DESIGNING EFFICIENT BARRIERS AND SEMAPHORES

FOR GRAPHICS PROCESSING UNITS

By

ROHAN MAHAPATRA

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

UNIVERSITY OF WISCONSIN-MADISON
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

MAY 2020

Dedicated to all the frontline coronavirus workers who are risking their lives to save the world from the global CODIV-19 pandemic.

ACKNOWLEDGMENT

My journey at UW Madison has been extremely challenging and rewarding. I would sincerely like to thank everyone who supported me along the way.

At the outset, I would like to express my sincere gratitude to my advisor Prof. Matthew D. Sinclair for his constant guidance, encouragement, constructive comments, and motivation which gave a concrete shape to my research endeavors. His breadth of knowledge and a wide range of skills always astonished me and continue to impress me. Deep in my heart, I have a deep respect for him. When I look back, I find myself as a novice without any research experience. Prof. Matt taught me how to think about the big picture ideas in computer engineering, frame tractable research problems, use tools to prototypes the ideas, and tricks to debug problems. Prof. Matt has given me ample opportunities to learn, grow, and explore. Prof. Matt has been instrumental in molding me into what I am today. I would also like to thank Prof. Dan Negrut for his constructive feedback which helped me improve my thesis.

This thesis is a result of collaboration with Preyesh Dalmia, a teammate who has been a crucial partner in my academic and research journey at UW Madison. I deeply appreciate the hours we spent brainstorming, implementing, and debugging projects. I learned a lot from his intellectual knowledge, persistence, optimism, and hardworking nature. I am excited to see where his audacious work will lead him next and wish him luck in all his future endeavors. I would also like to thank Shivangi Kamat who constantly motivated me during my research.

I am thankful to all the members of the Heterogeneous Architecture Lab Research Group which whom I have either directly or indirectly collaborated. They have been especially excellent in providing feedback on presentations, and on relinquishing their Linux jobs to provide me extra slots during conference deadlines.

I am also grateful to the faculty of both ECE and CS Departments at the University of Wisconsin-Madison for their all-out help and academic guidance which helped me in keeping the momentum of my academic journey.

Last but not least, a huge thanks goes to my family especially my father, mother, and sister who has always been a constant source of encouragement and boosted my morale with frequent calls withstanding the time zone difference across continents.

Finally, a very special thanks to all my friends who have been instrumental in helping me get my MS degree.

On, Wisconsin!

(Rohan Mahapatra)

Abstract

General-purpose GPU applications that use fine-grained synchronization to enforce ordering between many threads accessing shared data have become increasingly popular. Thus, it is imperative to create more efficient GPU synchronization primitives for these applications. Accordingly, in recent years there has been a push to establish a single, unified set of GPU synchronization primitives. However, unlike CPUs, modern GPUs poorly support synchronization primitives. In particular, inefficient support for atomics, which are used to implement fine-grained synchronization, make it challenging to implement efficient algorithms. Therefore, as GPU algorithms are scaled to millions or billions of threads, existing GPU synchronization primitives either scale poorly or suffer from livelock or deadlock issues because of increased contention between threads accessing the same shared synchronization objects. In this work, we seek to overcome these inefficiencies by designing more efficient, scalable GPU global barriers and semaphores. In particular, we show how multi-level sense reversing barriers and priority mechanisms for semaphores can be extended from prior CPU implementations and applied to the GPUs unique processing model in order to improve performance and scalability of GPU synchronization primitives. Our results show that proposed designs significantly improve performance compared to state-of-the-art solutions like CUDA Cooperative Groups, and scale to an order of magnitude more threads – avoiding livelock as the algorithms scale compared to prior open source algorithms. Overall, across three modern GPUs: the proposed barrier implementation reduces atomic traffic by 50% and improves performance by an average of 26% over a GPU tree barrier algorithm and improves performance by an average of 30% over CUDA Cooperative Groups for four full-sized benchmarks; the new semaphore implementation improves performance by an average of 65% compared to prior GPU semaphore implementations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter One

INTRODUCTION

Traditionally, GPUs focused on providing high performance for streaming, data parallel workloads with limited data reuse or data sharing and coarse-grained synchronization that usually only happened at kernel boundaries. In recent years, applications with different characteristics have also started taking advantage of the parallelism exposed by the GPUs. For example, recent work has shown how to use GPUs for persistent kernels [11, 25, 73], buddy allocation [15], and particle partitioning [10] while other work shows how GPUs use fusing kernels [1, 17, 34] and concurrent streams [38, 36]. Unlike traditional, streaming, data parallel general-purpose GPU (GPGPU) applications, these applications utilize fine-grained synchronization across many threads. Hence, they make use of synchronization primitives such as locks, semaphores, and barriers [41, 57, 61]. Traditionally, developers have relied on complex GPU solutions [18] or utilized CPU-side solutions [64] for providing device-wide synchronization.

Unfortunately, fine-grained synchronization like barriers and semaphores is not well supported on modern GPUs. Unlike multi-core CPUs, which have significant OS support and complex coherence protocols like MOESI that make synchronization relatively cheap, GPUs have limited OS support and use simple, software-driven coherence protocols. To enforce consistency, the GPU coherence protocols invalidate all valid data in local caches (e.g., L1) at the beginning of a synchronization pair (e.g., a load acquire) and write through all dirty

data from local caches at the end of a synchronization pair (e.g., a store release) [5, 3, 20, 26, 55, 56, 58]. Thus, GPU synchronization accesses, which are usually implemented with atomics, are expensive on GPUs. To partially compensate for this, GPU memory consistency models utilize scoped synchronization, which allows programmers to specify the level at which threads need to synchronize [14, 22, 23, 29, 30]. With scopes, if the synchronizing threads are part of the same thread block (TB), then the synchronization occurs locally with minimal overhead. However, if the synchronizing threads are not part of the same TB, which is usually the case for these next-generation workloads, expensive global synchronization is still required. We discuss the GPU coherence and consistency support further in Section 2.

In recent years, companies have started adding better hardware support for atomics [42, 40], such that compared to NVIDIA's Kepler and Maxwell GPUs, atomics are an order of magnitude faster on Pascal and Volta GPUs. However, this support only accelerates the atomic operations themselves, not the corresponding heavyweight operations to invalidate valid data in the cache or write through dirty data. Moreover, it does not address the issue that globally scoped atomic operations must still take place at the last level cache. Consequently, fine-grained synchronization is still extremely expensive, and often represents the bottleneck for emerging workloads that utilize it. In addition, synchronization overheads are exacerbated by the level of parallelism on GPUs: GPUs typically run kernels with millions or billions of threads, which leads to significant contention for the synchronization variables.

Recently NVIDIA proposed a hierarchy of synchronization methods based on CUDA Cooperative Groups (CCG) [41] that span across all levels of granularity from a small group of threads in a GPU to a multi-GPU device. This approach offers tight integration with CUDA and performs well, especially at low contention levels, but as we show in Section 5, suffers from high contention for shared synchronization variables as the number of threads joining the barrier increase. Researchers have also developed device-wide software barriers [71] which implement GPU lock-based synchronization using mutex variables and global counters with atomic operations and lock-free synchronization. Other work developed portable barriers,

which dynamically compute an estimate of the SM occupancy for a given kernel, allowing for a starvation-free inter-block barrier by restricting the number of blocks according to this estimate [60]. However, both of these approaches limit the number of TBs that can be launched on an SM to avoid deadlock. Another popular open source device-wide barrier is HeteroSync's tree barrier [63], which uses two consecutive barriers to properly support for context switching. However, this approach significantly increases global atomic traffic to update the shared counter variables [57]. We propose to overcome these issues by introducing a two-level sense-reversing tree barrier for GPUs. Although tree barriers and sense reversing barriers have been widely used for CPUs [21, 63], extending them to work well on GPUs is novel. We discuss related work further in Section 6.

Semaphores allow multiple TBs to enter the critical section simultaneously, potentially improving performance. However, current reader-writer GPU semaphore implementations suffer from significant scalability bottlenecks as the number of threads increase. This problem is exacerbated when a writer attempts to exit the semaphore and multiple reader TBs attempt to enter it simultaneously, leading to frequent deadlocks under high contention. Thus, we propose a priority mechanism for the semaphore to give TBs exiting the semaphore priority.

Overall, this paper makes the following contributions:

1. We propose a two-level, sense reversing barrier for GPUs that reduces the number of atomic transactions required for synchronization by 50% compared to the baseline tree barrier.

2. We propose a priority mechanism to reduce the contention for GPU semaphores, which reduces the time spent on synchronization and eliminates deadlock compared to the baseline thereby improving scalability.

3. We quantitatively evaluate the proposed schemes against state-of-the-art GPU synchronization primitives. Compared to HeteroSync's tree barrier and semaphore, on

average our barrier improves performance by 14% on Pascal, 24% on Volta, and 41% on Turing GPUs; our semaphore improves performance by 35% for Pascal, 89% for Volta, and 70% for Turing GPUs, respectively. Moreover, compared to CUDA's CCG, on average our barrier algorithm scales significantly better, improving performance by 28% on Pascal, 41% on Volta, and 20% on Turing at the maximum contention level for four larger benchmarks that utilize global barriers.

Chapter Two

BACKGROUND

In this section, we first discuss GPU coherence and consistency models and how they impact programs with fine-grained synchronization (Section 2.1). We then discuss synchronization primitives used for CPUs and why they are inefficient on GPUs, and finally discuss about current state-of-the-art GPU implementation of synchronization primitives (Section 2.2).

2.1 GPU Coherence & Consistency

Traditional GPU applications were highly parallel and had coarse-grained synchronization, which enabled GPUs to use a simple, software-driven coherence protocol [3, 20, 26, 56, 55, 58]. The coherence protocol invalidates the cache on acquires (synchronization reads) while releases (synchronization writes) must flush the local store buffer and bypass the L1 cache and be performed at a memory that is shared by all the SMs (usually the L2). Thus, GPUs perform global synchronization at the shared L2. However, emerging applications require fine-grained synchronization which makes use of atomics [1, 10, 11, 15, 17, 25, 27, 34, 50, 51] and performing an acquire for synchronization will invalidate the entire cache while a release waits until all previous writes have reached the shared L2. This makes fine-grained synchronizations extremely costly from a performance perspective. To overcome this performance loss, programmers use scope-based synchronizations [14, 22, 23, 29, 30].

Both AMD and NVIDIA's memory consistency models extend the sequentially consistent for data-race-free (SC-for-DRF) memory consistency model that is widely used in CPUs [8, 31] with scoped synchronization [14, 22, 23, 29, 30]. Thus, it is referred to as the sequentially consistent for heterogeneous-race-free (SC-for-HRF) memory consistency model. Although there are multiple scopes in SC-for-HRF, we will focus on the two most widely used variants: local and global. An atomic with a local scope is only guaranteed to be visible to other threads in the same TB, while an atomic with global scope is visible to all threads across the GPU. Therefore locally scoped synchronization (e.g., CUDA's __threadfence_block()) is significantly cheaper than globally scoped synchronization. Thus, programmers have an incentive to synchronize locally wherever possible. However, when synchronization is required across TBs, global scope (e.g., CUDA's __threadfence()) must be used.

2.2 GPU Synchronization Primitives

In recent years, several projects sought to develop a common set of GPU synchronization primitives. In this work, we compare against the state-of-the-art open source solution, HeteroSync, and the state-of-the-art solution provided by NVIDIA, CCG. We discuss additional related work further in Section 6.

Synchronization Primitives (SyncPrims) originally released a set of mutexes, barriers, and semaphores focused on GPU atomic performance [61]. HeteroSync extended SyncPrims to model memory accesses that required fine-grained synchronization[1], optimize the algorithms, and added both local and globally scoped versions of most algorithms [57]. NVIDIA also introduced the CCGs [41], which allows kernels to dynamically organize groups of threads that span across all levels of granularity from a small group of threads in a GPU to a multi-GPU device. These primitives allow for new patterns of cooperative parallelism within

[1]SyncPrims' memory accesses in the critical sections used local scratchpads, and thus did not require any fine-grained synchronization to ensure ordering.

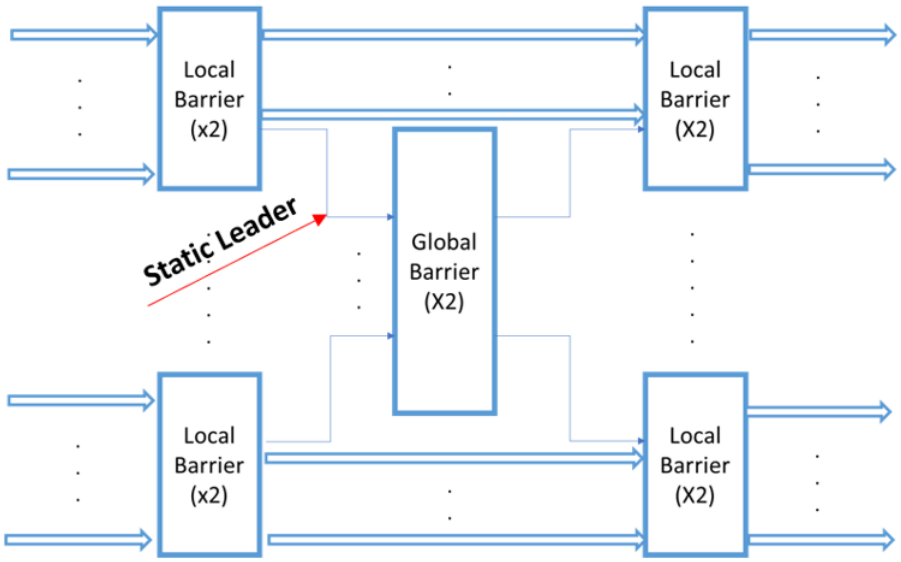

Figure 2.1 Baseline Two–Level Tree Barrier [57] with statically elected per-SM leader TB which joins the global barrier.

CUDA, including producer-consumer parallelism and global synchronization. Thus, we next describe HeteroSync's barrier and semaphore and CCG's barrier implementations.

2.2.1 Barriers

Figure 2.1 shows an illustration of HeteroSync's current two-level centralized tree barrier, which uses a hybrid local-global scope. In a centralized barrier, TBs increment a shared counter as they reach the barrier and spin until the counter indicates that all TBs are present. As compared to decentralized barriers, centralized barriers consume significantly lesser memory because of shared counters. In this implementation, all TBs on a Streaming Multiprocessor (SM) [6, 40] access unique data before joining a local barrier. In each TB, a single master thread performs the synchronization. Once all TBs have reached the local barrier, one designated leader TB from each SM joins the global barrier. As shown in Listing 2.1, all other TBs on a SM spin on a local variable until all TBs on this SM join the local barrier. Then, a statically assigned leader TB proceeds to join the global barrier as

```
1  __shared__ *done = 0;
2  // local scope store release
3  __threadfence_block();
4  atomicInc(perSMBarr, 0x7FFFFFFF);
5  while (!*done) {
6    if (atomicCAS(perSMBarr, numTBs_thisSM, 0) == 0) {
7      // local scope load acquire
8      __threadfence_block();
9      *done = 1;
10   }
11   __syncthreads();
12 }
```

Listing 2.1 Pseudo-code for baseline GPU per SM local barrier [57].

shown in Listing 2.2 while all other TBs proceed to join a second local barrier as illustrated in Listing 2.3. Crucially, in order to ensure correctness during potential context switches [2, 45, 65, 68, 67, 69, 72], HeteroSync's tree barrier uses a second barrier, which creates an effect similar to a sense-reversing barrier Instead of using the same value to represent the "done" and "not-done" states, barriers use opposite values [57]. Once the expected number of leader TBs reach the global barrier, these leader TBs proceed to join the second local barrier on the SM where other TBs from the same SM are spinning. However, this approach requires a significant number of additional atomics, which adversely impacts performance. Using a tree barrier partially mitigates this overhead by making many of the atomics locally scoped; nonetheless, it is not able to completely mitigate the overhead.

Although CCG is closed source, and thus not all of its implementation details are known, we disassembled and studied the SASS for a Volta `grid_group.sync()` (barrier) [39, 66]. Based on the SASS, CCG appears to use multiple different barrier implementations. Al-

```
1  __shared__ * done = 0;
2  // global scope store release
3  __threadfence();
4  // adds 1 for each TB that joins global barrier
5  atomicInc(globalBarr, 0x7FFFFFFF);
6  while (!*done) {
7    if (atomicCAS(globalBarr, numBarr, 0) == 0) {
8      // global scope load acquire
9      __threadfence();
10     *done = 1;
11   }
12   __syncthreads();
13   if (!*done) { DoBackoff(); }
14 }
```

Listing 2.2 Pseudo-code for baseline GPU global barrier [57].

though we could not determine when each barrier was dynamically selected, all of them use a single-level global memory barrier. The pseudo-code in Listing 2.4 shows the barrier that was selected for our configurations (Section 4). CCG utilizes an aggressive, single-level global memory barrier, similar to open source barrier implementations [60, 61]. Like HeteroSync, each TB elects a leader thread, and the remaining threads in the TB spin (`__syncthreads()`) waiting for the barrier to complete. Furthermore, CCG adds a sense direction counter that exploits integer overflow. Interestingly, CCG does not appear to perform any backoff, likely because it may increase latency when leaving the barrier. However, as we will show in Section 5, a single level barrier without backoff can also result in added contention, especially when many TBs join the barrier.

```
1  unsigned int * atom1 = perSMBarrBuffers[i];
2  unsigned int * atom2 = perSMBarrBuffers[i+1];
3  __shared__ int done1, done2;
4
5  cudaBarrierAtomic(atom1, &done1, numTBsThisSM);
6  // to avoid deadlock on context switches
7  cudaBarrierAtomic(atom2, &done2, numTBsThisSM);
```

Listing 2.3 Pseudo-code for baseline two-level tree barrier implementation [57]. The second call to the barrier is to create an effect similar to sense-reversing barrier.

2.2.2 Semaphores

Figure 2.2 provides a high-level overview of HeteroSync's reader-writer semaphore implementation. In this semaphore implementation, each SM has one writer TB that tries to write all the data, and $N-1$ reader TBs that try to read a subset of the data. When a TB tries to enter the critical section (the *post* sub-routine) as shown in Listing 2.5, it first acquires a mutex lock and checks to see if there is enough capacity in the semaphore for the TB. If there is capacity, then the TB updates the semaphore and releases the lock. Similarly, when a TB is leaving the critical section (the *wait* sub-routine), as shown in Listing 2.6, it acquires the mutex lock and updates the semaphore to remove itself from the semaphore before releasing the lock. When the size of the semaphore is greater than one, multiple reader TBs can enter the critical section simultaneously. Unlike on CPUs, which benefit from better OS support, both the post and wait sub-routines must utilize a lock to ensure ordering for accesses to the semaphore count ($semSize$) as shown in Listing 2.5. Thus, as the number of TBs increases, the mutex lock causes increased contention for reader and writer TBs trying to enter and exit the semaphore simultaneously. This occurs because a TB leaving the semaphore cannot exit since it cannot access the mutex variable, which other TBs trying to enter the semaphore

```
1  // global_barrier count initialized to 0
2  if (threadIdx == (0,0,0)) {
3    int direction, value_to_add;
4    if (blockIdx == (0,0,0)) {
5      valueToAdd = INT_MAX - total_blocks + 2;
6    } else { valueToAdd = 1; }
7    // 0: wait for negative, 1: wait for positive
8    direction = (global_barrier >= 0) ? 0 : 1;
9    __threadfence();
10   atomicAdd(&global_barrier, valueToAdd);
11   if (direction == 0) {
12     while(global_barrier >= 0) { ; }
13   } else {
14     while(global_barrier < 0) { ; }
15   }
16   __threadfence();
17 }
18 __syncthreads(); // other threads wait here
```

Listing 2.4 Pseudo-code for disassembled Volta CCG barrier [41].

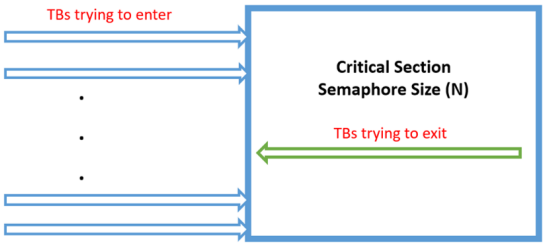

Figure 2.2 Baseline Semaphore Implementation [57].

keep accessing. One way to optimize the performance of these algorithms is to add backoff and make the TBs wait for a short period of time between each unsuccessful acquire.

```
1  while (!acqLock) {
2    // try to acquire the sem lock
3    if (atomicCAS(lock, 0, 1) == 0) {
4      // global scope load acquire
5      __threadfence();
6      acqLock = true;
7      if (isWriter) {
8        semSize -= maxSemCount;
9      } else {
10       --semSize;
11   } }
12   __syncthreads();
13 }
14 // global scope load acquire
15 __threadfence();
16 atomicExch(lock, 0);
```

Listing 2.5 Pseudo-code for the post routine of the baseline GPU semaphore [57]. Post enters the semaphore.

```
1  while (!acqLock) {
2    if (atomicCAS(lock, 0, 1) == 0) {
3      __threadfence();
4      acqLock = true;
5      // writers add the max value to the semaphore
6      if (isWriter) { semSize += maxSemCount; }
7      // readers add 1 to the semaphore
8      else { ++semSize; }
9  } }
10 // global scope store release
11 __threadfence();
12 atomicExch(lock, 0);
```

Listing 2.6 Pseudo-code for the wait routine of the baseline GPU semaphore [57]. Wait exits the semaphore.

Chapter Three

Design

3.1 Sense Reversing Barriers

As discussed in Section 2.2.1, HeteroSync's atomic tree barrier requires every TB to increment the global counter twice to avoid potential deadlock on a context switch. Current GPUs applications rarely switch contexts, although prior work has shown that it does happen, especially when higher priority tasks with hard real-time deadlines, such as graphics tasks arrive [45, 65, 68, 67, 69, 72]. Thus it is necessary to ensure correctness, similar to CCG's `direction counter`. The second barrier, which is necessary for correctness, causes significant overhead, even when a tree barrier is used to convert most of the atomics to locally scoped atomics. Thus, to overcome this issue, we propose to extend sense reversing barriers (SRBs) [21] from CPUs. Figure 3.1 shows a high-level overview of the proposed GPU SRB design.

Like the atomic tree barrier used in HeteroSync, our SRB utilizes a tree barrier to reduce contention. Further, we dynamically designate a leader TB to go and join the global barrier unlike previously where a statically elected leader TB was required to join the global barrier. We choose two levels for our tree barrier because this naturally conforms to the memory hierarchy and scoped synchronization of modern GPUs – locally scoped atomics can be performed in L1 caches co-located with each SM, while globally scoped atomics can

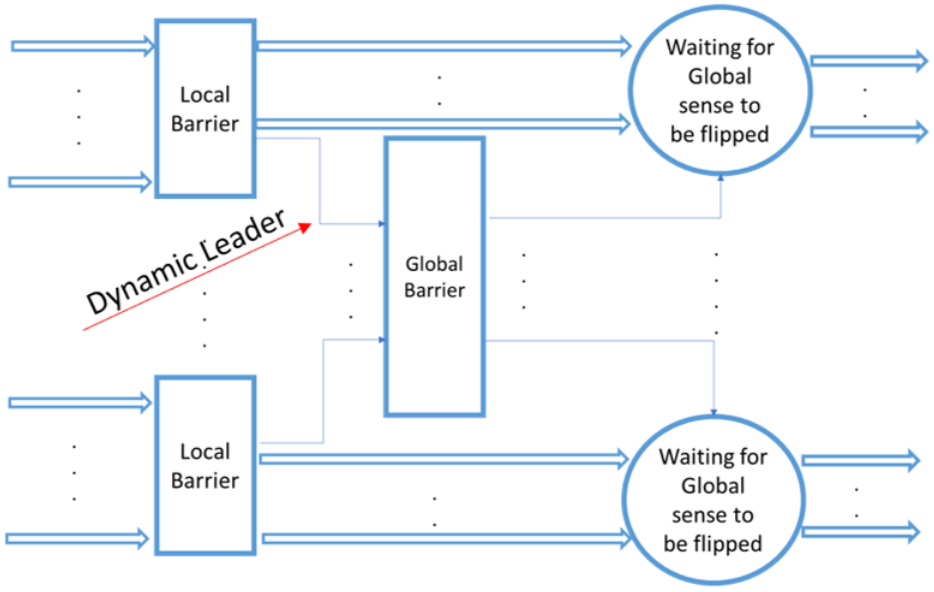

Figure 3.1 High-level overview of GPU Sense Reversing Barrier design with dynamically elected per-SM leader joining the global barrier. This leader TB also flips the sense.

be performed in L2 cache, which is shared across all SMs. Similar to the tree barrier in Listing 2.3, the SRB has two parts: a local barrier per SM (Listing 3.1) and a global barrier across all SMs (Listing 3.2).

Local Barrier: All TBs on a SM share a sense variable (`sense`), which is initialized to a constant value. As shown in Listing 3.1, TBs spin until their respective sense variable matches `sense`. As in CPU sense reversing barriers, `sense` matches when all TBs on a SM join the local barrier. Next, we dynamically designate a leader TB to proceed to the global barrier. To avoid overhead when a statically selected leader is not the last TB to join the local barrier (and thus may not be active), we dynamically make the last TB joining the local barrier the leader TB that joins to the global barrier.

Global Barrier: Next, all dynamic leaders increment a global counter for the global barrier across TBs. When this count reaches the total number of active SMs, all leader TBs have reached the global barrier and the global sense is flipped (Listing 3.2). Meanwhile, all other

```
1  while (*sense != s) {
2    if (atomicCAS(perSMBarr, numTBsThisSM, 0) == numTBsThisSM){
3      __threadfence();
4      *sense = s; // flip sense
5      *last_block = blockIdx.x; // dynamic leader
6    }
7    __syncthreads();
8  }
```

Listing 3.1 Pseudo-code for local (per SM) phase of proposed GPU sense-reversing tree barrier.

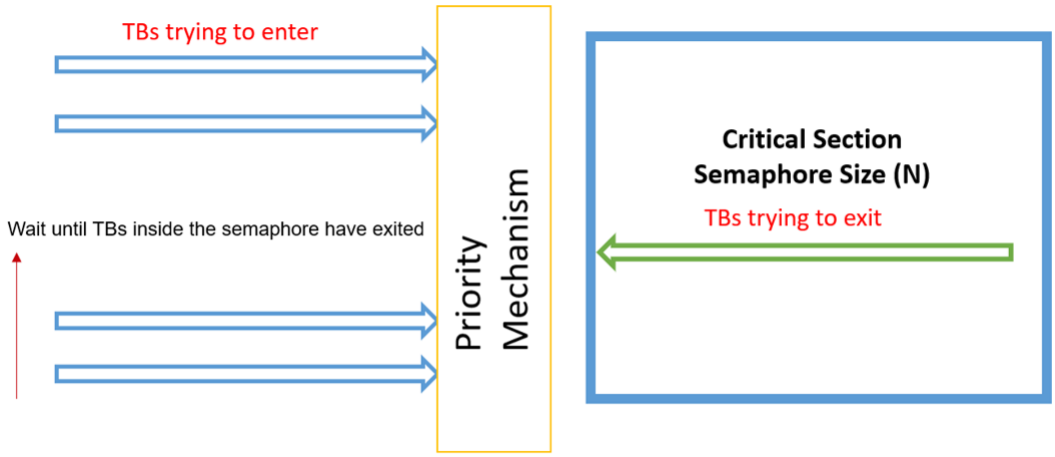

Figure 3.2 High-level overview of proposed priority semaphore.

TBs per SM except the leader spin, waiting for the global barrier to complete, as shown in Figure 3.1. Overall, this implementation retains the benefits of HeteroSync's tree barrier by keeping most thread's atomics local, while removing the second local barrier and dynamically designating a TB from the SM to join the global barrier.

```
1  while (*global_sense != *sense) {
2    // update global counter
3    if (atomicCAS(globalBarr, numBarr, 0) == numBarr) {
4      // global scope load acquire
5      __threadfence();
6      *global_sense = *sense; // flip sense
7    } else { DoBackoff(); }
8    __syncthreads();
9  }
```

Listing 3.2 Pseudo-code for global phase of proposed GPU sense-reversing tree barrier.

3.2 Semaphores

Semaphores allow multiple TBs to enter the critical section simultaneously based on the semaphore size. Typically, semaphores either require that the data accessed by each thread is unique, to avoid data races, or only allow readers to be in the semaphore simultaneously. As discussed in Section 2.2.2, due to the lack of OS and hardware support in modern GPUs, current semaphore implementations use mutex locks when updating the semaphore to ensure that multiple threads do not update the semaphore count simultaneously. However, this centralized mutex quickly becomes highly contented given the larger number of threads in GPU programs.

Moreover, this centralized mutex lock also creates a bottleneck for threads attempting to exit the semaphore, since they need to contend with threads trying to acquire the lock to enter the semaphore. We demonstrate that this leads to deadlock for most modern GPUs when the number of TBs/SM are scaled beyond 1 or 2 TBs/SM (Section 5.2). Even software back-off does not solve this deadlock, because it does not guarantee that threads trying to exit the semaphore can obtain the lock in a timely fashion. Thus, we propose to add a

```
1  acqLock = false;
2  while (!acqLock) {
3    // if any threads exiting semaphore, wait
4    while (atomicCAS(priority, 0, 0) != 0) {
5      DoBackoff();
6    }
7    if (atomicCAS(lock, 0, 1) == 0) {
8      // global scope load acquire
9      __threadfence();
10     acqLock = true;
11     if (isWriter) { semSize -= maxSemCount; }
12     else { --semSize; }
13   }
14 }
```

Listing 3.3 Pseudo-code for the core post routine of the proposed GPU priority semaphore. Post enters the semaphore.

priority mechanism to prioritize threads exiting the semaphore. This helps ensure forward progress and reduces the serialization penalty resulting from multiple tries at acquiring the lock variable by a group of TBs in which some are trying to exit while others enter the semaphore. Figure 3.2 shows a high-level overview of the proposed design.

Listings 3.3 and 3.4 show, respectively, the post and wait components of the proposed priority semaphore. The key difference between this code and the prior version shown in Listing 2.5 and Listing 2.6 is that by adding a priority mechanism, and checking this mechanism in post before attempting to acquire the lock, threads leaving the critical section are favored.

```
1  acqLock = false;
2  while (!acqLock) {
3    if (atomicCAS(lock, 0, 1) == 0) {
4      // global scope store release
5      __threadfence();
6      acqLock = true;
7    } else { atomicOr(priority, 1); }
8  }
9  // writers add the max value to the semaphore
10 if (isWriter) { semSize += maxSemCount; }
11 else { ++semSize; }
12 // global scope store release
13 __threadfence();
14 atomicExch(lock, 0);
15 atomicAnd(priority, 0);
```

Listing 3.4 Pseudo-code for the core wait routine of the proposed GPU priority semaphore. Wait exits the semaphore.

Chapter Four

Methodology

4.1 System Setup

To study the performance impact of our proposed algorithmic optimizations from Section 3, we run the baseline algorithms and the optimized version on three modern NVIDIA GPU architectures – GTX 1080 [46], Titan V [40] and RTX 2080Ti [43]. These GPUs are representative of the three most recent generations of GPUs. The system configurations are listed in Table 4.1. We use CUDA 10 for all experiments [37].

GPU Feature	**GTX 1080**	**Titan V**	**RTX 2080Ti**
Architecture	Pascal	Volta	Turing
# SMs	20	80	68
# CUDA Cores/SM	128	64	128
Max TBs/SM	32	32	16
GPU Base Clock	1480 MHz	1200 MHz	1350 MHz
L2 Cache Size	2816 KB	4608 KB	5632 KB
Memory	8 GB GDDR	16 GB HBM2	16 GB HBM2

Table 4.1 GPUs used to evaluate algorithms, with relevant system parameters.

4.2 Benchmarks

We use barrier and semaphore microbenchmarks and benchmarks to evaluate our proposed algorithms. The microbenchmarks allow us to quickly compare different synchronization approaches and different critical section sizes. For the baseline microbenchmarks, we use HeteroSync's tree barrier and semaphores with and without exponential software backoff, as described in Section 2. We also compare against CCG's barrier synchronization. All of the microbenchmarks perform a series of loads and stores to global memory locations. We also compare the performance of each barrier for four modern GPGPU benchmarks [9, 24], which are representative of larger GPU programs that use barriers. For BFS, SSSP, and PageRank, we use the bgg.gr (1 TB/SM), USA-road-d.NY (2 TBs/SM), USA-road-d.FLA (4 TBs/SM), USA-road-d.W (8 TBs/SM), and USA-road-dUSA-road-d.USA (16 TBs/SM) graphs from LonestarGPU [9] to model various levels of contention. Table 4.2 and Table 4.3 lists the synchronization primitive microbenchmarks and benchmarks used in the experiments.

4.3 Configurations

We conduct a series of experiments on the benchmarks in Table 4.2 and Table 4.3. For all experiments, we run the benchmark 10 times and take the average result to ensure we are seeing steady state behavior.

To ensure that each SM has the same number of TBs, we take advantage of NVIDIA's scheduler using a round robin arbitration scheme to schedule TBs across SMs [35, 49, 59].

To analyze the sensitivity of the algorithms under various loads, we vary the critical section size and contention level in isolation. First, we hold the contention level constant (by using 16 TBs/SM to model a high contention situation), and vary the critical section size by varying the number of global loads and stores each thread performs from 10 through 1000 in multiples of 10. For the barrier implementations, these accesses use the barriers to

Applications	**Description**
Barriers	
atomicTreeBarr [57]	Two-level atomic tree barrier
atomicTreeBarrSRB	Proposed two-level sense reversing tree barrier. ldst represents the number of load and store each thread performs.
Semaphores	
SpinSem1 [57]	Semaphore with size = 1
SpinSemEBO1 [57]	Semaphore with exponential software backoff and semaphore size = 1
SpinSem10 [57]	Semaphore with semaphore size = 10
SpinSemEBO10 [57]	Semaphore with exponential software backoff and semaphore size = 10
SpinSem120 [57]	Semaphore with semaphore size = 120
SpinSemEBO120 [57]	Semaphore with exponential software backoff and semaphore size = 120
PriorSem1	Proposed semaphore with writer priority flag and semaphore size = 1
PriorSem10	Proposed semaphore with writer priority flag and semaphore size = 10
PriorSem120	Proposed semaphore with writer priority flag and semaphore size = 120
PriorSemEBO1	Proposed semaphore with writer priority flag, exponential software backoff and semaphore size = 1
PriorSemEBO10	Proposed semaphore with writer priority flag, exponential software backoff and semaphore size = 10
PriorSemEBO120	Proposed semaphore with writer priority flag, exponential software backoff and semaphore size = 120

Table 4.2 Synchronization primitive microbenchmarks.

ensure that multiple threads are not accessing the same data simultaneously, while for the semaphores these accesses are all performed in the critical section. Second, to see the effect contention has on the algorithm's performance, we hold the critical section size constant (at 10 global loads and stores) and vary the number of TBs from 1 TB/SM to 32 TBs/SM (for

Applications	Description
BFS [9]	Graph traversal algorithm.
SSSP [9]	Graph algorithm which computes the shortest path of each node from a source node.
PageRank [9]	Ranks websites from search engine results.
Reduce [24]	Reduces array of elements into a single result.

Table 4.3 Synchronization primitive benchmarks.

Turing we vary from 1 TB/SM to 16 TB/SM; Turing allows a maximum of 16 TB/SM). For these experiments, we use weak scaling [16]. We also show the kernel GPU activity profiles provided by NVIDIA NVPROF [44] for both the baseline barrier/semaphore and the proposed implementations. Additionally, we present a breakdown of the synchronization and non-synchronization time for all implementations.

For the Semaphore, we compare execution times of the baseline implementation, proposed implementation, baseline implementation with exponential backoff and proposed implementation with exponential backoff across different contention levels. These different contention points are created by either varying the Semaphore size, number of TBs/SM, or number of load and store instructions while keeping the other two constants. We use semaphore sizes of 1, 10 and 120. These sizes are taken from HeteroSync and represent cases where the ratio of readers to writers varies.

Finally, we compute the breakdown of the synchronization and non-synchronization time for the microbenchmarks to understand the amount of time they spend on synchronization to access the critical section. Our optimizations seek to reduce the synchronization time by reducing the number of atomic transactions in the case of barriers, and the degree of contention in accessing the critical section in the case of semaphores.

Chapter Five

Evaluation

5.1 Barriers

This section compares and analyzes the performance of all three barriers with increase in the number of TBs to vary the contention level and increase in the length of critical section size.

5.1.1 Increase in contention level

Figure 5.4 compares the execution time of all three barriers, as the number of TBs vary. Overall, these results show that the sense reversing barrier (SRB) and CCG significantly outperform the baseline. On average, the SRB improves performance by 14% on the Pascal, 24% on the Volta, and 41% on the Turing GPU compared to the baseline atomicTreeBarrier. By removing the second barrier call, SRB significantly reduces the number of atomic accesses in the baseline by over 50%. Moreover, dynamically selecting a TB as the per SM leader further improves performance, by avoiding stalls when the static leader is not active. SRB's gains over the baseline also get better as contention increases; although the baseline also uses a tree barrier, the additional atomics it must perform relative to SRB cause execution time to continue increasing faster than SRB's does. This demonstrates how SRB's changes improve scalability. To further examine the differences between the baseline and SRB, Figure 5.5

breaks down their execution time in terms of synchronization and non-synchronization time for the Volta GPU. On average, SRB reduces the synchronization time by 78%, which in turn reduces SRB's overall execution time. Moreover, non-synchronization time only increases a little for SRB as contention increases, demonstrating that the reduction in synchronization time is not replaced by power-hungry active waiting – which would reduce synchronization time without reducing execution time.

CCG also provides significant benefits over the baseline, showing similar trends to SRB relative to the baseline. Notably, CCG outperforms SRB at lower levels of contention. This makes sense, because at low levels of contention, SRB's software backoff and hierarchical design offer less benefit. But, as the level of contention increases, and more TBs are joining CCG's single-level global barrier, SRB approaches and then outperforms CCG for Volta and Turing. However, for the Pascal GTX 1080, CCG always outperforms SRB for the microbenchmark. We believe this sizable performance difference is the result of Pascal having a much smaller number of SMs (Table 4.1), which reduces the number of TBs that need to be scheduled to create similar levels of GPU occupancy (and contention) compared to the Volta and Turing GPUs. Since the number of TBs is smaller for the same contention level, the number of accesses to the shared variables also decreases, making CCG the best choice for the microbenchmark on the Pascal GTX 1080. Nevertheless, since CCG and SRB significantly outperform the baseline, we focus on the comparison between CCG and SRB in the full-sized benchmarks that use barriers.

To further explore CCG and SRB's behavior, Figure 5.9 examines their performance as contention varies for four full-sized benchmarks that use barriers. Similar to the microbenchmarks, at lower contention levels CCG again outperforms SRB. However, unlike the microbenchmarks, SRB consistently outperforms CCG at much lowers of contention (e.g., 4 TBs/SM for Volta and Turing). This happens because SRB's use of locally scoped atomics reduces the number of global flushes and invalidations, which improves scalability and performance relative to CCG, which must globally flush and invalidate more frequently

due to its single-level design. This difference is further magnified since CCG does not perform backoff. Thus, as contention increases, SRB reduces unnecessary accesses to the shared synchronization variables. In general, for all four benchmarks, SRB's performance improvement over CCG is closely tied to the percentage of total execution time spent in global synchronization. For example, global synchronization is a major component in the *Reduce* benchmark. Hence for Reduce SRB outperforms CCG by an average of 62% at the maximum contention level, across the three GPU architectures. In contrast, in PageRank a smaller percentage of the overall execution time is spent in synchronization. As a result, SRB only improves performance by 4% on average over CCG. Finally, across all three GPUs, for BFS and SSSP, CCG consistently outperforms SRB for lower contention levels (e.g., 11% and 17% better for BFS and SSSP, respectively, on average at the lowest contention level). However, as contention increases, SRB's advantages again enable it to outperform CCG: at the maximum contention level SRB is 24% and 31% better than CCG on average across the three GPUs, respectively, for BFS and SSSP.

5.1.2 Increase in critical section length

Given these above results, we also evaluate how all three barrier algorithms scale as the size of the critical section varies from 10-1000 in Figure 5.13; these critical section sizes are evaluated at 1 TB/SM, 16 TBs/SM, and 32 TBs/SM for Pascal and Volta, and 1 TB/SM, 8 TBs/SM, and 16 TBs/SM for Turing. These results confirm the results from the prior two studies: as contention increases, SRB again starts to outperform CCG. Interestingly, as critical section size increases, CCG and SRB provide smaller benefits over the baseline. This occurs because an increasing amount of time in these applications is spent performing data accesses.

5.2 Semaphores

For the Semaphore, we compare execution times of the baseline implementation, proposed implementation, baseline implementation with exponential backoff, and proposed implementation with exponential backoff across different contention levels and different critical section lengths.

5.2.1 Increase in contention level

These different contention points are created by either varying the Semaphore size, number of TBs/SM, or number of load and store instructions while keeping the other two constants. Figures 5.17, 5.21, and 5.25 show the execution times of all the four implementations across different contention levels ranging from 1 TB/SM to the maximum TB/SM across different GPU architectures (Table 4.1). Across the data points available the proposed implementation shows a 35% performance gain over the baseline implementation on the Pascal, 89% on the Volta and 70% on the Turing GPU on average for semaphore of size 1. For the semaphores with less contentions, that is, 10 and 120, the gains are slightly more modest. For example, for size 10, the average performance gain is 29% for Pascal, 89% for Volta, and 66% for Turing. The baseline implementation with exponential backoff shows much better performance and is only 10% slower than the proposed solution on the Pascal and 5% slower on the Volta GPU for a semaphore of size 1. However, data for the baseline could be gathered across only a few data points due to the problem of livelock with increased contention for the centralized semaphore variable. We were only able to get data for the contention case of up to 4 TBs/SM on the Pascal GPU and 1 TB/SM on the Volta and Turing GPUs. The baseline implementation which uses exponential backoff to reduce the number of accesses to synchronization variables also livelocks at the same data points. In comparison, our proposed implementation alleviates the problem of livelock in the baseline implementation across all the data points considered. This demonstrates the value of the priority mechanism, which

prevents TBs entering the semaphore from stopping TBs trying to exit the semaphore, thus ensuring thus forward progress. The GPU kernel activity profile (Figures 5.30 and 5.31) shows that our priority mechanism also reduces the redundant tries to acquire the lock variable by TBs trying to enter the semaphore when there is already a writer TB inside the semaphore thereby reducing the overall synchronization time from 12% in baseline to 5% in the proposed implementation. Further, Figure 5.32 shows that the priority mechanism also reduces the amount of time TBs spends on the critical section (called as synchronization time) by 88% as compared to the baseline.

5.2.2 Increase in critical section length

Finally, as with the SRB, we evaluate the semaphores across different lengths of the critical sections and for different numbers of TBs/SM based on the GPU architecture. We see a similar trend to the sense reversing barrier for performance gain (Figure 5.29): with increasing critical sections size the overall gains decrease because the percentage of time spent in synchronization decreases as we increase the size of the critical section. Overall, these results show that SRB significantly improves on the baseline barrier and semaphore, and outperforms the state-of-the-art CCG barrier as contention increases.

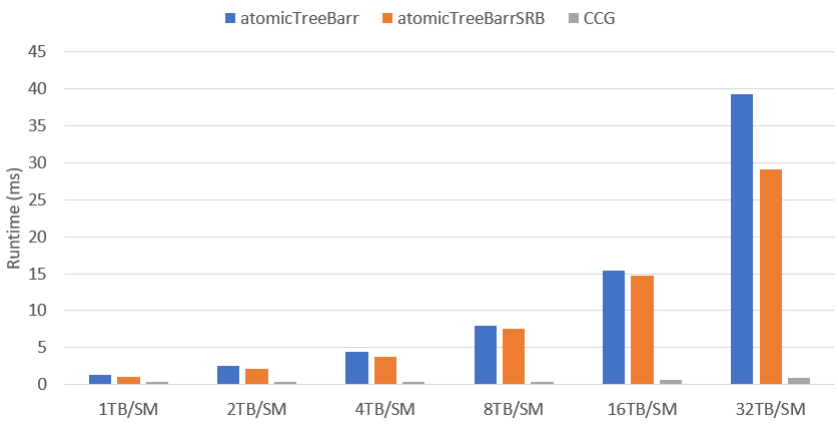

Figure 5.1 Pascal Architecture

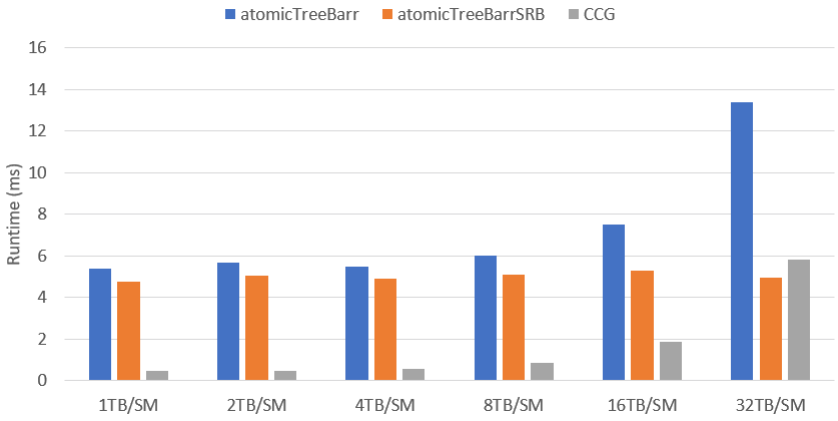

Figure 5.2 Volta Architecture

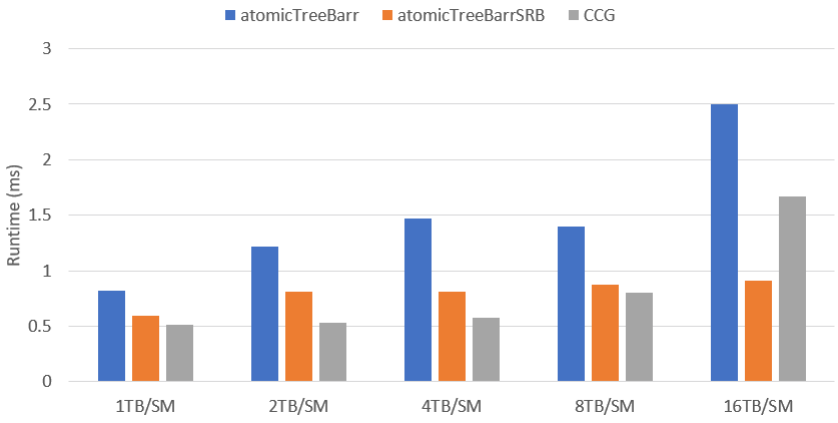

Figure 5.3 Turing Architecture

Figure 5.4 Execution time for all sense reversing barriers, as contention increases.

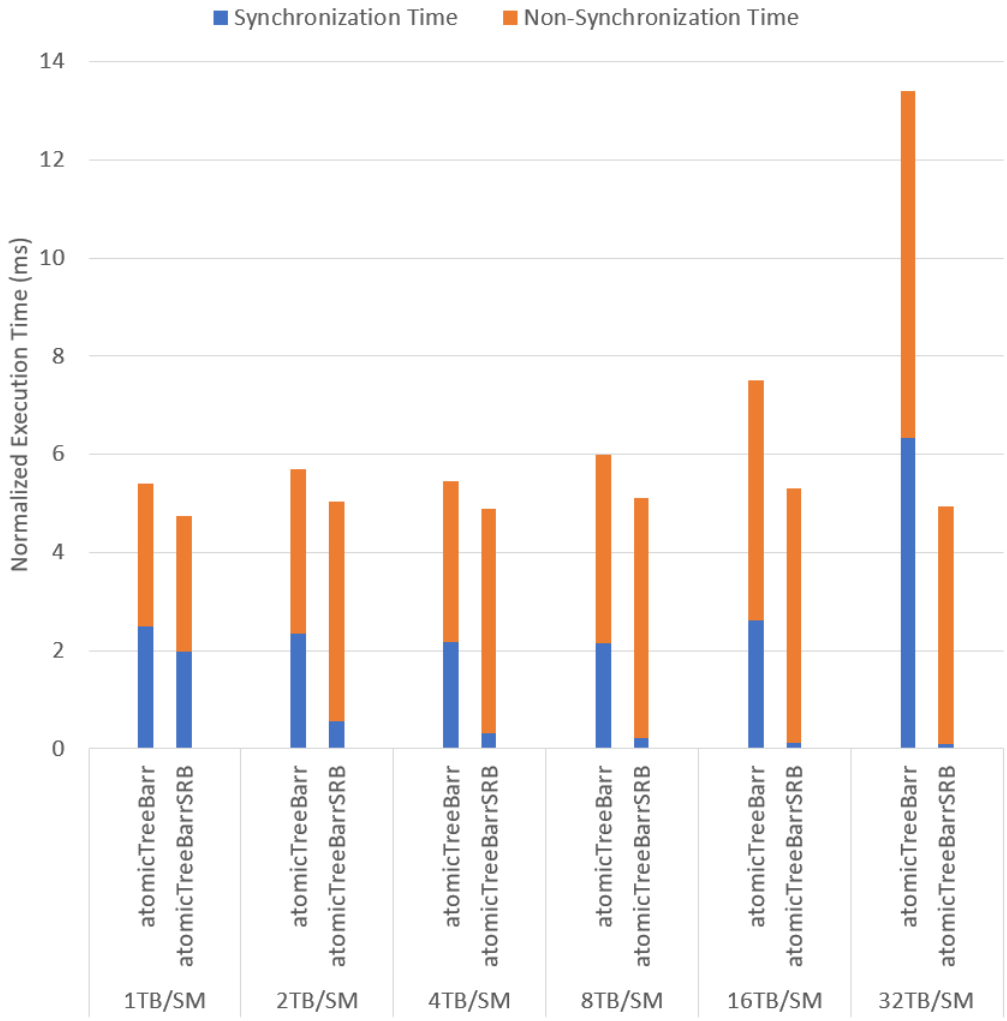

Figure 5.5 Comparison of sync and non-sync time for barriers in Volta GPU architecture.

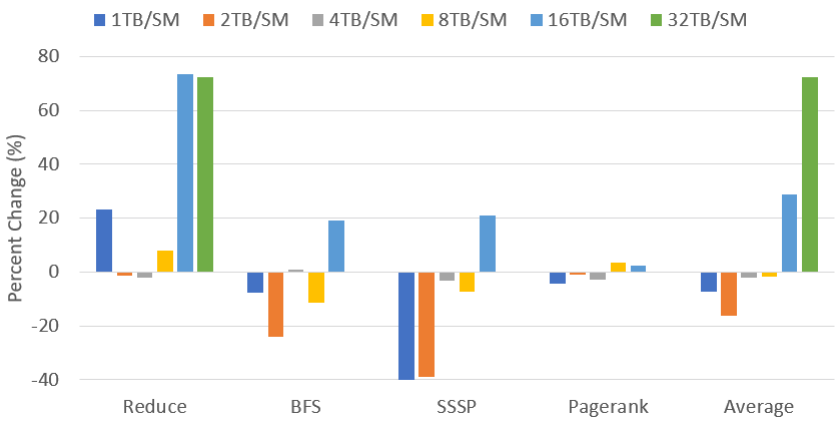

Figure 5.6 Pascal Architecture

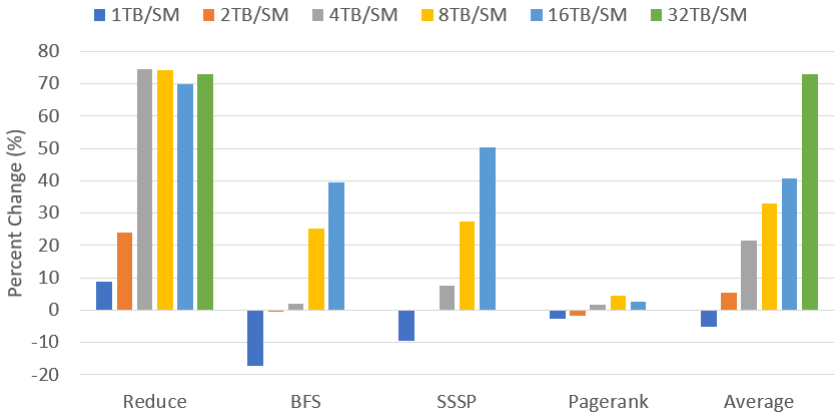

Figure 5.7 Volta Architecture

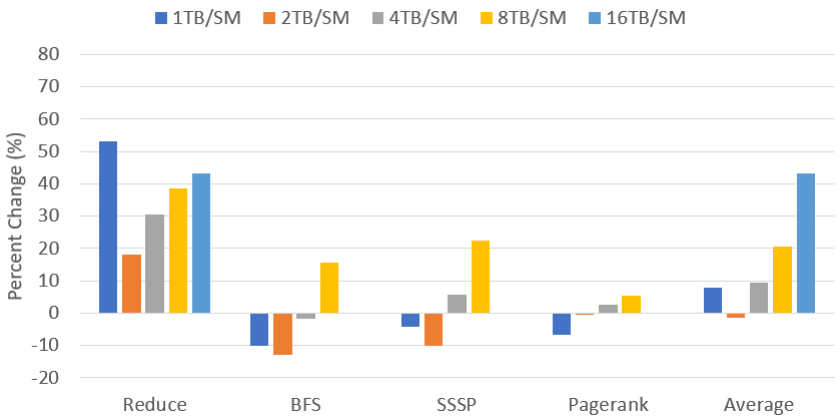

Figure 5.8 Turing Architecture

Figure 5.9 Execution time for benchmarks with barriers as contention varies.

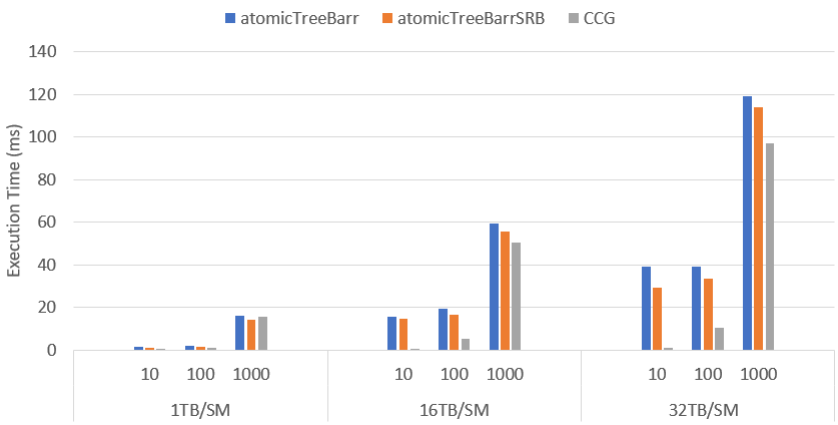

Figure 5.10 Pascal Architecture

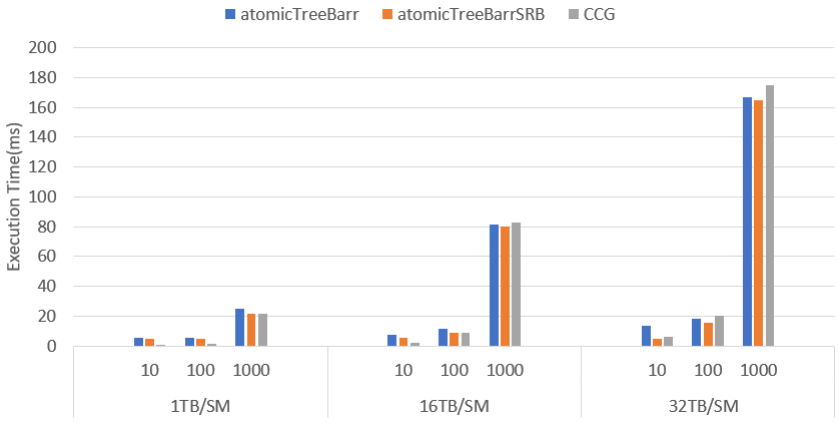

Figure 5.11 Volta Architecture

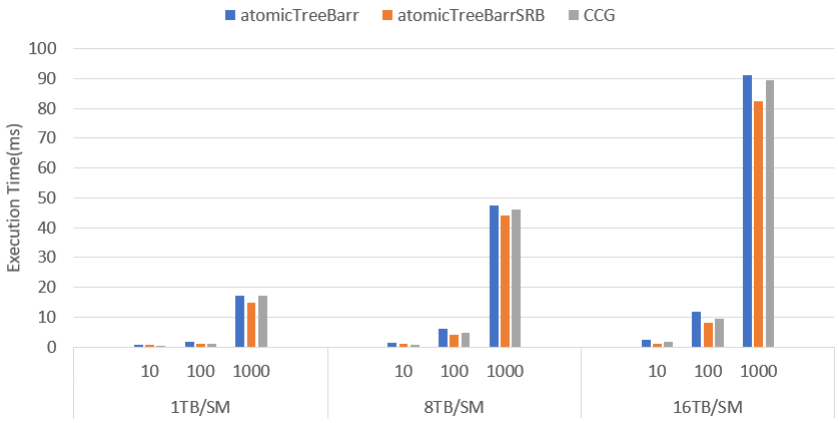

Figure 5.12 Turing Architecture

Figure 5.13 Execution time for barrier microbenchmarks as critical section size varies.

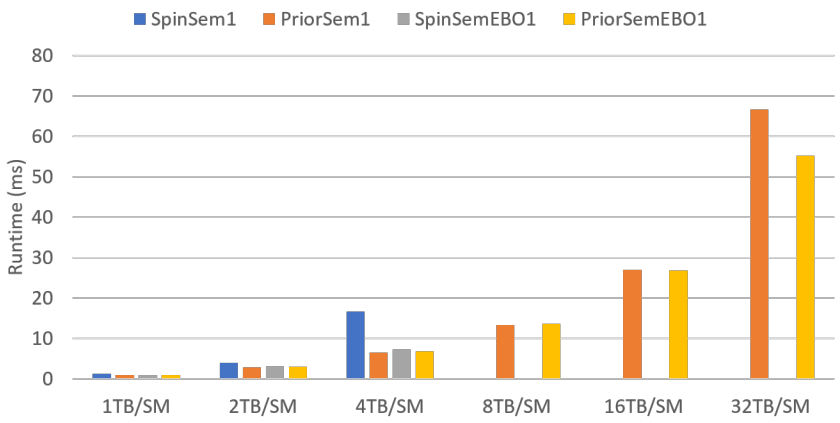

Figure 5.14 Pascal Architecture

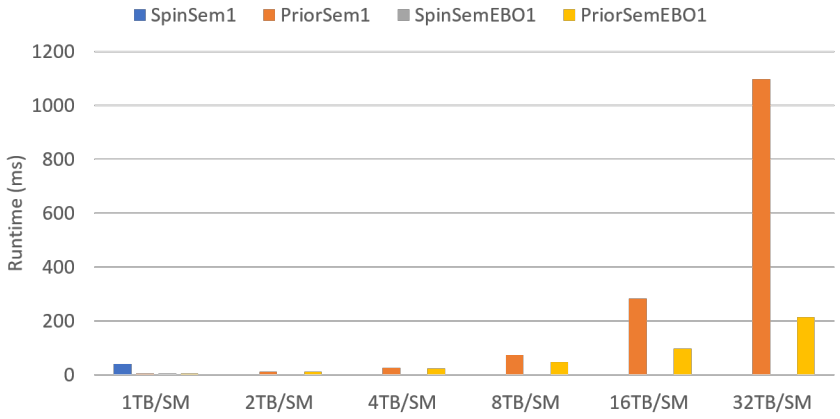

Figure 5.15 Volta Architecture

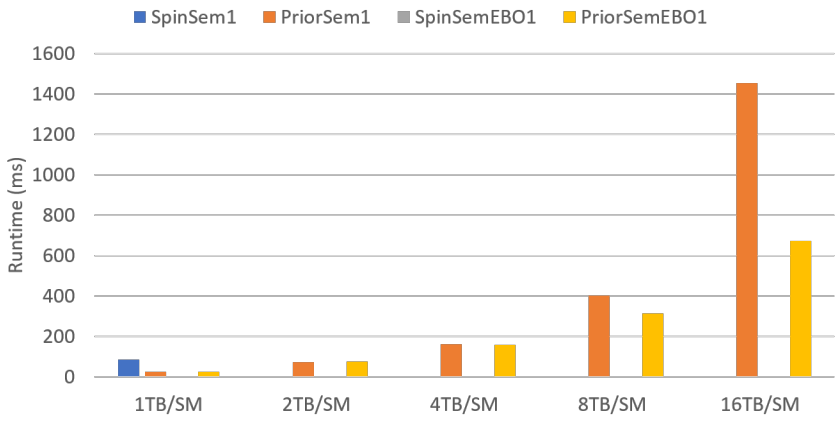

Figure 5.16 Turing Architecture

Figure 5.17 Execution time for all size 1 semaphores, as contention increases.

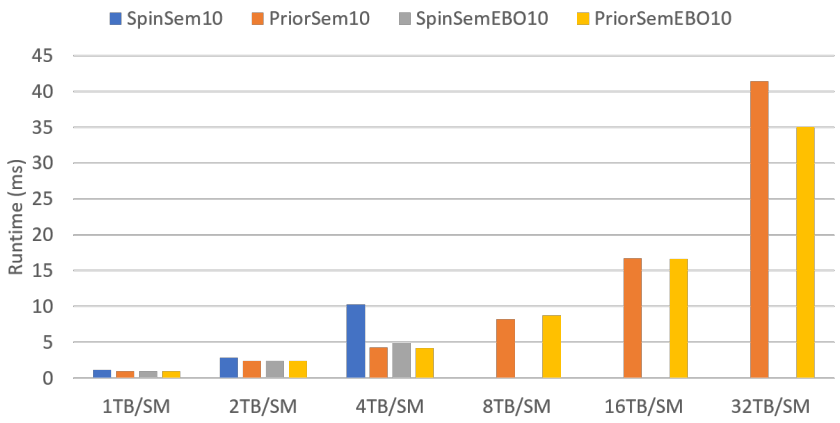

Figure 5.18 Pascal Architecture

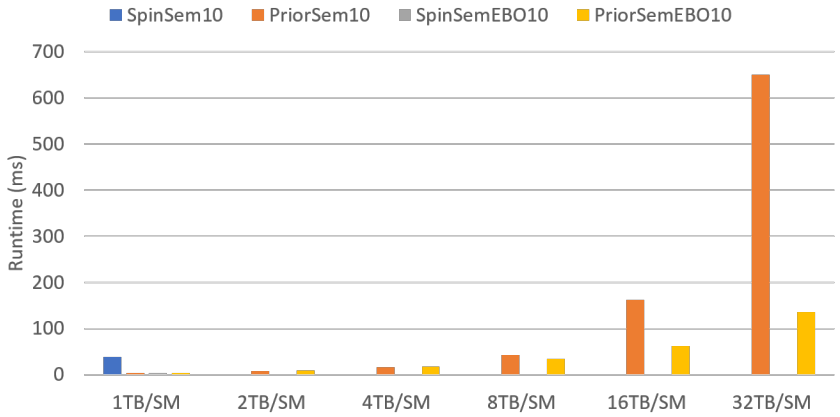

Figure 5.19 Volta Architecture

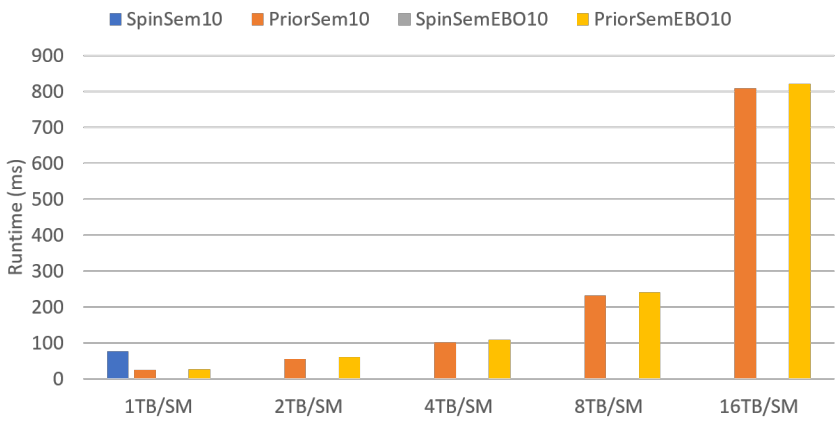

Figure 5.20 Turing Architecture

Figure 5.21 Execution time for all size 10 semaphores, as contention increases.

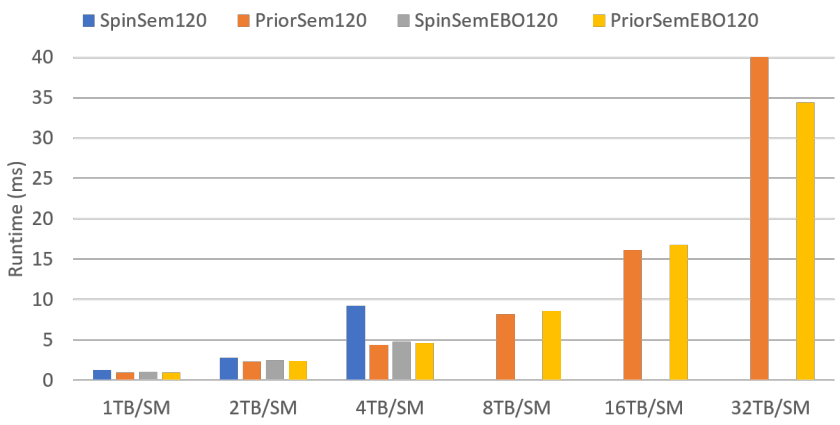

Figure 5.22 Pascal Architecture

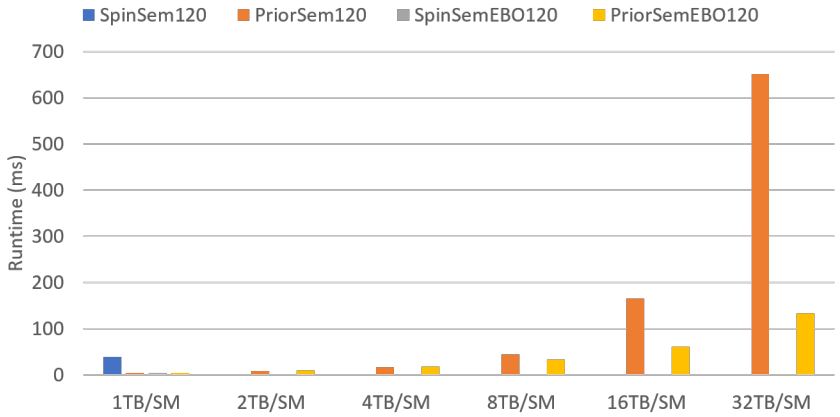

Figure 5.23 Volta Architecture

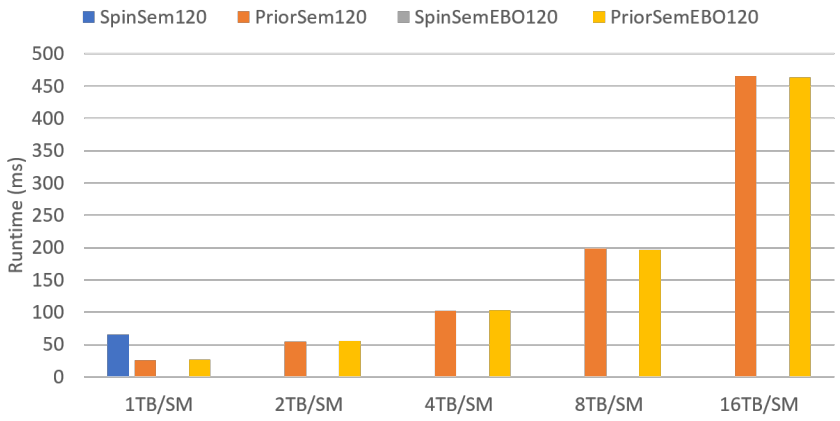

Figure 5.24 Turing Architecture

Figure 5.25 Execution time for all size 120 semaphores, as contention increases.

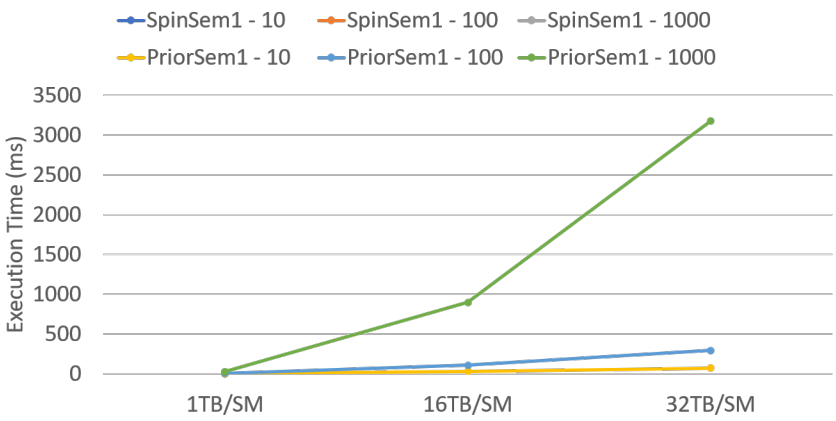

Figure 5.26 Pascal Architecture

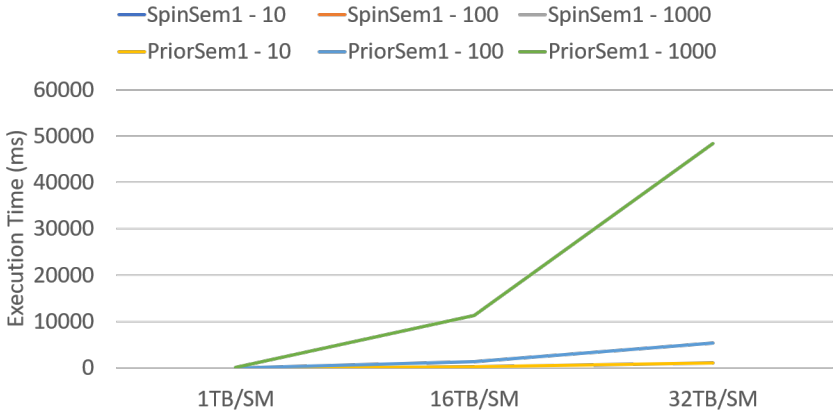

Figure 5.27 Volta Architecture

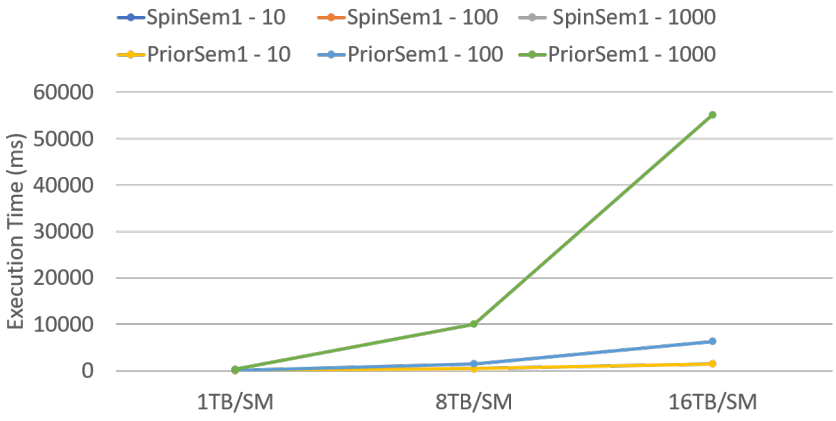

Figure 5.28 Turing Architecture

Figure 5.29 Execution time for semaphores as critical section size varies. The label shows the name of the microbenchmarks suffixed with the critical section size.

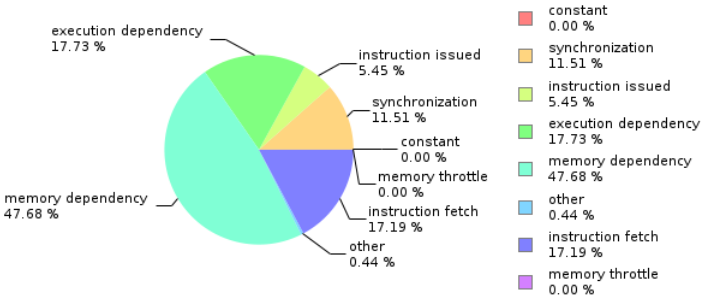

Figure 5.30 GPU kernel profile for the baseline semaphore.

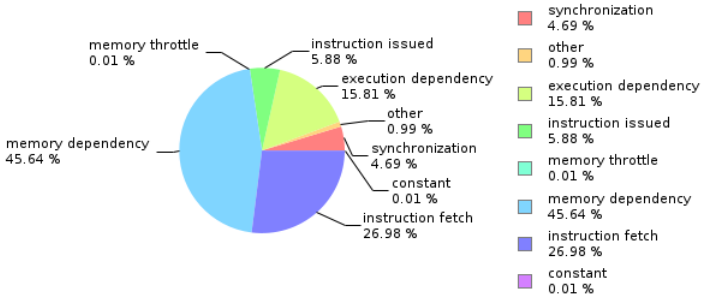

Figure 5.31 GPU kernel profile for the priority semaphore.

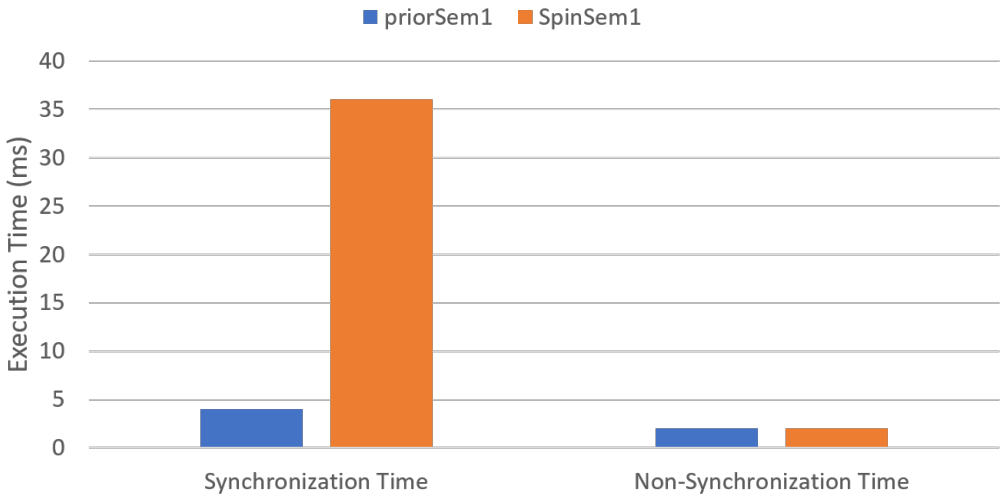

Figure 5.32 Comparison of semaphore sync and non-sync time.

Chapter Six

Related Work

CPU Synchronization Primitives: CPUs have a long history of developing efficient synchronization primitives for fine-grained synchronization. This support include centralized and decentralized mutexes [7, 32, 33, 47, 48, 54], ticket locks [19, 28], barriers [21, 63], and semaphores [12]. Some of these approaches, like ours, even use backoff techniques to reduce contention [7, 33]. However, GPU applications traditionally haven't utilized fine-grained synchronization. Moreover, the level of parallelism in GPUs necessitates simpler coherence protocols and less OS involvement.

Currently, most modern CPU operating systems include highly optimized synchronization primitives such as spin mutexes, barriers, and semaphores [13]. The simplest realization of a mutex on a CPU is a simple spin lock implemented using atomic instructions [52, 53, 62]. However, spin mutex locks and other synchronization primitives do not scale well on a GPU as the number of threads increases. This is because most CPU spin locks primitives are implemented using linked data structures which causes warp divergence in GPUs [70]. Further, as the number of threads increases, these mutexes use excessive atomics which degrades performance (Section 2.1). As a result, most of the CPU advancements in this space have yet to be applied to GPUs. Regardless, as GPU applications become increasingly general-purpose, the need for better synchronization techniques on GPUs has increased. Our work make strides in this direction and improves over the current state-of-the-art for GPUs.

GPU Synchronization Primitives: Although GPU synchronization support is still in its infancy compared to CPUs, prior work has laid the groundwork we build upon in this work. Prior work such as SyncPrims [61] developed micro-benchmarks consisting of synchronization primitives for discrete GPUs. HeteroSync [57] extended SyncPrims to resolve this issue, and created a single benchmark suite which implements various synchronization primitives, along with variants for locally and globally scoped atomics [22], as well as relaxed atomics [55]. However, as shown in Section 5 the barrier and semaphore implementations in HeteroSync do not scale well with the number of thread blocks and result in livelock. Our proposed approaches address these shortcomings and create optimized barriers and semaphores which both scale well as the number of thread block increases and also perform significantly lesser atomics. HeteroSync also includes a decentralized, two-level lock-free barrier, which extends a prior decentralized, single-level lock-free barrier [71]. Decentralized barriers trade off increased memory consumption for reduced contention, improved efficiency, and improved scalability. However, unlike our approach, it does not focus on centralized barriers and consumes significantly more memory.

Prior work also proposed a protocol that dynamically estimates a GPU kernel's occupancy for barriers [60]. Sorensen's work modifies existing inter-TB barrier to use OpenCL atomic operations and restricts the number of TBs based on the discovery protocol estimate. Since the restricted number of occupant TBs guarantees fair scheduling, they can reliably participate in an inter-TB barrier and ensure deadlock-freedom. Finally, GPU manufactures such as AMD and NVIDIA have started to add support for both intra- and inter-block synchronization. Recent AMD GPUs provide hardware support for synchronization as a part of the Global Data Share [4]. However, this hardware support is limited in terms of scope since only a fixed number of barriers can be run.

Chapter Seven

Conclusion

Although GPUs have traditionally run applications with only coarse-grained synchronization, in recent years an increasing number of workloads have started to utilize GPUs. Thus, efficient support for fine-grained GPU synchronization via barriers and semaphores is increasingly important. Recent work has made significant strides in both academia and industry. However, we found that these algorithms, when used on modern GPUs with millions of threads, often scale poorly or suffer from deadlock at high contention levels. To overcome these issues, we propose improvements to these state-of-the-art GPU barrier and semaphore synchronization primitives. Our results show that our algorithms significantly improve the performance and scalability of GPU barriers and semaphores for NVIDIA Pascal, Volta, and Turing GPUs and avoid deadlocks. In particular, the techniques proposed reduce global memory traffic, especially the number of atomics, for barriers, and avoids extraneous contention over entering and exiting the semaphore. This results in an average of 26% performance improvement for the sense reversing barrier relative to the baseline atomic tree barrier; the new semaphore implementation improves performance by 65% on average. Moreover, although NVIDIA's state-of-the-art CCG barrier outperforms our two-level SRB at low levels of contention, as contention increases, our approach scales much better. For example, at the highest level of contention, for four full-sized benchmarks our barrier outperforms CCG by 28% on Pascal, 41% on Volta, and 20% on Turing.

REFERENCES

[1] AmirAli Abdolrashidi et al. "Wireframe: Supporting Data-Dependent Parallelism through Dependency Graph Execution in GPUs". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 '17. Cambridge, Massachusetts: Association for Computing Machinery, 2017, pp. 600–611. ISBN: 9781450349529. DOI: 10.1145/3123939.3123976. URL: https://doi.org/10.1145/3123939.3123976.

[2] J. T. Adriaens et al. "The Case for GPGPU Spatial Multitasking". In: *IEEE International Symposium on High-Performance Computer Architecture*. HPCA. 2012, pp. 1–12.

[3] Johnathan Alsop et al. "Lazy release consistency for GPUs". In: *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. 2016, 26:1–26:13. DOI: 10.1109/MICRO.2016.7783729. URL: https://doi.org/10.1109/MICRO.2016.7783729.

[4] AMD. *AMD Graphics Core Next (GCN) Architecture*. https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf. 2016.

[5] AMD Research. *AMD's gem5 APU Simulator*. http://www.gem5.org/wiki/images/7/7a/2015_ws_03_amd-apu-model.pdf.

[6] T. Amert et al. "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed". In: *2017 IEEE Real-Time Systems Symposium (RTSS)*. Dec. 2017, pp. 104–115. DOI: 10.1109/RTSS.2017.00017.

[7] T. E. Anderson. "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors". In: *IEEE Trans. Parallel Distrib. Syst.* 1.1 (Jan. 1990), pp. 6–16. ISSN: 1045-9219. DOI: 10.1109/71.80120. URL: https://doi.org/10.1109/71.80120.

[8] Hans-J. Boehm and Sarita V. Adve. "Foundations of the C++ concurrency memory model". In: *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*. 2008, p. 68. DOI: 10.1145/1375581.1375591. URL: http://portal.acm.org/citation.cfm?doid=1375581.1375591.

[9] M. Burtscher, R. Nasre, and K. Pingali. "A Quantitative Study of Irregular Programs on GPUs". In: *IEEE International Symposium on Workload Characterization*. IISWC. Nov. 2012, pp. 141–151. DOI: 10.1109/IISWC.2012.6402918.

[10] Daniel Cederman and Philippas Tsigas. "On Dynamic Load Balancing on Graphics Processors". In: *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. Eurographics Association. 2008, pp. 57–64.

[11] Greg Diamos et al. "Persistent RNNs: Stashing Recurrent Weights On-Chip". In: *Proceedings of the 33nd International Conference on Machine Learning*. ICML. 2016, pp. 2024–2033.

[12] Edsger W. Dijkstra. "A tutorial on the split binary semaphore". circulated privately. Mar. 1979. URL: http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD703.PDF.

[13] J. I. Fiestas and R. E. Bustamante. "A Survey on the Performance of Different Mutex and Barrier Algorithms". In: *2019 IEEE XXVI International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*. 2019, pp. 1–4.

[14] Benedict R. Gaster, Derek Hower, and Lee Howes. "HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models". In: *ACM Trans. Archit. Code Optim.* 12.1 (Apr. 2015), 7:1–7:26. ISSN: 1544-3566. DOI: 10.1145/2701618. URL: http://doi.acm.org/10.1145/2701618.

[15] Isaac Gelado and Michael Garland. "Throughput-Oriented GPU Memory Allocation". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 27–37. ISBN: 9781450362252. DOI: 10.1145/3293883.3295727. URL: https://doi.org/10.1145/3293883.3295727.

[16] John L. Gustafson. "Gustafson's Law". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 819–825. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_78. URL: https://doi.org/10.1007/978-0-387-09766-4%5C_78.

[17] I. E. Hajj et al. "KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism". In: *49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. 2016, pp. 1–12.

[18] Mark Harris et al. "Optimizing parallel reduction in CUDA". In: *Nvidia developer technology* 2.4 (2007), p. 70.

[19] Bijun He, William N. Scherer, and Michael L. Scott. "Preemption Adaptivity in Time-Published Queue-Based Spin Locks". In: *High Performance Computing – HiPC 2005*. Ed. by David A. Bader et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 7–18. ISBN: 978-3-540-32427-0.

[20] B.A. Hechtman et al. "QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs". In: *20th International Symposium on High Performance Computer Architecture*. HPCA. Feb. 2014, pp. 189–200. DOI: 10.1109/HPCA.2014.6835930.

[21] Debra Hensgen, Raphael Finkel, and Udi Manber. "Two Algorithms for Barrier Synchronization". In: *Int. J. Parallel Program.* 17.1 (Feb. 1988), pp. 1–17. ISSN: 0885-7458. DOI: 10.1007/BF01379320. URL: https://doi.org/10.1007/BF01379320.

[22] Derek R. Hower et al. "Heterogeneous-race-free Memory Models". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. Salt Lake City, Utah, USA: ACM, 2014, pp. 427–440. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541981. URL: http://doi.acm.org/10.1145/2541940.2541981.

[23] HSA Foundation. *HSA Platform System Architecture Specification*. http://www.hsafoundation.com/?ddownload=4944. 2015.

[24] W. A. R. Jradi, H. A. Dantas do Nascimento, and W. Santos Martins. "A Fast and Generic GPU-Based Parallel Reduction Implementation". In: *Symposium on High Performance Computing Systems*. WSCAD. 2018, pp. 16–22.

[25] Farzad Khorasani et al. "In-Register Parameter Caching for Dynamic Neural Nets with Virtual Persistent Processor Specialization". In: *Proceedings of 51st IEEE/ACM International Symposium on Microarchitecture*. MICRO. 2018.

[26] Konstantinos Koukos et al. "Building Heterogeneous Unified Virtual Memories (UVMs) Without the Overhead". In: *ACM Trans. Archit. Code Optim.* 13.1 (Mar. 2016), 1:1–1:22. ISSN: 1544-3566. DOI: 10.1145/2889488. URL: http://doi.acm.org/10.1145/2889488.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105. URL: http://dl.acm.org/citation.cfm?id=2999134.2999257.

[28] Leslie Lamport. "A New Solution of Dijkstra's Concurrent Programming Problem". In: *Commun. ACM* 17.8 (Aug. 1974), pp. 453–455. ISSN: 0001-0782. DOI: 10.1145/361082.361093. URL: https://doi.org/10.1145/361082.361093.

[29] Lee Howes and Aaftab Munshi. *The OpenCL Specification, Version 2.0*. Khronos Group. 2015.

[30] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. "A Formal Analysis of the NVIDIA PTX Memory Consistency Model". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 257–270. ISBN: 9781450362405. DOI: 10.1145/3297858.3304043. URL: https://doi.org/10.1145/3297858.3304043.

[31] Jeremy Manson, William Pugh, and Sarita V. Adve. "The Java Memory Model". In: *SIGPLAN Not.* 40.1 (Jan. 2005), pp. 378–391. ISSN: 0362-1340. DOI: 10.1145/1047659.1040336. URL: https://doi.org/10.1145/1047659.1040336.

[32] John M. Mellor-Crummey and Michael L. Scott. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors". In: *ACM Trans. Comput. Syst.* 9.1 (Feb. 1991), pp. 21–65. ISSN: 0734-2071. DOI: 10.1145/103727.103729. URL: https://doi.org/10.1145/103727.103729.

[33] Robert M. Metcalfe and David R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks". In: *Commun. ACM* 19.7 (July 1976), pp. 395–404. ISSN: 0001-0782. DOI: 10.1145/360248.360253. URL: https://doi.org/10.1145/360248.360253.

[34] ML Munford et al. "AFM in-situ characterization of supported phospholipid layers formed by vesicle fusion". In: *Microscopy and Microanalysis* 11.S03 (2005), pp. 90–93.

[35] Andrew Nere, Atif Hashmi, and Mikko Lipasti. "Profiling Heterogeneous Multi-GPU Systems to Accelerate Cortically Inspired Learning Algorithms". In: *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium*. IPDPS '11. USA: IEEE Computer Society, 2011, pp. 906–920. ISBN: 9780769543857. DOI: 10.1109/IPDPS.2011.88. URL: https://doi.org/10.1109/IPDPS.2011.88.

[36] NVIDIA. *CUDA programming guide*. http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf. 2016.

[37] NVIDIA. *CUDA programming guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. 2016.

[38] NVIDIA. *CUDA Stream Management*. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/group__CUDART__STREAM.html. 2018.

[39] NVIDIA. *Cuda-gdb*. http://docs.nvidia.com/cuda/cuda-gdb/index.html/. Accessed Nov 15, 2017. NVIDIA.

[40] NVIDIA. *Inside Volta*. https://devblogs.nvidia.com/parallelforall/inside-volta. May 2017.

[41] NVIDIA. *Pascal P100*. https://devblogs.nvidia.com/cooperative-groups/. 2016.

[42] NVIDIA. *Pascal P102*. https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf. 2016.

[43] NVIDIA. *Turing*. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. 2016.

[44] NVIDIA Corp. *Profiler User's Guide*. https://docs.nvidia.com/cuda/profiler-users-guide/index.html. 2018.

[45] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. "Chimera: Collaborative Preemption for Multitasking on a Shared GPU". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: Association for Computing Machinery, 2015, pp. 593–606. ISBN: 9781450328357. DOI: 10.1145/2694344.2694346. URL: https://doi.org/10.1145/2694344.2694346.

[46] Pascal-Network. *The PASCAL Visual Object Classes Challenge 2012*. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[47] Gary L. Peterson and Michael J. Fischer. "Economical Solutions for the Critical Section Problem in a Distributed System (Extended Abstract)". In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC '77. Boulder, Colorado, USA: Association for Computing Machinery, 1977, pp. 91–97. ISBN: 9781450374095. DOI: 10.1145/800105.803398. URL: https://doi.org/10.1145/800105.803398.

[48] G.L. Peterson. "Myths about the mutual exclusion problem". In: *Information Processing Letters* 12.3 (1981), pp. 115–116. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(81)90106-X. URL: http://www.sciencedirect.com/science/article/pii/002001908190106X.

[49] Sooraj Puthoor et al. "Oversubscribed Command Queues in GPUs". In: *Proceedings of the 11th Workshop on General Purpose GPUs*. GPGPU-11. Vienna, Austria: Association for Computing Machinery, 2018, pp. 50–60. ISBN: 9781450356473. DOI: 10.1145/3180270.3180271. URL: https://doi.org/10.1145/3180270.3180271.

[50] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271.

[51] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018).

[52] I. Rhee. "Optimizing a FIFO, scalable spin lock using consistent memory". In: *17th IEEE Real-Time Systems Symposium*. 1996, pp. 106–114.

[53] M. A. Rivas and M. G. Harbour. "Evaluation of new POSIX real-time operating systems services for small embedded platforms". In: *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings*. 2003, pp. 161–168.

[54] Larry Rudolph and Zary Segall. "Dynamic Decentralized Cache Schemes for Mimd Parallel Processors". In: *SIGARCH Comput. Archit. News* 12.3 (Jan. 1984), pp. 340–347. ISSN: 0163-5964. DOI: 10.1145/773453.808203. URL: https://doi.org/10.1145/773453.808203.

[55] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. "Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA. Toronto, ON, Canada: ACM, 2017, pp. 161–174. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080206. URL: http://doi.acm.org/10.1145/3079856.3080206.

[56] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. "Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models". In: *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. Dec. 2015, pp. 647–659.

[57] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. "HeteroSync: A Benchmark Suite for Fine-Grained Synchronization on Tightly Coupled GPUs". In: *IEEE International Symposium on Workload Characterization*. IISWC. Oct. 2017.

[58] Inderpreet Singh et al. "Cache Coherence for GPU Architectures". In: *19th International Symposium on High Performance Computer Architecture*. HPCA. 2013, pp. 578–590. DOI: http://doi.ieeecomputersociety.org/10.1109/HPCA.2013.6522351.

[59] Dong Oh Son et al. "A Dynamic CTA Scheduling Scheme for Massive Parallel Computing". In: *Cluster Computing* 20.1 (Mar. 2017), pp. 781–787. ISSN: 1386-7857. DOI: 10.1007/s10586-017-0768-9. URL: https://doi.org/10.1007/s10586-017-0768-9.

[60] Tyler Sorensen et al. "Portable Inter-Workgroup Barrier Synchronisation for GPUs". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA. 2016, pp. 39–58.

[61] Jeff A. Stuart and John D. Owens. "Efficient Synchronization Primitives for GPUs". In: *CoRR* abs/1110.4623 (2011). arXiv: 1110.4623. URL: http://arxiv.org/abs/1110.4623.

[62] H. Takada and K. Sakamura. "Predictable spin lock algorithms with preemption". In: *Proceedings of 11th IEEE Workshop on Real-Time Operating Systems and Software*. 1994, pp. 2–6.

[63] Peiyi Tang and Pen-Chung Yew. "Software combining algorithms for distributing hot-spot addressing". In: *Journal of Parallel and Distributed Computing* 10.2 (1990), pp. 130–139. ISSN: 0743-7315. DOI: https://doi.org/10.1016/0743-7315(90)90022-H. URL: http://www.sciencedirect.com/science/article/pii/074373159090022H.

[64] Seiya Tokui et al. "Chainer: a next-generation open source framework for deep learning". In: *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*. Vol. 5. 2015, pp. 1–6.

[65] Y. Ukidave, X. Li, and D. Kaeli. "Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning". In: *IEEE International Parallel and Distributed Processing Symposium*. IPDPS. 2016, pp. 353–362.

[66] Oreste Villa et al. "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 372–383. ISBN: 9781450369381. DOI: 10.1145/3352460.3358307. URL: https://doi.org/10.1145/3352460.3358307.

[67] Zhenning Wang et al. "Quality of Service Support for Fine-Grained Sharing on GPUs". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 269–281. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080203. URL: http://doi.acm.org/10.1145/3079856.3080203.

[68] Z. Wang et al. “Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing”. In: *IEEE International Symposium on High Performance Computer Architecture*. HPCA. 2016, pp. 358–369.

[69] Bo Wu et al. “FLEP: Enabling Flexible and Efficient Preemption on GPUs”. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’17. Xi’an, China: ACM, 2017, pp. 483–496. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037742. URL: http://doi.acm.org/10.1145/3037697.3037742.

[70] Ping Xiang, Yi Yang, and Huiyang Zhou. “Warp-level divergence in GPUs: Characterization, impact, and mitigation”. In: *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*. IEEE Computer Society, 2014, pp. 284–295. DOI: 10.1109/HPCA.2014.6835939. URL: https://doi.org/10.1109/HPCA.2014.6835939.

[71] Shucai Xiao and Wu Feng. “Inter-Block GPU Communication via Fast Barrier Synchronization”. In: *IEEE International Parallel and Distributed Processing Symposium*. IPDPS. May 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470477.

[72] Q. Xu et al. “Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming”. In: *ACM/IEEE 43rd Annual International Symposium on Computer Architecture*. ISCA. 2016, pp. 230–242.

[73] Feiwen Zhu et al. “Sparse Persistent RNNs: Squeezing Large Recurrent Networks On-Chip”. In: *Proceedings of 6th International Conference on Learning Representations*. ICLR. 2018.