Dynamic Scheduling on Warp-Contention Heavy Workloads in GPU Architectures

Tanuj Nayak
Carnegie Mellon University
tnayak@andrew.cmu.edu

Preetansh Goyal
Carnegie Mellon University
preetang@andrew.cmu.edu

Gautam Jain
Carnegie Mellon University
gautamj@andrew.cmu.edu

Abstract

With an increased focus on General Purpose GPU programming, there is a growing interest in delivering performance on applications on a GPU that have traditionally done well on CPUs. Nvidia has recently introduced support for fine grained synchronization by including atomic instructions in CUDA[1]. Profiling these applications has shown that in cases of high contention, warps busy wait and cause underutilization of GPU resources. Our project aims to provide a technique, implemented in the GPGPUSim simulator, that detects warp-spinning during the execution of PTX instructions and adds logic to the scheduler to back off such detected warps. Our technique is able to get a reduction of 25-50% in total cycles in our hand-made contention-heavy benchmarks and also get a speedup of 10-25% in the transactions benchmark [2].

1 Introduction

1.1 Background

General Purpose Graphical Processing Units (GPGPU's) have come up in the past decade as an exciting source of computation that accelerates massively parallellizable workloads on large amounts of data. These architectures rely on executing programs in a manner that's known as Single Instruction Multiple Thread (SIMT) where a group of threads are made to run the same set of instructions synchronously in their own separate thread contexts. This model works really well to execute multiple instances of a single piece of straight line code over multiple independent shards of data. For Nvidia current GPU's, 32 such threads are executed in lock-step and are grouped into what are called a warp. AMD executes 64 threads (workitems) in lockstep and they call them a wavefront. Branching of code and irregular control flow becomes a deterrent to the SIMT model as the behaviour of threads in a warp starts to vary. This problem is solved by maintaining an *active mask* throughout the execution which represents the threads that will execute the next instruction. A SIMT stack is also maintained which allows the threads to reconverge after executing a branch instruction. However, this approach causes underutilization of the GPU resources and is called *branch divergence*. Therefore, GPU's SIMT model is not well-suited for Multiple Instruction Multiple Thread (MIMT) workloads. There has been a lot of work that looks to see how we can extract the power of GPU's better for MIMT workloads. Amongst this research, dealing with shared spin locks between warps has also gained prominence. There currently isn't much support for workloads that need to synchronize amongst threads. For example, if we have a kernel that inserts multiple values into a bucketed hash table, multiple threads inserting into the same bucket will be contending for a lock bit and a few of them will potentially be spinning. This results in sporadic branch divergence within a warp amongst threads in and not in the critical section as depicted in figure 1. It also results in many warps that get scheduled, just to spin on a lock and busy wait in place of other warps that can proceed with useful work or actually be holding the lock.

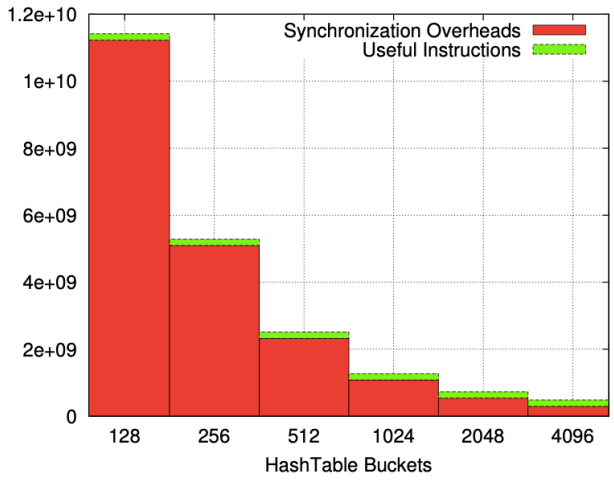

Figure 1. Contention in hashtable with decreasing number of buckets [3]

1.2 General Approach

In CPU architectures, the problem of thread contention is usually dealt with either at the software or hardware level by imposing a back-off in scheduling of a particular thread when it has unsuccessfully spun on a lock a few times. This project seeks to try to extend that same idea to GPU warps and see how well that works on MIMT workloads in a way that is friendly to GPU programmers. GPUs have been very successful in hiding memory latencies by scheduling out warps that issue longer memory accesses and replacing them with warps that can execute useful work in parallel. We believe that the latency due to busy waiting can also be hidden by

the same logic. We aim to achieve this by having the hardware detect repeated patterns in execution to derive whether or not the current thread is spinning on a lock and adjust the scheduler accordingly. The goal is to do this in a manner completely transparent to the GPU programmer.

We take an approach based on El-Tantawey *et al*'s [3] that dynamically detects warps that are not doing any useful work because they are spinning on a branch while waiting for a lock and prevents them from being scheduled for a fixed quantum of cycles. We perform a greedy method that backs-off (prevents scheduling) a warp at a set predicate instruction once it finds sufficient repetition in a fixed size history of executed setp instruction and operands to such instructions. Note that these set predicate PTX instructions are usually the equivalent of a loop guard checker.

1.3 Related Work

There have been a few works related to the effort of bringing MIMT workloads to SIMT architectures. One of them is Criticality Aware Warp Acceleration (CAWA) brought forth by Lee *et al.* [4]. This work intends to optimize a superclass of spinning overheads that's more than just spinning on locks and also includes taking memory loads and pipeline hazards into account. The authors make a system that dynamically detects the "criticality" of a warp within a thread block via profiling and collecting various hardware metrics such as stalls. It devises a greedy scheduler that prioritizes such critical warps. This worked to beat greedy-than-oldest scheduling by 17% on average on workloads that included breadth-first search and B+ tree searches. This criticality metric, however, would deem warps that spin for a long time in our context to be critical and would prioritize them.

Another work that's related to this is Hardware Queue Locking (HQL), which was introduced by Yilmazer and Kaeli [6]. This paper took a different perspective on making atomic usage in GPU's more efficient by making newer atomic primitives within GPU's. These atomic primitives provide locks at a cache line granularity that queues up requests from the same compute unit. This enables easy passing of the lock between threads using the same compute unit. This new API also allows the authors to make this a blocking synchronization primitive. Whenever a thread acquires a lock, the primitive sets a scheduler state to block the current thread. They develop contention-intensive microbenchmarks on their own that include a hash table to show that this technique reduces the number of instructions executed by a GPU by up to 84% when compared against the traditional GPU atomic primitives.

El-Tantawy *et. Al* have worked on the general problem of migrating MIMT workloads to SIMT architectures. In their paper [3], they devised an adjustment to warp schedulers that dynamically detects warps that are spinning on a lock and deprioritizes them in the warp scheduler for a GPU SM unit accordingly. The paper's main contributions were the spinning detection method called Dynamic Detection of Spinning (DDOS) and the Backoff Warp Scheduler (BoWS) that went alongside it. DDOS operates by keeping a history of program counter values within a warp to trace whether it is in a loop or not. After that, it must detect whether or not such a loop is the one that is making progress. It achieves this by examining the operands to the set predicate instruction that take place before each iteration of a loop in order to see which threads within a warp should have their active mask values adjusted for the next iteration of the loop. If a thread is not making any progress on a loop, then it is expected that the operands to the set predicate instruction will not change. This is much like how in a loop that does not make progress, the operands to the loop guard comparison instruction are not liable to change. By keeping a history of setp instruction operands and program counter values, most spinning loops can be detected in the hardware. If sufficient repetition is determined in these history registers of all threads in a warp, the state of the the warp is determined to be "spinning". The execution paths of warps that are spinning are profiled and used to update an SM-wide statistics data structure called a "prediction table". The state of the warp and this prediction table are used in conjunction to feed the scheduler information about whether a given warp is spinning on a non-progressing branch and should be backed off by a fixed amount of cycles or not. This was determined on contention-heavy benchmarks to improve performance by 1.5x over CAWA.

2 Approach/Design

Our approach's main component is one that makes GPU hardware infrastructure to detect that a warp only has threads stuck in a loop that is spinning aimlessly. This operates by only keeping track of `setp` PTX instructions that are executed. Within a warp, we maintain a fixed-size recent *history table* of `setp` instructions that were executed along with their operands. Each row in the table corresponds to a specific program counter that would be a setp instruction and the columns for this row encode a history of operands to the setp instruction for the row. When a new entry is added to this table that causes sufficient repetition amongst the columns in a row of the table, we consider the current thread to be spinning for that instruction. If all active threads within a warp for one program counter are deemed to be spinning then the entire warp is deemed to be spinning and we set a backoff amount on the scheduling state for the warp. Figure 2 shows a sample history table for a thread. The thread has two `setp` instruction PCs in the history and window of operands (storing 5 sets of operands) representing the history. For the `setp` instruction at PC=272, since all the values in the window are same, it can be used to determine spinning at that instruction.

Program Counter	Operand History Window				
184	(0000 , 0000)	(0000 , 0000)	(0000 , 0001)		
272	(0000 , 0001)	(0000 , 0001)	(0000 , 0001)	(0000 , 0001)	(0000 , 0001)

Figure 2. History table

We implemented the backing off by creating a separate data structure that holds the positive backoff amount set for backed-off warps. For the sake of simplicity, we decremented the backoff amount for each warp in this queue every cycle. This can be made more efficient by representing these warps via a min-heap where each warp is keyed on the cycle count at which they are expected to finish their backoff period. This is still in line with our goal to see to what extent our method decreases the overall cycle count of running contention heavy programs on a SIMT architecture. Any warp that is in this backed off queue is not allowed to be scheduled until it completes its backoff period (or if all warps are backed-off).

This concludes our main greedy design for dynamic detection and warp back off on contention. We wanted to compare the performance of this greedy model with what El-Tantawey's work on DDoS and BoWS to get a better insight into where their performance gains come from. We also implemented the prediction table described in [3] and compared the performance of our system versus theirs. The prediction table contains rows, each of which corresponds to a backward branch. Each row also contains a "confidence" column for the concerned backward branch of that row. If a warp in a "spinning state" is deemed to take a backward branch at a program counter x, it increases the confidence value of that row by 1. If a warp in a non-spinning state is deemed to take a backward branch that's listed in the prediction table, the confidence value is decremented by 1. Once we have that that a particular entry in the table has a confidence value over a particular threshold, the branch that corresponds to that entry is deemed to be *spin inducing*. Now each time any other spinning warp to take such a spin inducing branch, the warp is backed off immediately.

This is different from our approach in that our greedier approach doesn't maintain such explicit confidence values. One can say that our history table operand values for a particular row encode an implicit form of a confidence. Furthermore, we are setting the point of decision of backing off at the setp instruction as opposed to when a backward branch is taken. Further, El-Tantawy et. Al[3] only profiled a single thread of a warp calling it the frontier thread. The results of the thread were sent to the prediction table which is common to all the warps running on a single SM (Streaming Multiprocessor). Our hypothesis is that the greedy approach that only tracks setp instructions over all the threads in the warps can actually provide better results on benchmarks with very high amount of contention. By profiling all threads, we can eliminate confidence checking over an SM and also avoid a warp being backed off if one of it's threads was actually able to get a lock. We investigate this hypothesis by comparing our greedy approach with the prediction table technique while keeping a normal scheduler without back-off as the baseline. We implemented their prediction table technique alongside our greedy approach for comparison.

Listing 1. Global Increment Benchmark

```
1     compute threadId
2     compute id = f(threadId)
3     bool done = false
4     while(!done) {
5         if(atomicCAS(&lock[id], 0, 1) == 0) {
6             array[id] += 1;
7             done = true;
8             atomicExch(&lock[id], 0);
9         }
10    }
```

Listing 2. Global Increment Benchmark PTX

```
1  BB0_2:
2  mov.u32 %r9, 1;
3  mov.u32 %r10, 0;
4  atom.global.cas.b32 %r11, [%rd2], %r10, %r9;
5  setp.ne.s32      %p2, %r11, 0; // PC-184
6  @%p2 bra BB0_4;
7  membar.gl;
8  mov.u16 %rs3, 1;
9  cvt.u8.u16       %rs6, %rs3;
10 ld.global.u32 %r12, [%rd3];
11 add.s32 %r13, %r12, 1; // PC-232
12 st.global.u32 [%rd3], %r13;
13 membar.gl;
14 atom.global.exch.b32 %r14, [%rd2], 0; // PC-256
15
16 BB0_4:
17 cvt.u16.u8       %rs4, %rs6;
18 setp.eq.s16      %p3, %rs4, 0; // PC-272
19
20 BB0_5:
21 cvt.u8.u16       %rs6, %rs1;
22 ret;
```

3 Experimental Setup

GPGPU-Sim [5] is a cycle level simulator modeling contemporary graphics processing units (GPUs) running GPU computing workloads written in CUDA. GPGPU-Sim models each SM as a shader object. A shader can have multiple schedulers (representing parallelism within a SM) that each have a fixed number of warp slots to execute. Based on the program's requirement they populate these slots with warps and schedule them based on a GTO (Greedy Than Oldest)

or LRR (Loose Round Robin) technique. Each scheduler simulates a cycle by searching the scheduled list of warps for a warp with a ready instruction to issue. We conducted all our experiments using GPGPU-Sim and added our backoff code in these schedulers. Our primary goal was to achieve a reduction in the number of cycles that need to be simulated for a program to finish execution.

We created a development benchmark (Global Increment Benchmark) in which each thread is responsible for adding a value to a global array by taking the lock for the position it is going to perform the append on. We chose this benchmark for a variety of reasons:

- It simulates a real world operation that will be commonly used by applications that require fine grained synchronization i.e manipulating a value in a global array.
- It is easy to control the amount of contention in such a program by varying the contention for a lock within a warp or amongst warps, therefore making it extremely suitable for analysis.
- It is easy to read the PTX for this program and allows debugging for the history table and the prediction table. As given in (2) the PTX shows that there are two setp instructions that can be tracked in our history table and can be checked for spinning.

We created a second benchmark (HashTable Benchmark) that would randomize contention amongst threads. This is based on the well known hashtable data structure. Our program maintains some number of buckets and each thread is responsible for picking up random numbers to insert in the hashtable based on the hash value of the number to insert. The thread will take the lock for the bucket it needs to insert in and will spin until it gets the lock.

The final benchmark is the Transactions Benchmark[2]. It simulates a number of transactions where an amount is taken from a source account to a destination account. The transactions are created by setting random numbers amongst a specified upper limit of accounts for the source and destination. Each thread is responsible for a small number of transactions to execute and it takes the lock for both the source account and the destination account and spins until it gets both the locks.

All the experiments were completed on GPGPU-Sim [5] with our implemented code.

4 Results/Analysis

The first results we show are for the Global Increment Benchmark (GIB) run on the SM2_GTX480 configuration provided by GPGPU-Sim. GTX480 consists of 15 SM's with 2 schedulers each. We reduce the benchmark to just work on 1 SM. Each scheduler has slots for 24 warps and we fill up all the slots by controlling the number of threads spawned by the

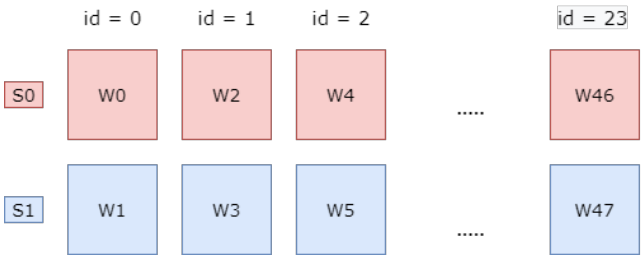

Figure 3. Array index mapping for Listing 1. The red warps belong to scheduler 0 and blue warps belong to scheduler 1

CUDA kernel. The index mapping is shown in Figure 3 . There is intra warp thread contention also but it boils down to taking the lock 32 times for a warp, therefore we ensure there is enough work for a warp to do. Moreover, there is no contention between warps in a single scheduler therefore there is enough useful work to do if one warp starts busy waiting. The results are shown in Figure 4.

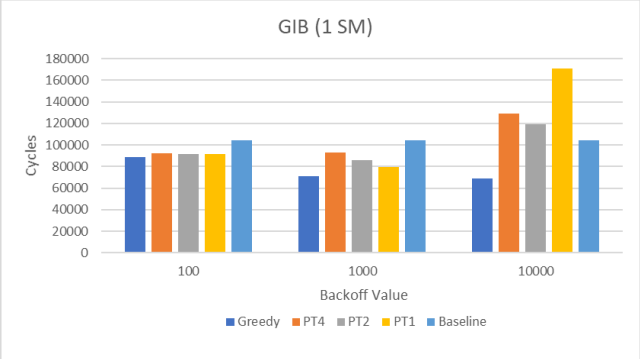

Figure 4. Global Increment Benchmark Results for 1 SM

From the baseline we can get a cycle reduction of upto 33%. If we expand the SM's to 15 to activate the entire GTX480, we can get a reduction of upto 50%. The prediction table in comparison only gets a reduction of upto 24%. The array increment is a memory operation and it takes a large number of cycles as it fetches the variable from the global shared memory taking up many cycles. A small backoff is not able to hide the latency of this memory operation and subsequently the busy waiting. A warp which is busy waiting is backed off for cycles that are very small as compared to the memory access. But once we start increasing the backoff to thousands of cycles we start having an effect on the total cycles. Now the warps that are spinning are backed off and we always allow other warps that can take the lock and execute the increment to begin working. The backoff allows for maximum utilization possible in the program. Another important observation is that the greedy backoff technique does better than the prediction table technique with varying confidence value (PT4 has confidence value 4). This is in

accordance with our hypothesis that for high contention a greedy technique is better than using a prediction table. This is because a prediction table will take many more cycles to flag a warp to be backed off. With a lower confidence value of 2 the prediction table works better than with a value of 4. This is because the speculative cycles are lower in 2 than 4.

The hashtable benchmark has no such clear mapping of locks acquired by a thread and there is no direct control for contention. Since the buckets are decided randomly, it's likely that one of the threads in a warp gets the lock. Therefore we classify this benchmark as a low contention application. As we can see from Figure 5, the results for greedy and prediction table are similar to the baseline and in some small values of backoff slightly better where the prediction tables don't incur a high enough confidence value to activate backoff. We can get a 4.3% reduction in cycles in the hashtable benchmark in the greedy approach while the prediction table gets 10.7%.

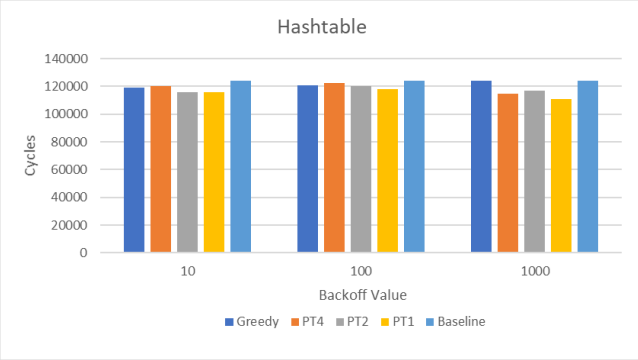

Figure 5. Hashtable Benchmark Results for TITAN V

Now we move to a mid to high contention benchmark of transactions. As described earlier, there is a global set of transactions created on the host CPU and each thread gets a disjoint subset of the transactions to perform. Each transaction is a withdrawal from a source account and a deposit into a destination account. The number of accounts controls the amount of contention possible in the benchmark. We conduct experiments with 23k threads (fully activate GTX480) and a total of 122k transactions. We test the prediction table with 2k, 4k and 8k accounts in order to have high, mid and low contention respectively. Results are shown in Figures 6, 7, 8.

As given by the results 8k has almost no contention and the baseline, prediction table and greedy perform similarly. In 4k and 2k accounts, we start noticing contention as the number of cycles for baseline start increasing, the increase in cycles is not as high for greedy and prediction table. This implies that we are hiding the extra latency due to contention by backing off. Again we see that for mid level contention the greedy technique performs as well as the prediction table and sometimes better. We can get a 27.4% reduction in cycles in the transactions benchmark (2k accounts) in the greedy approach while the prediction table gets 26.3%.

5 Surprises and Lessons Learned

We had spent a substantial amount of time setting up the GPGPU-Sim simulator framework for development. Out initial bet was on the pre-configured VM image provided on the website but after multiple hours spent to update the VM's old version and to subsequently use that for our benchmarks, we decided to drop this approach. Then we used AWS instances and GHC machines to setup GPGPU-Sim. To use the GHC machines, we had to perform considerable work to work around the case where GPGPU did not support CUDA 10.2 (the version in GHC machines).

GPGPU-Sim architecture is based on a shader class which was primarily a graphics term but means a streaming multiprocessor for all intents and purposes. The issuing logic is very simple, although seems a bit wasteful as they iterate over all the warps in the prioritized list and choose one which is ready and capable of issuing a valid instruction. GTO (Greedy than oldest) in theory executes the warp which last issued a instruction first in the next cycle. This property though does not hold when a long memory access is being conducted or there is a stall. This gave us some weird results when we tried GIB on eight warps. The opportunity for backing off and swapping in a useful warp was minimal.

6 Conclusions and Future Work

From the results, we conclude that our greedy approach can achieve similar results to the prediction table technique in mid level or low level contention benchmarks. On the other hand, we are able to get a 10% more reduction with high contention benchmarks. The gains are from the implicit confidence values in the history window checked over all the threads therefore removing the need for a prediction table. We therefore save on an extra data structure that needs to be maintained by the scheduler for each SM and needs to be updated by each warp. Moreover, the backoff is applied much more quickly as the confidence value takes a lot more speculative cycles for reaching the appropriate confidence value. This is also evidenced by the better results shown by PT2 over PT4 consistently in all the benchmarks as it takes up lesser speculative cycles.

For the future work, there can be a notion of useful cycles for instructions executed in the backoff factor to calculate it adaptively. We used a fixed backoff factor when a warp was spinning. Finding the average cycles taken by instructions to complete can help to eliminate the overheads due to a shorter or longer backoff factor.

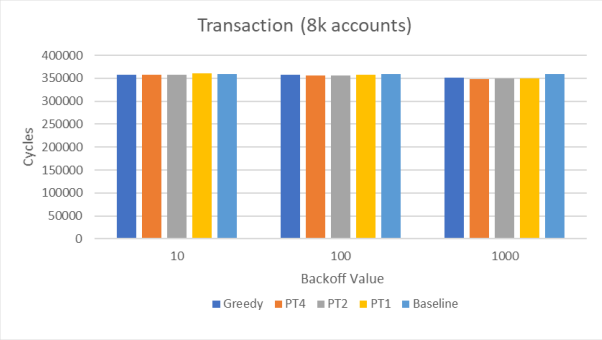

Figure 6. Transaction benchmark for 8k accounts

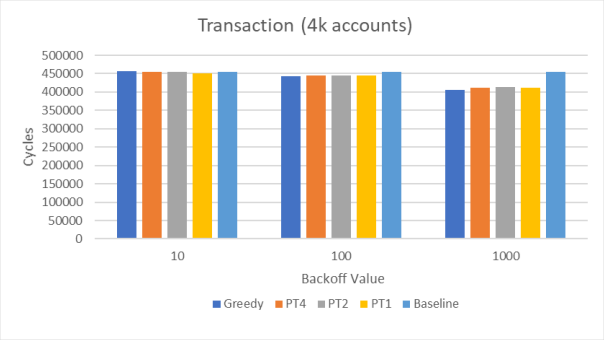

Figure 7. Transaction benchmark for 4k accounts

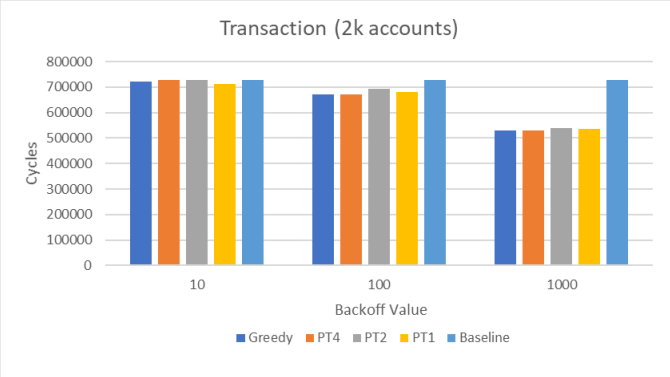

Figure 8. Transaction benchmark for 2k accounts

References

[1] Nvidia Corporation. [n.d.]. Atomic Instructions. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

[2] ElTantawy. 2016. Transaction Benchmark. https://github.com/ElTantawy/mimd_to_simt/tree/master/testCases

[3] Ahmed ElTantawy and Tor Aamodt. 2018. Warp Scheduling for Fine-Grained Synchronization. 375–388. https://doi.org/10.1109/HPCA.2018.00040

[4] Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. 2015. CAWA: coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads. *ACM SIGARCH Computer Architecture News* 43, 515–527. https://doi.org/10.1145/2872887.2750418

[5] Tor M. Aamodt Timothy G Rogers Mahmoud Khairy, Zhesheng Shen. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *47th IEEE/ACM International Symposium on Computer Architecture (ISCA)*. IEEE/ACM.

[6] A. Yilmazer and D. Kaeli. 2013. HQL: A Scalable Synchronization Mechanism for GPUs. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 475–486. https://doi.org/10.1109/IPDPS.2013.82