Optimistic Parallelism on GPUs

Min Feng[1], Rajiv Gupta[2], and Laxmi N. Bhuyan[2]

[1] NEC Laboratories America
[2] University of California, Riverside**

Abstract. We present speculative parallelization techniques that can exploit parallelism in loops even in the presence of dynamic irregularities that may give rise to cross-iteration dependences. The execution of a speculatively parallelized loop consists of five phases: scheduling, computation, misspeculation check, result committing, and misspeculation recovery. While the first two phases enable exploitation of data parallelism, the latter three phases represent overhead costs of using speculation. We perform misspeculation check on the GPU to minimize its cost. We perform result committing and misspeculation recovery on the CPU to reduce the result copying and recovery overhead. The scheduling policies are designed to reduce the misspeculation rate. Our programming model provides API for programmers to give hints about potential misspeculations to reduce their detection cost. Our experiments yielded speedups of 3.62x-13.76x on an nVidia Tesla C1060 hosted in an Intel(R) Xeon(R) E5540 machine.

1 Introduction

Many top-500 supercomputers today have adopted Graphics Processing Units (GPUs) for high performance computing. A number of research works [9, 2, 3] have explored loop-level data parallelism using GPUs, whose massive number of computing units are ideal for accelerating data-parallel computations. The presence of dynamic irregularities prevents existing techniques from parallelizing the loops for GPUs. Therefore optimizing performance in their presence has been widely studied [20, 3, 9, 18]. In this work, we consider a new class of dynamic irregularities in loops that may cause cross-iteration dependences at runtime. In particular, we have identified two types of dynamic irregularities, illustrated in Figure 1, that may dynamically cause cross-iteration dependences to arise preventing the loops from being parallelized by compilers for GPUs.

Dynamic irregular memory accesses refer to memory accesses whose memory access patterns are unknown at compile time. They may result in infrequent cross-iteration dependences at runtime. In Figure 1(a) the memory access patterns of $A[P[i]]$ and $A[Q[i]]$ are determined by the runtime values of the elements in arrays P and Q. It is possible that an element in array A is read in one iteration and written in another causing a dynamic cross-iteration dependence.

Irregular control flow is introduced by conditional statements, which may cause execution of paths that may give rise to cross-iteration dependences at

** This work is supported by NSF grants CNS-1157377 and CCF-0905509 to UCR.

```
for (i=0; i<n; i++) {
    ... = A[P[i]];
    A[Q[i]] = ...;
}
```

(a) Irregular memory access

```
for (i=0; i<n; i++) {
    ... = A[i];
    if (A[i]) A[i+1] = ...;
}
```

(b) Irregular control flow

Fig. 1. Examples of dynamic irregularities that cause cross-iteration dependences.

runtime, as illustrated in Figure 1(b), where each iteration of the loop usually only reads $A[i]$. In the loop, there is a conditional branch that guards a write to $A[i + 1]$, which is to be read in the next iteration. The true outcome of the branch condition gives rise to a cross-iteration dependence.

Software thread-level speculation (TLS) [13, 5, 7] has been used with success to parallelize loops that may contain cross-iteration dependences for execution on CPUs. However, developing similar speculative techniques for GPUs is challenging due to the architectural differences between CPUs and GPUs. This is due to the need for logically separate space to store results of thousands of threads and high overhead of complicated thread synchronizations [8].

This paper presents a speculative execution framework for GPU computing. It parallelizes loops that may contain cross-iteration dependences caused by above dynamic irregularities. The execution of a speculative parallel loop consists of five phases: scheduling, computation, misspeculation check, result committing, and misspeculation recovery. For efficiency, we develop a scheduling policy that is optimized for different types of cross-iteration dependences to reduce the misspeculation rate. We reduce the runtime overhead by performing misspeculation check on the GPU. We optimize the result committing procedure to reduce the size of data transferred between the CPU and GPU. Recovery is performed on the CPU for as few iterations as possible to minimize its runtime overhead. We present programming constructs for specifying speculatively parallel loops. Our implementation achieves 3.62x-13.76x speedups for speculatively parallelized loops on nVidia Tesla C1060 hosted in a Intel Xeon E5540 machine.

2 Execution Framework

Figure 2 gives the overview of executing a speculative parallel loop using GPUs. The procedure consists of five phases: *scheduling*, *computation*, *misspeculation check*, *result committing*, and *misspeculation recovery*, among which *computation* and *misspeculation check* are performed on the GPU. The five phases are repeated until the entire loop is finished.

Scheduling, performed on the CPU, determines the proper number of iterations to execute on the GPU – assigning too many iterations to the GPU can cause excessive misspeculations while assigning too few iterations limits performance by leaving the GPU underutilized. In the Computation phase the GPU executes the iterations in parallel by speculating on the absence of cross-iteration dependence while tracking the irregular memory accesses and control flow. Next the GPU performs the Misspeculation Check in two steps: detection and localization. Misspeculation detection is used to determine whether the iterations have been executed correctly. If misspeculation is detected, the localization step

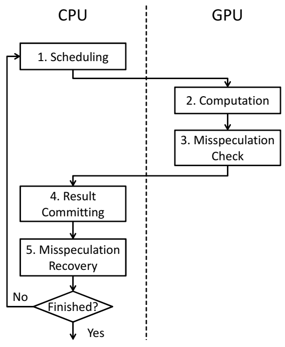


Fig. 2. Execution framework of a speculative parallel loop with GPUs.

identifies the iterations that were executed incorrectly. In addition, we identify the correctly computed results so they can be copied back to the CPU memory. To make misspeculation checks efficient, they are performed in parallel on the GPU. Result committing phase copies the results from the GPU memory to the CPU memory. Finally, Misspeculation Recovery phase re-executes the iterations where misspeculation occured on the CPU or GPU depending on whether a few or a large number of iterations are to be executed. In the subsequent sections we first illustrate the GPU part of the execution model, i.e., the second and third phases. Then we will elaborate on the fourth and fifth phases, which are performed on the CPU. Finally, we will describe our scheduling policy.

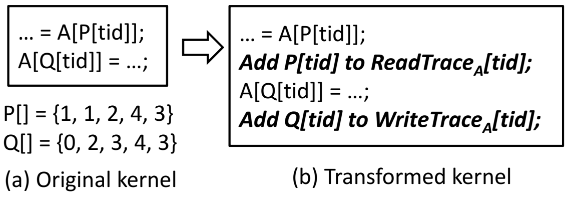
3 Speculative Execution on GPUs

In this section, we describe how a loop is speculatively executed in parallel on GPUs. The infrequent cross-iteration dependences in a speculative parallel loop are usually caused by two types of dynamic irregularities – irregular memory accesses and irregular control flow. We elaborate the strategies for speculative execution for the types of irregularities separately.

3.1 Irregular memory accesses

Figure 3(a) shows the kernel of the loop example given in Figure 1(a). The conversion from loops to GPU kernels has been studied in [9, 2]. In this example, *tid* is the GPU thread ID and each GPU thread executes one iteration of the loop. Depending upon the runtime values of the elements in arrays P and Q, two iterations of the loop may read and write the same element of array A, causing cross-iteration dependence at runtime. Consider the runtime values of arrays P and Q shown in Figure 3(a). Because iteration 1 (starting from 0) writes $A[2]$ and iteration 2 reads $A[2]$, there exists a RAW cross-iteration dependence. Similarly, since both iteration 2 and 4 write to $A[3]$, there is a WAW dependence between them. Therefore, the results computed by iterations 2 and 4 will be incorrect following parallel execution. Our speculative execution of this kernel on GPUs consists of three phases: execution with memory access tracking, misspeculation detection, and misspeculation localization.

Memory access tracking – to detect cross-iteration dependences, we track which elements of the arrays with irregular access patterns are accessed in each



```
… = A[P[tid]];
A[Q[tid]] = …;
```

P[] = {1, 1, 2, 4, 3}
Q[] = {0, 2, 3, 4, 3}

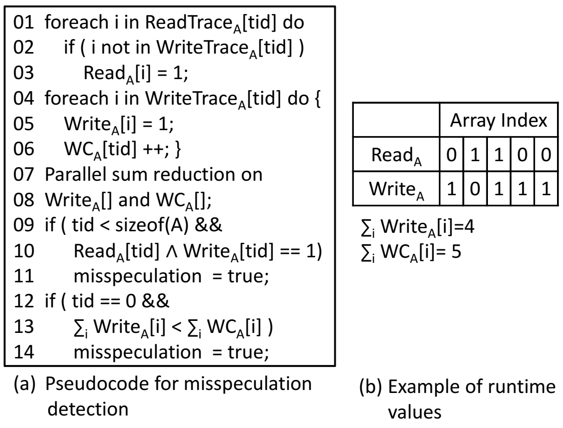
(a) Original kernel

```
… = A[P[tid]];
Add P[tid] to ReadTrace_A[tid];
A[Q[tid]] = …;
Add Q[tid] to WriteTrace_A[tid];
```

(b) Transformed kernel

Fig. 3. Code transformation of a loop with irregular memory accesses.

iteration. This is done by inserting a tracking operation after each irregular memory access. For low tracking overhead, we use two static arrays of a predefined size for each iteration to store the indices of the read and written elements. We assume that we know the maximum number of elements that will be accessed in each iteration. This is often true, including for all benchmarks used in our experiments. Figure 3(b) shows the transformed kernel with a tracking operation after each irregular access to array A.



```
foreach i in ReadTrace_A[tid] do
   if ( i not in WriteTrace_A[tid] )
      Read_A[i] = 1;
foreach i in WriteTrace_A[tid] do {
   Write_A[i] = 1;
   WC_A[tid] ++; }
Parallel sum reduction on
Write_A[] and WC_A[];
if ( tid < sizeof(A) &&
     Read_A[tid] ∧ Write_A[tid] == 1)
     misspeculation = true;
if ( tid == 0 &&
     ∑_i Write_A[i] < ∑_i WC_A[i] )
     misspeculation = true;
```

(a) Pseudocode for misspeculation detection

	Array Index				
$Read_A$	0	1	1	0	0
$Write_A$	1	0	1	1	1

$\sum_i Write_A[i]=4$
$\sum_i WC_A[i]= 5$

(b) Example of runtime values

Fig. 4. Misspeculation detection.

Misspeculation detection checks whether a cross-iteration dependence was encountered during parallel execution. Unlike recent speculative parallelization techniques [7] for CPUs, which only need to detect RAW dependences, speculative parallelization on GPUs requires detecting RAW, WAR, and WAW. Speculative parallelization on CPUs resolves WAR and WAW dependences by committing the results of the iterations in a sequential order. However, these techniques cannot be efficiently implemented on GPUs because they require complicated synchronizations. Since all computations are performed simultaneously on GPUs, we need to detect all kinds of dependences. Additionally, in some cases, privatizing a shared array can solve WAR and WAW dependences on the array. However, privatization is not always possible. A shared array can be privatized only if the compiler can guarantee that every read access to an element is preceded by a write access to the same element within the same iteration. Since this is not always true in real applications, speculative parallelization on GPUs should be able to detect WAR and WAW dependences.

We simplify the traditional shadow memory-based misspeculation detection method [13] and adapt it for GPU computing. We perform the misspeculation detection on the GPU to exploit the data parallelism in the detection procedure. Our lightweight misspeculation detection method only detects the existence of cross-iteration dependences. Only if there is a dependence, we perform the misspeculation localization phase to determine in which iterations misspeculation has occurred. Figure 4(a) shows the pseudocode for misspeculation detection for the kernel example given in Figure 3. In our implementation, all trace data are stored in the global memory. The detection procedure is as follows.

1. Compute $Read_A$ in parallel (line 1–3). $Read_A[i]$ is set when $A[i]$ is read but not written in an iteration. It records the elements of array A which are read-only in some iteration(s). It is safe to allow multiple blocks to simultaneously update $Read_A$ since each update is just setting an element to 1.
2. Compute $Write_A$ and WC_A in parallel (line 4–6). $Write_A[i]$ is set when $A[i]$ is written in an iteration. It records the elements of array A that have been written. WC_A stores the number of elements written in each iteration. We allow multiple blocks to simultaneously update $Write_A$.
3. Compute sums of $Write_A$ and WC_A in parallel (line 7–8). The sum of $Write_A$ is the number of elements that have been written, where multiple writes to the same element in an iteration count as 1. The sum of WC_A is the number of writes to array A.
4. Compute the intersection of $Read_A$ and $Write_A$ in parallel using threads $0 \dots sizeof(A)$ (line 9–11). If $\exists i, Read_A[i] \wedge Write_A[i] = 1$, then $A[i]$ is read-only in some iteration(s) and written in some other iteration(s). In this case, misspeculation occurs due to a RAW or WAR dependence. In some cases, an element may be read and written in an iteration(s) and also written in another iteration(s). We treat such dependences as WAW dependences, which are detected in the next step.
5. Compare the sum of $Write_A$ and WC_A. If $\sum_i Write_A[i] < \sum_i WC_A[i]$, then there must exist multiple iterations that write the same element. This indicates the existence of WAW dependences.

Figure 4(b) shows the calculated values of array $Read_A$ and $Write_A$ for the P and Q given in Figure 3(a). The values indicate that there exist both RAW/WAR and WAW dependences in the kernel execution. $Read_A[2]$ and $Write_A[2]$ are both equal to 1 since iteration 1 writes $A[2]$ and iteration 2 reads $A[2]$. $\sum Write_A[i] = 4$ is smaller than $\sum WC_A[i] = 5$ since $A[3]$ is written in two iterations. The read and write of $A[3]$ happens in the same iteration. Thus, $A[3]$ is not recorded in $Read_A$. Therefore, there is no dependence detected on $A[3]$.

Misspeculation localization method identifies not only the misspeculated iterations but also the incorrect elements of arrays with irregular access patterns. With the information of incorrect elements, we can optimize the copying of results from the GPU to the CPU. The localization procedure is also parallelized for performance. Figure 5(a) shows the misspeculation localization for the kernel example. The details of the localization procedure are described below.

1. Compute RW_A in parallel (line 1–2) by intersecting $Read_A$ and $Write_A$, which indicate elements that are read and written in different iterations. To

```
01 for (i=$BK_A$[tid]; i<$BK_A$[tid+1]; i++)
02    $RW_A$[i]=ReadA[i] & WriteA[i];
03 if (tid < N)
04    for(i=$BK_T$ [tid]; i<$BK_T$[tid+1]; i++)
05        foreach j in $WriteTrace_A$[i] do
06            $WW_A$[tid][j] ++;
07 Combine $WW_A$[1][] … $WW_A$[N][];
08 foreach i in $ReadTrace_A$[tid] do
09    if ( $RW_A$[i] == 1)
10        Misspec[tid] = 1;
11 foreach i in $WriteTrace_A$[tid] do
12    if ($WW_A$[i] > 1)
13        Misspec[tid] = 1;
14 if (Misspec[tid] == 1)
15    foreach i in $WriteTrace_A$[tid] do
16        $Wrong_A$[i] = 1;
17 Parallel reduction on $Wrong_A$[]
18 and Misspec[];
```

	Array Index				
RW_A	0	0	1	0	0
WW_A	1	0	1	2	1
$Wrong_A$	0	0	0	1	0
	Thread ID				
Misspec	0	0	1	0	1

$Wrong_{A\ reduced}$[]={3}
$Misspec_{reduced}$[]={2, 4}

(a) Pseudocode for misspeculation localization

(b) Example of runtime values

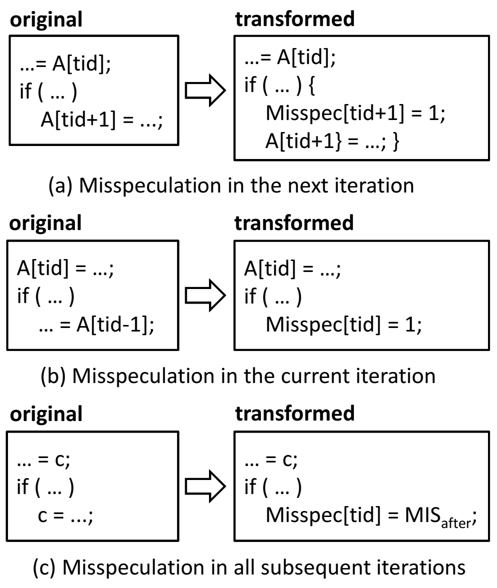
Fig. 5. Misspeculation localization.

calculate RW_A in parallel, we divide array A into blocks. Block boundaries are stored in BK_A. Each thread calculates RW_A for one block of array A.

2. Compute WW_A in parallel (line 3–7). WW_A stores the number of iterations that write each element. $WW_A[i]$ is larger than 1 if $A[i]$ is written in multiple iterations. We use the first N threads to calculate WW_A in parallel. Each of the N threads calculates partial WW_A using arrays $WriteTrace_A$ from a block of threads. The block boundaries are stored in BK_T. A reduction merges the values of these subsets. The total size of WW_A is $N * sizeof(A)$.
3. Check RW_A in each thread (line 8–10). If $RW_A[i]$ is set and $A[i]$ is read in the current thread, then the iteration performed by the current thread reads an element that is written in another iteration. The iteration is misspeculated due to a RAW/WAR dependence. Array $Misspec$ stores such iterations.
4. Check WW_A in each thread (line 11–13). If $WW_A[i]$ is larger than 1 and $A[i]$ is written in the current thread, then the iteration performed by the current thread writes an element that is written in some other iteration(s). The iteration misspeculates due to a WAW dependence. $Misspec[tid]$ is set when the iteration calculated by the current thread is involved in a misspeculation.
5. Compute $Wrong_A$ in parallel (line 14–16), which indicates the incorrect elements of array A. An element is incorrect only when it is written by at least one misspeculated iteration.
6. Perform parallel reductions on $Wrong_A$ and $Misspec$ to store the incorrect elements and misspeculated iterations in lists. The CPU uses these to perform commit and recovery instead of having to inefficiently scan sparse arrays $Wrong_A$ and $Misspec$.

Figure 5(b) shows the values of RW_A, WW_A, $Wrong_A$, and $Misspec$ for the P and Q given in Figure 3(a). Since iteration 2 reads $A[2]$ which is written by iteration 1, iteration 2 is involved in misspeculation. Since iteration 4 writes $A[3]$ which is written by multiple iterations, it is misspeculated. As $A[3]$ is written by misspeculated iterations 2 and 4, its value is incorrect.

3.2 Irregular Control Flow



```
…= A[tid];
if ( … )
   A[tid+1] = …;
```

```
…= A[tid];
if ( … ) {
   Misspec[tid+1] = 1;
   A[tid+1} = …; }
```

(a) Misspeculation in the next iteration

```
A[tid] = …;
if ( … )
   … = A[tid-1];
```

```
A[tid] = …;
if ( … )
   Misspec[tid] = 1;
```

(b) Misspeculation in the current iteration

```
… = c;
if ( … )
   c = …;
```

```
… = c;
if ( … )
   Misspec[tid] = MIS_after;
```

(c) Misspeculation in all subsequent iterations

Fig. 6. Loops with irregular control flow.

Figure 6 shows three types of cross-iteration dependences that are caused by irregular control flow. In Figure 6(a), the true branch condition causes a write to an element that is read in the next iteration and thus causing misspeculation in the next iteration. In Figure 6(b), the true branch condition reads an element that is written in the previous iteration and thus causing misspeculation in the current iteration. In Figure 6(c), the true branch condition writes a scalar variable that is read in all iterations and therefore makes them all wrong. We parallelize such loops by speculating the branch will not be executed. To verify the correctness of the parallel execution, we must monitor the execution of these branches. Once these branches are executed, we should be able to detect the misspeculation and identify the misspeculated iterations.

The cross-iteration dependences in the branches can be either marked by the programmer or detected by the static data race detection techniques. The static data race detection techniques identify dependences in a conservative way. Therefore, they may cause false misspeculations. Programmers can better identify the branches using their knowledge of the application. We propose a programming model that allows programmers to mark such branches (Section 6).

Once we have identified the cross-iteration dependences in the branches, we transform the branches for speculative execution in two steps.

1. In the branches that cause cross-iteration dependences, we insert an operation for recording the misspeculated iterations. We use the same array *Misspec* to store the misspeculated iterations as shown previously.
2. We remove the statements in a branch from the GPU kernels if the branch execution will cause previous or current iterations to misspeculate.

We explain the rationale behind this transformation using examples. Figure 6 gives the transformed code for the branches. In Figure 6(a), we insert an operation that marks the next iteration as misspeculated. The statements in the branch are kept since they will not pollute previous iterations. In Figure 6(b), we insert an operation that marks the current iteration as misspeculated. Since the current iteration is misspeculated, executing the statements in the branch is meaningless. Therefore, we remove the statements from the branch. In Figure 6(c), the operation inserted in the branch sets a special flag in $Misspec$. The flag indicates that all subsequent iterations including the current iteration are misspeculated. Since executing the statements in the branch also make previous iterations wrong, we remove the statements from the branch so that the results of previous iterations will be correct. In this branch, the current iteration is included in the misspeculated iterations because the statements in the branch need to be re-executed during recovery.

Having identified which iterations have misspeculated, we next identify the incorrect elements in the output array (i.e., incorrect results). Since the memory accesses are regular, we can use polyhedral tools to capture the mapping between the iterations and array elements. Once the mapping is known, the elements that are written in the misspeculated iterations can be easily found. These elements are incorrect and should be stored in array $Wrong$ as shown in the previous section. As in the previous section, we use GPU to perform parallel reductions on $Wrong_A$ and $Misspec$ to store the incorrect elements and misspeculated iterations in lists. This reduces the commit and recovery overhead on the CPU.

4 Scheduling

The synchronization granularity is critical to the GPU performance. Scheduling more iterations in one assignment may not give better performance because larger number of iterations in one assignment may cause excessive misspeculations. However, if we reduce the synchronization granularity to lower the misspeculation rate, we will also increase the kernel launching overhead. Thus, when scheduling iterations, we need to balance the above factors.

For *loops with irregular memory accesses* scheduling more iterations in one assignment will increase the chance of dependences between iterations. The optimal assignment size cannot be found since the cross-iteration dependences are unknown at compile time. Therefore we propose a runtime scheme.

In the first assignment, we schedule n/m iterations to the GPU, where m is the number of elements written in each iteration and n is the number of elements in the array. If we assign more than n/m iterations, there must exist two iterations that writes the same element. From the second assignment, we adjust the assignment size based on the observed misspeculation rate. If the misspeculation rate is higher than a predefined threshold, we halve the assignment size to reduce the misspeculation rate in the next round of scheduling. If the misspeculation rate stays zero for a number of consecutive iterations, we double the assignment size for better utilizing the large number of stream processors on the GPU. We do not increase the assignment size beyond n/m.

For *loops with irregular control flow*, in the first two cases in Figure 6, we schedule as many iterations as possible in one assignment. This is because the number of misspeculated iterations are almost solely determined by the number of iterations that execute the branches. Therefore we can only change the misspeculation rate if we schedule the iterations that execute the branches as the first or last iteration in an assignment, which is very unlikely.

For the third example in Figure 6, where all subsequent iterations are marked misspeculated if the branch is executed, we measure the average interval between two iterations that executes the branch at runtime and uses the interval as the assignment size when scheduling. This is because once an iteration executes the branch, all subsequent iterations are misspeculated. Therefore, we want the iterations that execute the branch to appear near the end of an assignment.

5 Commit and Recovery

The commit and misspeculation recovery are performed on the CPU. Figure 7 shows the pseudocode of commit and misspeculation recovery for the example given in Figure 3. The procedure is described next in detail.

```
copyFromGPUToCPU(Misspec);
copyFromGPUToCPU(Wrong$_A$);
if ( sizeof(Wrong$_A$) == 0 )
  copyFromGPUToCPU(A);
else {    // copy only correct part of array A
  prepend(-1, Wrong$_A$);
  append(size(A), Wrong$_A$);
  for (i=0; i<size(Wrong$_A$); i++)
    copyFromGPUToCPU(A[Wrong$_A$[i]+1 ... Wrong$_A$[i+1]-1]);
}
for (i=0; i<size(Misspec); i++)
  reexecute(Misspec[i]);
```

Fig. 7. Commit and misspeculation recovery for the example given in Figure 3.

1. We first copy the reduced arrays $Misspec$ and $Wrong$ from the GPU to to CPU. These arrays are required for the commit and misspeculation recovery. This step has very low overhead since the arrays are usually very small.
2. We then commit the data back to the CPU. For an array, if all elements are correct, we directly copy the whole array from the GPU to the CPU and overwrite the original array on the CPU. If misspeculation is detected, we scan array $Wrong$ and only copy the correct elements between the wrong elements stored in array $Wrong$.
3. Finally, we perform the misspeculation recovery step that reexecutes the misspeculated iterations on the CPU. For loops with irregular memory accesses, we scan array $Misspec$ and redo every iteration inside. For the loops with irregular control flow, we perform recovery depending on the misspeculation type. For the first two cases in Figure 6, where only one iteration is misspeculated with the execution of the branch, we redo every misspeculated

iteration in array *Misspec*. For the third case in Figure 6, where all subsequent iterations are misspeculated, we only reexecute the first iteration on the CPU. All remaining misspeculated iterations are assigned to the GPU in the next scheduling assignment.

Array *Wrong* can also be used to reduce the copy-in (copy from the CPU to GPU) overhead. For loops with irregular memory accesses, we do not know the array elements that will be accessed in an assignment of iterations. Therefore, we keep the whole array in the GPU memory. After the recovery procedure, all elements that are re-calculated on the CPU are stored in array *Wrong*. In the next assignment, we only copy the elements stored in array *Wrong* from the CPU to GPU. All other elements in the GPU memory are already up-to-date.

6 Programming Speculative Parallel Loops on GPUs

Our extensions to OpenMP basically tell the compiler which variables/branches to speculate on. Code offload and data transfer is handled by OpenMPC. To extend OpenMP for GPUs, previous works [2] have introduced the `target` clause, which can be applied to worksharing constructs:

```
#pragma omp for target(device)
```

This clause is also similar to the `target device` clause introduced in OpenMP 4.0. The intent of the `target` clause is to specify the device on which a given computation will be executed. The valid device specified by the `target` clause can be cuda, cell, and etc. We use `target(cuda)` for loop parallelization on GPUs.

6.1 Irregular Memory Accesses

To enable speculative parallelization of loops with irregular memory accesses, we introduce the `speculate` clause:

```
#pragma omp for speculate(array)
```

The `speculate` clause is designed to be used with worksharing constructs. Programmers can specify which arrays may cause cross-iteration dependences in the `speculate` clause. The memory accesses to these arrays will be monitored at runtime for misspeculation check. Although the compiler can identify the arrays that have irregular access patterns [18], not all of them will cause cross-iteration dependence at runtime. Programmers can better identify which arrays need to be monitored. This construct was useful in parallelizing a loop from the benchmark `ocean`, a Boussinesq fluid layer solver.

6.2 Irregular Control Flow

To enable speculative parallelization of loops with irregular control flow, we introduce the `branch` construct:

```
#pragma omp branch misspeculate(iterations)
```

The `branch` construct is designed to be inserted at the beginning of a branch that will cause cross-iteration dependences once its branch condition is true. The `misspeculate` clause is used to specify the misspeculated iterations if the branch is executed. A loop that is parallelized with worksharing constructs and contains the `branch` construct will be executed speculatively using the scheme described in Section 3.2.

The `iterations` expression in the `misspeculate` clause is designed to allow the following forms: absolute iterations, relative iterations, and iteration ranges. Absolute iterations can be expressed as `(i)`, where i is the iteration index. For example, `(10)` denotes the 10^{th} iteration. Relative iterations can be expressed as `(+i/-i)`, where i is the relative iteration index. For example, `(+1)` denotes the next iteration. Iteration ranges can be expressed as `(i:j)`, where i and j can be either absolute iteration index or relative iteration index. For example, `(+0:+4)` denotes the current iteration and next four iterations. Multiple iterations can be separated by comma in the expression. For example, `(-1,+1)` denotes the previous and next iterations. We found this construct useful in parallelizing a loop from benchmark `mdg`, which dynamically calculates water molecules in the liquid state at room temperature and pressure.

7 Evaluation

We implemented our framework whose core components consist of: a source-to-source translator and a runtime library. The translator is based on OpenMPC [9], which is an OpenMP-to-CUDA compiler. The programmers use pragmas to annotate the variables or control flow. The runtime library implements the core steps. We used an nVidia Tesla C1060 as our platform which includes a single chip with 240 cores organized as 30 streaming multiprocessors. The device is connected to a host system consisting of Intel Xeon E5540 processors. The machine has CUDA 3.0 installed. The benchmarks are summarized in Table 1.

Benchmark	Irregularities	% of time	# of pragmas
ocean	irregular memory accesses	45%	1
trfd	irregular memory accesses	6%	1
fftbench	irregular memory accesses	20%	1
mdg	irregular control flow	94%	2
strcat	irregular control flow	99%	2
gothic	irregular control flow	99%	2
alvinn	irregular control flow	97%	8

Table 1. Benchmark summary: benchmark name, type of irregularities, percentage of total execution time taken by the loop, and number of pragmas inserted.

7.1 Performance Overview

Figure 8 shows the speedups for the loops considered. The baseline is the sequential execution time of the loops on the host system. Bars higher (lower) than 1 indicate speedup (slowdown).

For each benchmark there are four bars – the first bar shows the performance of our technique with all optimizations. The rest of the bars show the performance with different optimizations individually omitted (for discussion of optimization results see Section 7.2). The speedups for the fully optimized version are between 3.62x and 13.76x, with five (out of seven) benchmarks achieving over 5x. *The speedups demonstrate the effectiveness of our framework in using GPUs for irregular loops considered.*

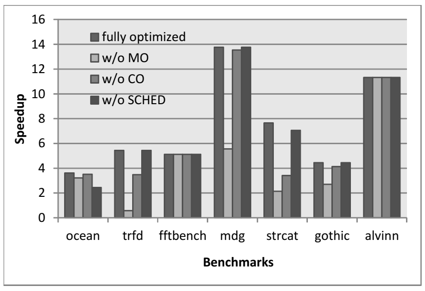


Fig. 8. Loop speedups for different optimization.

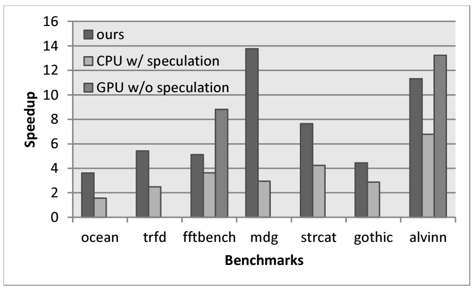


Fig. 9. A comparison with other approaches.

Figure 9 compares the loop speedups achieved by our approach, speculative parallelization on a CPU, and non-speculative parallelization on a GPU. We implemented non-speculative GPU versions for two benchmarks – `fftbench` and `alvinn` since they do not have cross-iteration dependences at runtime. The other benchmarks cannot be parallelized in a non-speculative way without changing the algorithms. From the figure, we can see that our approach always outperforms speculation on the CPU. This is because we can run the benchmarks using more concurrent threads on the GPU and the transactional memory on the CPU has high time overhead.

7.2 Effectiveness of the Optimizations

Let us examine Figure 8 to study the effectiveness of optimizations. The second bar ("w/o MO" in Figure 8) gives the performance without misspeculation optimization (i.e., misspeculation detection without misspeculation localization and re-executing all iterations on the host system once misspeculation is detected). The third bar ("w/o CO" in Figure 8) shows the performance without copy optimization (i.e., copying all data between CPU and GPU for every assignment of iterations). The last bar ("w/o SCHED" in Figure 8) shows the performance without our scheduling policy (i.e., scheduling all iterations to the GPU in the first assignment). These three groups of bars are intended to show the importance of misspeculation localization, copy optimization, and our scheduling policy. Figure 10 shows the misspeculation rate (i.e., the number of iterations re-computed on the CPU divided by the total number of iterations) with and without the optimizations. We can see that our optimizations greatly reduces the misspeculation rate. The details of each benchmark will be described next.

For the `ocean` benchmark, our scheduling policy improves the performance by around 32% over the one ("w/o SCHED") with minimum speedup. Our scheduling policy decreases the size of each assignment so that there is almost no misspeculation after the first few assignments. Misspeculation and copy optimizations do not improve the performance much since no misspeculation occurs in most assignments of iterations. Misspeculation optimization improves the performance of `trfd` greatly because there is always only one misspeculation in each execution of the loop. Without misspeculation optimization, we have to always re-execute all iterations on the CPU, which apparently will cause slowdown.

Benchmark	Misspeculation Rate	
	w/o opts	w/ opts
ocean	0.53%	0.14%
trfd	100%	0.52%
fftbench	0.0%	0.0%
mdg	0.0018%	0.0018%
strcat	41.23%	2.10%
gothic	0.92%	0.67%
alvinn	0.0%	0.0%

Fig. 10. Misspeculation rates

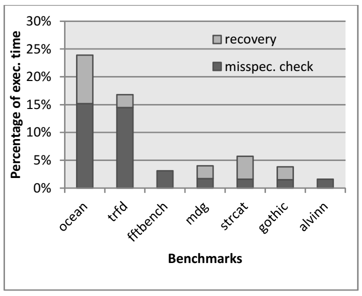


Fig. 11. Time overhead.

Copy optimization improves its performance by 36%. The copy-in (i.e., copy from CPU to GPU) overhead is greatly reduced as we only copy elements that are re-computed on the CPU (for recovery) to the GPU memory for every assignment. Our scheduling policy does not have much impact on the performance of `trfd` since there is only one misspeculation for each execution of the loop. The speedup of `fftbench` is partially offset by the number of memory access tracking. Since no cross-iteration dependences occur at runtime for the test input, misspeculation localization and recovery are never performed. Therefore, none of the optimizations has a performance impact. The speedup of `mdg` is high because the loop body has a lot of computation which can fully utilize the massively parallel architecture of the GPU. Also, only a few iterations execute the branch at runtime. Therefore, most computations are performed in parallel. With misspeculation optimization, we only re-execute the first misspeculated iteration on the CPU. The rest of the misspeculated iterations are assigned to the GPU in the next assignment of iterations. If we re-execute all misspeculated iterations on the CPU, the performance will be degraded by 60%. The speedup for `strcat` is good since the misspeculation rate is very low due to the rapid growth of buffer size. Misspeculation optimization improves the performance by 72% for the same reason as in `mdg`. Copy optimization is critical for performance of `strcat`. By avoiding copying the correct results back and forth between the CPU and GPU, we improve the performance by 55%. In `gothic`, a misspeculation makes all subsequent iterations incorrect. Thus, misspeculation optimization greatly reduces iterations executed on the CPU and improves the performance. The speedup of `alvinn` is high because no misspeculation happens in our experiments. The increment of the *weight* pointer does not change according to the input and thus the optimizations make no impact.

Figure 11 shows the time overhead (recovery and misspeculation check) as the percentage of the loop execution time. The time of computation and copy is a necessity for all GPU computations. The misspeculation check overhead for the `ocean` benchmark is the highest among all benchmarks because it requires memory access tracking, and its misspeculation check needs to detect both RAW/WAR and WAW dependences. The recovery overhead is high for `ocean` since the first few schedules of the loop cause many misspeculations. The misspeculation check overhead for `trfd` is lower than `ocean` since its misspeculation check only needs to detect WAW dependences. In `fftbench` and `alvinn`, since no cross-iteration dependence occurs for the test input, misspeculation localization and recovery are never performed. The recovery overhead for `mdg` and `gothic` is low since we only re-execute the first misspeculated iteration on the CPU. In `strcat`, the overhead for misspeculation check is low since we only monitor the execution of the branch that reallocates the buffer.

8 Related Work

Speculative execution has been used to explore task-level parallelism on multi-GPU systems [6]. Usually, the runtime system must block the execution of a kernel until its predecessors in the control flow graph (CFG) have finished. On multi-GPU systems, the performance is limited by the runtime system's inability to execute more kernels in parallel. Diamos and Yalamanchili [6] alleviated this problem by speculating the control flow between kernels. Unlike their work, this paper explores thread-level speculative parallelism in a kernel. An exploratory study has been done for speculative execution on GPUs [10, 11]. They explored the hardware implementation of speculative execution operations on GPU architectures to reduce the software performance overheads. The GPU-TLS system [19] adapted CPU speculative parallelization techniques for GPU use. Like previous speculative parallelization works for CPUs, it only checks RAW dependences at runtime and handles other types of dependences (i.e., WAW and WAR) by keeping the sequential order of iteration commits. Paragon [14] is the work closest to ours. It is a framework to speculatively run possible parallel loops on a GPU. However, unlike our framework, Paragon is not able to locate the misspeculated iterations. Therefore, on misspeculation, Paragon has to throw away any result on the GPU and re-execute the entire loop sequentially on the CPU.

Instead of causing cross-iteration dependences, irregularities may severely limit the efficiency of GPU computing due to the warp organization and SIMD execution model of GPUs. Zhang et al. [20] proposed runtime optimizations with the support of a CPU-GPU pipeline scheme to remove thread divergences. Baskaran et al. [3] use a polyhedral compiler model to optimize affine memory accesses in regular loops. Yang et al. [18] presented an optimizing compiler for memory bandwidth enhancement, data reuse, parallelism management, etc.

Several programming models have been proposed for GPU computing including OpenCL [15], CUDA [12], OpenACC [16], PGI Accelerator [17], OmpSs [2] which is based upon OpenMP standard [4], and Par4All [1]. *None of these programming models support speculative parallelization for GPU computing.*

9 Conclusion

We presented a framework for employing GPUs to speculatively parallelize loops that may have cross-iteration dependences at runtime due to irregularities. Several optimizations were proposed to improve the performance, including parallelizing misspeculation check on the GPU, optimizing the procedure of result committing and misspeculation recovery, and adaptive scheduling policy for different types of cross-iteration dependences. Our implementation achieves 3.62x-13.76x speedup for the seven parallelized loops.

References

1. Mehdi Amini, Onig Goubier, Serge Guelton, Janice Onanian Mcmahon, Francois xavier Pasquier, Gregoire Pean, and Pierre Villalon. Par4All: From convex array regions to heterogeneous computing. In *IMPACT*, 2012.
2. Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An extension of the StarSs programming model for platforms with multiple GPUs. In *Euro-Par*, pages 851–862, 2009.
3. Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *ICS*, pages 225–234, 2008.
4. L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE computational science & engineering*, 5(1):46–55, 1998.
5. Francis H. Dang, Hao Yu, and Lawrence Rauchwerger. The r-lrpd test: Speculative parallelization of partially parallel loops. In *IPDPS*, 2002.
6. G. Diamos and S. Yalamanchili. Speculative execution on multi-gpu systems. In *IPDPS*, pages 1–12, 2010.
7. C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI*, pages 223–234, 2007.
8. Wuchun Feng and Shucai Xiao. To GPU synchronize or not GPU synchronize? In *ISCAS*, pages 3801–3804, 2010.
9. Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP*, pages 101–110, 2009.
10. Shaoshan Liu, Christine Eisenbeis, and Jean-Luc Gaudiot. Speculative execution on GPU: An exploratory study. In *ICPP*, pages 453–461, 2010.
11. Shaoshan Liu, Christine Eisenbeis, and Jean-Luc Gaudiot. Value prediction and speculative execution on GPU. *IJPP*, 39(5):533–552, 2011.
12. John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.
13. Lawrence Rauchwerger and David Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *PLDI*, pages 218–232, 1995.
14. Mehrzad Samadi, Amir Hormati, Janghaeng Lee, and Scott Mahlke. Paragon: Collaborative speculative loop execution on gpu and cpu. In *GPGPU*, pages 64–73, 2012.
15. John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, 2010.
16. Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC: first experiences with real-world applications. In *Euro-Par*, pages 859–870, 2012.
17. Michael Wolfe. Implementing the pgi accelerator model. In *GPGPU*, 2010.
18. Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. In *PLDI*, pages 86–97, 2010.
19. Chenggang Zhang, Guodong Han, and Cho-Li Wang. GPU-TLS: an efficient runtime for speculative loop parallelization on GPUs. In *CCGrid*, pages 120–127, 2013.
20. Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In *ICS*, pages 115–126, 2010.