Lightweight Dependency Checking for Parallelizing Loops with Non-deterministic Dependency on GPU

Hongyuan Liu*, King Tin Lam*, Huanxin Lin*, Cho-Li Wang*, Junchao Ma†
*Department of Computer Science, The University of Hong Kong
{hyliu, ktlam, hxlin, clwang}@cs.hku.hk
†Shannon Cognitive Computing Laboratory, Huawei Technologies
{kelvin.majunchao}@huawei.com

***Abstract*—General-purpose GPUs have been prevalent for a decade. Nevertheless, GPU programming remains an onerous job practically exclusive to veteran developers who must know both domain-specific knowledge and GPU architecture well. Although current parallelizing compilers that automatically parallelize and offload sizable loops onto the GPU have helped in unfettering the power of the GPU with minimal programming effort, there are still a family of loops that carry statically non-deterministic data dependencies and cannot be parallelized. To tackle this issue, we propose two lightweight dependency checking schemes that are very different from existing conservative compilers to assist parallelizing loops with non-deterministic data dependencies. Our schemes feature linear work complexity for memory operations, lower memory consumption compared to previous work, and minimal false positives by leveraging the lockstep execution on the GPU's SIMD lanes. Experiments done using microbenchmarking and real-life applications on the latest advanced AMD discrete GPUs show that our schemes can achieve 2.2× speedup over existing solutions in dependency-free cases while only taking about 20% of time compared to existing solutions in the case with statically unproven loop-carried dependencies.**

***Index Terms*—GPGPU; Dependency Checking; Loop Parallelization; Code Generation;**

I. Introduction

The many-core computing revolution has begun to impact not only supercomputers but also almost every sort of computing devices including our smartphones. Unleashing the full potential of many-core computing to sustain computational performance will require fundamental advances in both computer architecture and programming models. With the breakdown of Dennardian scaling, we can expect processors are going towards more asymmetric coupled cores in the dark silicon era. This has been evidenced by the skyrocketing use of heterogeneous computing architectures; of these, GPUs (graphics processing units) are taking over CPUs as regards the many-core role of a computer, thanks to their higher power efficiency and cost effectiveness [4], [16], [19].

Parallel programming on GPUs is however intrinsically so complex that it is not easy for even veteran programmers to master it. To unburden programmers from onerous programming work and to accelerate existing code with minimal porting efforts, auto-parallelization emerges as a promising approach, albeit far from a panacea. The *polyhedral model* [31] is a powerful abstraction for analyzing and transforming loops of affine iterations and data domain to exploit statically proven parallelism. Polyhedral compilation considers sequential loops as polyhedra, mapping iteration domains and data domains to multicore or many-core processor domains, and transforms them into multithreaded code (e.g. OpenCL, CUDA, OpenMP) [7], [12], [27]. While polyhedral compilers are widely used for loop parallelization, they rely on static analysis which has well-known limitations on the sorts of loops that can be analyzed and parallelized.

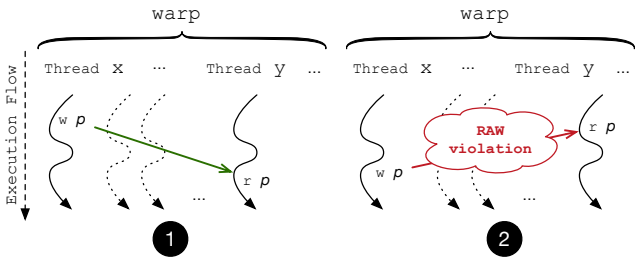


Fig. 1. Different cases of intra-warp data dependency: (1) $\mathtt{r}\ p$ in thread y is able to read the value written by $\mathtt{w}\ p$ in thread x ($x < y$), which was reported as a dependency violation by previous research, but is actually not (2) Intra-warp data dependency violation

Loops with non-affine loop bounds or subscripts are common in many applications, and it is difficult to detect data dependencies at compile time for code that uses indirect addressing, pointers, recursion, or indirect function calls. In reality, a large portion of such loops are actually parallelizable [34]. However, due to the hardship of determining data dependencies at compile time, existing techniques regard all loops with statically non-deterministic data dependencies as non-parallelizable and let them run sequentially. This conservative approach would be overkill for a great many potentially parallelizable loops. To tackle this issue, we attempt to offload loops with non-deterministic data dependencies to GPUs. With lightweight data dependency checking runtime support implemented as a safety net, the GPU can be used to accelerate those statically unproven loops as well to exploit potentially huge parallelism inherent in a lot of applications.

Designing such a dependency checking scheme is challenging since a large amount of states need to be maintained in memory. They may compete with application data for the relatively scarce memory resource on the GPU. Even though prior research work achieves checking inter-iteration

data dependency at runtime, the incurred overhead limits the scalability. Paragon [22] proposes a scheme for running loops speculatively on the GPU. It detects whether data dependency violation happens during the speculative execution phase. However, it is suboptimal in three aspects. First, Paragon may result in false positive cases like the one depicted in Fig. 1, which may limit the scope of loops to be offloaded. Second, its memory overhead is significant as it records all accesses to the read/write sets. Since GPU memory is far less than host memory, it may fail to run big loops. Third, it has limited scalability as it needs quadratic step complexity to determine whether a dependency occurs or not by comparing the read set and write set pairwisely. GPU-TLS [32] reduces false positive cases by exploiting the GPU's lockstep execution model. It uses a deferred update strategy, caching all read and write addresses in global memory. Also, written address-value pairs are cached in shared memory, which easily becomes the bottleneck of stream multiprocessor occupancy [2], potentially serializing workgroup execution. As both of the works use read/write sets, for programs that are hard to determine the number of write operations per iteration, they need costly profiling procedures to configure the sizes of read/write sets, limiting their practicality. Using a Bloom filter-like approach to tracking read/write sets may help reduce memory consumption, but the associated hashing algorithms can give more false positives of dependency. It would be inefficient when ported in software to commodity GPUs which consist of many compute cores and a relaxed memory model.

This paper makes the following contributions to address these issues:

- We propose two lightweight dependency checking schemes that work without using read and write sets, reducing $O(n^2)$ step complexity to linear, where n is the number of memory operations in a GPU thread. Our schemes are of high scalability even compared with the blindly parallelized version[1] of the loop.
- Our schemes feature more precise data dependency detection on the GPU. The first scheme trims a large portion of false positives, while the second is free from all false positives and false negatives in dependency checking.
- Both of our schemes support early termination of the entire kernel when a dependency is detected. This saves a significant amount of speculative execution time when the kernel execution takes long.
- Each scheme has its own advantages. Our first scheme, namely *two-pass inspector-executor*, runs faster in general; it requires fewer memory operations but more static analysis to generate code. Another scheme dubbed *Warp-Intrinsic Speculative Execution (WISE)* has a wider range of use cases. It executes the parallel code of the loop and detects data dependencies on the fly.

The rest of the paper is organized as follows. Section II gives a summarized background of this work. Section III and Section IV present our two-pass and WISE schemes respectively. A complexity analysis on the various schemes follows in Section V. In Section VI, we evaluate the performance of the schemes through experiments. Section VII reviews related work. Finally, Section VIII concludes this paper.

[1]Its generated kernel code is regarded unsafe and may give incorrect output.

II. Background

OpenCL (Open Computing Language) [3] is one of the most widely used programming frameworks for general-purpose GPU computing. Various compute device vendors (e.g. NVIDIA, AMD, Intel) provide OpenCL implementations. The GPU programming model requires the programmer to explicitly exploit data-level parallelism. From a programmer's perspective, a GPU consists of multiple *stream multiprocessors (SMs)*, or *compute units* in OpenCL terms. A special program called *kernel* is run on the GPU, which separates the solution space into blocks, namely *workgroups*[2]. The kernel is launched from the host side via a command queue, with a NDRange parameter which contains a n-dimensional array of workgroups specified by programmer.

There are two-level work schedulers on the GPU. Hardware workgroup scheduler (CTA scheduler) assigns those workgroups to stream multiprocessor in a many-to-one manner. The number of workgroups could be assigned to an SM simultaneously is limited by three factors of SM: the max number of threads, shared memory, registers. When a workgroup finishes execution, the workgroup scheduler will continue assigning another pending workgroup to the vacant SM. The second level of work scheduler is the *warp-scheduler*. Warp is the basic unit of execution flow of a stream multiprocessor, consisting of 64 threads (AMD) or 32 threads (NVIDIA), where instructions are executed in lockstep on the SIMD lanes of the GPU. The warp-scheduler employs switched execution of warps in order to hide memory access latency and to boost pipelining of instructions.

GPUs adopt a relaxed memory consistency model for scalability sakes, therefore it is necessary to take care of the visibility issue when programming on it. A typical GPU memory subsystem is organized in a hierarchy of the following: (1) per-thread registers; (2) per-SM L1 caches and shared memory (a.k.a. local memory in OpenCL); (3) a unified L2 cache shared by all SMs; and (4) the off-chip global memory.

III. Two-Pass Eager Dependency Checking

Our two-pass *eager* dependency checking scheme follows an inspector-executor model. The entire execution consists of two phases: the inspector run and executor run. During the inspector run, we use two passes to check for all possible dependencies in a loop. The first pass checks for *Write-After-Write (WAW)* dependencies, followed by the second pass checking for *Read-After-Write (RAW)* and *Write-After-Read (WAR)* dependencies. We design a low-overhead intra-warp timestamping mechanism (Section III-A), based on which we can avoid reporting false positives of detected dependencies.

[2]Known as *thread blocks* and *cooperative thread arrays (CTAs)* in CUDA and PTX terminology respectively

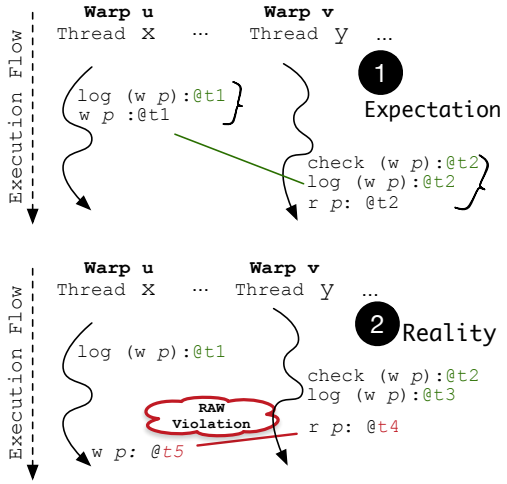


Fig. 2. Inaccuracy of recording global timestamp: (1) It is expected that every logging or memory operation is done atomically. One is led to think of using the timestamp to check for dependency violation based on the occurrence order. (2) But in reality, if threads x and y are in different warps, transactional logging and memory operations do not exist since any warp could be switched off anytime. Detecting dependency violation in this way leads to both false positives and false negatives.

Details of our dependency checking algorithm using the timestamp will follow in Section III-B.

A. Intra-Warp Timestamp Recording

To record a timestamp on the GPU for ordering memory operations is challenging due to the relaxed memory consistency model adopted by the GPU's memory/cache hierarchy. Although the `clock64` register that carries the current cycle number could be considered a timestamp, it does not imply the order of memory operations being issued would follow the order of visiting this register. Consider the case as shown in Fig. 2. Since the hardware schedulers are transparent to the programmer, the actual order of reads and writes to location x made by different threads cannot be recorded easily. Applying fine-grained locks may solve the problem, however at a cost of inducing significant runtime overheads and risks of deadlock. Software transactional memory is not a good solution either; it is as hard as performing dependency checking. Therefore, as a tradeoff between cost and efficiency, we design an *intra-warp timestamp* to facilitate memory operation ordering.

We have two design goals for our intra-warp timestamp. First, we expect the timestamp is maintained incrementally for each memory operation within a warp. Second, the timestamp should be consistent, regardless of how many times we run the GPU kernel with the same read/write pattern. Design such a timestamp is challenging for a number of reasons. First of all, GPGPU programming, conforming to the SPMD (Single Program Multiple Data) model, actually allows divergent control flows or branch-divergent threads. As shown in Fig. 3, threads x and y follow different control flows, and run on different lanes of same SIMD engine. The GPU kernel compiler inserts reconvergence instructions after each time a divergent branch finishes execution in some alternate manner. At any point that

Listing 1
INTRA-WARP TIMESTAMPING

```
__local volatile int timestamp[WORKGROUP_SIZE /
    WARPSIZE];
warpId = flattened NDRange index / WARPSIZE;
++timestamp[warpId];
// increment timestamp per memory operation
```

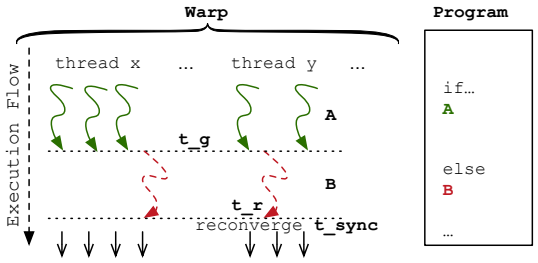


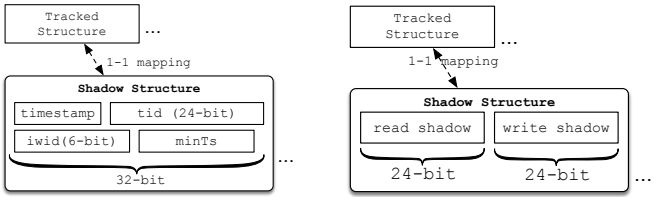
Fig. 3. Timestamp synchronization under divergent control flow: the order of green block (A) and red block (B) is unpredictable. Thus, timestamp at the reconvergence point $t_{\text{sync}} = max(t_{\text{r}}, t_{\text{g}})$, where t_{r} is the latest timestamp of the red block (B) and t_{g} is the latest timestamp of the green block (A).

threads reconverge, the timestamp must equal the maximum one among the threads in a warp in order to meet our first design goal. Since the execution order of *if* branch and *else* branch depends on the branch prediction instructions generated by the compiler, it may vary across compilers with various branching features.

One may be surprised at our solution that lies in adding a deliberate *data race* on shared memory. Our timestamp recording mechanism leverages the lockstep execution within a warp. We declare a 32-bit integer variable for each warp and allocate it in on-chip shared memory for low-latency access. It is visible to all threads in the warp by declaring it with the `volatile` modifier, which instructs the compiler not to cache it in registers since it may be modified by more than one threads [9]. Listing 1 shows our ideas. Although the `timestamp` variable is incremented by all threads in the warp, it is exactly incremented by one only. It is obvious that this timestamp design complies with our two goals. After the control flows reconverge, the `timestamp` variable equals the maximum one of all threads in the warp. Note that our timestamping technique would not cause shared memory bank conflicts [1], [2] because threads in a warp access the same 32-bit word at the same time.

B. Dependency Checking

According to OpenCL memory consistency model, memory has load / store consistency within a thread. Therefore, data dependencies happened within an iteration need not be checked when mapped to a GPU thread. Our method uses the kernel of the parallelized loop as input, which can be generated manually or automatically by parallelizing compilers. We refer to the arrays that may contain data dependencies at runtime as *shared arrays*. Non-affine expressions with subscripts accessing shared arrays, leading to statically non-



(a) 2-pass (b) WISE

Fig. 4. Data structures used in our schemes

deterministic memory accesses, are called *index expressions*. In the two-pass scheme, two OpenCL kernels, known as *inspector kernels*, are generated to inspect each shared array a for different types of data dependencies. The first one checks for WAW dependencies, and the second for RAW and WAR dependencies.

Data Structures. We use a shadow array S whose size equals that of the largest shared array. Each address of a shared array a to be checked is mapped to one element in the shadow array S, i.e. 64-bit for tracking a memory address. Each element of S array is a 64-bit integer struct consisting of two parts. In the first part (upper 32 bits), the rightmost 24 bits are used to record thread id, since the maximum flattened global work size is typically 2^{24}, and the leftmost 8 bits record the timestamp when the corresponding element is accessed. The second part (the next 32 bits) is used to record the earliest time a thread accessed the tracked place, and is composed of a 6-bit intra-warp id (`iwid`) and a timestamp value (`minTs`). (See Fig. 4a). The shadow array might be reused as we may check shared arrays step by step. Both inspector kernels contain a global variable called `raceflag`, which is initialized to false, and denotes whether a dependency is detected.

Inspector Kernel Generation. The overall inspector kernel generation rule is shown in Fig. 5. We show exp_1 as a write access to array a, and exp_3 as a read access. Specifically, if either of exp_1 and exp_3 is non-affine expression regarding to iteration domain variables, then array a is a shared array and all expressions mentioned are index expressions. Inspector kernels are generated using a simplified variant of data flow analysis. We collect all variables involved in the index expressions into a set V. From the parallelized loop kernel, we traverse the control flow diagram generated by the abstract syntax tree (AST) to maintain V in a backward manner. When a variable in V is written, we add all variables on the right side of the assignment statement into V. Finally, we eliminate those statements without variables in V in a forward manner. In essence, this effectively prunes away most of the compute-intensive code (similar to dead code elimination used in compilers), producing a streamlined kernel version that enables a lightweight dependency checking phase considering memory operations alone.

WAW Checks. At the beginning, we initialize all elements of the shadow array `S` to represent that the corresponding addresses have never been accessed. When a write access is at place p of a shared array a, we combine `timestamp`

Parallelized Loop Kernel

```
...
a[exp1] = exp2
...= ...op a[exp3] op ...
...
```

WAW checking for array a

```
fillBuffer(S, 0)
raceFlag = false
// maintain timestamp in each memory access
// WS : WARPSIZE
kernel {
    tid = get NDRange index
    t = combine(timestamp, tid)
    v = atomic_xchg(&S[exp1], t)
    if (first time access)
      record iwid and minTs
    (prevtimestamp, prevtid) = decombine(v)
    if (inSameWarp(prevtid, tid)) {
      if (tid mod WS < prevtid mod WS or
        prevtimestamp = timestamp)
        *raceFlag = true
    } else *raceFlag = true
}
```

RAW/WAR checking for array a

```
if (raceFlag) return;
// maintain timestamp in each memory access
kernel {
    tid = get NDRange index
    localId = tid mod WS
    (wtime, writeTid) = S[exp3]
    wLocalId = writeTid mod WS
    if (not inSameWarp(writeTid, tid))
      *raceFlag = true
    else if (localId < iwid and timestamp > minTs)
      *raceFlag = true
    else if (wLocalId<localId and wtime>timestamp)
      *raceFlag = true
    else if (iwid<localId<wLocalId and
        mTs<timestamp<wTime)
      *raceFlag = true
}
```

Fig. 5. Dependency checking code generation

and the current flattened NDRange thread ID (`tid`) into one register variable by bit operation. Then an `atomic_xchg` is performed on `S`[p], which sets the current value and returns the previous value in that place atomically. We extract the previous value to get the corresponding `tid`. The result of a write operation check could be one of the following three cases. (1) The previous value equals the initial value; this implies the current write operation is the first-time access to place p. Current NDRange index and `timestamp` are assigned to `S`[p]. To find all RAW/WAR violations in the second pass, we record intra-warp id and timestamp to last 32-bit `S`[p]. Note that once written, the last 32-bit of shadow array element would not be modified. (2) The previous warp id is the same as the current warp id; we therefore compare id within a warp (local warp id). If the former one is less than or equal to the current local warp id, the dependency would not cause a

violation. Otherwise, the race flag is set to indicate an intra-warp WAW dependency violation. (3) The previous warp id is not equal to the current warp id. This implies an inter-warp WAW dependency. As a result, the race flag is set to be `true`. Setting the race flag to `true` need not be atomic since it can only switch from `false` to `true`.

RAW/WAR Checks. Provided that no WAW dependency is detected in a, the second inspector kernel responsible for RAW/WAR checks is carried out. This time, the shadow array `S` contains values from WAW checking pass. Each element of `S` stores a quaternion, namely (`timestamp`, `tid`, `iwid`, `minTs`), which could be fetched through only one 64-bit read operation. The `timestamp` marks the last time that the tracked place was written by thread `tid`. `iwid` and `minTs` are intra-warp id and timestamp at the first time the tracked place was written. All memory operations are time-stamped during the second inspector kernel run. But our scheme need not record read locations. Instead, we check S to confirm whether the current read location has been written by other warps during the time we read p. If the memory location is written by other warps, RAW or WAR dependency exists since the order of warp is agnostic. If not, we further check intra-warp RAW or WAR cases. Three cases would cause an intra-warp dependency violation: (1) if current intra-warp thread id (`localId`) is less than that of the thread id first written in p (`iwid`), and current timestamp is greater than the first written timestamp `minTs`, there is an intra-warp WAR dependency violation. (2) If current thread id is greater that of the thread written in p and current timestamp is less than the written time, an intra-warp RAW dependency violation is reported. (3) If the thread id and the timestamp of read operation is between the first time and the last time written to a certain place, it is regarded as an intra-warp dependency violation. Although the summarized logs recorded in the WAW phase might still result in false positive cases during RAW/WAR checks, they belong to very extreme cases.

IV. Warp-Intrinsic Speculative Execution (WISE)

While the two-pass eager dependency checking scheme fits many application scenarios, it has a drawback that shadows its usefulness for certain kinds of compute-intensive applications. If the index expression comes from "heavy computation" that is impossible to eliminate by dataflow analysis, then the scheme could degrade the execution by multiple times for repeating the same compute-intensive part. Listing 2 shows an example where there is one shared array *array* and one index array *ind*. The result of the index array is computed by calling some function that involves heavy computations, which cannot be eliminated through static analysis. To cope with this problem, we design another scheme, dubbed *Warp-Intrinsic Speculative Execution (WISE)*, which detects data dependencies in a speculative and on-the-fly manner. The idea may look similar to Thread-Level Speculation (TLS), which is widely used in multicore CPU design and usually supported at hardware level (e.g. as a variant of the cache coherence protocol [25]) or software level (e.g. working as

Listing 2
Index array subject to heavy computation

```
for (int i = 0; i < n; i++)
{
   ind[i] = heavyComputation();
   array[ind[i]] = i;
}
```

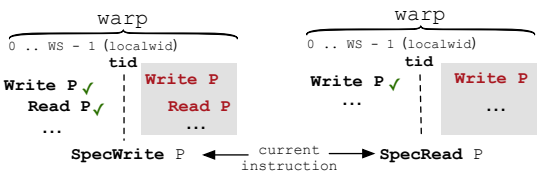


Fig. 6. Illustrating speculative read and write semantics (Algorithm 1 and 2) graphically: memory operations under shaded area are regarded intra-warp dependency violations.

a C++ template [20]). On a multicore CPU, these pieces of work use sophisticated synchronization techniques to realize misspeculation recovery. However, the impact of misspeculations could be considerable more serious on the GPU, and the recovery can incur more significant overheads due to the relaxed memory model and the obstacle of performing inter-SM orchestration. It comes to our knowledge that recent research on speculative loop execution on GPUs [22], [29], [32] do not attempt to recover from misspeculation. Instead, when a misspeculation (i.e. dependency violation) occurs, the result coming from the loop-parallelized kernel is discarded. The loop that speculated wrongly turns back to the CPU eventually. Despite the speedups achieved over the sequential loop execution on the GPU, their solutions suffer from quadratic time and space complexity [22], [32], false positives of data dependency [22], and even correctness issues [29]. In contrast, our WISE scheme, employing implicit timestamps guaranteed by lockstep execution in the warp, features precise dependency checking that is free of false positives (we call WISE an *fp-free* scheme) and on-the-fly detection that allows early kernel termination once a dependency gets detected. As a result, WISE is very lightweight in view of its linear work complexity as the number of memory operations to be tracked scales.

The data structure we used in WISE is shown in Fig. 4b. Different from our two-pass approach, we need another shadow array (the read copy) in this case for recording addresses of read operations to shared arrays. In total, we need 64 bits (two 32-bit integers) for tracking each address used in the program. Algorithm 1 and Algorithm 2 show the speculative read and speculative write operations used in WISE respectively. The code generation of a WISE-augmented program is very simple as the dependency checking is enabled by simple substitutions of memory operations that contain statically non-deterministic data dependencies with speculative ones. Therefore, WISE it is more suitable for parallelizing loops with a complicated control flow.

Speculative Write. The procedure shown in Algorithm 1

Algorithm 1 Speculative write to address a

```
1: function SPECWRITE(a, value)
2:     tid = get NDRange Index
3:     cur_wid = tid / WARP_SIZE
4:     cur_localwid = tid % WARP_SIZE
5:     prev_tid = atom_xchg(writeShadow_a, tid)
6:     prev_wid = prev_tid / WARP_SIZE
7:     prev_localwid = prev_tid % WARP_SIZE
8:     if hasWritten(prev_tid) then
9:         if prev_wid ≠ cur_wid then
10:            return inter-warp WAW data dependency
11:        else
12:            if cur_localwid < prev_localwid then
13:                return intra-warp WAW dependency
14:    r_tid = atom_or(readShadow_a, 0)
15:    calculate r_wid, r_localwid from r_tid
16:    if hasRead(r_tid) then
17:        if r_wid ≠ cur_wid then
18:            return inter-warp RAW or WAR dependency
19:        else
20:            if cur_localwid < r_localwid then
21:                return intra-warp RAW dependency
22:    [a] = value
```

checks both the write shadow and read shadow. It first checks for inter-warp WAW dependencies (line 9), followed by intra-warp WAW dependencies (line 12). If the address being written has not been touched by any thread in other warp, it goes on further checking for intra-warp RAW / WAR dependencies (line 16 to 21). Line 14 reads the read shadow, using `atomic_or` to force bypassing the L1 cache, from which we know the read information of the address. Line 17 to 18 checks for inter-warp RAW or WAR dependencies. If the currently checked address is read by other warp, it reports an inter-warp dependency violation; otherwise it checks for intra-warp RAW or WAR dependencies (see Fig. 6). The unshaded area of Fig. 6 represents memory operations that caused benign data dependencies (false positive cases in Paragon) while the shaded area denotes real dependency violations. Finally, it updates the value of the given address.

Speculative Read. Similar to speculative write, the speculative read as shown in Algorithm 2 works for the situation depicted in Fig. 6. It checks the write shadow of the corresponding address first (line 5). If it is not written before, we return the value of the address. Otherwise, it checks for inter-warp RAW / WAR dependencies (line 8 to 9), and then checks for intra-warp WAR dependencies (line 10 to 11). If no dependency violation is detected, it updates the read shadow to be the maximum of its old value and current thread id; this guarantees a monotonically increasing read shadow. Finally, it returns the value of the required address.

Race Flag Propagation. We design a *race flag propagation* strategy to support early termination of the whole kernel once a dependency is detected. Although on NVIDIA GPUs, it is possible to use `trap` instruction [9] to terminate the whole grid of the executing kernel. However, this method could be broken in three ways: (1) it cannot work on GPUs from other vendors (worsened cross-compatibility); (2) it would mask

Algorithm 2 Speculative read to address a

```
1: function SPECREAD(a)
2:     tid = get NDRange Index
3:     cur_wid = tid / WARP_SIZE
4:     cur_localwid = tid % WARP_SIZE
5:     w_tid = atom_or(writeShadow_a, 0)
6:     if hasWritten(w_tid) then
7:         calculate w_wid, w_localwid from w_tid
8:         if w_wid ≠ cur_wid then
9:             return inter-warp RAW or WAR dependency
10:        if cur_localwid < w_localwid then
11:            return intra-warp WAR dependency
12:    readShadow_a = atom_max(readShadow_a, tid)
13:    return [a]
```

actual errors that occur within the GPU, raising obstacles to debugging; (3) its behavior depends on the device driver. Therefore, we design a higher-level mechanism to support universal and graceful termination. We keep a local "race" flag in each workgroup, declared using the `volatile` modifier, and we check its value before a speculative operation is performed. At the time that a dependency is detected from a thread, the thread sets both the global and local flags of that workgroup, and then returns. Therefore, the threads in the same workgroup could see the local race flag before the next speculative operation. Moreover, it checks the value of global flag at the beginning of thread execution. If the global race flag is true, the thread itself and all the following workgroups would return. When the former portion of workgroups have detected dependencies while the later assigned workgroups do not start, this technique can make those later assigned workgroups terminate at the time they are assigned to SMs. The race flag propagation technique is especially useful for WISE, and is applied to the two-pass approach as well.

V. Analysis and Comparison

In this section, we present a theoretical analysis, mainly on time and space complexity, to compare our schemes with previous work including Paragon [22] and GPU-TLS [32]. Paragon is a CPU-GPU cooperative framework that speculatively executes programs on the GPU. If a dependency gets detected, Paragon will discard the parallelized result and re-execute the task on the CPU. Its dependency checking algorithm consists of three phases. The speculation phase is run first, producing the execution results as well as collecting the read and write sets. The second phase compares the read set and write set recorded, entry by entry, through a speculative execution phase. The third phase determines the dependency using information collected from the first two phases. GPU-TLS is our previous work which also speculatively executes loops. It makes use of the on-chip shared memory as a scratchpad for logging and caching deferred update entries. Global memory is used to record read and write sets. With the *intra-warp value forwarding* technique, it would not report false positives of dependency violation. After the speculative execution, the dependency checking phase of GPU-TLS compares read and write sets in a manner similar

TABLE I
MEMORY COMSUMPTION COMPARISON

GPU-TLS	Paragon	2-pass	WISE
$O(t\max_i\{w_i\}+\sum_i s_i)$	$O(t\max_i\{w_i\}+\sum_i s_i)$	$O(\max_i\{s_i\})$	$O(\sum_i s_i)$

to what Paragon does. Finally, GPU-TLS commits the result sequentially as for cached address-value pairs to host memory. It employs some spin-locks tailored to GPUs [23], [26] to keep the commit order. However, the lock has an issue that when many workgroups are running on the GPU concurrently, its efficiency drops significantly [26].

Memory Complexity. We denote the write time on array a in an iteration as w_a, the size of shared array a as s_a, and the number of threads as t. The comparison of memory consumption is shown in Table I. The space complexity of our work does not depend on the write time per iteration. Hence, it is expected to scale better in programs with large memory footprints. Our counterparts, Paragon and GPU-TLS, share a common problem: when the count of memory operations per thread varies significantly (i.e. unbalanced memory workload per thread), they waste lots of extra memory to allocate read/write (r/w) sets according to the maximum number of r/w per thread. Worse still, both schemes may need a profiling phase to determine the r/w set sizes when the number of r/w's in some threads is not determined statically.

Time Complexity. The second phase of Paragon constitutes a dominant part of the step complexity, requiring $O(xy)$ to identify intra-iteration dependency, where x and y are the counts of read and write operations respectively. The third phase performs an atomic summation or a summing reduction. Regarding GPU-TLS, its time complexity is not superior either. The dependency checking phase of GPU-TLS is of $O(t)$ step complexity and of $O(t\max_i\{w_i\})$ work complexity, where t is the number of parallel threads. In contrast, our schemes are of linear work complexity with respect to memory operations, i.e. resulting in an additional constant of computations as memory operations scale. Hence, our schemes are expected to scale better.

Concerns on Atomic Operations. While we extensively use atomic operations in our schemes, our *race flag propagation* technique ensures that the contentious performance issue with using atomic operations would not happen. Atomic operation is actually improving with the development of hardware [21]. Existing research [10] even shows that several parallel algorithms such as prefix scan and reduction could be accelerated when using atomic operations appropriately in recent GPUs.

VI. EXPERIMENTAL EVALUATION

A. Testing Environment Setup

The experimental platform consist of an AMD A10 7850K CPU and an AMD R9 290X GPU with 4GB GDDR5 global memory. The discrete GPU has 2816 stream processors clocked at 1040MHz in 44 compute units (SMs). The PC is installed with AMD APP SDK 3.0 [24]. The CPU and GPU communicate over the PCI express bus 3.0 (x16) with 8GB/sec bandwidth. We use Microsoft C/C++ Optimizing Compiler (version 18.00.40629) for x86 to compile our host code. For stability reasons, the programs are compiled with `-O0` and `-cl-opt-disable` options. We collect host-side timings which include data transfer time for fairness to judge speedup over the CPU.

Listing 3
OUR SYNTHETIC LOOP USED IN MICROBENCHMARKING

```
for (int i = 0; i < N; i++) {
   double temp = 0.0;
   for (int j = 0; j < M; j++) {
      temp = a[r[i]];
      a[w[i]] = dumb_computation(temp);
   }
}
```

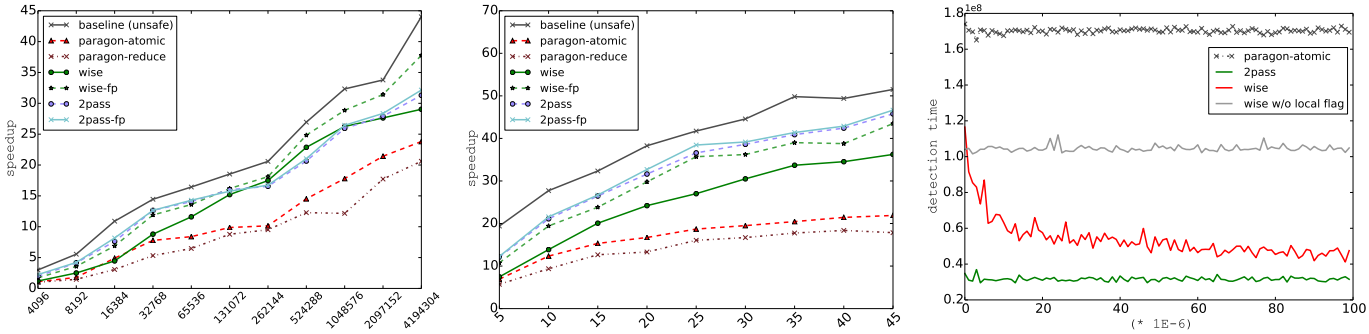
B. Evaluation Methodology

First, we design a microbenchmark that can manually adjust the dependency type and size, as shown in Listing 3. The outer loop is parallelized as a GPU kernel. We perform our experiments to verify three aspects: (1) scalability vs. the loop size (by varing N); (2) scalability vs. the number of memory accesses per iteration (by varing M); (3) agility of detection against the dependency density [28]. Adjusting the values in array `r` and `w`, dependencies can be added to the loop. The microbenchmarking experiments are described in Section VI-C. Second, we evaluate our schemes using the loops extracted from various real-life scientific computing applications. We compare the speedup of our schemes with our closest work, Paragon [22]. This part of experimental evaluation is described in Section VI-D. We do not compare with GPU-TLS [32] in this part as its progressive approach that resolves Write-After-Write dependencies by commits with sequential semantics kept would not gain advantages in this evaluation. Actually, GPU-TLS leaves RAW / WAR dependencies unresolved, so head-to-head comparison with it is not insightful.

C. Microbenchmarking

In the first two experiments, we mainly focus on the performance when no dependency case not actually exists. Two variants of Paragon (*paragon-atomic* and *paragon-reduce*) are used to compare with our schemes *2pass*, *2pass-fp*, *wise*, *wise-fp*. The suffix *-fp* means the schemes may report all false positive cases as Paragon does. The kernel corresponding to a blindly parallelized loop serves as the baseline for comparison; we refer to it as *unsafe* since it merely executes the loop in parallel on the GPU, ignoring all data dependency issues. We record the sequential (T_s) and parallel execution time (T_p) and calculate the speedups as T_s/T_p. In the third experiment, we test the detection time in dependency-existent cases. *wise w/o local flag* is the variant that we turned off race flag propagation. We do not include *-fp* cases here, and we do not test *paragon-reduce* any more as it performs always slower than *paragon-atomic*.

1) Scalability vs. Loop Size: We set $M = 5$ as it is common to see a kernel with that number of read/write operations per thread. To know the scalability of each scheme, we vary the



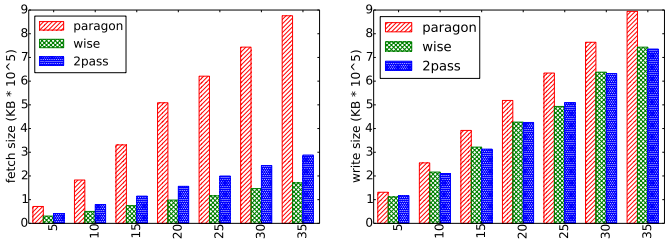
(a) Performance against loop size (b) Performance against #memory accesses (c) Performance against dependency density

Fig. 7. Performance of the microbenchmark

loop size and take the speedup over single-threaded CPU execution. Array r and w in the microbenchmark are configured to ensure the loop is dependency-free. Therefore, all of the three mechanisms need to finish the whole execution on the GPU. The results are shown in Fig. 7a. In this experiment, both variants of Paragon achieve less speedup than all variants of our schemes. *wise* is slower than *wise-fp* for about 32% on average since detecting intra-warp false positives needs another atomic operation and more complicated control flow. On the contrary, *2pass* and *2pass-fp* achieve almost the same speedup. This implies the overhead incurred by false positive detection in *2pass* is trivial (less than 5%). For loop sizes larger than 262,144, the performance of *2pass* and *wise* is similar because they share the same work complexity, but for loop sizes smaller than 131,072, *2pass* scales the best.

2) Scalability vs. Memory Accesses: We test the scalability against the number of memory accesses M per GPU thread. The loop size is fixed as 32,768, which is a size to obtain good speedup for data-parallel applications on the GPU. Fig. 7b shows that along the increment of memory accesses per iteration, the speedups given by the two Paragon schemes remain stalled (even noted a slowdown in Paragon-reduce when $M = 45$). This is because of Paragon's $O(n^2)$ time complexity for completing its read/write set comparison and calculation phase. Further, as shown in Fig. 8, Paragon's memory fetches are about five times more than *wise*, and 3.4 times more than *2pass*. The write size of Paragon is also much larger than that of both our schemes. Since memory operation needs hundreds of cycles on the GPU, "frugal" or judicious use of memory operations is always the key to performance. In contrast, our schemes achieved speedup curves similar to the baseline (*unsafe*) case, thanks to our sustained scalability of memory access. Among our schemes, *wise* achieves the least speedup for it uses the largest number of memory operations with regard to M.

3) Agility of Dependency Detection: We have tested our microbenchmark with a dependency-free configuration. This time we vary the dependency density (dd) and compare the detection time of different schemes. The loop size is fixed as a relatively large number 1,048,576 for more obvious detection



(a) Fetch size (b) Write size

Fig. 8. Fetch and write sizes vs. #memory accesses in the microbenchmark

time comparison. As shown in Fig. 7c, unlike all of our schemes, the execution time of Paragon remains constantly high since it has no ability to terminate early. The experiment also shows that when scaling from $dd = 10^{-6}$ to $dd = 10^{-4}$, the detection time of *wise* keeps dropping by 56% compared with *wise w/o local flag*. This proves the usefulness of the race flag propagation technique. After that, it is stabilized around a level with small fluctuations, approximately using only 28% of Paragon's detection time when dependency density is greater than 0.004%. Finally, *2pass* obtains the shortest execution time to determine if a loop carries dependency. It takes only 17.6% of Paragon's time, and 50% of *wise*'s time when $dd > 2 \times 10^{-5}$.

D. Application Benchmarking

In this part of experiments, we study the performance of our schemes and Paragon using seven real-life scientific applications. The loops of the first six contain no dependency at runtime, but this cannot be determined statically. The last one cannot be parallelized actually, thus early dependency detection and falling back to the CPU will minimize the loss of speedup. We show the results in Fig. 9. For most of the applications, *2pass* achieves the greatest speedup, *wise* ranks the second, and both win over Paragon in all cases.

1) Computational Fluid Dynamics (CFD): CFD [30] contains a loop over a number of edges; each edge is associated with two nodes. To calculate the force between each pair of nodes, the loop reads an edge and acquires the associated

nodes, but the data dependencies involved cannot be statically determined. Our test result shows that *2pass* achieves the best speedup (29% over Paragon and 20% over *wise*).

2) Molecular Dynamics (MD): In MD [30], a two-level loop is used to compute the forces between every pair of molecules. Molecules are paired up via an array called `partners`. Accesses to the input and output arrays `X` and `Y` are conditionally dependent upon the values in the `partners` array. Therefore, parallel reads and writes to the arrays may contain statically non-deterministic dependencies, but only WAW dependencies could occur in the loop. In this experiment, *2pass* obtains 55% and 10% higher speedups than Paragon and *wise* do respectively.

3) Inverse Permutation Vector (IPVEC): *Ipvec* [22] contains a loop that rearranges all elements of the input vector based on another index vector and puts the results in the output vector. This access pattern is common in parallel computing, and works as an integral part of many algorithms. We use a vector of 65,536 elements as input. Since the loop entails lots of memory operations but no computation, the speedup obtained on the GPU is not significant. With high throughput of PCIe 3.0 bus and latest GDDR5 memory, *Ipvec* still obtains a speedup of around 2×. This time, *2pass* performs only 1% faster than *wise* and 16% faster than Paragon.

4) Forward Elimination with Level Scheduling (FWD): FWD [22] is a two-level loop using forward elimination to solve linear equations. It operates on sparse matrices stored in CSR (compressed sparse row) format and update the matrices in-place. Thus, its parallelism is statically undecidable for existing parallelizing compilers. Testing shows that our schemes achieve at least 26% higher speedup than Paragon does.

5) Sparse Matrix-Vector Multiplication (SpMv): The sparse matrix in SpMv [5] is again stored in CSR format. We adapt the naive `spmv_csr_scalar_kernel` to a nested loop whose outer loop is parallelizable. We modify the loop slightly to write the result back to the original row of the matrix, so statically non-deterministic reads and writes are done on the matrix. Experiment with a $3{,}000^2$ sparse array of 10% non-zero randomly distributed elements shows that *wise* and *2pass* perform 75% and 110% faster than Paragon respectively.

6) Breadth-first search (BFS): A parallelizable BFS loop, adapted from *bfs* in Rodinia benchmark [8], is used for this experiment. Each GPU thread id corresponds to an iteration of the BFS loop. However, existing auto-parallelization tools fail to parallelize this loop due to its non-affine memory accesses for extending the frontier and marking the visited flag. Testing with the graph-65536 dataset provided by Rodinia shows that our schemes outperform Paragon by 30%. For a degree-unbalanced graph, i.e. a graph subject to power-law distribution (a common case for social networks), Paragon will need even more space to record memory addresses.

7) 0-1 knapsack (KNAP): 0-1 knapsack is a two-level loop containing statically non-deterministic dependencies across the inner and outer loops. It implements a dynamic programming algorithm that has heavy dependencies on the outer loop. We simulate a case that the outer loop is parallelized mistakenly,

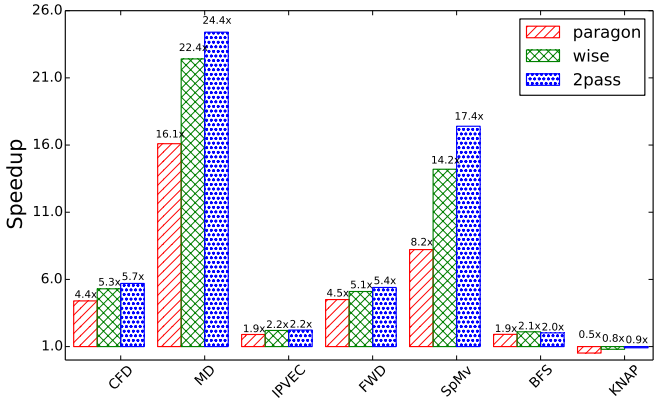


Fig. 9. Performance of real-life programs

in which we show that both of our schemes hand it over to CPU earlier than Paragon does, resulting in less slowdown than that of Paragon. The input has 5,000 items and the weight of the knapsack is 600. Eventually we use the CPU to finish the program, yet our mechanisms incur less overhead since our dependency checking can be terminated early.

VII. Related Work

We summarize two areas of related work as follows.

Data Dependency Detection in Loop Parallelization. Methods of data dependency detection in loop parallelization can be classified into two categories. The first category [14], [15], [17] is offline profiling for data dependencies to assist parallelization. Once classified as parallelizable, the loop is always parallelizable, regardless the program state or input. The time and space overheads for classifying loops and reporting places, where dependencies occur, are not an issue. However, for cases in which input or other program states have impacts on loop parallelism, online profiling [11], [22], [29], [32] is required. It runs the underlying loop to detect dependency violations at runtime, using various techniques that minimize the profiling overheads. Our schemes fall into the second category.

Data Race Detection. Data race detection on GPUs to assist debugging and verification is gaining research attention in recent years. GPUVerify [6] is a tool for statically checking intra-workgroup data races in CUDA or OpenCL kernels. GRace [33] finds static analysis not sufficient and goes for detecting data races at runtime (while retaining traditional compile-time detection techniques). However, it could only perform detection on the GPU's shared memory because it needs to scan and compare addresses stored in a warp table, leading to both quadratic time and space complexity, which are unacceptable when extended to global memory. HAccRG [13] proposes hardware techniques for the GPU to reduce the data race detection overhead. Their work could perform detection on shared memory and global memory, but the feature is not supported by current commodity GPUs. Although data race detection is quite similar to checking inter-iteration dependency violation in concepts, it could make false reports as not all data races are actual dependency violations. Transactional memory (TM) provides an alternative approach

that avoids data race in parallel execution [18]. However, hardware TM has limitations on the sizes of read/write sets. Porting software TM systems to many-core GPUs does not fit either due to the prohibitively high time and space overheads.

VIII. Conclusion

In this paper, we propose two lightweight dependency checking schemes to help parallelize loops with statically non-deterministic data dependencies. With intra-warp timestamp support, our first scheme (*2pass*) checks for WAW and RAW/WAR dependencies step by step. It is a lightweight approach that prunes away compute-intensive code through basic data flow analysis and inspects statically non-deterministic memory accesses only. The second scheme (*wise*), which checks dependencies on the fly and executes speculatively, is more apt for programs with rather unpredictable memory access patterns. Further, *wise* is able to terminate kernel execution sooner compared to previous schemes based on speculative execution. Both of our schemes reduce false positives by leveraging lockstep execution of the GPU's SIMD engine. Theoretical and experimental analyses show that our schemes scale well, achieving higher speedups while shrinking memory complexity and work complexity compared with previous work.

Acknowledgement

This research is supported by a Huawei research grant (No. 200007692).

References

[1] "OpenCL Optimization Guide – AMD," 2015.

[2] "Best practices guide," http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/, 2016.

[3] "OpenCL - The open standard for parallel programming of heterogeneous systems," https://www.khronos.org/opencl/, 2016.

[4] "TOP500 Supercomputer Sites," http://top500.org, 2016.

[5] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, 2008.

[6] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: A Verifier for GPU Kernels," in *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012, pp. 113–132.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "PLuTo: A practical and fully automatic polyhedral program optimization system," in *Proc. of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[9] N. Compute, "PTX: Parallel thread execution ISA version 2.3," *Dostopno na: http://developer.download.nvidia.com/compute/cuda/3*, vol. 1, 2010.

[10] I. J. Egielski, J. Huang, and E. Z. Zhang, "Massive atomics for massive parallelism on GPUs," in *Proc. of the 2014 International Symposium on Memory Management (ISMM)*, 2014, pp. 93–103.

[11] M. Feng, R. Gupta, and L. N. Bhuyan, "Speculative parallelization on GPGPUs," in *Proc. of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012, pp. 293–294.

[12] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in LLVM," in *Proc. of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2011.

[13] A. Holey, V. Mekkat, and A. Zhai, "HAccRG: Hardware-accelerated data race detection in GPUs," in *Proc. of the 42nd International Conference on Parallel Processing (ICPP)*, Oct 2013, pp. 60–69.

[14] A. Ketterlin and P. Clauss, "Profiling data-dependence to assist parallelization: Framework, scope, and optimization," in *Proc. of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 437–448.

[15] M. Kim, H. Kim, and C.-K. Luk, "SD3: A scalable approach to dynamic data-dependence profiling," in *Proc. of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010, pp. 535–546.

[16] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, no. 11, pp. 32–38, 2005.

[17] Z. Li, A. Jannesari, and F. Wolf, "An efficient data-dependence profiler for sequential and parallel programs," in *Proc. of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2015, pp. 484–493.

[18] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke, "Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory," in *Proc. of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 166–176.

[19] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE micro*, no. 2, pp. 56–69, 2010.

[20] C. E. Oancea and A. Mycroft, "Software thread-level speculation: An optimistic library implementation," in *Proc. of the 1st International Workshop on Multicore Software Engineering (IWMSE)*, 2008, pp. 23–32.

[21] D. Patterson, "The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges," 2009.

[22] M. Samadi, A. Hormati, J. Lee, and S. Mahlke, "Leveraging GPUs using cooperative loop speculation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 1, p. 3, 2014.

[23] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[24] A. Staff, "OpenCL and the AMD APP SDK 3.0," 2016.

[25] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proc. of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000, pp. 1–12.

[26] J. A. Stuart and J. D. Owens, "Efficient synchronization primitives for GPUs," *arXiv preprint arXiv:1110.4623*, 2011.

[27] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 54, 2013.

[28] C. von Praun, R. Bordawekar, and C. Cascaval, "Modeling optimistic concurrency using quantitative dependence analysis," in *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008, pp. 185–196.

[29] Z. Wang, D. Powell, B. Franke, and M. OBoyle, "Exploitation of GPUs for the parallelisation of probably parallel legacy code," in *Compiler Construction*, 2014, vol. 8409, pp. 154–173.

[30] C.-Z. Xu and V. Chaudhary, "Time stamp algorithms for runtime parallelization of DOACROSS loops with dynamic dependences," *IEEE Trans. Parallel Distrib. Syst. (TPDS)*, vol. 12, no. 5, pp. 433–450, 2001.

[31] J. Xue, *Loop tiling for parallelism*. Springer Science & Business Media, 2012, vol. 575.

[32] C. Zhang, G. Han, and C.-L. Wang, "GPU-TLS: An efficient runtime for speculative loop parallelization on GPUs," in *Proc. of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2013, pp. 120–127.

[33] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal, "GRace: A low-overhead mechanism for detecting data races in GPU programs," in *Proc. of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011, pp. 135–146.

[34] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke, "Uncovering hidden loop level parallelism in sequential applications," in *Proc. of the 14th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2008, pp. 290–301.