Independent Forward Progress of Work-groups

Alexandru Duțu*, Matthew D. Sinclair*†, Bradford M. Beckmann*, David A. Wood*†, Marcus Chow*‡

*AMD Research, †University of Wisconsin – Madison, ‡University of California – Riverside

***Abstract*—GPUs have evolved from providing highly-constrained programmability for a single kernel to using pre-emption to ensure independent forward progress for concurrently executing kernels. However, modern GPUs do not ensure independent forward progress for kernels that use fine-grain synchronization to coordinate inter-work-group execution. Enabling independent forward progress among work-groups (WGs) is challenging as pre-empted kernels may be rescheduled with fewer hardware resources. This can lead to oversubscribed execution scenarios that deadlock current hardware even for correctly written code. Prior work addresses this problem by requiring programmers to specify resource requirements and assuming static resource allocation, which adds scheduling constraints and reduces portability.**

We propose a family of novel hardware approaches — trading off hardware complexity for performance — that provide independent forward progress in the presence of fine-grain inter-WG synchronization and dynamic resource allocation. Additionally, we propose new waiting atomic instructions compatible with proposed C++20 extensions. Our final design, Autonomous Work-Groups (AWG), uses hints from regular and waiting atomics to cooperatively schedule WGs within a kernel, improving efficiency and virtualizing hardware resources. In non-oversubscribed scenarios, AWG outperforms a busy-waiting baseline (which deadlocks in oversubscribed scenarios) by 12x on average for benchmarks that use different mutexes and barriers for fine-grained, WG granularity synchronization. Furthermore, AWG outperforms other solutions that do not deadlock in the oversubscribed case, such as fixed-interval round-robin context switching or naively extending monitor/mwait to GPUs, by 2.6x and 2.2x, respectively.

I. INTRODUCTION

Graphics Processing Units (GPUs) are massively throughput-oriented processors with a hierarchy of execution abstractions that provide high performance acceleration for applications in a variety of fields: from high-performance computing [1] to machine learning [2][3]. GPUs continue to scale to larger sizes with improvements in assembly and manufacturing technology. At the highest level of a GPU's execution hierarchy, kernels are executed from memory-backed queues, and below that, work-groups (WGs), wavefronts, and work-items (WIs) make up the lower levels. Modern GPUs support the simultaneous execution of multiple different types of kernels, from different queues belonging to the same application or even different applications.

In order to manage the simultaneous execution of these diverse kernels, GPUs allow higher priority kernels to pre-empt lower priority kernels [4][5][6], without ensuring the same resources will be available to individual kernels upon resumption. The correctness of many GPU workloads, especially traditional streaming, data parallel workloads, is unaffected by a potential loss of resources. However, for emerging workloads that perform synchronization within a kernel [2][7][8], kernel-level pre-emptive scheduling can cause inter-WG deadlock – *even when the programmer has written correct code*. For instance, Sorensen *et al.* showed that simple global barrier synchronization results in deadlock on current GPUs when the number of WGs within a kernel oversubscribe the available resources [9].

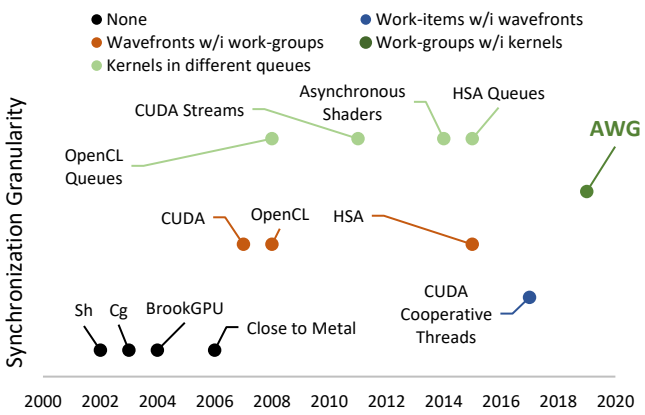

Figure 1: A timeline of independent forward progress for GPUs, at different synchronization granularities.

More broadly, Independent Forward Progress (IFP) rules have been a predominant issue ever since software developers started to write general-purpose GPU (GPGPU) kernels. However, thus far, prior work has not addressed or only partially addressed the IFP of WGs within a kernel. Figure 1 summarizes the appearance of IFP rules for a variety of GPGPU application programming interfaces (APIs). Initial APIs such as Sh [10], Cg [11], or BrookGPU [12] completely ignored IFP rules before CUDA and OpenCL [13][14] defined IFP for wavefronts within a WG. Later OpenCL, CUDA, and HSA [15][16] defined IFP at the kernel granularity by allowing kernels from different queues to synchronize with each other.

More recently, programming models have been addressing forward progress rules at the previously neglected intermediate levels. For example, Sorensen *et al.* [9] avoided inter-WG deadlocks by developing a software method that dynamically discovers resource availability. However, their mechanism cannot adjust to mid-execution resource reductions, as depicted in Figure 2, and has not yet been adopted by popular programming models. A new execution abstraction, cooperative groups, has been introduced as an alternative to avoid mid-execution resource losses [17]. Cooperative

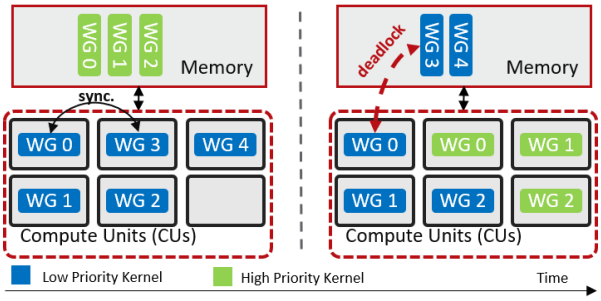

Figure 2: Inter-WG synchronization under dynamic allocation of resources.

groups provide IFP for WIs within a wavefront and a restrictive amount of IFP for WGs within a kernel. However, cooperative groups require programmers to explicitly define how many WIs may synchronize and ensure that kernels do not oversubscribe GPU resources. Furthermore, cooperative group kernel dispatches must wait until enough resources become available (i.e., no dynamic allocation of resources). Thus, when a cooperative group encompasses many WGs, the associated kernel launch can encounter significant scheduling delay, making them less attractive for synchronization across multiple WGs.

In this work we propose Autonomous Work-Groups (*AWG*), a family of alternative hardware-centric approaches that provide IFP for an arbitrary number of WGs executing in dynamic resource environments. AWG introduces *waiting atomic* instructions for efficient GPU synchronization and scheduling. Waiting atomics are atomic instructions with an extra operand that indicates the expected value of the synch variable for the atomic to succeed. If the atomic fails, the synchronization variable and the expected value form a *condition* for a WG to *wait* on before being resumed. These waiting atomics effectively provide hardware support for a recent C++20 extension proposal [18]. AWG adds specialized hardware to the GPU last level cache (LLC) to efficiently monitor *waiting conditions* and relies on virtual memory when its internal hardware structures reach capacity. Thus, AWG effectively virtualizes the GPU execution resources such that inter-WG IFP within a kernel is ensured.

We evaluate AWG across a wide variety of fine-grained synchronization benchmarks. Our results show that AWG significantly outperforms existing solutions both when the GPU is oversubscribed (i.e., kernel's WGs exceed available resources) and non-oversubscribed. In a non-oversubscribed scenario, AWG is 12x faster than a busy-*waiting* baseline for applications that utilize one synchronization variable for an entire WG. When the GPU is oversubscribed, compared with a simple fixed interval timeout mechanism, AWG is 2.6x faster on average. Furthermore, AWG outperforms hardware synchronization approaches similar to *monitor/mwait* by 2.2x on average.

In summary, AWG addresses IFP of WGs within a kernel and provides the following contributions:

- **Portability**. We extend existing GPU execution abstractions to provide inter-WG IFP, improving portability across different amounts of resources from GPUs of different sizes.
- **Kernel Scheduling.** We relax constraints on kernel scheduling by assuming dynamic resource allocation. This opens the door to new GPU scheduling opportunities for low latency kernels.
- **WG Scheduling.** We evaluate different WG scheduling policies and converge on using atomic operations to support cooperative WG scheduling.
- **Virtualization.** We virtualize the number of synchronization variables, waiting values, and waiting WGs allowed in inter-WG synchronization by extending the firmware of the programmable micro-controller already available in current GPUs.

II. Synchronization And Scheduling

Multicore CPUs use atomic instructions to modify and poll synchronization variables and rely on the pre-emptive OS scheduler to ensure IFP. Waiting CPU threads can release resources using the *yield* system call, which has high overhead, or busy-*wait* on the synchronization variable, which can be inefficient [19][20][21]. To address the latter, current x86 CPUs provide more efficient inter-thread synchronization through special *monitor* and *mwait* [22] instructions. The monitor instruction specifies an address range for hardware monitoring. The mwait instruction causes the processor to wait, in an implementation specific power state, until a different processor writes the monitored address, or an unmasked interrupt, reset, or far control transfer in between monitor and mwait occurs. CPUs support mwait by forwarding write invalidations to all sharers. Because mwait can return before the condition has been met, mwait provides relaxed support for Mesa semantics [23][24][25][26][27]. Mesa semantics allow the producer thread to continue execution after synchronization notifications and defines these notifications as *hints*. As a result, the consumer cannot assume its waiting condition holds true when it resumes execution and must recheck its waiting condition.

Synchronization and IFP are more complex on GPUs due to their massive parallelism and hierarchical execution model. The programmer specifies the number of WIs within a grid and the number of WIs within a WG. The grid is then split into WGs, which are comprised of one or more wavefronts. Wavefronts are the set of WIs executed on the GPU's SIMD resources in lock-step fashion. Figure 3a) depicts the GPU architecture. Due to their hierarchical execution model, GPUs include multiple levels of synchronization and forward progress rules associated with each level. Figure 3 depicts synchronization of: WIs within a WF (Figure 3b), WFs within a WG (Figure 3c), kernels in different queues (Figure 3d), and WGs within a kernel (Figure 3e). The following

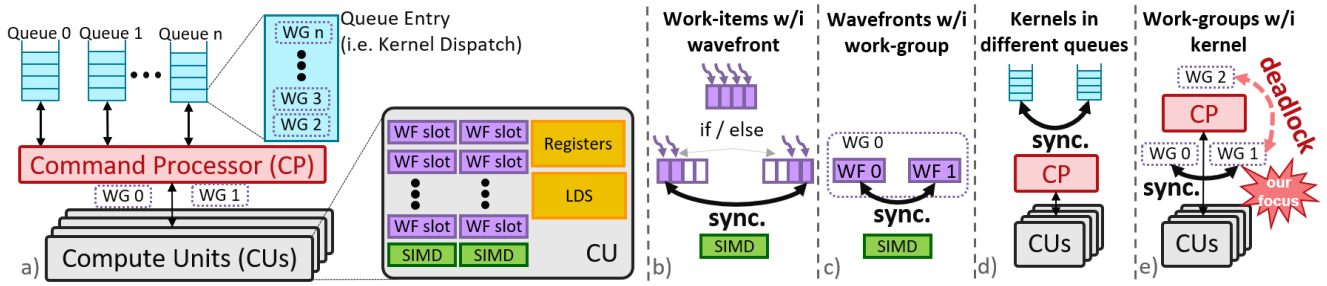

Figure 3: GPU Synchronization and Scheduling.

subsections detail these rules and prior efforts to improve them.

A. *Intra-Wavefront Synchronization*

Recent GPUs have introduced new synchronization capabilities within a wavefront to allow WIs to synchronize even in case of divergent control flow [17]. However, programmers may need to inform the compiler/hardware of inter-WI reconvergence points – using a new synchronization function called__*syncwarp()* – in order to achieve good performance. This new functionality allows WI-level IFP in the case of intra-wavefront synchronization (Figure 3b), and therefore is orthogonal to inter-WG synchronization.

B. *Wavefronts within WG Synchronization*

GPUs use thread-level parallelism to tolerate long latency operations and need barrier instructions when synchronizing wavefronts, as shown in Figure 3c. GPUs assign all wavefronts within a WG to a single compute unit (CU) and local barriers are commonly used when data is shared through the CU's scratchpad memory. Within a WG, wavefront IFP is commonly provided by scheduling wavefronts in a round-robin fashion, allowing each wavefront fair access to execution resources [13][14]. Recent work has improved upon round-robin scheduling, without compromising inter-wavefront IFP, by performing synchronization conscious scheduling of wavefronts [28] or adding compiler support [29].

C. *Inter-Kernel Synchronization*

Figure 3d illustrates that GPUs provide IFP for synchronizing kernels by context switching all the resident WGs of kernels originating from different queues [13][14][15][30][31][32] [33]. Pre-emptive kernel scheduling completely relinquishes all kernel-allocated hardware resources and is commonly used to execute high priority jobs with real-time deadlines, such as a compute kernel, whose output is consumed by a graphics kernel, pre-empting background compute kernels [4]. Kernel pre-emption provides inter-kernel IFP in a similar fashion to OS-managed CPU threads.

Prior work has also explored improving multi-tasking on GPUs [34][35][36][37][38][39][40][41]. For example, KLAP [36] uses kernel aggregation and kernel promotion in context of irregular applications that use CUDA Dynamic Parallelism. Kernel promotion amortizes kernel launch latency by overlapping dependent kernels. Broadly, prior work utilizes a variety of techniques, including kernel aggregation, lightweight task spawning systems, compiler support, persistent threads, idempotence, and WG context switching to better utilize GPU resources and avoid inter-kernel synchronization when possible. Although these approaches can significantly improve performance by amortizing kernel launch latency, they do not attempt to address inter-WG IFP.

D. *Limited Inter-WG Synchronization*

Historically, GPUs have benefited from limited WG-level scheduling. Figure 3e shows how WGs within a kernel are sequentially dispatched until execution resources (i.e., functional units and registers) and memory resources (i.e., scratchpad) are saturated. Additional younger WGs are dispatched when execution resources become available, limiting inter-WG synchronization to just the currently executing WGs.

Kernels that over-subscribe available GPU resources can deadlock when using inter-WG synchronization. Consider a program written as follows. A shared variable is expected to be updated by a producer WG that is not yet scheduled for execution. An older consumer WG, which is already scheduled for execution (i.e., resident), is waiting for the producer to update the shared variable. However, the consumer WG will not be able to make forward progress and release its resources until the waiting condition is satisfied and the waiting condition will not be satisfied if the producing WG cannot be scheduled for execution, because of lack of available resources. Additionally, when pre-empted kernels are rescheduled for execution, the scheduler may not provide the same execution resources as before, resulting in over-subscription.

The AMD GCN ISA manual discusses a global data share (GDS) memory, shared among all resident WGs and accessible through special atomics [42]. The GDS supports inter-WG synchronization through specialized ordered append/consume operations, supporting a finite number of synchronization events and synchronizing WGs at a time. Liu proposed EffiSync [43], an architecture designed for efficient synchronization that further generalizes inter-WG synchronization by virtualizing synchronization events. However,

Table 1: Baseline GPU model.

8 Compute Units, each with the following configuration:	
Clock	2GHz
SIMD units	2
SIMD width	64
Wavefronts per SIMD	20
Memory Hierarchy (64 B block size):	
1 Instruction Cache / 4 CUs	32 KB, 8-way set assoc., 4 cycles
1 Scalar Cache / 4 CUs or CP	16 KB, 8-way set assoc., 4 cycles
L1 cache / CU	32 KB, 16-way set assoc., 30 cycles
L2 cache shared	512 KB, 16-way set assoc., 50 cycles
DRAM	DDR3, 4 Channels, 1 GHz

these approaches do not consider an arbitrary number of synchronization variables, waiting conditions, and waiting WGs.

To prevent deadlock, CUDA 9 introduced a new execution abstraction, cooperative groups, which allows inter-WG synchronization. When using cooperative groups, the programmer has to use new API functions to launch a GPU kernel and to manage WGs within a cooperative group. This adds programmability and portability costs. More importantly, the program must wait until the scheduler ensures that all WGs within a cooperative group can be resident, limiting the kernel scheduler to static resource assignment.

Prior work has also looked into ways to provide inter-WG synchronization despite not ensuring IFP. First, several papers explored using persistent threads to avoid oversubscription [44] and the overhead of launching multiple kernels [45][46]. Second, prior work created software implementations of a variety of synchronization primitives for GPUs including various mutexes and barriers [9][47][48][49]. However, all these rely on persistent threads which assume static resource allocation across a kernel lifetime to ensure inter-WG IFP. Finally, Wireframe [50] specifies dependencies between WGs statically and presents a dependency-aware WG scheduler. Although it is not the focus of this work, AWG does not burden the programmer with a specialized WG dependency and can handle dynamic dependencies between WGs using atomic operations.

III. METHODOLOGY

We use a modified version of the gem5 simulator [51] to model a tightly-coupled GPU system (i.e., APU). Furthermore, we model in detail multiple WG scheduling policies described in the remainder of this paper, including memory operations for WG context saving and restoring, latencies for accessing new hardware structures, and periodic checking of conditions. Table 1 summarizes our baseline GPU model.

Choosing benchmarks to assess the behaviour of AWG is challenging because few GPU benchmarks use inter-WG synchronization. As a result, prior work has often focused on microbenchmarks to evaluate the behaviour of their system [9][52][53][54]. We select the HeteroSync suite [55] to further explore the design space of inter-WG synchronization. All benchmarks in HeteroSync require inter-WG fine-grained synchronization, use busy-*waiting*, and are representative of the most widely used forms of synchronization currently used on GPUs. In our experiments, we use both locally (L) and globally (G) scoped [53][56] synchronization variables. The selected benchmarks include multiple different centralized and decentralized mutexes and barriers, several performing exponential backoff in software. These benchmarks also cover a wide set of characteristics, as highlighted in Table 2.

IV. AUTONOMOUS WORK-GROUPS

We design AWG by proposing a family of novel autonomous architectures that provide inter-WG IFP, through cooperative scheduling of WGs, and evaluate their tradeoffs. Predominantly, these architectures rely on the set or a subset of the following four main components: 1) new *waiting* instructions that can be used by compilers, runtimes, libraries, or programmers to indicate opportunities for WGs to efficiently *wait* on synchronization operations, 2) a Synchronization Monitor (SyncMon) that observes accesses to monitored synchronization variables, 3) firmware extensions to the existing Command Processor (CP) to track waiting WGs, coordinate context switching, and check conditions that cannot be monitored by SyncMon, and 4) a virtualization interface between the SyncMon and the CP. Figure 4 presents a high-level overview of our family of architectures.

Table 2: Inter-WG synchronization benchmarks, constituting a busy-waiting Baseline: HeteroSync, Hash Table, and Bank Account [G = total number of WGs, L = number of WGs per CU, n = number of WIs per WG, d = size of shared data structure].

Benchmark	Abbreviation	Description	Granularity (# WIs per sync var)	# sync vars	# of conds per sync vars	# waiters per cond	# updates per sync var until condition met
SpinMutex	SPM_G	Test-and-set lock	n	1	1	G	2
FAMutex	FAM_G	Centralized ticket lock	n	1	G	1	1
SleepMutex	SLM_G	Decentralized ticket lock	n	G	1	1	1
AtomicTreeBarr	TB_LG	Two-level tree barrier	n	G/L	1	L	L
LFTreeBarr	LFTB_LG	Decentralized two-level tree barrier	n	G	1	1	1
SpinMutexLocal	SPM_L	Test-and-set lock local scope	n	G/L	1	L	2
FAMutexLocal	FAM_L	Centralized ticket lock local scope	n	G/L	L	1	1
SleepMutexLocal	SLM_L	Decentralized ticket lock local scope	n	G	1	1	1
AtomicTreeBarrLocalExch	TBEX_LG	Two-level tree barrier	n	G/L	1	L	L
LFTreeBarrLocalExch	LFTBEX_LG	Decentralized two-level tree barrier	n	G	1	1	1

A. *Challenges in Cooperative WG Scheduling*

Given the WG scheduling granularity considered by this work, there are some challenges in effectively addressing the problem of inter-WG IFP.

Context switching overhead. GPUs have significantly larger contexts than CPUs, leading to higher overheads. WGs can have up to 1024 WIs, each with their own vector registers. In addition, WIs within a WG share Local Data Share (LDS) memory and wavefronts within a WG have their own scalar registers. Figure 5 shows that the WG context size ranges from 2 to 10 KB for our benchmarks [55] [57]. Thus, it is important to avoid context switches whenever possible.

Virtualizing the number of waiting WGs and conditions. Given the finite size of the SyncMon, a limited number of conditions can be simultaneously monitored by the SyncMon. Additionally, a limited number of waiting WGs can also be stored on chip. Thus, to be able to synchronize WGs when GPU resources are oversubscribed, we need to overcome limited monitoring capabilities of the SyncMon.

Virtualizing the data structures used for scheduling. The CP requires additional data structures to hold waiting WGs, waiting conditions, and different queues that track the state of waiting WGs as they are stalled, context switching out, waiting, ready, or context switching in. Having finite hardware structures holding these data structures is challenging as they can easily overflow.

Synchronization Contention. Some synchronization primitives exhibit performance-challenging behaviors by generating additional memory traffic, while trying to acquire a synchronization variable [58]. These types of synchronization primitives are usually the simplest ones to program, therefore providing support for such primitives constitute an important programmability aspect.

B. *Our Approach*

Motivated to ensure inter-WG IFP, we take a systematic approach in overcoming previously mentioned challenges. First, we take a holistic approach to reduce unnecessary context switches. After looking at the different scheduling policies, we identified cooperative scheduling as better suited for high throughput computing devices with large WG context size. To achieve cooperative scheduling, we use *waiting* instructions to provide hints at opportunities when WGs can yield their hardware resources and be context switched out. These instructions are necessary for IFP of WGs, and context switching may occur when they are executed to allow other WGs to proceed. Moreover, we context switch out a WG only if there are other WGs ready to be resumed or started, meaning only if the kernel over-subscribes GPU resources.

Furthermore, as depicted in Figure 4, we propose specialized hardware (i.e., SyncMon) to enhance cooperative WG scheduling by notifying when WGs can be resumed for execution. We overcome SyncMon's limited monitoring abilities by defining a virtualization interface used to communicate to the CP of extra conditions and waiting WGs.

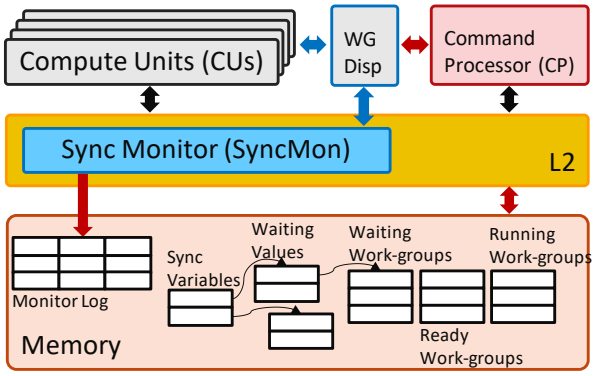

Figure 4: Autonomous Work-Groups Overview.

Contention on synchronization variables has been a well-studied phenomenon in computer architecture since the design of the first multi-processor computers [58][59]. We investigate contention mitigating resume policies. Additionally, in case of latency sensitive or low-contention synchronization, such as global barriers or spin mutexes with few contending WGs, context switching can incur a performance penalty. To solve this, before context switching a WG out, we stall waiting WGs for a predicted time period, and only context switch out a WG if its condition has not been met when the period expires. AWG predicts the stall period by recording the mean number of cycles at which conditions are met.

Overall, we evaluate exponential backoff with sleep and propose autonomous architectures with different *waiting* and resume policies each varying in degree of necessary hardware support: simple timeout mechanism and different types of special instructions which arm a hardware monitor that can resume all or one waiting WG, when a condition is met. Figure 6 depicts the timelines for all these architectures. Furthermore, we analyze them by comparing with a *Baseline* configuration composed of software busy-*waiting* and hardware deadlocking in case of over-subscription. However, in this subsection we show quantitative results for non-over-subscribed GPUs in order to build up the trade-offs for each autonomous architecture; we discuss the hardware necessary to implement our final solution, AWG, in Section V and we quantitatively compare these architectures to each other and to AWG for over-subscribed GPUs in Section VI.

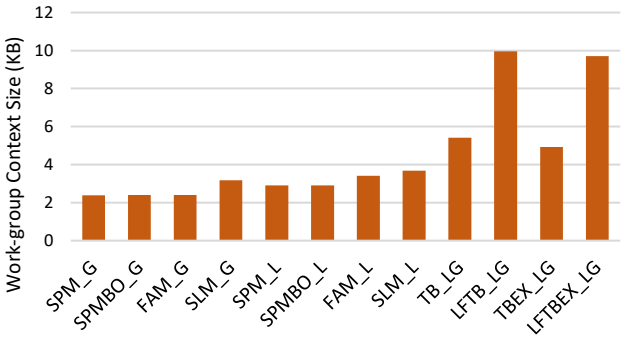

Figure 5: Work-group context size.

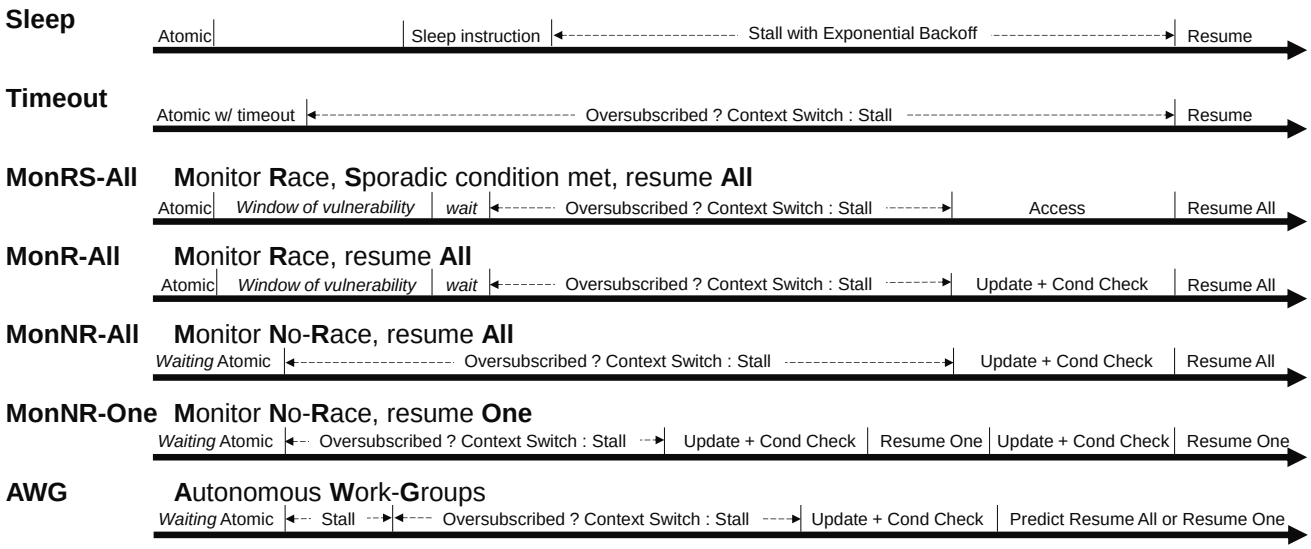

Figure 6: Timeline signatures for different cooperative WG scheduling policies.

C. *Waiting Policies*

i. *Sleep and Exponential Backoff*

Current GPUs provide support for optimizing synchronization by waiting, through instructions such as *s_sleep*, which stalls a wavefront for a fixed number of cycles [42][60]. Sleep instructions have low hardware overhead and provide ease of programmability. However, they support limited timeout periods and do not wait for a specific event, making them less adaptable to system dynamics.

Figure 7 illustrates that exponential backoff—doubling the sleep time in software after each failed retry, up to a maximum backoff interval—improves performance for many workloads. Increasing the maximum backoff interval—X thousands cycles for the label *Sleep-Xk*—decreases contention on synchronization variables. This improves performance to a point, but eventually becomes counterproductive because WGs that could make progress sleep too long. These results also show there is no one best static sleep configuration across the different synchronization primitives. More importantly, sleep instructions do not release hardware resources while sleeping, failing to provide inter-WG IFP when the GPU is over-subscribed.

ii. *Simplistic Hardware Support: Fixed Timeout*

To provide IFP for over-subscribed kernels, we first investigate a *Timeout* architecture. In the non-oversubscribed case, Timeout stalls a WG for a fixed interval of time. Unlike Sleep, this interval does not represent a maximum backoff interval. In the over-subscribed case, Timeout yields its resources by context switching out for a fixed timeout interval. Figure 8 shows that there is no single best static timeout interval: different synchronization primitives prefer different timeouts. More importantly, for some timeout intervals, Timeout performs much worse than the busy-*waiting* Baseline for non-oversubscribed kernels. These results motivate additional hardware support for *waiting*.

iii. *Relaxed Hardware Support*

As discussed in Section II, modern CPUs implement monitor and mwait instructions to detect when synchronization conditions (may) have been met. We extend this approach to

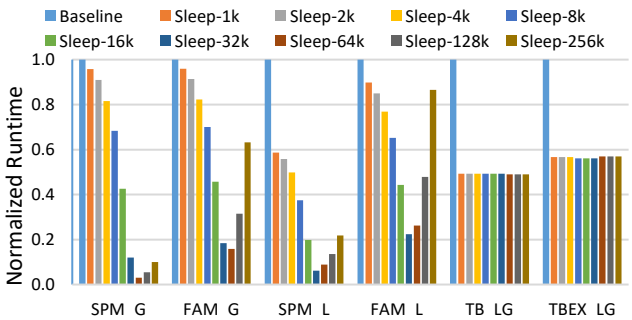

Figure 7: Exponential backoff with s_sleep, normalized to the Baseline where s_sleep is not used.

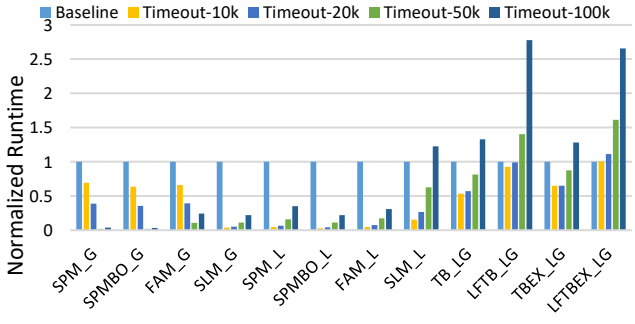

Figure 8: Timeout interval runtime, normalized to the Baseline.

GPUs, by proposing *MonRS-All*, where waiting WGs can hint yielding of hardware resources with special *wait* instructions and notifications are supported through a simplistic hardware monitor (i.e., SyncMon). Unlike CPUs, which maintain ownership-based coherence across all caches, GPUs use write-through caches and perform atomics at the shared last-level cache (e.g., L2) [54]. Because of these limitations, GPUs cannot rely on coherence protocols to detect a write. In this case, the simplistic SyncMon observes memory accesses and if a monitored address is accessed it will notify corresponding waiting WGs to resume, without checking their waiting condition. Therefore, we call these notifications sporadic.

Compared to Timeout, this waiting policy better reacts to system dynamics by monitoring synchronization variables. However, such relaxed hardware support for synchronization on GPUs is dominated by unnecessary resuming of WGs. Although this works well for decentralized synchronization primitives, where there are few updates to a given variable, it is inefficient at waiting for all centralized synchronization primitives. Figure 9 shows the wait efficiency of MonRS-All, as the number of executed atomic instructions normalized to an oracular *MinResume* configuration which does not resume WGs unnecessarily. MinResume achieves this by spreading out when waiting WGs are resumed, such that WGs will not contend when retrying to acquire sync variables. MonRS-All executes up to two orders of magnitude more atomic instructions in some cases, because it unnecessarily resumes waiting WGs.

iv. Enhanced Hardware Support

To overcome the shortcomings of sporadic notifications, we enhance the hardware support for synchronization by proposing MonR-All. MonR-All uses the same special *wait* instruction to hint a waiting WG can yield its hardware resources, except it enhances SyncMon to check waiting conditions as synchronization variables are being updated. The SyncMon registers waiting WGs, checks waiting conditions when monitored addresses are written, and resumes all waiting WGs associated with a met condition. Figure 9 shows MonR-All to have better wait efficiency than MonRS-All. Decentralized synchronization primitives are unaffected because they can have at most one write operation per synchronization variable. Nevertheless, providing enhanced hardware support exhibits a clear performance advantage for centralized primitives due to reduced resuming of WG.

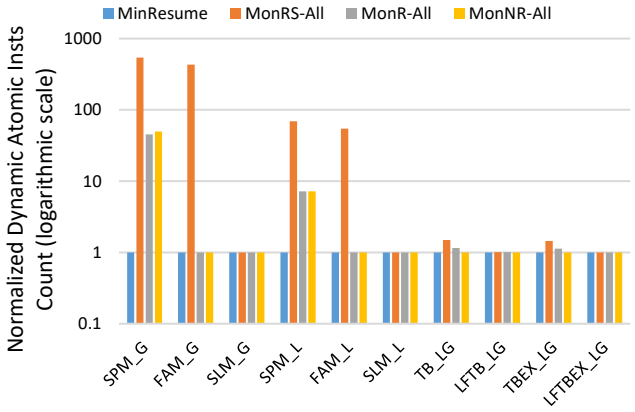

Figure 9: Wait efficiency, number of atomic instructions executed normalized to MinResume.

However, this approach has a data race that causes a window of vulnerability when monitoring addresses. As shown in Figure 10, for a decentralized ticket mutex there is a data

```
1.  __device__ void decentralizedTicketMutexLock(...) {
2.    __shared__ bool acquired;
3.    ...
4.    if (isMasterThread) {
5.      myQueueLoc = atomicAdd(queueTailPtr, 16);
6.      atomicExch(&acquired, false);
7.    }
8.    __syncthreads();
9.    while (!atomicLoad(&acquired)) {
10.     if (isMasterThread) {
11.       if (atomicLoad(myQueueLoc) == 1) {  //<- lock acquire
12.         atomicExch(&acquired, true);
13.       }
14.     }                                   window of vulnerability
15.     __syncthreads();
16.     if (!atomicLoad(&acquired)) {
17.         wait(myQueueLoc, 1);          //<- arming SyncMon
18.     }
19.   }
20. }
21. __device__ void decentralizedTicketMutexUnlock(...) {
22.   ...
23.   if (isMasterThread) {
24.     atomicExch(myQueueLoc, -1);
25.     atomicExch(nextQueueLoc, 1);          //<-  lock release
26.   }
27. }
```

```
1.  __device__ void decentralizedTicketMutexLock(...) {
2.    __shared__ bool acquired;
3.    ...
4.    if (isMasterThread) {
5.      myQueueLoc = atomicAdd(queueTailPtr, 16);
6.      atomicExch(&acquired, false);
7.    }
8.
9.    __syncthreads();
10.
11.   while (!atomicLoad(&acquired)) {
12.     if (isMasterThread) {                          expected value
13.       if (atomicCmpWait(myQueueLoc, 1) == 1) { //<- lock acquire
14.                           // arming SyncMon if comparison failed
15.         atomicExch(&acquired, true);
16.       }
17.     }
18.   }
19. }
20.
21. __device__ void decentralizedTicketMutexUnlock(...) {
22.   ...
23.   if (isMasterThread) {
24.     atomicExch(myQueueLoc, -1);
25.     atomicExch(nextQueueLoc, 1);                 //<- lock release
26.   }
27. }
```

Figure 10: Decentralized ticket lock implementations, with wait instructions and waiting atomic instructions. We are showing these new instructions as intrinsic functions.

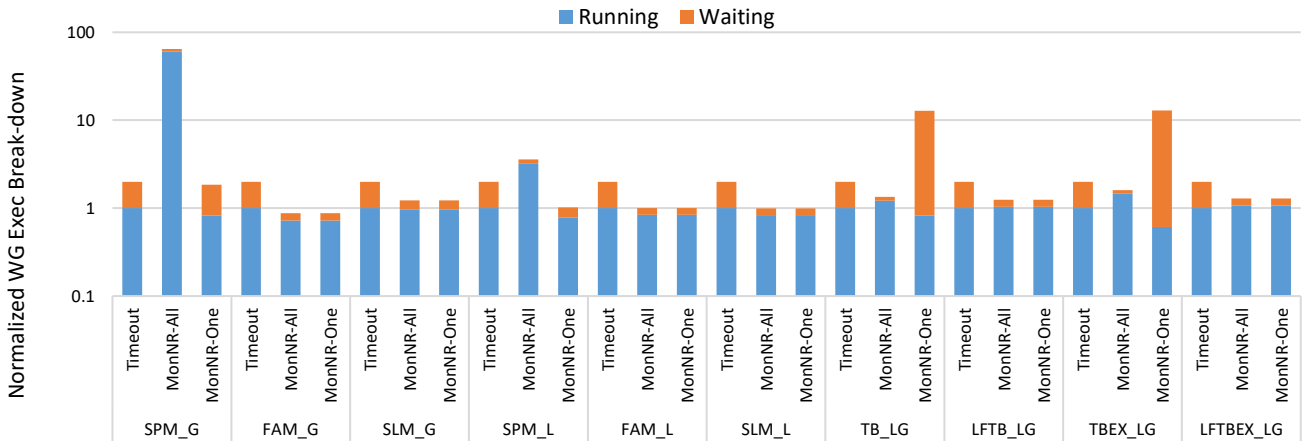

Figure 11: WG execution time break-down, normalized to Timeout.

race between arming the monitor and the atomics updating synchronization variables. In this synchronization primitive each WG places itself in a queue by atomically incrementing the tail pointer and then polls its queue entry. A queue entry value of -1 indicates the mutex is locked and the value of 1 indicates the mutex is unlocked. Initially, only the first queue entry is unlocked until the first WG locks the entry to begin its critical section. Then when the first critical section completes, the associated WG unlocks the next queue entry. Thus, the next WG in the queue can grab the lock and proceed executing its critical section. Even though in program order *wait* precedes the atomic exchange operation, the SyncMon can observe the lock release first and the *wait* succeeding it. This makes the SyncMon unable to provide inter-WG IFP by itself, as deadlock arises when it misses updates to monitored addresses. We next show how to overcome this shortcoming.

D. *Waiting Atomics*

Rather than introducing new *wait* instructions, we identify the synchronization points where WGs can naturally yield their resources by enhancing atomic instructions. To remove the window of vulnerability we propose MonNR-All, which provides hardware support for sync variables through new waiting atomic instructions. This hardware support fully embraces the synchronization library extension recently proposed to the C++20 standard [18].

All waiting atomics have an extra operand that specifies the expected value of the synchronization variable. If a waiting atomic fails when comparing the acquired value with the expected value, the WG associated with that atomic enters in a *waiting* state. This means that updates will not be missed by the SyncMon and the associated WG can be context switched out at least until its waiting condition is met, ensuring inter-WG IFP. For example, a compare-and-swap instruction is a perfect candidate for a waiting atomic, as it already has an operand for an expected value. However, there are synchronization primitives which synchronize using atomic load operations that do not have this information. Thus, we propose a new *compare-and-wait* atomic instruction that performs a load, compares the retrieved value with the expected value, and waits on the expected value if the comparison fails. Figure 10 demonstrates the use of the proposed *compare-and-wait* instruction, when implementing a decentralized ticket lock, and exemplifies an implementation with *wait* instructions for comparison.

E. *Resume Policies*

Figure 9 shows that MonR-All and MonNR-All can further improve their wait efficiency compared to a MinResume that avoids unnecessarily resuming waiting WGs. Waiting atomics can be slightly less inefficient at mitigating contention as they register waiting WGs earlier, which captures more waiting WGs per condition and therefore resumes more waiting WGs when their condition is met. This produces slightly higher contention for MonNR-All than MonR-All, which uses *wait* instructions. To address contention on synchronization variables, we propose another resume policy, *MonNR-One*. In MonNR-One, the SyncMon observes atomic updates to monitored addresses, resumes only one waiting WG once a condition is met, and continues to monitor the same condition. The rest of the waiters are resumed when a different update to the monitored address meets the condition or after a fixed timeout interval. In Figure 11, we break-down the WG execution time based on the two states a WG can have in the non-over-subscribed case: running and waiting on synchronization. *MonNR-One* performs well in cases of high contention such as spin mutexes. However, it performs poorly for global barriers, where MonNR-All performs well because it wakes up all the waiters at the barrier at once. To address this imbalance between MonNR-All and MonNR-One, we introduce another policy, AWG, to predict the number of WGs to resume.

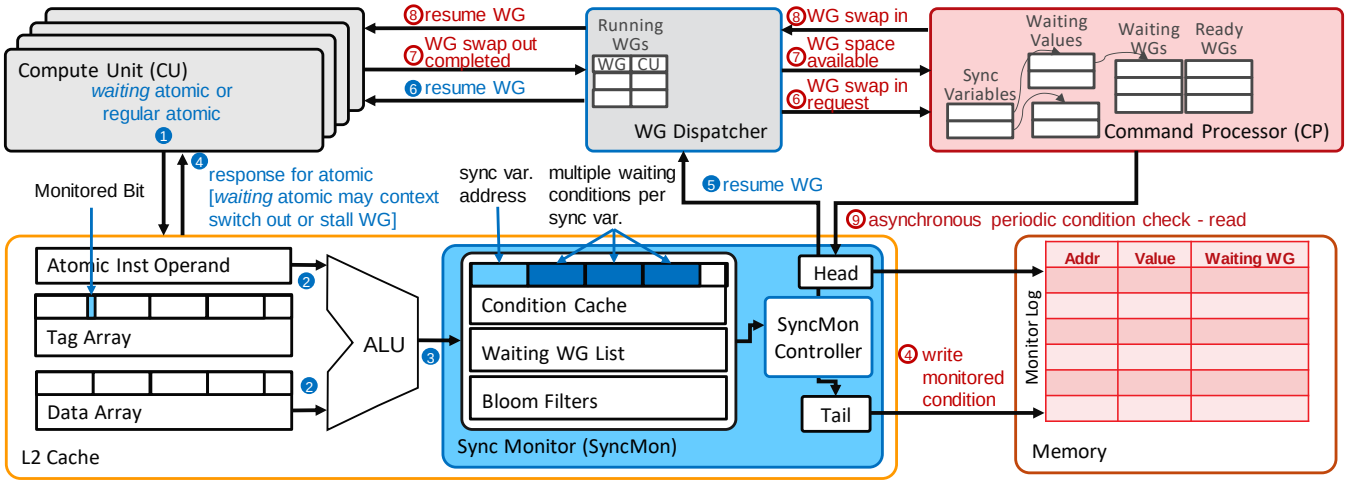

Figure 12: Autonomous Work-Groups micro-architecture.

V. AWG: Detailed Description

In this section, we detail AWG's hardware implementation and walk through how AWG ensures WG IFP.

A. Hardware Components

AWG relies on current GPU abilities to perform atomic operations at its last level cache (i.e., L2) [42] and extends the atomic instructions to *wait* on certain conditions in case the atomic operation fails. As previously introduced, AWG relies on two hardware components (the SyncMon and the CP) to effectively support these waiting atomics.

Figure 12 presents AWG's detailed hardware implementation. The SyncMon is added to the L2 cache banks to monitor sync variables for specific waiting conditions. The SyncMon is a distributed design similar to current GPU L2 cache microarchitecture and can provide both scalable high throughput and enhanced hardware support for Mesa semantics. However, only a finite number of waiting conditions and WGs can be cached in the SyncMon block.

To extend beyond finite hardware resources and handle scenarios when the synchronization variable working set exceeds the L2 cache capacity, AWG supports a virtualization interface [61][62] composed of a Monitor Log and a protocol for reading and writing the Monitor Log. The Monitor Log is a circular buffer residing in global memory that stores entries composed of the monitored address, the waiting value, and the waiting WG ID. In the rare situation when the SyncMon reaches its capacity limit for storing either waiting conditions or waiting WGs, it writes additional entries to the Monitor Log. If the Monitor Log is full, the waiting atomic fails its comparison, however, the associated WG does not enter a waiting state as it normally does. Instead, the WG continues executing and retries its waiting atomic (i.e., Mesa program semantics) until the CP processes the Monitor Log and frees some entries.

AWG relies on the CP to handle the Monitor Log entries and perform WG context switching. The CP is not involved in the common case when the kernel does not oversubscribe the GPU and WGs are simply stalled on the CUs. Instead, the CP focuses on tracking context switched out WGs, and it detects spilled sync variables by parsing the Monitor Log written to by the SyncMon. To efficiently track waiting WGs, the CP uses an in-memory data structure and updates their status changes between stalled, context switched out, ready, or resuming. The CP also periodically checks the waiting conditions of spilled sync variables. The Monitor Log may contain younger waiting conditions than the SyncMon Cache. This can lead to fairness issues that can be addressed with different replacement policies. We leave this study for future work.

To improve the inefficiencies of MonNR-All and MonNR-One, detailed in previous section, AWG predicts the number of WGs to resume. The prediction mechanism counts the number of waiting WGs and uses one counting Bloom filter per monitored address to count the number unique updates to the associated address. AWG will resume all waiters for global barriers, when it detects there are more than one waiting WGs per waiting condition and more than two unique updates to the sync variable. AWG will resume waiters one by one when it detects there are multiple waiting WGs per waiting condition and at most two unique updates. Once a condition has been met, all waiting WGs have resumed, and the address is not monitored, the associated Bloom filter is reset. If AWG's prediction is incorrect, eventually the stalled WGs will time out and be activated.

B. Detailed Mechanism

The dispatcher is responsible for assigning a unique ID to each dispatched WG. This WG ID is used by AWG throughout the entire cooperative scheduling process, from registering waiting WGs and managing their waiting conditions and memory storage for their contexts, to communicating to the

CU and indicating what WGs should be context switched. The CP is involved only in high-latency operations (i.e., context switching and Monitor Log operations) and is not on the critical path.

Figure 12 walks through all major microarchitectural events when monitoring sync variable conditions and scheduling WGs. In the figure, the fast, common path for detecting when conditions are met is depicted in blue, while the slower, uncommon operations that occur when WGs are context switched out or sync variables are spilled to the Monitor Log is depicted in red. When a WG executes a waiting atomic instruction ❶, its generated memory access will include the operand for an expected value and the issuing WG ID. Assuming all atomics are performed at the L2, AWG extends each L2 cache tag with one monitor bit, to indicate monitored addresses and pins monitored cachelines such that they are not evicted.

Eventually the waiting atomic operation will arrive at the L2 cache ❷ along with its expected successful value. If the operation (e.g., comparison) fails ❸, then the executing WG is considered waiting on the specified condition. In this case, the monitored bit in the tag is set and the waiting WG ID will be stored in the SyncMon cache along with its waiting condition. The SyncMon communicates the desired WG's waiting state back to the CU as part of the atomic operation response ❹. The desired waiting state can have two values: stalled (i.e., not executing but still holding hardware resources), or context switched (i.e., not executing and not holding any hardware resources).

Subsequent atomic operations check monitored bit and if it is set, they pass the updated value to the SyncMon ❸. If the SyncMon determines a condition is met, the monitored bit is cleared and the SyncMon informs the dispatcher to resume the associated waiting WG(s) ❺. In the common case when the WGs are simply stalled and still consuming CU resources, the dispatcher requests the CUs to resume the WGs ❻. In the uncommon case where the waiting WGs are context switched out, the dispatcher initiates context-switch-in operations through the CP ⑥, evaluates available resources and picks the CUs that can accommodate the WGs.

The SyncMon is a finite hardware structure and AWG uses the CP to handle cases when the SyncMon is full and cannot monitor additional conditions. For instance, when a WG's waiting condition or ID cannot be inserted into the SyncMon cache ❸, the WG's information bypasses the SyncMon cache and is written to the tail of the Monitor Log ④. When the SyncMon requests a WG context switch upon executing a waiting instruction, the CU informs the CP, via dispatcher, when the operation completes ⑦. The CP will then context switch in and resume, via dispatcher, ready WGs ⑧. Periodically, the CP block removes the valid entries between the Monitor Log's head and tail pointers and stores the associated waiting WG information into a more look-up efficient data structure. The CP then uses that data structure to check conditions and determine which WGs must be swapped back in, when conditions are met ⑨.

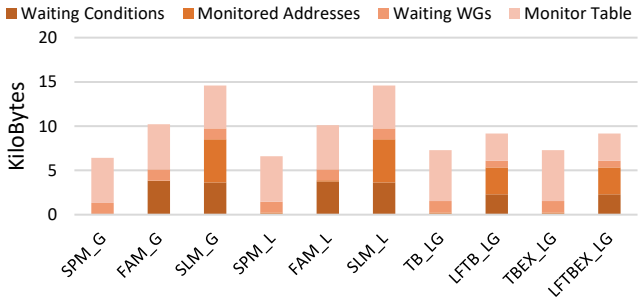

Figure 13: Size of data structures used by the Command Processor for WG scheduling.

C. *Hardware Overhead*

AWG extends the GPU CP with new firmware to perform WG scheduling. AWG's SyncMon condition cache is logically 4-way set associative with 256 sets, it can hold a total of 1024 waiting conditions. Waiting conditions are calculated by hashing the monitored address and waiting value together. Specifically, the address is shifted left with log of number of cache entries, after subtracting log of cacheline size, and bitwise ORed with the waiting value. The result is further hashed with a universal hash function [63]. To identify the waiting WGs for each waiting condition, each SyncMon condition cache entry holds two 9-bit pointers (head and tail) into a separate waiting WG list. This list holds up to 512 waiting WG IDs and collectively the condition cache and WG list have a total size of 26112 bits or 3.18 KB. To predict the number of WGs to resume, AWG adds 512 Bloom filters, each storing 24 bits and using 6 hash functions, with a total overhead of 12288 bits or 1.5 KB. We have configured the Bloom filters to have a small false positives probability (2.1%) when recording unique values observed to monitored addresses. Finally, AWG adds one monitored bit per L2 tag, which results in an additional 1 KB overhead to the overall L2 cache size. It is important to note that all aspects of AWG's hardware components can be addressed sliced and distributed across a large, high-bandwidth L2 cache design.

Figure 13 displays the size of all the data structures used by the CP for scheduling. Here we show the maximum Monitor Log size assuming no SyncMon Cache. In addition to data structures used for scheduling, the CP also allocates memory for holding WG contexts. This memory varies between 0.74 – 3.11 MB across all evaluated benchmarks.

D. *Benefits of Our Design*

Reducing interference with kernel scheduling. AWG decouples pre-emptive scheduling of kernels and concurrent multi-kernel execution from scheduling WGs within a kernel. In this way, AWG relaxes the pre-emption constraints of kernel scheduling, which improves performance and allows the

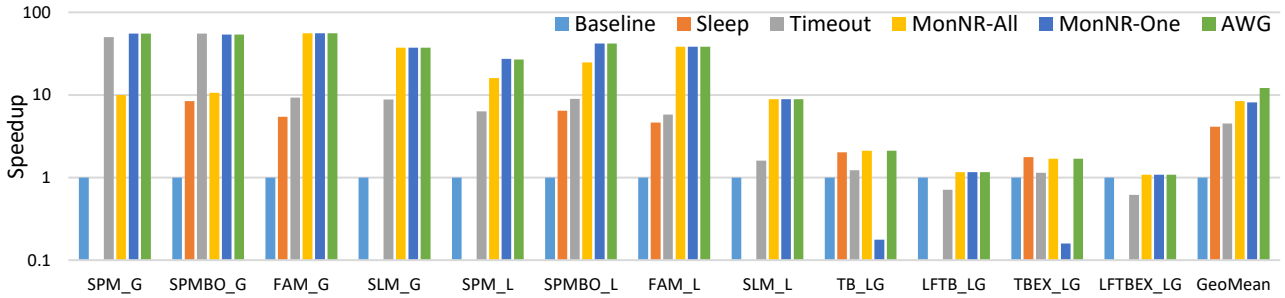

Figure 14: Speedup normalized to Baseline in non-oversubscribed scenario.

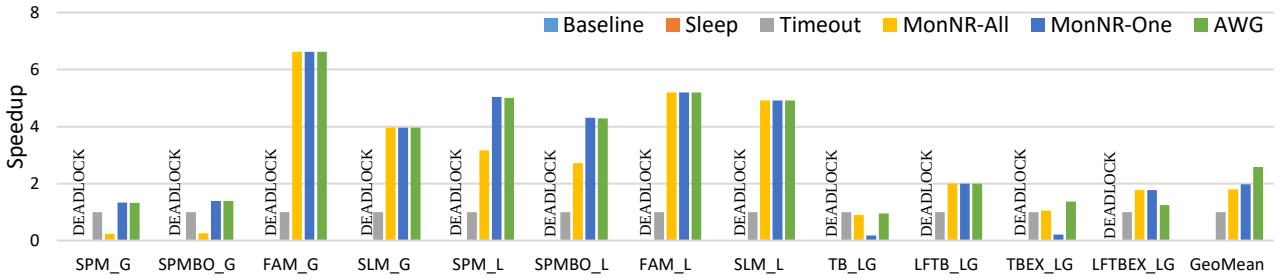

Figure 15: Speedup normalized to Timeout in oversubscribed scenario.

GPU to be more responsive to high priority kernels while, at the same time, ensuring the IFP of lower priority kernels.

Employing WG context switching when necessary. Given that GPU kernels can be large, a pre-emptive WG scheduling approach can be poor performing and inefficient. AWG employs cooperative WG scheduling and tracks individual WG's waiting condition such that context switches are only performed when WGs are ready to execute.

Virtualizing waiting conditions and WGs. A solution to WG IFP that is bounded by the size of hardware structures imposes serious challenges to programmers. AWG combines two methods for condition checking and leverages virtual memory to overcome hardware monitoring limitations.

Effective contention mitigation. AWG reduces contention by predicting the number of waiting WGs needed to be resumed when a condition is met.

VI. Performance Analysis

In this section we evaluate AWG using two experiments, a non-oversubscribed experiment where resources are constant throughout the entire kernel lifetime, and an oversubscribed experiment where resources vary during its lifetime. Specifically, our oversubscribed experiment starts with 8 CUs and after 50 µs the WGs from one CU are context switched out. This emulates a kernel scheduling scenario where resource availability varies across kernel scheduling time slices, or a high-priority pre-emption scenario where a lower priority kernel dynamically loses resources. In these oversubscribed experiments, the baseline deadlocks when context-switched WGs (i.e., due to kernel level scheduling) fail to release a synchronization variable before being swapped out. Moreover, these oversubscribed inter-WG synchronization scenarios are not supported in current GPUs, including the ones with CUDA cooperative groups.

Figure 14 shows that when non-oversubscribed, AWG outperforms the Baseline with a geometric mean of 12x in speedup, across the HeteroSync benchmarks. It outperforms Sleep and Timeout because it better utilizes execution resources and the memory hierarchy, whereas Sleep or Timeout use fixed time intervals that do not fit the dynamic behaviour of our benchmarks. Note that Sleep only appears for the benchmarks that have been modified to use exponential backoff with the sleep instruction (see Section IV.C).

Both MonNR-All and MonNR-One perform poorly for certain benchmarks. MonNR-All is deficient on benchmarks where multiple waiters contend on synchronization variables where only one WG can enter a critical section at a time. In contrast, MonNR-One manages contention well for these benchmarks by resuming only one WG immediately and waiting for additional condition met events to resume other waiters. However, MonNR-One displays performance deficiency in case of centralized tree barriers where there are multiple waiters per condition and all of them are expected to start immediately. Meanwhile AWG outperforms both by predicting when to resume one waiting WG versus multiple waiting WGs.

Finally, Figure 15 shows AWG's speedup across all benchmarks for the oversubscribed scenario. Baseline techniques, such as cooperative groups, do not support such a scenario. AWG predicts the number of waiters to resume, whereas MonNR-All and MonNR-One have fixed strategies for selecting the number of WGs to resume. This proves particularly beneficial for centralized synchronization primitives. For some tree barriers (i.e., TB_LG and LFTBEX_LG), AWG is slower because of the stall time prediction. Barriers are latency sensitive and predicting a too long stall time adds context switch overhead to the application's critical path.

Nevertheless, AWG has an average speed up of 2.5x over Timeout.

VII. Related Work

In addition to the GPU synchronization and scheduling work discussed in Section II, other work virtualizes GPU resources in attempt to escape the bound on occupancy imposed on kernels when scheduling WGs. Virtual Thread [64] allows more WGs in flight than the hardware limit and relies on context switching to increase thread level parallelism. Jeon *et al.* decreased the number of physical registers while maintaining the number of architectural registers unchanged [65]. Zorua gives the appearance of higher hardware resource availability to account for different phases in the application, which have different requirements on resources and therefore can have a different number of resident WGs at each phase [66]. However, Zorua does not address inter-WG synchronization and context switches WGs at phase boundaries without regard to potential synchronization deadlock.

VIII. Conclusion

In this paper, we proposed AWG, an architecture designed to provide WG IFP for arbitrarily sized kernels which use inter-WG synchronization. AWG uniquely identifies synchronization operations as opportunities for cooperative WG scheduling and uses enhanced hardware support for synchronization at the last-level cache. We also demonstrated that WG scheduling policies can significantly impact performance for kernels that use this type of synchronization and AWG uses this insight to select the appropriate number of WGs to resume.

This work overcomes current challenges in virtualizing GPU hardware resources at the WG level by using the CP and virtual memory. In addition, our work provides a sense of autonomy to the WG execution abstraction, freeing the GPU from additional constraints when scheduling kernels with inter-WG synchronization.

Acknowledgement

The authors would like to thank Marc Orr, Tanmay Gangwani, Tony Gutierrez, Tony Tye, Brian Sumner, Mike Mantor, Mark Fowler, Steve Reinhardt, Mark Hill, and the anonymous reviewers for their feedback during various stages of the project.

References

[1] Thiruvengadam Vijayaraghavany, Yasuko Eckert, Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, Bradford M. Beckmann, William C. Brantley, Joseph L. Greathouse, Wei Huang, Arun Karunanithi, Onur Kayiran, Mitesh Meswani, Indrani Paul, Matthew Poremba, Steven Raasch, Steven K. Reinhardt, Greg Sadowski, and Vilas Sridharan, "Design and Analysis of an APU for Exascale Computing," in *Proceedings - International Symposium on High-Performance Computer Architecture*, 2017, pp. 85–96.

[2] Gregory Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh, "Persistent RNNs : Stashing Recurrent Weights On-Chip," in *International Conference on Machine Learning*, 2016.

[3] Mohammad Shoeybi Sercan Arık, Mike Chrzanowskii, Adam Coates, Gregory Diamos, Andrew Gibiansky, Yongguo Kang, Xian Li, John Miller, Andrew Ng, Jonathan Raiman, Shubho Sengupta, "Deep Voice: Real-time Neural Text-to-Speech Sercan," in *Computers and Mathematics with Applications*, 2013, vol. 65, no. 10, pp. 1471–1482.

[4] Advanced Micro Devices, "Asynchronous Shaders Whitepaper," 2015, p. 9.

[5] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero, "Enabling Preemptive Multiprogramming on GPUs," in *ACM SIGARCH Computer Architecture News*, 2014, vol. 42, no. 3, pp. 193–204.

[6] Guoyang Chen, Yue Zhao, and Xipeng Shen, "EffiSha : A Software Framework for Enabling Efficient Preemptive Scheduling of GPU," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 3–16.

[7] Sercan Arik, Mike Chrzanowski, Adam Coates, Gregory Diamos, Andrew Gibiansky, Yongguo Kang, Xian Li, John Miller, Andrew Ng, Jonathan Raiman, Shubho Sengupta, and Mohammad Shoeybi, "Deep voice: Real-time neural text-to-speech," in *34th International Conference on Machine Learning, ICML 2017*, 2017, vol. 1, no. Icml, pp. 264–273.

[8] Feiwen Zhu, Jeff Pool, Michael Andersch, Jeremy Appleyard, and Fung Xie, "Sparse Persistent RNNs: Squeezing Large Recurrent Networks On-Chip," in *arXiv preprint arXiv:1804.10223*, 2018.

[9] Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić, "Portable Inter-workgroup Barrier Synchronisation for GPUs," in *OOPSLA: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 39–58.

[10] Michael D. Mccool, Zheng Qin, and Tiberiu S. Popa, "Shader Metaprogramming," 2002, pp. 57–68.

[11] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard, "Cg: a system for programming graphics hardware in a C-like language," in *ACM Transactions on Graphics*, 2003, vol. 22, no. 3, p. 896.

[12] Mike Houston, "Brook for GPUs : Stream Computing on Graphics Hardware," 2001.

[13] CUDA Nvidia, "Compute unified device architecture programming guide," 2007.

[14] Aaftab Munshi, "The opencl specification," in *Hot Chips 21 Symposium (HCS), 2009 IEEE*, 2009, pp. 1–314.

[15] AMD, "HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer, and Object Format (BRIG)," 2019. [Online]. Available: http://www.hsafoundation.com/standards/. [Accessed: 16-Aug-2019].

[16] AMD, "HSA Platform System Architecture Specification," 2019. [Online]. Available: http://www.hsafoundation.com/standards/.

[Accessed: 16-Aug-2019].

[17] NVIDIA, “Nvidia Tesla V100 Gpu Architecture,” 2017.

[18] Bryce A. Lelbach, Olivier Giroux, and JF Bastien, “The C++20 Synchronization Library,” 2018. [Online]. Available: https://isocpp.org/files/papers/p1135r1.html. [Accessed: 16-Aug-2019].

[19] Henry Wong, Anne Bracy, Ethan Schuchman, Tor M. Aamodt, Jamison D. Collins, Perry H. Wang, Gautham Chinya, Ankur Khandelwal Groen, Hong Jiang, and Hong Wang, “Pangaea : A Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor Categories and Subject Descriptors,” in *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 52–61.

[20] Sameer Kumar, Yanhua Sun, and Laximant V. Kalé, “Acceleration of an asynchronous message driven programming paradigm on IBM Blue Gene/Q,” in *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, 2013, pp. 689–699.

[21] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy, “Supporting fine-grained synchronization on a simultaneous multithreading processor,” in *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, 1999, pp. 54–58.

[22] AMD, “AMD64 Architecture Programmer’s Manual Volume 3: General-Purpose andSystem Instructions,” 2017, no. 24594.

[23] Per Brinch Hansen, “Structured Multi- programming,” in *Communications of the ACM*, 1972, vol. 15, no. 7, pp. 574–578.

[24] Butler W. Lampson, and David D. Redell, “Experience with processes and monitors in Mesa,” in *Communications of the ACM*, 1980, vol. 23, no. 2, pp. 105–117.

[25] James R. Goodman, Mary K. Vernon, and Philip J. Woest, “Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors,” in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989, pp. 64–75.

[26] James R. Goodman, and Philip J. Woest, “The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor,” in *ACM SIGARCH Computer Architecture News*, 1988, vol. 16, no. 2, pp. 422–431.

[27] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith, “The Tera Computer System *,” in *Proceedings of the 4th international conference on Supercomputing*, 1990, pp. 1–6.

[28] Jiwei Liu, Jun Yang, and Rami Melhem, “SAWS: Synchronization Aware GPGPU Warp Scheduling for Multiple Independent Warp Schedulers,” in *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*, 2015, pp. 383–394.

[29] Ahmed Eltantawy, and Tor M. Aamodt, “MIMD Synchronization on SIMT Architectures,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–14.

[30] Thomas Bradley, “Hyper-Q Example,” 2013. [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf. [Accessed: 16-Aug-2019].

[31] NVIDIA, “CUDA Stream Management,” 2019. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/group__CUDART__STREAM.html. [Accessed: 16-Aug-2019].

[32] AMD, “HIP: Heterogeneous-computing Interface for Portability,” 2019. [Online]. Available: https://github.com/ROCm-Developer-Tools/HIP/. [Accessed: 16-Aug-2019].

[33] Justin Luitjens, “CUDA Streams: Best Practices and Common Pitfalls,” 2014. [Online]. Available: http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf. [Accessed: 16-Aug-2019].

[34] Zhen Lin, North Carolina, Lars Nyland, North Carolina, and North Carolina, “Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching,” in *SC’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, no. November, pp. 898–908.

[35] Zhen Lin, and Michael Mantor, “GPU performance vs. thread-level parallelism: Scalability analysis and a novel way to improve TLP,” in *ACM Transactions on Architecture and Code Optimization (TACO)*, 2018, vol. 15, no. 1, pp. 1–21.

[36] Izzat El Hajj, Juan Gomez-Luna, Cheng Li, Li-Wen Chang, Dejan Milojicic, and Wen-mei Hwu, “KLAP : Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[37] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili, “Dynamic Thread Block Launch : A Lightweight Execution Mechanism to Support Irregular Applications on GPUs,” in *ACM SIGARCH Computer Architecture News*, 2015, vol. 43, no. 3S, pp. 528–540.

[38] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo, “Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing,” in *Proceedings - International Symposium on High-Performance Computer Architecture*, 2016, vol. 2016-April, pp. 358–369.

[39] Hancheng Wu, Da Li, and Michela Becchi, “Compiler-Assisted Workload Consolidation for Efficient Dynamic Parallelism on GPU,” in *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016*, 2016, pp. 534–543.

[40] Guoyang Chen, and Xipeng Shen, “Free launch: optimizing GPU dynamic kernel launches through thread reuse,” in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 407–419.

[41] Jason Jong, Kyu Park, Ann Arbor, Yongjun Park, and Ann Arbor, “Chimera : Collaborative Preemption for Multitasking on a Shared GPU,” in *ACM SIGARCH Computer Architecture News*, 2015, vol. 43, no. 1, pp. 593–606.

[42] AMD, “Graphics Core Next Architecture, Generation 3,” 2016. [Online]. Available: http://developer.amd.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf. [Accessed: 16-Aug-2019].

[43] Jiwei Liu, “Efficient Synchronization for GPGPU,” 2018.

[44] Kshitij Gupta, and Jeff Stuart, “A Study of Persistent Threads Style Programming Model for GPU Computing,” in *Nvidia GTC*, 2012.

[45] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter Mccardwell, Alejandro Villegas, and David Kaeli, “Hetero-Mark, A Benchmark Suite for CPU-GPU Collaborative Computing,” in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[46] Mehmet E. Belviranli, Peng Deng, Laxmi N. Bhuyan, Rajiv Gupta, and Qi Zhu, “PeerWave: Exploiting Wavefront Parallelism on GPUs with Peer-SM Synchronization,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 25–35.

[47] Jeff A. Stuart, and John D. Owens, “Efficient Synchronization Primitives for GPUs,” in *arXiv preprint arXiv:1110.4623*, 2011, p. 13.

[48] Shucai Xiao, and Wu Chun Feng, “Inter-block GPU communication via fast barrier synchronization,” in *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, 2010, pp. 1–11.

[49] Tyler Sorensen, Hugues Evrard, and Alastair F. Donaldson, “GPU schedulers: how fair is fair enough?,” in *29th International Conference on Concurrency Theory (CONCUR 2018)*, 2018, no. 23, pp. 1–17.

[50] Amir Ali Abdolrashidi, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi Narayan Bhuyan, and Daniel Wong, “WIREFRAME : Supporting Data-dependent Parallelism through Dependency Graph Execution in GPUs,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 600–611.

[51] Anthony Gutierrez, Bradford M. Beckmann, Alexandru Dutu, Joseph Gross, John Kalamatianos, Onur Kayiran, Michael Lebeane, Matthew Poremba, Brandon Potter, Sooraj Puthoor, Matthew D. Sinclair, Mark Wyse, Xianwei Zhang, Akshay Jain, and Timothy G. Rogers, “Lost in Abstraction : Pitfalls of Analyzing GPUs at the Intermediate Language Level,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 608–619.

[52] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson, “GPU Concurrency: Weak Behaviours and Programming Assumptions,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS ’15*, 2015, vol. 43, no. 1, pp. 577–591.

[53] 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*, 2017, pp. 161–174.

[54] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve, “Efficient GPU synchronization without scopes: Saying no to complex consistency models,” in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2015, vol. 05-09-Dece, pp. 647–659.

[55] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve, “HeteroSync: A benchmark suite for fine-grained synchronization on tightly coupled GPUs,” in *Workload Characterization (IISWC), 2017 IEEE International Symposium on*, 2017, pp. 239–249.

[56] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood, “Heterogeneous-race-free Memory Models,” in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 427–440.

[57] Wilson Wai Lun Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt, “Hardware Transactional Memory for GPU Architectures,” in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 296–307.

[58] Tong Li, Alvin R. Lebeck, and Dan J. Sorin, “Spin detection hardware for improved management of multithreaded systems,” in *Parallel and Distributed Systems, IEEE Transactions on*, 2006, vol. 17, no. 6, pp. 508–521.

[59] Mithuna Thottethodi, Alvin R. Lebeck, and Shubhendu S. Mukherjee, “Self-tuned congestion control for multiprocessor networks,” in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 107–118.

[60] NVIDIA, “CUDA Instruction Set Reference,” 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-ref. [Accessed: 16-Aug-2019].

[61] Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood, “Fine-grain task aggregation and coordination on GPUs,” in *Proceedings - International Symposium on Computer Architecture*, 2014, pp. 181–192.

[62] Benedict R. Gaster, and Lee Howes, “Can GPGPU programming be liberated from the data-parallel bottleneck?,” in *Computer*, 2012, vol. 45, no. 8, pp. 42–52.

[63] J. Lawrence Carter, and Mark N. Wegman, “Universal Classes of Hash Functions,” in *Journal of Computer and System Sciences*, 1979, vol. 18, no. 2, pp. 143–154.

[64] Myung Kuk Yoon, Keunsoo Kim, Sangpil Lee, Won Woo Ro, and Murali Annavaram, “Virtual Thread : Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 609–621.

[65] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram, “GPU register file virtualization,” in *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*, 2015, pp. 420–432.

[66] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenw, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B. Gibbons, and Onur Mutlu, “Zorua: A holistic approach to resource virtualization in GPUs,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–14.