GPU-TLS: An Efficient Runtime for Speculative Loop Parallelization on GPUs

Chenggang Zhang, Guodong Han, Cho-Li Wang
Department of Computer Science
The University of Hong Kong
Hong Kong, China
{*cgzhang, gdhan, clwang*}*@cs.hku.hk*

***Abstract*—Recently GPUs have risen as one important parallel platform for general purpose applications, both in HPC and cloud environments. Due to the special execution model, developing programs for GPUs is difficult even with the recent introduction of high-level languages like CUDA and OpenCL. To ease the programming efforts, some research has proposed automatically generating parallel GPU codes by complex compile-time techniques. However, this approach can only parallelize loops 100% free of inter-iteration dependencies (i.e., DOALL loops). To exploit runtime parallelism, which cannot be proven by static analysis, in this work, we propose GPU-TLS, a runtime system to speculatively parallelize possibly-parallel loops in sequential programs on GPUs.**

GPU-TLS parallelizes a possibly-parallel loop by chopping it into smaller sub-loops, each of which is executed in parallel by a GPU kernel, speculating that no inter-iteration dependencies exist. After dependency checking, the buffered writes of iterations without mis-speculations are copied to the master memory while iterations encountering mis-speculations are re-executed. GPU-TLS addresses several key problems of speculative loop parallelization on GPUs: (1) The larger mis-speculation rate caused by larger number of threads is reduced by three approaches: the loop chopping parallelization approach, the deferred memory update scheme and intra-warp value forwarding method. (2) The larger overhead of dependency checking is reduced by a hybrid scheme: eager intra-warp dependency checking combined with lazy inter-warp dependency checking. (3) The bottleneck of serial commit is alleviated by a parallel commit scheme, which allows different iterations to enter the commit phase out of order but still guarantees sequential semantics.

Extensive evaluations using both microbenchmarks and real-life applications on two recent NVIDIA GPU cards show that speculative loop parallelization using GPU-TLS can achieve speedups ranging from 5 to 160 for sequential programs with possibly-parallel loops.

***Keywords*-GPGPU; Speculative Loop Parallelization; Thread-Level Speculation (TLS); GPU-TLS**

I. Introduction

During the last few years, we have witnessed the dominance of multicore processors in high performance computing and their success in continuing the computing performance leap beyond the decade-long approach of raising the clock speed. It is however difficult to envision hundreds of traditional CPU cores to pack on a chip to achieve continual performance growth, as well as low cost and energy. The use of hundreds of accelerator cores (such as GPUs) in conjunction with a handful of host CPU cores, on the other hand, appears to be a sustainable roadmap. With the promise of cheaper and greener HPC environment, the interest in GPUs for efficient coprocessing is at an all-time high.

Although initial success has been achieved, up to now GPUs can only accelerate data-parallel loops with static parallelism. Exploiting static parallelism only may already be enough for some domains like image processing, computation in which is usually embarrassingly parallel with statically analyzable parallelism. However, in some other domains like computational physics or chemistry, loops with dynamic parallelism are common. As the parallelism cannot be proven statically, existing techniques fail to parallelize these loops on GPUs even if they may have high degree of parallelism at the runtime, limiting the scope of GPUs' applicability in general-purpose applications. Designing software solutions to support parallel execution of workloads with dynamic parallelism on GPUs is important to continue the success of GPGPU.

Thread Level Speculation (TLS) is a technique proposed to parallelize loops with dynamic parallelism on multi-core or multi-processor CPUs. Although there have been many TLS designs on CPUs, none of them works efficiently when ported to GPUs, due to the large differences between GPUs and CPUs in both hardware architecture and execution model. Most design dimensions of TLS need to be reviewed on GPUs: (1) TLS with a serial commit mismatches with GPUs' scalable hardware resources. A scalable design is desired. (2) The meta-data used must be memory-efficient due to the relatively limited memory of GPUs. (3) More effort should be made to reduce the mis-speculation rate. This is because the larger number of threads increases the possibility of mis-speculations among speculative threads.

Based on careful analysis of GPU features, we have designed GPU-TLS, an efficient TLS runtime for GPUs. We make the following contributions:

- We propose a loop speculative parallelization framework on GPUs using sliding windows. This design decouples the memory overhead from the loop iteration number and utilizes partial parallelism in a large loop incrementally.
- We make use of the lockstep execution model in GPUs and propose an intra-warp value forwarding technique, which reduces mis-speculation rate.
- We propose a hybrid dependency checking scheme with eager intra-warp and lazy inter-warp checking. The eager intra-warp checking enables early abort when there are RAW dependencies that cannot be satisfied. The lazy inter-warp checking is rather lightweight and does not introduce any false positive mis-speculation.
- We propose a deadlock-free parallel commit scheme, which avoids the serial commit bottleneck in existing TLS designs and is scalable to thousands of GPU threads.

Our experiments show that GPU-TLS, when used to parallelize loops with dynamic parallelism, achieves speedups ranging from 5 to 160 on two platforms shipped with different NVIDIA GPU cards.

II. GPUs and Adapting Software TLS to GPUs

In this section, we describe the GPU architecture and execution model, systematically survey existing software TLS designs and analyze their deficiency when ported to GPUs.

A. GPU Architecture and Execution Model

The modern Graphical Processing Unit (GPU) architecture consists of two major parts: the computation sub-system and the memory sub-system. The computation sub-system is composed of streaming multiprocessors, which contain a series of simple streaming processors. GPU features a very complicated memory hierarchy. It is shipped with both off-chip and on-chip memory modules. The off-chip memory is large in size (e.g., 3GB for C2050). The on-chip memory is small in size but features a short access latency. Different vendors release their software development tools and APIs to support programming on GPU, such as CUDA from NVIDIA and OpenCL from Khronos group. Take CUDA for example, it defines kernels consisting of a grid of threads, which is further divided into a series of thread blocks. Each thread block is comprised of some GPU threads and these threads are divided into multiple groups, each of which contains 32 threads and is named as a warp. Each thread in the same warp executes the same instruction by means of lock-step synchronization (SIMD execution) in the same Streaming Multiprocessor (SM).

B. Deficiency of Existing TLS Designs when Ported to GPUs

Most existing TLS designs [1] [2] [3] have a serial commit phase. The serial commit bottleneck is a mismatch with the scalable hardware resources of GPUs. In the literature, there are only two TLS systems free of this bottleneck. The system proposed by Peter *et al.* [4] simulates the hardware TLS design in software and uses a huge access structure to store the access information. The other system proposed by Cosmin *et al.* [5] uses direct update memory versioning scheme, in which the commit is done on the fly along with each speculative write. This design has the downside of introducing potential false positive mis-speculations. The reason is that with direct update scheme, WAR (Write After Read) and WAW (Write After Write) dependencies can also result in mis-speculations apart from RAW (Read After Write) dependencies. As the number of speculative threads is large in GPUs, the possibility of false dependencies (i.e., WAR and WAW) can not be neglected. To design an efficient TLS with thousands of GPU threads, we need a scalable commit scheme.

As for dependency checking, there are basically four different schemes in the literature. The scheme by comparing read set with version numbers [1] is highly coupled with the serial commit protocol and cannot work if we replace serial commit with a scalable parallel commit scheme. All-software TLS design using load vector and store vector to do dependency checking [4] has been proven to cause too much memory overhead [5] and is not suitable for GPUs, which have limited memory. Among the existing dependency checking schemes in the literature, the one using load vector [3] could probably be the most scalable one. However, it has a disadvantage of introducing potentially large degree of mis-speculations. The reason is that the RAW dependency is detected by the producer thread by comparing the entries in the write set and the load vector. For a given address, when the corresponding entry in the load vector is larger than the ID of the checking thread, the thread has to conservatively reckon that all succeeding threads have violated RAW dependencies. TLS designs on GPUs call for a new dependency checking scheme.

III. GPU-TLS Design

A. Overview

We show the framework of loop parallelization using GPU-TLS in Figure 1. Through analysis and profiling, programmers/compilers select possibly-parallel loops in sequential programs and transform them using the GPU-TLS library. The transformed loops are then speculatively executed on GPUs. For each selected loop, we chop it into several sub-loops and for each sub-loop we launch a GPU kernel to speculatively execute the loop iterations. The speculative execution of a GPU kernel has four phases: speculative execution, dependency checking, commit and mis-speculation recovery. In the first phase, GPU executes the loop iterations in parallel by speculating that inter-iteration dependencies do not exist. During the execution, each thread buffers the possibly unsafe memory updates instead of updating the master memory. Also, the memory accesses are tracked using meta-data to aid the later mis-speculation checking. The dependency checking phase checks whether the speculation is successful using the memory access meta-data and reports the potential violation locations (i.e., which speculative threads have violated inter-iteration dependencies). For threads that are not reported to have violated dependencies by the dependency checking, the commit phase copies their buffered memory updates to the master memory. When launching a GPU kernel with k threads, $violation\_idx$ is initialized to k. During the dependency checking phase, this variable is updated to show the earliest violation location. Suppose after kernel execution $violation\_idx$ equals p ($0 < p \leq k$), we can tell that threads $T_0 \sim T_{p-1}$ have executed correctly and committed the memory updates to the master memory. If $violation\_idx$ equals k, we know that no mis-speculation has happened and no mis-speculation recovery is needed. Otherwise, the mis-speculation recovery of threads $T_p \sim T_{k-1}$ is done by re-executing them. Instead of launching a new GPU kernel to re-execute the failed threads only, we use a sliding window approach by combining the executions of some iterations in the new sub-loop together with the mis-speculation recovery of failed iterations in the old sub-loop (shown in the right bottom of Figure 1).

B. Intra-warp Value Forwarding

Existing software TLS designs adopt four different approaches to satisfy potential inter-thread RAW dependencies [6]. The "speculation" approach speculates that there are no inter-iteration RAW dependencies and speculative reads return values in the master memory. Instead, in the "value prediction" approach, a speculative read returns the value produced by a value predictor. The "value forwarding" approach does best-effort value forwarding to fetch the potential write values

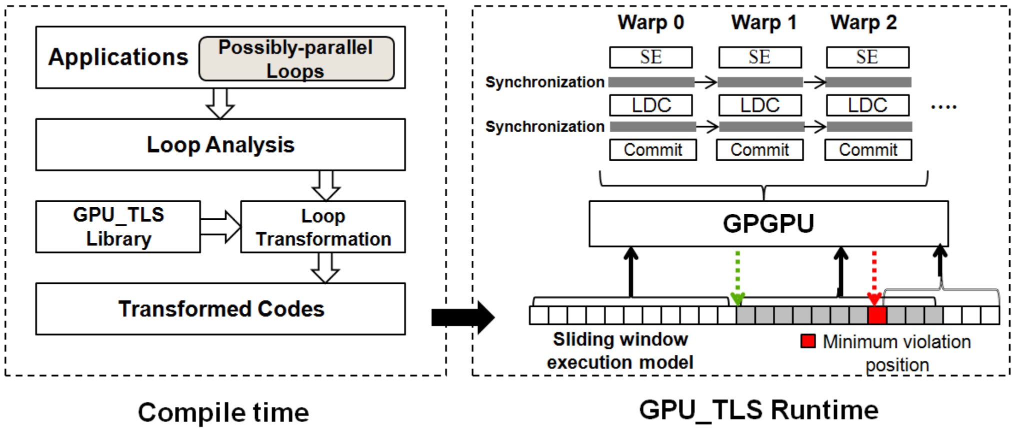


Figure 1. GPU-TLS Overview

produced by preceding threads. The "synchronization" approach assumes there are inter-thread RAW dependencies and waits until the producer thread has produced the values on speculative reads.

When doing speculative loop parallelization on GPUs, the "speculation" approach is not desirable because we have thousands of speculative threads and extremely optimistic speculation may encounter many mis-speculations and result in poor performance. At the same time, the "synchronization" approach may lead to deadlock under some patterns of RAW dependencies or result in a waste of hardware resources. The pioneering work of value prediction on GPUs reports that value prediction can hardly gain speedup [7]. As a result, in GPU-TLS we adopt the "value forwarding" approach. In previous TLS designs with value forwarding, the scope of the value forwarding is either the whole loop [4] or a sliding window [8]. In GPU-TLS, we choose a warp as the scope to implement value forwarding (this is named intra-warp value forwarding). The benefit of this design is two-fold: (1) no explicit synchronization is needed when doing the value forwarding due to the lockstep execution model; (2) we could use on-chip share memory to speedup the value forwarding.

We show the algorithm of speculative read with intra-warp value forwarding in Algorithm 1. On a speculative read from the address r_addr in thread T_k, we satisfy intra-warp RAW dependencies. To achieve this, basically we go through the write sets of thread T_k and the preceding threads in the same warp to forward the appropriate write value to the read. This introduces some set traversal overhead. The meta-data $minWriter\_ary$ helps us to alleviate the overhead by avoiding some unnecessary set traversal: we only check threads that may have written to r_addr instead of all preceding threads in the warp (line 13). We only update the read set when no prior writes to r_addr exist in the same thread. If a speculative read operation consumes the write value produced by a write operation in a preceding thread, we record the ID of that thread in the meta-data $rProSet\_ary$. Otherwise, we store the value "-1" in $rProSet\_ary$. This meta-data will be used in eager intra-warp dependency checking. Also, we need to check and update the meta-data $maxReader\_ary$ if necessary.

Algorithm 1 Speculative Read with Intra-warp Value Forwarding

Input: memory address to read from: r_addr, thread ID: k
Output: the speculative read value

```
1:  $idx \leftarrow$ hash($r\_addr$)
2:  $minWriter \leftarrow minWriter\_ary[idx]$
3:  if $minWriter > k$ then
4:      go to line 23
5:  else
6:      for each entry $w\_addr$ in the write set of thread $T_k$ do
7:          if $w\_addr == r\_addr$ then
8:              $r\_value \leftarrow$ the entry in $wValueSet\_ary$
9:              go to line 27
10:         end if
11:     end for
12:     $warpFirstTh \leftarrow \lfloor \frac{k}{W} \rfloor \cdot W$
13:     for $i \leftarrow k$-1 to MAX($warpFirstTh$, $minWriter$) do
14:         for each entry $w\_addr$ in the write set of thread $T_i$ do
15:             if $w\_addr == r\_addr$ then
16:                 $r\_value \leftarrow$ the entry in $wValueSet\_ary$
17:                 $rProSet\_ary[\, rIdx\_ary[k] \,][\, k \,] \leftarrow i$
18:                 go to line 25
19:             end if
20:         end for
21:     end for
22: end if
23: $r\_value \leftarrow *r\_addr$
24: $rProSet\_ary[\, rIdx\_ary[k] \,][\, k \,] \leftarrow$ -1
25: update read set of thread $T_k$
26: update $maxReader\_ary$ if necessary
27: return $r\_value$
```

C. Hybrid Dependency Checking

GPU-TLS adopts a hybrid dependency checking scheme: eager intra-warp and lazy inter-warp checking. During the **SE** phase, each speculative write checks whether threads in the same warp have violated intra-warp RAW dependencies (Algorithm 2). This is achieved by checking the read sets of succeeding threads in the same warp. The set traversal overhead could be alleviated with the help of the meta-data $maxReader\_ary$: we only traverse the read sets of threads [k+1, MIN($warpLastTh$, $maxReader\_ary$[hash(w_addr)])] (line 4), in which $warpLastTh$ denotes the last thread in the warp. For each succeeding thread T_i, we check whether address w_addr appears in its read set (line 5-6). If this happens, we need to further check the value of the corresponding entry in $rProSet\_ary$. There are three possible cases. In the first case, if the value is "-1", we know that thread T_i has read the value from the master memory (Figure 2(a)). In the second case, if the value is not "-1" but is smaller than or equal to k, we can infer that thread T_i has read the value produced by a preceding thread in the same warp (Figure 2(b)). In the third case, if the value is larger than k, we can tell that thread T_i has read the right value (Figure 2(c)). In the first two cases, we find RAW dependency violations and set the corresponding element in the warp dependency array to "TRUE" (line 6-9) because we can tell that thread T_i has consumed a wrong value.

In the lazy dependency checking phase, we only check inter-warp RAW dependencies. The reason is that with the intra-warp value forwarding and eager dependency checking, the intra-warp RAW dependencies must have either been satisfied or the dependency violations have been detected. We do the lazy dependency checking with the help of the meta-data $minWriter\_ary$. Basically, to check whether the thread T_k has violated RAW dependencies, we go through its read set and for each entry r_addr, we first check whether $minWriter\_ary$[hash(r_addr)] is smaller than the ID of the first thread in the warp. If the condition satisfies, we can tell that a thread in a preceding warp has written to r_addr, but this does not necessarily mean a RAW dependency violation has happened. We need to further check whether T_k has been forwarded a value by intra-warp value forwarding (line 5-8). If this happens, we can infer that no RAW dependencies have been violated (Figure 3(b)). Otherwise, we find one RAW dependency violation (Figure 3(a)). Note that since the current thread can arrive at the lazy dependency checking phase, it's impossible that intra-warp RAW dependency violations have happened (Figure 3(c)).

Algorithm 2 Speculative Write with Eager Intra-warp Dependency Checking

Input: memory address to write to: w_addr, speculative write value: w_value, thread ID: k

```
1: $idx \leftarrow$ hash($w\_addr$)
2: $maxReader \leftarrow maxReader\_ary[idx]$
3: $warpLastTh \leftarrow (\lfloor \frac{k}{W} \rfloor + 1) \cdot W$ - 1
4: for $i \leftarrow k$+1 to MIN($warpLastTh$, $maxReader$) do
5:     for each entry $r\_addr$ in the read set of thread $T_i$ do
6:         if $r\_addr == w\_addr$ && the corresponding entry in $rProSet\_ary \leq k$ then
7:             calculate warp ID $wid$ of thread $T_i$
8:             $dependency\_found[wid] \leftarrow TRUE$
9:         end if
10:    end for
11: end for
12: update the write buffer of thread $T_k$
13: update $minWriter\_ary$ if necessary
```

D. Scalable Parallel Commit

Any parallel execution of a sequential loop needs to maintain the sequential semantics for correctness. For example, if two iterations I_i and I_j ($i < j$) in a loop both write to the memory address $addr$, any parallelization of the loop needs to make sure that the value in $addr$ after the parallel execution is that written by I_j. To achieve this, most existing software TLS solutions have adopted a very conservative approach by enforcing iterations to enter the commit phase in iteration order. For example, iteration I_3 can only do the commit after I_2, the commit of which is started only after I_1. The performance penalty of serial commit may not be that serious due to the limited number of speculative threads in CPUs, but the situation in GPUs is totally different: letting thousands of speculative threads do the commit serially could easily become a performance bottleneck. Also, the lockstep execution model of threads in a warp make it easy to encounter deadlock if a serial commit is adopted. To solve the above problems, we have proposed a parallel commit scheme in which speculative threads could enter the commit phase out of iteration order after all the preceding threads have finished their LDC phases. The sequential semantic is guaranteed by additional checking when committing each speculative write. The efficiency and correctness of the parallel commit is guaranteed by the lock-free algorithm 4. The algorithm makes sure that the commit never gets dead-lock, especially in the same warp. We use an additional piece of meta-data named $maxWriter\_ary$ to record the ID of the maximum thread that has ever written to each memory address. Before a thread T_k commits each (w_addr, w_value) pair in its write buffer, it first checks whether the value of the entry $maxWriter\_ary$[hash($addr$)] is smaller than or equal to k (line 7). If this condition holds, T_k carries out the commit and updates $maxWriter\_ary$[hash($addr$)] to k (line 8-10). On the contrary, if $maxWriter\_ary$[hash($addr$)] is larger than k, we know that a later thread has committed a speculative write to $addr$; to avoid WAW dependency violations, T_k discards the commit of this speculative write.

Algorithm 3 Lazy Dependency Checking

```
1: for each entry $r\_addr$ in the read set of thread $T_k$ do
2:     $idx \leftarrow$ hash($r\_addr$)
3:     $minWriter \leftarrow minWriter\_ary[idx]$
4:     $warpFirstTh \leftarrow \lfloor \frac{k}{W} \rfloor \cdot W$
5:     if $minWriter < warpFirstTh$ && the corresponding entry in $rProSet\_ary$ == -1 then
6:         calculate warp ID $wid$ of thread $T_k$
7:         $dependency\_found[wid] \leftarrow TRUE$
8:     end if
9: end for
```

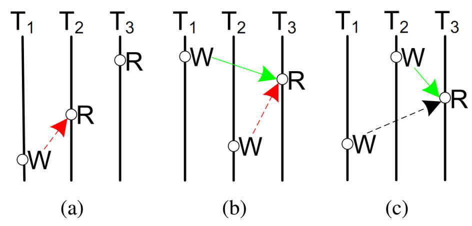


Figure 2. Different cases in eager intra-warp dependency checking: (a)no prior intra-warp value forwarding exists, (b)prior read fetches a wrong value through intra-warp value forwarding, (c)prior read fetches the correct value through intra-warp value forwarding.

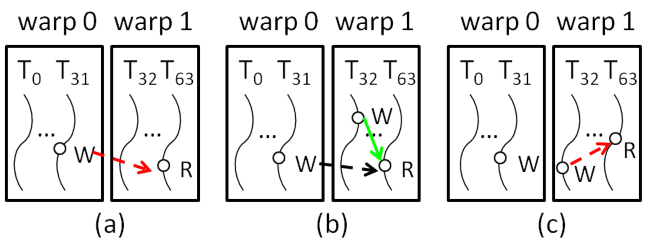


Figure 3. Different cases in lazy inter-warp dependency checking: (a)inter-warp RAW without intra-warp value forwarding, (b)inter-warp RAW with intra-warp value forwarding, (c)both inter- and intra-warp RAW without value forwarding

Algorithm 4 Deadlock-free Parallel Commit

```
1:  for each ($w\_addr, w\_value$) in write buffer of thread $T_k$ do
2:      $idx \leftarrow$ hash($w\_addr$)
3:      $exitTheLoop \leftarrow FALSE$
4:      while $!exitTheLoop$ do
5:          $flag \leftarrow$ atomicCAS(&($lock\_vector[idx]$), 0, 1)
6:          if $flag$ == 0 then
7:              if $maxWriter\_ary[idx] \leq k$ then
8:                  $*w\_addr \leftarrow w\_value$
9:                  $maxWriter\_ary[idx] \leftarrow k$
10:                 __threadfence()
11:             end if
12:             $lock\_vector[idx] \leftarrow 0$
13:             $exitTheLoop \leftarrow TRUE$
14:         end if
15:     end while
16: end for
```

IV. Evaluation Methodology

A. Platform Settings

We carry out the experiments on two platforms, as shown in Table I. Platform 1 is an IBM iDataPlex dx360 M3 server and Platform 2 is a desktop PC. On both platforms, the GPU cards are connected to the main board through PCIe x16 Gen 2 bus. The OS installed is 64-bit Scientific Linux 5.5 and the compilation flag used in gcc and nvcc is "-O2". In all experiments, the default GPU configuration of 48KB shared memory and 16KB L1 cache is used.

Table I
Experiment platforms

	Platform 1	**Platform 2**
CPU	Intel Xeon X5650	Intel Core i7 870
CPU frequency	2.66GHz	2.93GHz
Cores per socket	6	4
Socket count	2	1
Total core count	12	4
Host memory	48GB ECC DDR3 RAM, 1333MHz	8GB DDR3 RAM, 1333MHz
GPU	NVIDIA Tesla M2050	NVIDIA GeForce GTX580
CUDA capability	2.0	2.0
GPU clock rate	1.15GHz	1.54GHz
SM #	14	16
SP # per SM	32	32
Total SP #	448	512
GPU warp size	32	32
Register # per SM	32768 32-bit	32768 32-bit
Shared memory per SM	48 KB/16KB (configurable)	48 KB/16KB (configurable)
L1 cache size	16KB/48KB (configurable)	16KB/48KB (configurable)
L2 cache size	768KB	768KB
Global memory size	3GB	1.5GB
Memory interface width	384-bit	384-bit
Memory bandwidth	148GB/s	192.4GB/s
OS kernel	Linux 2.6.18-194.3.1.el5 x86_64	Linux 2.6.18-194.3.1.el5 x86_64
CUDA Toolkit	64-bit CUDA 4.0	64-bit CUDA 3.2
C Compiler	GCC 4.1.2	GCC 4.1.2
CUDA Compiler	64-bit NVCC 4.0, V0.2.1221	64-bit NVCC 3.2, V0.2.1221

B. Benchmarks

1) Microbenchmarks: To facilitate micro-benchmarking, we design a synthetic loop (shown in Figure 4) to simulate loops with different characteristics. N represents the number of loop iterations, which is an important indicator of the degree of parallelism within a loop. Sw and Sr decide the size of the write set and read set respectively. M controls the workload in each iteration. We use accesses to a shared array A through subscripted subscripts (using the r and w arrays) to simulate statically unanalyzable dependencies. By configuring the values of elements in the r and w arrays, we can simulate loops with different dependency conditions in terms of dependency type (i.e., RAW, WAR, WAW), number of dependencies, etc. By varying the values of different parameters (N, Sw, Sr, M, w, r), we can simulate loops with different patterns, which enables us to have a deep understanding of the performance of GPU-TLS.

```
#define N 7168 /*iteration number*/
#define Sw 5     /*write set size*/
#define Sr 5     /*read set size*/
#define M 10
/*the workload per iteration*/

#define MAX(a, b) ((a>b)?(a):(b))
#define S MAX(Sw, Sr)

double func(double v){
   for(int i = 0; i < M; i++)
      dumb_computation(v);
}
```

```
void main(){
   for(int i = 0; i < N; i++){
      double temp = 0;

      for(int j = 0; j < S; j++){
         if(j < Sr)
            temp = A[ r[i * Sr + j] ];
         if(j < Sw)
            A[ w[i * Sw + j] ] = func(temp);
      }
   }
}
```

Figure 4. Synthetic loop used in the experiments

2) Real-life applications: Besides the synthetic loop, we also use three real-life benchmark programs in the experiments.

The first program is a Molecular Dynamics (MD) loop [9] as illustrated in listing 1. In the nested loop, the accesses to array Y go through one level of redirection by the $partners$ array. The parallel reads and writes to the shared array Y issued in the outer loop iterations may pose some dependencies. We speculatively parallelize the outer loop by speculating that accesses to Y in different iterations will not violate RAW dependencies.

The second program is a Computational Fluid Dynamics (CFD) loop [9] shown in listing 2. In this loop, the subscripts ($n1$ and $n2$) in accessing a shared array Y depend on the values of elements in an array $edge$ and cannot be statically decided. We speculatively parallelize this loop by speculating that accesses to Y in different iterations will not result in RAW dependency violations.

The third program is blackscholes, a benchmark from Intel RMS benchmark [10]. It uses the Black-Scholes partial differential equation [11] to calculate the prices for a portfolio of European options analytically. We need very little speculation in this program: the statically unanalyzable inter-iteration dependencies stem from the potential updates to a shared variable $numOfErrors$ when errors occur in the pricing; the only speculation needed is that an error will not appear.

Listing 1. A Molecular Dynamics (MD) loop

```
ATOM X[NumAtoms], Y[NumAtoms];
ATOM *partners[NumAtoms];

for (i=0; i<NumAtoms; i++) {
  for each element j in partners[i] {
    Y[i] += force(X[i], X[j]);
    Y[j] += force(X[i], X[j]);
  }
}
```

Listing 2. A Computational Fluid Dynamics (CFD) loop

```
NODE X[NumNodes], Y[Numnodes];
struct {
  NODE LeftNode, RightNode;
}edge[NumEdges];

for (i=0; i<NumEdges; i++) {
  n1 = edge[i].LeftNode;
  n2 = edge[i].RightNode;
  Y[n1] += f(X[n1], X[n2])
  Y[n2] += g(X[n1], X[n2])
}
```

V. Experimental Results and Analysis

A. Overall Speedups

We first carry out experiments to evaluate the performance potential of GPU-TLS using the selected real-life applications.

We vary the problem size (i.e., the number of iterations) by using different values of $NumAtoms$ in MD, $NumEdges$ in CFD and $NumOptions$ in blacksholes. The values we use are 512*k (k = 1, 2, 3, $\cdots$, 16). For each selected problem size, we first run the loop sequentially on CPU and then execute the parallel version enabled by GPU-TLS on GPU. We record the sequential (T_s) and parallel execution time (T_p) and calculate the speedups as T_s/T_p. We carry out the experiments on both Platform 1 and 2 and the speedups achieved when there are no runtime inter-iteration RAW dependencies are shown in Figure 5 and 6 respectively.

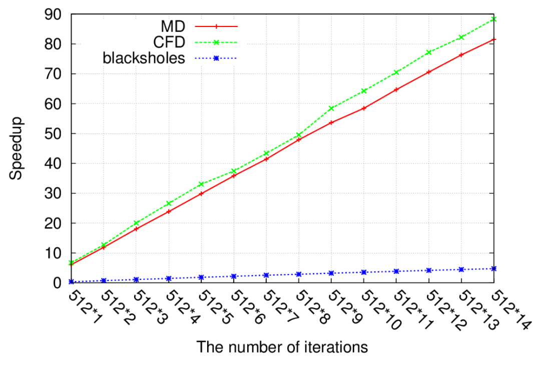


Figure 5. Speedups achieved for different problem sizes on Platform 1

We have several observations from the two figures. Firstly, as the problem size increases, we can generally get larger speedups. For example, on Platform 2, when the loop has only 512 iterations, the speedup we can achieve in CFD is 16.4, while the speedup value goes as high as 104 when the number of iterations increases to 512*16. This shows that to make the best use of GPUs the problem size should be large enough. Secondly, we can get larger speedups for CFD and MD compared with blacksholes. This is due to the heavier computation in the loop body of these two applications, parallel execution of which on GPUs gets more benefits. Thirdly, we can generally get a larger speedup on Platform 2 than on Platform 1. For example, for the blacksholes loop with 512*14 iterations, we observe a speedup of 4.8 on Platform 1 while on Platform 2 this value is 13.4. The reasons are two-fold: (1) the GTX580 GPU shipped with Platform 2 has a higher clock rate than the M2050 GPU on Platform 1 (1.54GHz vs. 1.15GHz); (2) GTX580 also has a larger memory bandwidth than M2050 (192.4GB/s vs. 148GB/s).

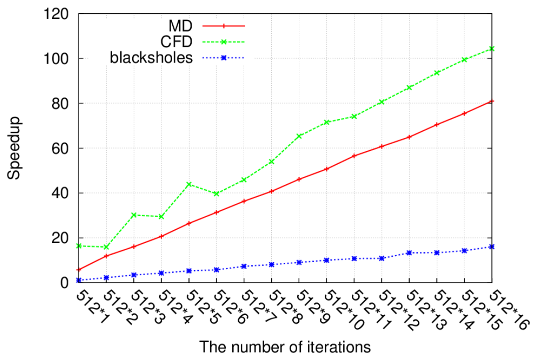


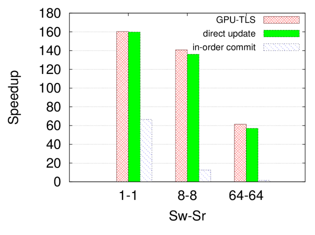
Figure 6. Speedups achieved for different problem sizes on Platform 2

From the observations, we conclude that GPU-TLS favors loops with large problem size and heavy computation in the loop body. A tool selecting proper loops from possibly-parallel ones through static analysis and dynamic profiling could be a great supplement to GPU-TLS.

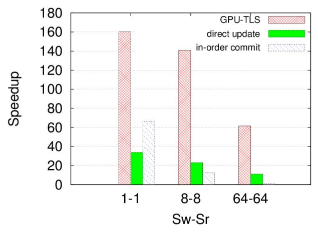
B. Effect of Dependency Patterns

In this section, we evaluate the effect of different dependency patterns to the speedups achievable by GPU-TLS using the synthetic loop on Platform 1. We test four kinds of loops: loops without runtime dependencies (i.e., DOALL loops), loops with runtime WAR dependencies only, loops with WAW dependencies only, loops with RAW dependencies only. The speedups achieved for different write and read set sizes are shown in Figure 7. For each kind of loops, besides the speedups achieved by GPU-TLS, we also test the speedups of two schemes in related work for comparison purpose. The "direct update" scheme is the porting of an existing TLS on CPUs with direct update memory versioning approach [5]. The "in-order commit" scheme is the porting of another TLS on CPUs with deferred update memory versioning and serial commit phase [3]. By configuring the values of the w and r arrays, we simulate loops with different dependency patterns. In the cases of WAW, WAR and RAW dependencies, we add 1% dependencies evenly among the iterations.

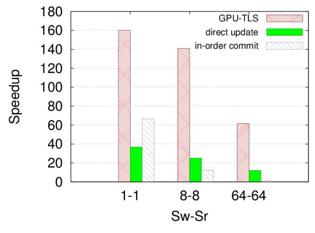
By comparing Figure 7(a), 7(b) and 7(c), we can see that GPU-TLS and "in-order commit" are not affected much by WAR and WAW dependencies while the "direct update" scheme performs much worse when there are WAR and WAW dependencies. This is as expected because the deferred update memory versioning approach used by GPU-TLS could avoid mis-speculations caused by false dependencies and is thus free of re-executions while the direct update scheme will result in lots of re-executions due to the mis-speculations in face of any form of dependencies (i.e, RAW, WAW and WAR). From Firue 7(d), we observe that when there are true dependencies (RAW), the speedups achieved by all the three schemes drop due to re-executions caused by mis-speculations; however, GPU-TLS performs the best in all cases.



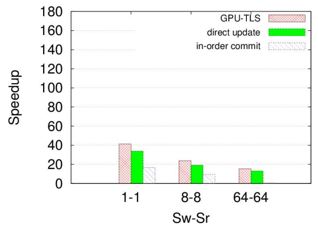
(a) Speedups for DOALL loops



(b) Speedups for WAW loops



(c) Speedups for WAR loops



(d) Speedups for RAW loops

Figure 7. Speedups achieved for loops with different dependency types

C. Evaluation of Parallel Commit

Finally, we are interested to see the effectiveness of the parallel commit scheme in GPU-TLS in helping to avoid the serial commit bottleneck. Figure 8 shows the execution time comparison of serial commit versus parallel commit on Platform 1 using the synthetic loop, which is configured to be DOALL. Seven groups of experiments with different write set and read set sizes are carried out. From the figure, we can see that using parallel commit the execution time grows relatively slowly with Sw-Sr. However, the serial commit scheme grows radically when Sw is larger than 16. When Sw-Sr is 1-1, the parallel commit scheme is 4.7 times faster than serial commit while this number grows to 8 when Sw-Sr is 64-64. These results show that parallel commit performs much better than the serial commit alternative, especially when the write set size is large.

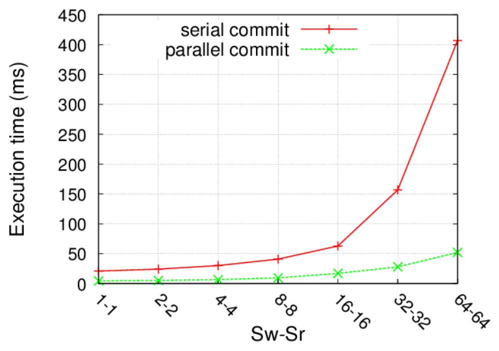


Figure 8. Execution time comparison: serial versus parallel commit

VI. Related Work

A. Speculative Loop Parallelization on CPUs

The BOP system [12] proposed by Ding *et al.* uses a process based runtime model to speculatively execute Potentially Parallel Regions (PPRs) on multi-core CPUs. The copy of memory pages to the master copy when speculation succeeds is heavyweight. CorD execution model [1] proposes allocating separate memory for different speculative threads. In their design, complicated synchronization is needed, making it not suitable for GPUs, which have limited synchronization support. Also, their design adopts a serial commit carried out by the centralized commit manager, which could easily become the performance bottleneck. The parallel commit scheme proposed in our design avoids the large overhead. STMLite [2] presents a TLS design based on an underlying STM. TLS and STM have some essential differences [13] and building TLS on STM may have to suffer some unnecessary overheads. LRPD Test [14] does the pioneering work of speculative loop parallelization on CPUs. Although they have privatization support, they cannot parallelize a loop when there are non-privatizable inter-iteration WAR or WAW dependencies. As we adopt deferred-update memory versioning scheme in our design, all inter-iteration WAR and WAW dependencies can be respected, qualifying loops with WAR and WAW dependencies as parallelizable. Oancea *et al.* [5] propose using direct-update memory versioning scheme to implement a TLS system on CPUs. They argue that the mis-speculation caused by inter-iteration WAR and WAW dependencies is acceptable. However, the same argument may not hold on GPUs, where we have thousands of speculative threads executing simultaneously. The deferred-update scheme we adopt in GPU-TLS avoids the mis-speculation caused by inter-iteration WAR and WAW dependencies and thus reduces the possibility of mis-speculation.

B. Loop Parallelization on GPUs

Lee *et al.* [15] propose a compiler framework for translating OpenMP programs into GPGPU programs. Leung *et al.* [16] present an extension to JIT compiler to implement automatic parallelization for GPUs. Calvert [17] designs a compiler that takes Java bytecode as input and generates GPU kernels from loops annotated with a special *@Parallel* flag. JCudaMP [18] devises a Java compiler framework to translate OpenMP-like annotated loops to JNI calls to CUDA kernels. All of the above-mentioned work focuses on simple DOALL loops or DOACROSS loops with statically known inter-iteration dependencies only, leaving the large portion of loops with statically unanalyzable dependencies un-accelerated. Our work proposes accelerating loops with dynamic parallelism on GPUs, expanding the scope of workloads parallelizable on GPUs. Liu *et al.* [7] carry out the initial exploratory study of implementing value prediction and TLS on GPUs. In their design, they control GPU thread in an ad-hoc approach. A GPU thread enters polling state when a dependency violation is detected. This ad-hoc approach can easily introduce branch divergence among the threads in the same warp, resulting in stalls and wastes of hardware resources. Di *et al.* [19] propose accelerating DOACROSS loops on GPUs. However, their solutions are algorithm-level modifications to a specific application instead of being generic. Diamos *et al.* [20] solve the problem of executing multiple kernels speculatively on multiple GPUs, exploiting coarse-grained Kernel Level Parallelism (KLP), which is orthogonal to our work.

VII. Conclusion and Future Work

This paper proposes GPU-TLS, a runtime system that enables speculative parallelization of loops with statically unanalyzable dependencies on GPUs. It utilizes partial parallelism in a large loop by chopping it into sub-loops and executing the iterations in a sliding window approach. The intra-warp value forwarding scheme reduces mis-speculation rate by utilizing the lockstep execution model of threads in a warp. The parallel commit scheme avoids the serial bottleneck and is scalable to thousands of GPU threads. Loops with dynamic parallelism are parallelized using GPU-TLS on two recent NVIDIA GPU cards and speedups from 5 to 160 are observed.

GPU-TLS works as a runtime library, which exposes APIs to compilers or programmers. To facilitate loop parallelization using GPU-TLS, designing a speculative parallelizing compiler that can automatically extract speculative threads from sequential loops and instrument the original memory accesses with APIs provided by GPU-TLS is our future plan. Also, using GPUs in cloud environments to provide HPC services is a trend [21]; how to coordinate multiple GPU cards to speculatively parallelize large loops using GPU-TLS would be another direction to explore.

Acknowledgment

This research is supported by Hong Kong RGC grant HKU 718011E.

References

[1] C. Tian, M. Feng, V. Nagarajan, and R. Gupta, "Copy or discard execution model for speculative parallelization on multicores," in *MICRO-41. 2008 41st IEEE/ACM International Symposium on Microarchitecture, 2008.*, 2008, pp. 330–341.

[2] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke, "Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 166–176. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542495

[3] C. E. Oancea and A. Mycroft, "Software thread-level speculation: an optimistic library implementation," in *Proceedings of the 1st international workshop on Multicore software engineering*, ser. IWMSE '08. New York, NY, USA: ACM, 2008, pp. 23–32. [Online]. Available: http://doi.acm.org/10.1145/1370082.1370090

[4] P. Rundberg and P. Stenstrm, "An all-software thread-level data dependence speculation system for multiprocessors," *Journal of Instruction-Level Parallelism*, vol. 3, p. 2002, 2001.

[5] C. E. Oancea, A. Mycroft, and T. Harris, "A lightweight in-place implementation for software thread-level speculation," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 223–232. [Online]. Available: http://doi.acm.org/10.1145/1583991.1584050

[6] J. Steffan, C. Colohan, A. Zhai, and T. Mowry, "Improving value communication for thread-level speculation," in *Eighth International Symposium on High-Performance Computer Architecture, 2002.*, 2002.

[7] S. Liu, C. Eisenbeis, and J.-L. Gaudiot, "Speculative execution on gpu: An exploratory study," in *Proceedings of the 2010 39th International Conference on Parallel Processing*, ser. ICPP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 453–461. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2010.53

[8] M. Cintra and D. R. Llanos, "Design space exploration of a software speculative parallelization scheme," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, pp. 562–576, June 2005. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2005.69

[9] C.-Z. Xu and V. Chaudhary, "Time stamp algorithms for runtime parallelization of doacross loops with dynamic dependences," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, pp. 433–450, 2001.

[10] P. Dubey, "Recognition, Mining and Synthesis Moves Computers to the Era of Tera," *Technology@Intel Magazine*, Feb. 2005.

[11] Blackscholes. [Online]. Available: http://en.wikipedia.org/wiki/Black-Scholes

[12] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang, "Software behavior oriented parallelization," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 223–234. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250760

[13] J. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui, "Unifying thread-level speculation and transactional memory," in *Middleware*, 2012.

[14] L. Rauchwerger and D. Padua, "The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization," in *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, ser. PLDI '95. New York, NY, USA: ACM, 1995, pp. 218–232. [Online]. Available: http://doi.acm.org/10.1145/207110.207148

[15] S. Lee, S.-J. Min, and R. Eigenmann, "Openmp to gpgpu: a compiler framework for automatic translation and optimization," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504194

[16] A. Leung, O. Lhoták, and G. Lashari, "Automatic parallelization for graphics processing units," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '09. New York, NY, USA: ACM, 2009, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1596655.1596670

[17] P. Calvert, "Parallelisation of java for graphics processors," Master's thesis, Cambridge University, 2010.

[18] G. Dotzler, R. Veldema, and M. Klemm, "Jcudamp: Openmp/java on cuda," in *Proceedings of the 3rd International Workshop on Multicore Software Engineering*, ser. IWMSE '10. New York, NY, USA: ACM, 2010, pp. 10–17. [Online]. Available: http://doi.acm.org/10.1145/1808954.1808959

[19] P. Di, Q. Wan, X. Zhang, H. Wu, and J. Xue, "Toward harnessing doacross parallelism for multi-gpgpus," in *Proceedings of the 2010 39th International Conference on Parallel Processing*, ser. ICPP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 40–50. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2010.13

[20] G. Diamos and S. Yalamanchili, "Speculative execution on multi-gpu systems," *2010 IEEE International Symposium on Parallel Distributed Processing IPDPS*, vol. 4, pp. 1–12, 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5470427

[21] Announcing cluster gpu instances for amazon ec2. [Online]. Available: http://aws.amazon.com/cn/about-aws/whats-new/2010/11/15/announcing-cluster-gpu-instances-for-amazon-ec2/