memory shared only by threads within the same thread block. Access latency to both the registers and shared memory is extremely low. The next level of memory is *local memory*, which is an off-chip memory private to a single thread. Local memory is mainly used as spill memory for local arrays. Mapping arrays to shared memory instead of spilling to local memory can provide much better performance. Finally, the last level of memory is *global memory*, which is an off-chip memory that is accessible to all threads in the grid. This memory is used primarily to stream data in and out of the GPU from the host processor. The latency for off-chip memory is 100-150x more than that for on-chip memories. Two other memory levels exist called the *texture memory* and *constant memory*. Texture memory is accessible through special built-in texture functions and constant memory is accessible to all threads in the grid.

The CUDA programming model is an abstraction layer to access GPUs. NVIDIA GPUs use a single instruction multiple thread (SIMT) model of execution where multiple thread blocks are mapped to streaming multiprocessors (SM). Each SM contains a number of processing elements called Streaming Processors (SP). A thread executes on a single SP. Threads in a block are executed in smaller execution groups of threads called *warps*. All threads in a warp share one program counter and execute the same instructions. If conditional branches within a warp take different paths, called *control path divergence*, the warp will execute each branch path serially, stalling the other paths until all the paths are complete. Such control path divergences severely degrade the performance.

In modern GPUs, such as the NVIDIA GTX 560, there are 8 SMs each with 48 SPs. Each SM processes warp sizes of 32 threads. The memory sizes for this GPU are: 48K of registers per SM and 1GB of global memory shared across all threads in the GPU.

3. Motivation

Parallelizing an existing single-threaded application for a multi-core 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.

```
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));
16     }
17   }
18 }
```

(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]];
4    }
5  }
```

(b)

```
1  void VectorAdd(int n,
2          float *c, float *a, float *b)
3  {
4    for(int i=0; i<n; i++){
5      *c = *a + *b;
6      a++;
7      b++;
8      c++;
9    }
10 }
```

(c)

Figure 2: Code examples for (a) non-linear array access, (b) indirect array access, (c) array access through pointer

Therefore, several automatic parallelization techniques for GPUs have been proposed to exploit more parallelism.

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 at runtime. Removing these dependencies speculatively can dramatically improve the parallelization. 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.

Parallel non-analyzable codes usually contain three types of loops: *non-linear array access* , *indirect array access* and *array access through pointers*. The rest of this section illustrates these kinds of loops.

Non-linear array access: If a loop accesses an array with an index that is not linear with respect to the loop's induction variables, it is hard to statically disambiguate the loop-carried dependencies. To illustrate, Figure 2(a) shows the `make_lattice()` function in the *milc* benchmark from SPEC2006. This function tries to manipulate 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 2(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 2(b) shows codes for forward elimination of a matrix in compressed sparse row (CSR) format where suffix `L` denotes the array for lower triangular matrix. Forward elimination is generally used as a part of gaussian elimination algorithm, which changes the matrix to a triangular form to solve the linear equations. CSR format consists of 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 2 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 do-all loop.

Array access through pointers: Many applications are modularized taking pointers as parameters. This makes it difficult for static compilers to parallelize even with a simple loop. Figure 2(c) shows a function that simply adds two vectors taking pointers as parameters. If there is a possibility that the pointer `c` overlaps with the operands either `a` or `b`, the loop cannot be parallelized. Conservative static compiler will give up parallelizing the loop if there is a little chance of pointers overlapping each other. If the runtime behavior shows that there is a very low probability of pointers overlapping each other, it is valuable to speculatively parallelize the loop at the runtime.

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

4. Paragon Execution System

The main goal of Paragon's execution system is to automatically extract fine-grained parallelism from its sequential input code and generate efficient C/CUDA code to run on heterogeneous systems. 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 parts and runs them speculatively on the GPU. In order to guarantee the correctness of the execution, like all speculative execution systems such as transactional memory systems, Paragon has checkpointing. At each checkpoint, before starting speculative thread execution, the system takes a snapshot of the architectural state. Storing copies for a few registers at transaction threads in a CPU core is relatively cheap. For GPUs, however, with thousands of threads running on the GPU, naively checkpointing large register files would incur significant overhead [12].

However, since GPUs and CPUs have separate memory systems, there is no need for special checkpointing before launching a speculative kernel. Paragon always keeps 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 remove the overhead of recovery, 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

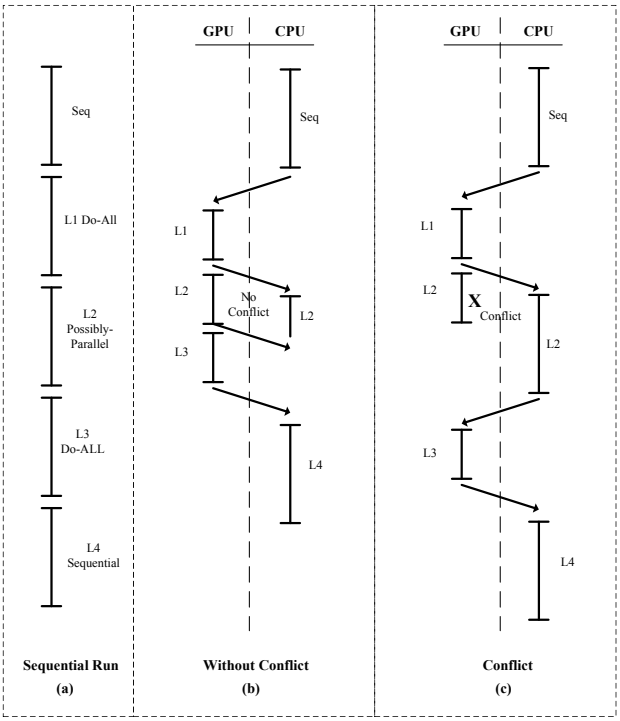

Figure 3: *An example of running a program with Paragon. (a) sequential run (b) execution without any conflict (c) execution with conflict.*

the GPU's results and waits for safe execution to finish and uses its result to run the next kernel.

Figure 3 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. Loops *L1* and *L3* are parallel. *L2* is a possibly-parallel loop that has complex or data-dependent cross-iteration dependency so the compiler is unable to guarantee safe parallel execution of this loop. *L4* has cross-iteration dependencies and it is classified as a sequential loop. Paragon launches the conflict management unit's (CMU) thread on the CPU. The conflict management unit is responsible for managing GPU-CPU transfers and running kernels on the GPU or the CPU. In order to run a kernel on the CPU, the CMU launches another thread on the CPU. For simplicity, Figure 3 just has one line for the CPU but there are two threads running on the CPU while Paragon executes kernels on the CPU.

In this example, Paragon starts execution by running the sequential part on the CPU. After running the sequential code, Paragon transfers the data needed for the execution of loop *L1* to the GPU and runs *L1* in parallel. Since loop *L2* 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. By using asynchronous concurrent execution, Paragon launches the CUDA kernel for loop *L2* at the same time. It should be noted that if *L2* reads and writes to the same array, Paragon should wait for data to be completely transferred to the CPU, and then launch the GPU kernel. CPU executes the safe and sequential version of *L2* after it receives the data needed for execution of *L2* from the GPU. Paragon checks for conflicts in parallel execution of possibly-parallel loops like *L2*. The conflict detection process is done in parallel on the GPU and will set a conflict flag if it detects any dependency violation. After loop *L2* 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, the CMU stops the working thread which executes loop *L2* on the CPU and uses the GPU data to execute loop *L3*. The second case is when a conflict is found in parallel execution of loop *L2*. In this case, Paragon waits for CPU execution to finish, then transfers data needed for the loop *L3* to the GPU. Since loop *L3* is parallel, this loop will be executed on the GPU. In order to run the sequential loop *L4*, Paragon copies the output of *L3* to the GPU.

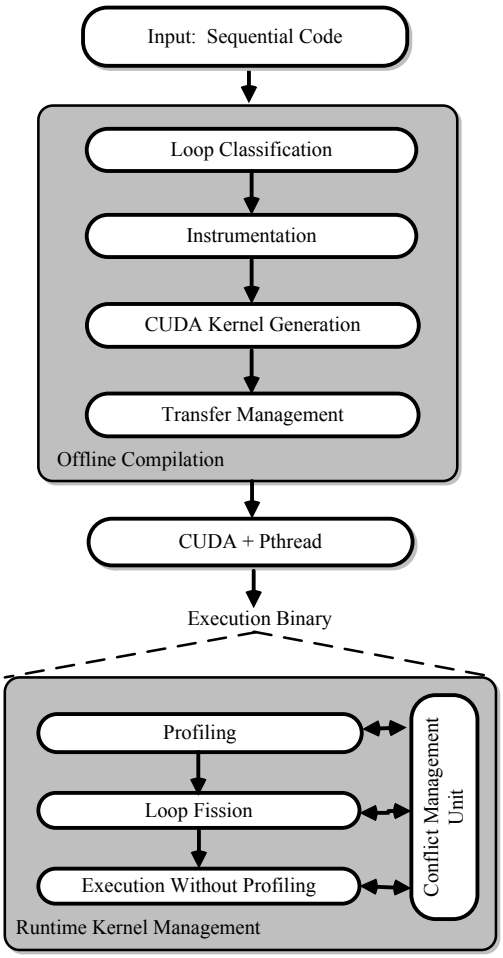

Figure 4: *Compilation flow in Paragon.*

The rest of this section illustrates different parts of the Paragon execution system. As shown in Figure 4, Paragon consists of two main steps: offline compilation and runtime management. Offline compilation performs loop classification to determine whether or not it is safe to parallelize code segments. Based on this information, loop classification categorizes different loops into three categories: parallel, 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 unit. The sequential loops will be run on the CPU therefore Paragon generates only C code for such loops. For parallel loops, CUDA kernels will be generated. Details of this compilation will be discussed in section 4.1. Code generation generates the CPU and GPU code with instrumentation for possibly-parallel loops. The purpose of the instrumentation is to find any possible conflict in the execution of unsafe kernels. These will be discussed

```
1 #pragma unroll
2 for (i = 0; i < iterationsPerThread ; i ++){
3  perform iteration #(i * blockDim + threadId)
4 }
```

(a)

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

(b)

Figure 5: Generated CUDA code for parallel loops with (a) Fixed trip count, (b) Variable trip count.

in section 4.1.2. After compiling all loops, transfer management puts all necessary memory copy instructions in the code.

Runtime kernel management profiles all possibly-parallel loops executing on the GPU and in case of misspeculation, runs a recovery process to guarantee the correctness of execution. Runtime management is also responsible for transferring data between the CPU and GPU.

During runtime, each possibly-parallel loop will be profiled to find any dependency between different iterations of that loop. After profiling, possibly-parallel loops will be categorized as a parallel or sequential loop based on the number of dependencies found in the profiling 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. For those loops with rare conflicts, a loop fission optimization is used to break that loop into multiple parallel loops.

The main unit of runtime management is the conflict management unit. The CMU uses profiling information to decide which kernels should be executed on the GPU and which of them should be run on the CPU. The CMU also takes care of data movement between CPU and GPU especially in misspeculation cases.

4.1 Offline Compilation

This section describes Paragon's CUDA compilation process. Paragon generates CUDA kernels for parallel and possibly-parallel loops and instruments the possibly-parallel kernels to make detection of any dependency violation possible at runtime.

4.1.1 Loop Classification and Kernel Generation

Distributing the workload evenly among thousands of threads is the main key to gaining good performance on the 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 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 is always equal to 256 and 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 is always a power of two to make it easier to handle on the GPU. In this case, number of blocks will be:

$$Number\_of\_Blocks = \frac{Trip\_Count}{Threads\_per\_Block} \times \frac{1}{Iterations\_per\_Thread}$$

If the trip count is not known during 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 previous runs of this kernel. As shown in Figure 5b each thread will run iterations until no iterations are left.

We could use the same method for loops with a fixed trip count, 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 can degrade the performance. Another advantage is that for loops similar to the loop in Figure 5a 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. If the outer loop has instructions that are not in the inner 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, 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: Another common loop is a reduction loop. A reduction operation generally takes a large array as an input, performs computations on it, and generates a single element as an 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. After this detection phase, Paragon replaces the reduction actor with a highly optimized kernel in its output CUDA code. In a single reduction loop, Paragon generates two kernels. The first kernel, called the initial reduction kernel, chunks up the input array and lets each block reduce a different data chunk . Since there is no global synchronization between threads of different blocks, the results of these blocks are written back to global memory. Subsequently, another kernel, called the merge kernel, is launched to merge the outputs from different blocks of the initial reduction kernel down to one element. The merge kernel has one block that is used to compute the reduction output of the initial kernel's output.

In both kernels, data is initially read from global memory, reduced and written to shared memory, and read again from shared memory and reduced to the final result.

Figure 6 shows Paragon's resulting CUDA code for the initial reduction kernel. In the first phase, the input array in global memory is divided into chunks of data. Each thread computes the output for each chunk, and copies it to shared memory. The amount of shared memory usage in each block is equal to $Threads\_per\_Block$.

In the next phase, the results stored in shared memory are reduced in multiple steps to form the input to the merge kernel. At each step of this phase, the number of active threads performing reduction are reduced by half. They continue until the number of active threads equals the number of threads in a single warp(line 10 in Figure 6). At this point, reducing the number of threads any fur-

```
1 Initial_Kernel<<<reductionBlocks, threads>>>
2   /* Global memory reduction phase */
3
4   Result = 0;
5   numberOfThreads = BlockDim * gridDim;
6   for ( index=tid; index<size; index+=
        numberOfThreads)
7     Result = Result Input[Index];
8   SharedData[tid] = Result;
9   activeThreads = blockDim;
10  while (activeThreads > WARP_SIZE){
11    if (tid <activethreads)
12      activeThreads /=2;
13      sync();
14      SharedData[tid] = SharedData[tid+
            activeThreads];
15  }
16
17  /* Shared memory reduction phase */
18  Stride = WARP_SIZE;
19  if (tid < WARP_SIZE)
20    while (stride > 1){
21      sync();
22      SharedData[tid] = SharedData[tid + stride
            ];
23      stride /=2;
24    }
25  if (tid == 0)
26    Output[bid] = SharedData[0];
```

Figure 6: The initial reduction kernel's CUDA code.

ther would cause control-flow divergence and inferior performance. Therefore, Paragon keeps the number of active threads constant and just has some threads doing unnecessary computation (line 20 in Figure 6). It should be noted that after each step, synchronization is necessary to make shared memory changes visible to other threads.

Finally, the thread with $tid = 0$ computes the final initial reduction result and writes it back to global memory.

If there are multiple do all loops and the innermost loop is a reduction loop, Paragon maps outer loops to the blocks and each block computes the reduction. Since each block is independent of other blocks, there is no need to launch the merge kernel.

If the outer loops have a high number of iterations, Paragon may assign each reduction process to one thread. Therefore, iterations of outer loops will be distributed between threads and each thread executes one instance of the innermost loop.

4.1.2 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. Paragon checks for the dependency between different threads for possibly-parallel loops on the fly. All this conflict detection process is done on the GPU in parallel. This process is done with two kernels: the execution kernel and the checking kernel.The first kernel does the computations and also tags store and load addresses, and the checking kernel checks marked 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.

Paragon detects the arrays that can cause conflicts, and for those arrays, it allocates write-log and read-log arrays. Using a bloom filter for keeping track of thousands of threads at the same time requires large signatures. Accessing these signatures on the GPU can cause control flow divergence which leads to performance degradation on the GPU. Instead of using a bloom filter, Paragon stores all memory accesses in read-log and write-log arrays separately. During execution, each store to a conflict array will be marked in a write-log and each load from that array will be marked in a read-log.

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

(a)

```
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;
8 }
```

(b)

Figure 7: Generated CUDA code for example code in Figure 2b. (a) the execution kernel code with instrumentation, (b) the checking kernel.

Since the order of execution of threads on the GPU is not known, any two threads which write to the same address from different threads can cause a conflict. This conflict may result in a wrong output. In addition to the output dependency, writes to and reads from the same address by two different threads may violate the dependency constraints. Since the number of writes to each address is important, Paragon uses CUDA atomic incrementation instructions to increment the number of writes for each store as shown in Figure 7. If the number of writes to one address is more than one then there is a write dependency between iterations of the loop, and that loop is not parallelizable. One write to and one read from the same address will cause a conflict, and the GPU's result may not be correct anymore. Since the number of reads from each address is not important, For decreasing the overhead of checking Paragon does not increment read-log elements atomically. Per each read, Paragon sets the corresponding bit in the read-log.

After completion of the execution kernel, the checking kernel will be launched. This kernel checks the read-log and write-log for conflicts. The easiest implementation of the checking kernel is to launch one thread per address and check all addresses as shown in Figure 7, but this method can decrease performance. Instead of checking all addresses, it will be faster 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 first regenerates addresses that threads wrote to them. Paragon starts from the index of write accesses and goes up on the data flow graph to find instructions that generate write addresses and puts all of them in the checking kernel. Line 5 of Figure 7 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.

Some of the loops only write to the conflicted array so a write-write dependency is the only source of conflicts in these loops. In this case, there is no need to launch the checking kernel because the *atomicInc* function returns the old value of the write-log element.

For each conflicted array write, 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 one, it shows that another thread already wrote to the same element and this may result in conflict.

Each thread that finds a conflict will set the conflict flag. This flag will be sent to the CPU. Based on the conflict flag, the conflict management unit makes further decisions. These decisions will be discussed in section 4.2.3.

4.1.3 Transfer Management

After generating CUDA codes, Paragon inserts copying instructions between the kernels. All live-in and live-out variables for all kernels are determined by Paragon. The transfer management compiler pass starts with sequential CPU codes. For calling the first GPU kernel, all live-in variables of that kernel will be transferred to the GPU. After each kernel, transfer management puts copying instruction based on 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 puts copying instructions in both directions. Runtime management will decide how to move the data at runtime based on the place of correct data.

4.2 Dynamic Runtime Management

4.2.1 Profiling

This section describes profiling loops to find the dependency between iterations and how Paragon uses this information to improve the performance of the generated code. Profiling helps us to find possibly-parallel loops for which a compile-time dependency analysis cannot guarantee safe parallel execution of a given loop. The first time that Paragon runs a possibly-parallel loop, it will run it on the CPU with a thread called *working thread*. Another thread called *profiling thread* will profile the loop in parallel to decrease the overhead of profiling. The profiling thread keeps track of all memory accesses. Since Paragon only profiles possibly-parallel loops and does it in parallel with real execution, profiling has a negligible overhead.

The profiling thread executes all instructions except stores and keeps track of the number of conflicts. If there was no conflict, the profiling thread marks the kernel as a parallel kernel for the conflict management unit. If there were many conflicts, the profiling thread marks the loop as sequential.

After all kernels are categorized based on the profiling results, Paragon enters the execution phase and runs the kernels without profiling. 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.

4.2.2 Loop Fission

In some loops conflicts are really rare. for these loops, if there are enough iterations, Paragon uses loop fission optimization to divide the iteration space of the loops into multiple groups. Paragon runs each group separately in parallel. Since conflicts are rare in these cases, most of the generated loops will be parallel.

This optimization will be more beneficial for future heterogeneous systems that have a high CPU-GPU transfer rate. For desktop systems, since transferring data between CPU and GPU is expensive, checkpointing between all consecutive subloops produced by loop fission may result in poor performance.

4.2.3 Conflict Management Unit

Conflict management unit is a thread running on the CPU and its responsibility is to manage GPU-CPU transfers and run kernels speculatively on the GPU. The CMU 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 CMU 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 conflict management unit makes different decisions. If the next kernel should be run on the GPU, the CMU 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 conflict management unit runs the sequential code on the CPU.

If there was no conflict in the GPU execution and the next kernel is parallel then the CMU will launch the next kernel. Otherwise it transfers live-out variables and runs the next kernel on the CPU.

5. Experiments

A set of benchmarks from the sparse matrix library are used for evaluation to show Paragon's performance for loops with indirect array accesses. We re-implemented another set of benchmarks from the Polybench benchmark suite in C with pointers to show Paragon's performance for loops with pointers. We compiled these benchmarks with Paragon, and compared their performance with the original and hand-optimized unsafe parallelized C code. Paragon compilation phases are implemented in the backend of the Cetus compiler [17] and its C code generator is modified to generate CUDA code. Paragon's output codes are compiled for execution on the GPU using NVIDIA nvcc 3.2. GCC 4.1 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.

5.1 Loops with Pointers

In this section, we want to show Paragon's performance for loops with pointers. We rewrote 6 benchmarks from the Polybench benchmark suite [28] with pointers. We also implemented unsafe parallel versions of these benchmarks for a CPU with 1, 2, and 4 threads. Figure 8 compares the performance of these benchmarks compared to sequential runs on the CPU. The GPU dataset shows the results for when we compile the loops with Paragon but without any instrumentation. In this case, we assume that the compiler can detect that these loops are parallel. Paragon dataset shows the performance of Paragon's generated code with instrumentation. In this case, we check all possibly-parallel loops on the fly. CPU 4, 2, and 1 are unsafe parallel CPU versions without any checks for conflicts. All these different versions are compared with sequential runs on the CPU without any threading. The thread creation overhead makes the performance of the CPU code with one thread worse than the sequential code.

Figure 9 shows the overhead of Paragon execution compared to the GPU execution without any instrumentation. This Figure also breaks down the overhead into three groups: write-log maintenance, read-log maintenance, and checking kernel execution.

FDTD, Finite Difference Time Domain method, is a powerful computational technique for modeling electromagnetic space. This benchmark has three different stencil loops and all these loops are highly memory intensive. In memory intensive loops, adding more memory accesses to check the conflicts can degrade the performance. Therefore, the performance gap between GPU and Paragon is high for *FDTD*.

The *Siedel* benchmark uses the Gauss-Siedel method which is an iterative method used to solve a linear systems of equations. Siedel is another stencil benchmark with more computation than FDTD. *Jacobi* is another stencil method to solve linear systems; We have one dimensional and two dimensional versions of this

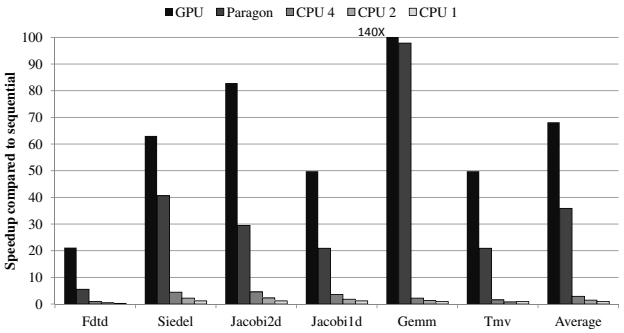

Figure 8: *Performance comparison of Paragon with unsafe parallel versions on the GPU and CPU with 4, 2, and 1 threads for loops with pointers.*

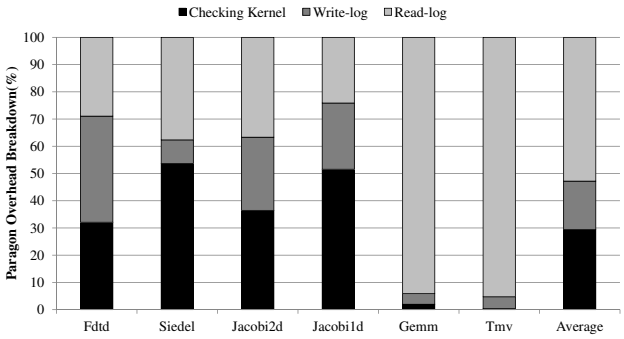

Figure 9: *Breakdown of Paragon's overhead compared to unsafe parallel version on the GPU for loops with pointers.*

benchmark. As shown in Figure 9, the read-log, write-log, and checking kernel have similar effects on these benchmarks.

gemm is a general matrix multiplication benchmark that has three nested loops. The innermost loop is reduction 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. Since the matrix multiplication benchmark has an order of magnitude more reads than writes, maintaining the read-log is the main source of the overhead compared to the GPU without checking.

mvt is a matrix transpose vector product 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. Maintaining the read-log is the main source of the overhead in this benchmark because of the high number of reads.

On average, the unsafe parallel GPU version is 2x faster than Paragon. The reason is that in loops with pointers, all arrays can cause conflicts, and Paragon takes care of all reads and writes which leads to 2x more memory accesses. These extra memory accesses can degrade the performance of memory-intensive loops. Compared to the unsafe parallelized version on the CPU, Paragon is 12x faster than CPU execution with 4 threads. Paragon is 24x and 37x faster than 2 and 1 thread execution, respectively.

5.2 Indirect or nonlinear array accesses

Figures 10 and 11 show the results for eight benchmarks with indirect array accesses. These benchmarks have loops that cannot be analyzed by traditional compilers. For each sparse matrix benchmark, we generated matrices randomly with one percent nonzero elements. Since memory accesses in these benchmarks are irregular, the GPU's performance is lower than regular access benchmarks. In these loops, contrary to the pointer loops, Paragon marks arrays that can cause conflicts. Since Paragon only checks memory accesses for these arrays, the overhead of Paragon is lower in these benchmarks compared to the pointer loops.

Saxpy adds one sparse array with one dense array and the result array will be dense as well. Since this benchmark only updates a small percentage of the output array, there is no need to check all elements of the output for a conflict. The checking kernel which checks for conflicts will check only written addresses. We need to add address generation instructions to the checking kernel.

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. This loop will be mapped to CUDA without any instrumentation. The second part has a loop which 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 the another array and puts the results in the output array. Since this benchmark is not sparse, the checking overhead is high. As can be seen in Figure 10, the GPU version is 5x faster than Paragon.

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.

Saxpy, *House*, *Ipvec*, *ger* and *gemver* only update the conflicted arrays. Since the *atomicInc* function 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. In these benchmarks, the Paragon overhead is the result of maintaining write-log as shown in Figure 11.

The next benchmark is *SOR*, a Multicolor SOR sweep in the EllPack format, and its code is shown in Figure 12. This benchmark has three loops: the outer loop is do-across and the two inner loops are parallel, but traditional static compilers cannot easily detect that. By looking at line 4 of the code, it seems that there can be a read-write dependency between *y[i]* and *y[ja[i,j]]*, but the main diagonal of matrix A is stored separately in another matrix. Since A does not have a main diagonal, there will be no dependency between iterations of loops in lines 2 and 3 of Figure 12.

Forward Elimination with Level Scheduling (FWD) is another method which is used in solving linear systems. FWD's code is similar to SOR and it has both reads and writes to the conflicted array. Since this benchmark updates the sparse matrix, the number of memory addresses that have been updated is low. Consequently, the overhead of launching the checking kernel is low but adding memory accesses to the execution kernel to maintain the read-log and write-log is expensive in this benchmark.

As can be seen in Figure 10, the performance of Paragon-generated code is 2.4x better than the parallel version of the code running on the CPU with 4 threads. Paragon is 3x and 5.2x faster than 2 and 1 thread execution, respectively. It should be noted that in the CPU version, we assumed that there is no conflict between different iterations. Figure 10 shows that running safely on the GPU is better than running unsafely on the CPU for these data parallel loops. The unsafe GPU version is 1.6x faster than Paragon's generated code on average.

6. Related Work

As many-core architectures have become mainstream, there has been a large body of work on static compiler techniques to automat-

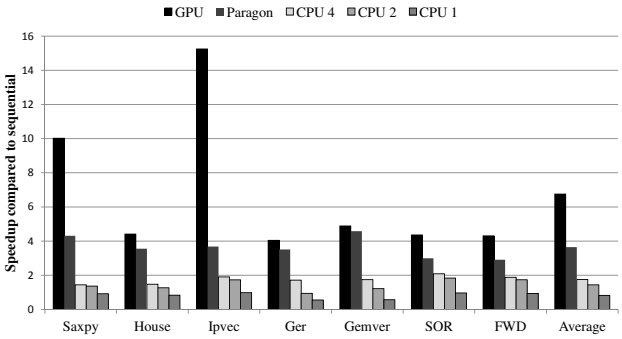

Figure 10: *Performance comparison of Paragon with unsafe parallel versions on the GPU and CPU with 4, 2, and 1 threads for loops with indirect accesses.*

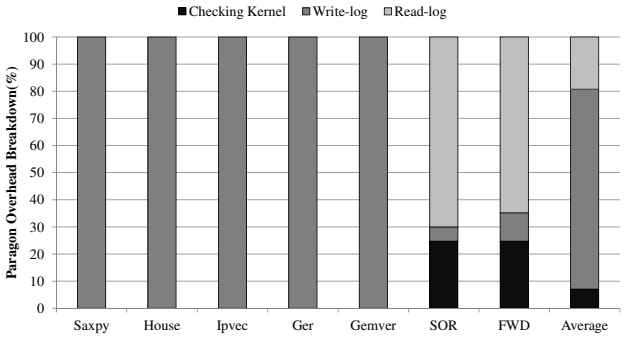

Figure 11: *Breakdown of Paragon's overhead compared to unsafe parallel version on the GPU for loops with indirect accesses.*

ically parallelize applications to utilize thread-level-parallelism [1, 5, 9, 24, 27, 31]. One of the most challenging issues in automatic parallelization is to discover loop-carried dependencies. Although various research projects on loop-dependence analysis [2, 15] and pointer analysis [8, 26] 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 recovery procedures in case of conflicts between threads, are necessary. Many previous works [4, 7, 11, 14, 16, 21–23, 29, 30, 32, 34, 35, 37, 38] have focused on designing and refining these steps for the decades. One of the main approaches is hardware-based TLS [16] which checks and recovers memory violations in hardware to minimize the overhead. Unfortunately, hardware TLS requires complicated logic and area to keep track of memory accesses for each thread. so the capability is only available to fine-grained parallelism, which is not desirable for thread-level-parallelism. Another approach for implementing TLS is software-based TLS [4, 7, 11, 14, 21, 29, 30].

There are previous works that have focused on generating CUDA code from sequential input [3, 13, 19, 33, 36]. hiCUDA [13] 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. [3] 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 [36], by using C pragma preprocessor directives, programmers help the compiler to generate efficient CUDA code. Tarditi et al. [33] proposed accelerator, in which programmers use the C# language and a library to write their programs and let the compiler generate efficient GPU code.The work by Leung et al [19] proposes an extension to a Java JIT compiler that executes program on the GPU.

While none of the previous works on automatic compilation for GPUs considered speculation, there are other works [10, 20] which studied the possibility of speculative execution on the GPGPU Gregory et al. [10] 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 GPU kernels. This approach leveraged the possibility of speculatively partitioning several kernels on multiple GPUs. Liu et al. [20] 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

```
1 for(col=1; col<=ncol; col++)
2   for(j=1; j<=ndiag; j++)
3     for(i=iaL[col]; i<iaL[col+1]-1; i++)
4       y[i] = y[i] - A[i,j] * y[ja[i,j]];
```

Figure 12: Main part of multicolor SOR sweep's code in the EllPack format

techniques which re-executes the whole loop in case of a dependency violation.

Recent works [6, 12] 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.

7. Conclusion

GPUs provide an attractive platform for accelerating parallel workloads. Due to their non-traditional execution model, developing applications for GPUs is usually very challenging, and as a result, these devices are left under-utilized in many commodity systems. Several languages, such as CUDA, have emerged to solve this challenge, but past research has shown that developing applications in these languages is a challenging task because of the tedious performance optimization cycle or inherent algorithmic characteristics of an application, which could make it unsuitable for GPUs. 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 such as laptops and mobile systems.

In this work, we proposed *Paragon*: a collaborative static/dynamic compiler platform to speculatively run possibly-data-parallel pieces of sequential applications on GPUs. Paragon utilizes the GPU in an opportunistic way for loops that are categorized as possibly-data-parallel by its loop classification phase. While running the loop speculatively on the GPU, Paragon monitors the dependencies using a light-weight kernel management unit, and transfers the execution to the CPU in case a conflict is detected. Paragon resumes the execution on the GPU after the dependency is executed sequentially on the CPU. Our experiments showed that Paragon achieves up to a 12x speedup compared to unsafe CPU execution with 4 threads and the unsafe GPU version performs less than 2x better than safe execution with Paragon.

Acknowledgement

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 grant CNS-0964478. We would like to thank Griffin Wright and Gaurav Chadha for their comments.

References

[1] F. Aleen and N. Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 241–252, 2009.

[2] Banerjee and Utpal. *Speedup of ordinary programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1979.

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

[4] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *SPAA '02: 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 99–108, 2002.

[5] W. Blume et al. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

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

[7] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Proc. of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–36, June 2003.

[8] B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proc. of the '00 Conference on Programming Language Design and Implementation*, pages 57–69, June 2000.

[9] K. Cooper et al. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, Feb. 1993.

[10] G. Diamos and S. Yalamanchili. Speculative execution on Multi-GPU systems. In *2010 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2010.

[11] Z.-H. Du et al. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proc. of the '04 Conference on Programming Language Design and Implementation*, pages 71–81, 2004.

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

[13] T. Han and T. Abdelrahman. hicuda: High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):52–61, 2010.

[14] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 2007.

[15] P. K., K. X., and K. D. The direction vector i test. *IEEE Journal of Parallel Distributed Systems*, 4(11):1280–1290, 1993.

[16] V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. pages 85–92, New York, NY, USA, 1998. ACM Press.

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

[18] 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. 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*, pages 451–460, 2010.

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

[20] S. Liu, C. Eisenbeis, and J.-L. Gaudiot. Value prediction and speculative execution on gpu. In *International Journal of Parallel Programming*, volume 39, pages 533–552, 2011.

[21] W. Liu et al. POSH: A TLS compiler that exploits program structure. In *Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 158–167, Apr. 2006.

[22] M. Mehrara, J. Hao, P. chun Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proc. of the '09 Conference on Programming Language Design and Implementation*, pages 166–176, June 2009.

[23] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke. Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism. In *Proc. of the 17th International Symposium on High-Performance Computer Architecture*, pages 87–98, Feb. 2011.

[24] S. Moon, B. So, and M. W. Hall. Evaluating automatic parallelization in SUIF. *Journal of Parallel and Distributed Computing*, 11(1):36–49, 2000.

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

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

[27] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 105–118, Nov. 2005.

[28] Polybench. the polyhedral benchmark suite, 2011. http://www.cse.ohio-state.edu/ pouchet/software/polybench.

[29] M. Prabhu and K. Olukotun. Exposing speculative thread parallelism in SPEC2000. In *Proc. of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 142–152, June 2005.

[30] C. G. Quinones et al. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proc. of the '05 Conference on Programming Language Design and Implementation*, pages 269–279, June 2005.

[31] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, 2004.

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

[33] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 325–335, 2006.

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

[35] N. Vachharajani, R. Rangan, E. Raman, M. Bridges, G. Ottoni, and D. August. Speculative Decoupled Software Pipelining. In *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 49–59, Sept. 2007.

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

[37] A. Zhai, C. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–183, Oct. 2002.

[38] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 85–96, Nov. 2002.