EFFICIENT SYNCHRONIZATION FOR GPGPU

by

Jiwei Liu

B.S., Zhejiang University, 2010

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Ph.D. in Electrical Engineering

University of Pittsburgh

2018

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Jiwei Liu

It was defended on

June 12, 2018

and approved by

Jun Yang, Ph.D., Professor, Department of Electrical and Computer Engineering

Rami Melhem, Ph.D., Professor, Department of Computer Science

Youtao Zhang, Ph.D., Associate Professor, Department of Computer Science

Kartik Mohanram, Ph.D., Associate Professor, Department of Electrical and Computer Engineering

Heng Huang, Ph.D., Professor, Department of Electrical and Computer Engineering

Wei Gao, Ph.D., Associate Professor, Department of Electrical and Computer Engineering

Dissertation Director: Jun Yang, Ph.D., Professor, Department of Electrical and Computer Engineering

EFFICIENT SYNCHRONIZATION FOR GPGPU

Jiwei Liu, PhD

University of Pittsburgh, 2018

High-performance General Purpose Graphics processing units (GPGPUs) have exposed bottlenecks in synchronizations of threads and cores. The massively parallel computing cores and complex hierarchies of threads present new challenges for synchronizations at different granularities. Performance of GPU is hindered by inefficient global and local synchronizations. I propose hardware-software cooperative frameworks for efficient synchronization of GPGPU to address the following issues.

To provide efficient global synchronization (Gsync), an API with direct hardware support is proposed. The GPU cores are synchronized by an on-chip Gsync controller. Partial context switch is employed to guarantee deadlock-free execution. The proposed Gsync avoids expensive API calls and alleviates data thrashing. Prioritized warp scheduling is used to increase the overlap of context switch with kernel execution.

To efficiently exploit the inherent parallelism of producer-consumer problems, a flexible wait-signal scheme is proposed at thread-block level. I propose dedicated APIs to express fine-grained static and dynamic dependencies with hardware support. The proposed scheme can accelerate wavefront, graph and machine learning applications. The architectural design of on-chip wait-signal controller eliminates busy wait loop and long-latency memory operations. I also propose thread block dispatch scheduling to address the problem of load imbalance and large context switch overhead.

To reduce stall due to synchronizations, a synchronization-aware warp scheduling is proposed to coordinate multiple warp schedulers upon synchronization events. Both performance and hardware utilization are improved by resolving the barrier sooner.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

Graphics Processing Units (GPUs) were originally designed to accelerate computer graphics and vision related tasks such as texture mapping [3], ray tracing [4], vertex shading [5] and etc, which are commonly used in video games and 3D rendering [6]. Dating back to 1995, the first GPU was introduced as a 3D graphics add-in chip [7]. During the early 2000s, the major goal of GPU design is how to deliver more polygons with high frame rates to increase the realism of 3D rendering [6]. The nature of geometric and graphics computation is highly parallel and heavily floating-point intensive. The raw throughput requirement of such operations impose a vast pressure on the traditional general-purpose CPU processors. Inspired by the good performance of GPU, scientific researchers and software developers started looking for ways to apply GPUs beyond computer graphics to more general purpose computing.

Programming GPUs used to be notoriously difficult for any simple tasks other than graphics applications [6]. Programmers have to be aware of the low-level hardware details such as the fixed graphics pipelines. Programming languages and interfaces were also limited to graphics-centered APIs such as OpenGL [8], DirectX [9], C for Graphics (CG) [10].

Numerous efforts have been made in the research and industry community to improve GPU programmability. The most successful projects include Compute Unified Device Architecture (CUDA) [11] form NVIDIA and Open Computing Language (OpenCL) [12]. Both CUDA and OpenCL are general-purpose programming models for parallel computing. CUDA is vendor-specific model which is available only to NVIDIA GPUs and accelerators while OpenCL is an open cross-platform standard for all vendors which supports not only GPUs but also CPUs, digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other hardware accelerators [12, 13]. CUDA and OpenCL support C as well as

other high level programming languages. With widely uptake of CUDA and OpenCL within the academic and industry community, GPU is turned from graphics-centric platform into general-purpose usage, hence the term general-purpose GPU (GPGPU) computing.

Nowadays GPGPU is becoming increasingly popular for a wider range of applications including machine learning, data mining, computational biology and many more scientific and commercial applications. Deep learning and deep neural networks (DNNs) have recently gained attention due to their superior performance and ground-breaking results in various application domains such as computer vision [14], speech recognition [15], and natural language processing [16]. GPGPU is the most successful hardware to accelerate DNN applications [14, 17, 18, 19], including both training and inference, thanks to its tremendous compute horsepower and highly optimized software libraries such as cuDNN [17], tensorflow [20], caffe [21] and so on.

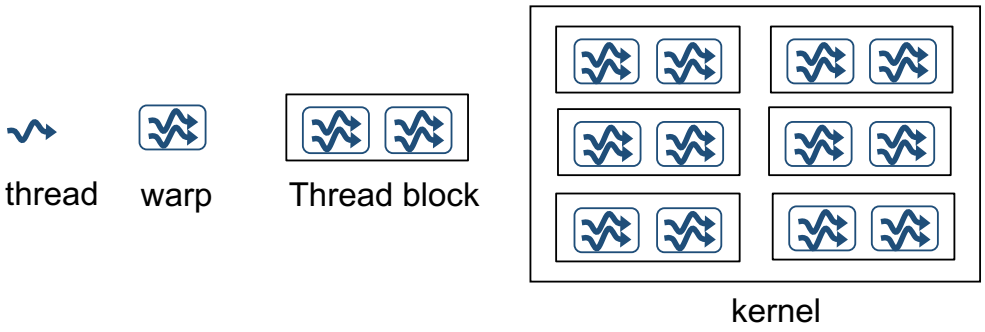

Figure 1: Thread hierarchy of GPU programs

A typical GPU program consists of multiple kernels of thousands of threads. The thread hierarchy is shown in Figure 1. These threads are organized in thread blocks, or TB in short [11, 22] (also known as work groups [23]). A kernel is simply a grid of TBs. Within a TB, a group of threads that are executed in lockstep and scheduled as a unit is referred to as a warp or a wavefront. A warp consists of 32 threads on NVIDIA hardware [11] and a wavefront consists of 64 threads in AMD hardware [23]. As shown in Figure 2, a GPU consists of Streaming Multiprocessors (SM) (also known as Compute Units) with their own shared memories and L1 data caches. Each SM has one or more independent warp schedulers that can quickly switch contexts between threads and issue instructions from different ready

warps. To initialize a kernel, the TB dispatcher dispatches TBs to SMs until all SMs reach their maximum capacities [22]. Then the warp scheduler within a SM picks a ready warp to execute in each cycle. The warp is selected based on a warp scheduling policy such as the round robin policy (RR).

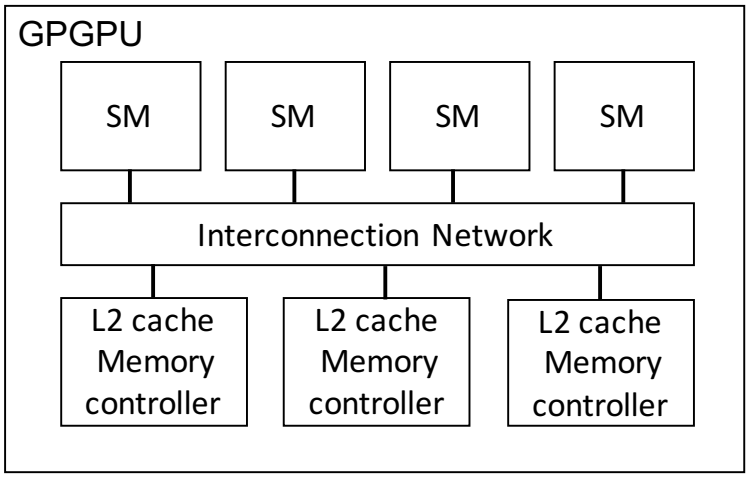

Figure 2: General GPGPU architecture. SM: Streaming Multiprocessor

In efficient parallel algorithms, threads cooperate and share data to perform collective computations. To share data, the threads must synchronize. The granularity of sharing varies from algorithm to algorithm, so thread synchronization should be flexible. Historically, programming models for GPU, including CUDA and OpenCL, have provided a single and simple construct for synchronizing cooperating threads: a barrier across all threads of a thread block, as implemented with the *syncthreads* or *barrier* function. Such barrier synchronization is a point in a program where a thread may not proceed further until other parallel threads, in this case threads from the same TB, reach this point. On the other hand, synchronization of TBs or threads from different TBs are not allowed within the kernel. The only operation to achieve such synchronization is to terminate the current kernel and launch the kernel again. All threads of the kernel are synchronized at the kernel boundary hence a global synchronization of all threads are realized. This shows a fundamental design philosophy of current GPU synchronization. By not allowing threads in different TBs to perform barrier synchronization with each other, GPU run-time system does not need to deal with

any constraint while executing different TBs. In other words, TBs can execute in any order relative to each other since none of them need to wait for each other. This flexibility enables transparent scalability [11, 22]. Without changing the code, the performance of the program scales naturally with different hardware configurations (different number of cores).

However, programmers often need to define and synchronize groups of threads larger or smaller than thread blocks in order to enable greater performance, design flexibility, and exploit finer parallelism. Therefore hardware and architectural modifications are necessary for efficient, flexible and reliable synchronization mechanisms.

1.1 THE CHALLENGES FOR GPGPU SYNCHRONIZATION

1.1.1 Inefficient Global Synchronization

Current GPU global synchronization (Gsync) across the entire grid of threads is inefficient. Gsync is typically achieved by slicing a kernel at the synchronization points, often leading to repeated kernel launches [11, 12]. Such an approach incurs overhead due to API calls and data thrashing. Previously, these overhead are deemed acceptable since many parallel processing have long-latency kernels which outweighs those overheads. However, with the recent development of real-time deep learning applications such as image recognition in self-driving cars [24] and speech recognition [15] in instantaneous speech to text (STT), the response time of inferring one image or one sentence (not training) is crucial. In those scenarios, each kernel finishes in several hundreds of microseconds, which makes kernel launch Gsync-related overhead more pronounced. Existing software workarounds such as atomic operations and memory flags may lead to deadlock when the GPU cannot simultaneously host all the threads [25].

1.1.2 Lack of Wait-Signal Support

GPU performance on applications with irregular parallelism and producer-consumer dependencies has suffered from a limited support for synchronization primitives. For these prob-

lems, global synchronization unnecessarily involves all the thread blocks. Existing software partial synchronization across thread blocks use atomic operations or memory flag-arrays, which rely on busy-wait loops that can cause underutilization of hardware resources and may lead to deadlock for large data sizes [26]. Specialized Warps [27] present a hardware solution that achieves partial synchronization among the warps of a thread block but do not extend to synchronization between thread blocks. Cooperative groups [28] can also synchronize thread blocks but are not flexible and efficient enough to express fine-grained dependencies. Heterogeneous system architecture (HSA) and Asynchronous Task Management Interface (ATMI) [13, 29] implement wait-signal at kernel level with dedicated APIs and task queues, which is less effective utilizing GPU hardwares compared to thread block (TB) level wait-signal schemes. Dynamic parallelism [11, 13] is also a kernel-level optimization for data dependent workloads but is not efficient for applications with static dependencies due to excessive launching of children kernels for small tasks.

One classic application to demonstrate the importance of wait-signal mechanism is Recurrent neural network (RNN). RNNs are an important family of deep learning models which can learn the sequential pattern of input data. Multi-layer RNNs have two unique properties: 1) wavefront parallelism across layers and time steps and 2) reusing weights over multiple time steps during inference. Previous GPU-accelerated RNNs exploit these two optimization opportunities separately. Specifically, the wavefront synchronization is realized with kernel-level APIs such as *hsa_signal_wait_acquire* or *cudaStreamSynchronize*. Preserving the weights using on-chip registers are achieved by persistent thread models at the thread block level which rely on global synchronization barriers to enforce the dependencies.

1.1.3 Synchronization Oblivious Warp Scheduling

Traditional warp scheduling policies of GPGPU have been optimized to increase memory latency hiding capability [30], improve cache behavior [31], and alleviate branch divergence [32]. However, warp scheduling policies do not account for the synchronization behavior among warps under the presence of multiple warp schedulers. A warp may stall the cores due to a thread incurring a long latency operation. Hence, GPGPU SMs can switch to

different warps to continue execution and stay active. On every cycle, a scheduler makes the decision on which warp to issue next. An uninformed execution order of warps may create unnecessary scheduling stalls and SM idle times. Barrier synchronizations, which are used to coordinate execution among threads, are adopted in a wide range of general purpose applications. The most common barrier is the TB-level barrier, where all threads of a TB have to arrive at a single point before any of them can proceed. A TB is assigned entirely to one SM. I made a key observation that the warps within a TB are distributed evenly to each scheduler of an SM to balance the load. Hence, each scheduler only sees part of one TB, and performs scheduling independently of other schedulers. Their scheduling decisions and order are also entirely different. When synchronization barriers are present, hitting the same barrier on different schedulers could be very far apart in time, causing earlier warps to stall for a long time waiting for the latest warp to clear out the barrier. None of the existing warp schedulers address this issue due to their synchronization unawareness.

1.2 THESIS OVERVIEW

The problems to be solved are summarized in Figure 3. Figure 3(a) shows the global synchronization where all threads and TBs are synchronized at a global barrier. Since the number of TBs can exceed the capacity of GPU, synchronizing all TBs requires marking the status of each TB and context switch out and in TBs appropriately. Figure 3(b) shows the problem of implementing wait-signal scheme at TB level. Producers and consumers are placed in different TBs. Consumer TBs should be blocked until a signal is received from the producer TB it depends on (the red arrow in Figure 3(b)). Figure 3(c) shows the problem of synchronization oblivious warp scheduling. The warps of the same TB are split and distributed to each warp scheduler. If a within-TB barrier is used in the kernel, the barrier will introduce additional delay since the two warp schedulers do not communicate with each other. To solve these problems, I propose several architectural techniques as summarized in Table 1 with the major challenges for each scheme. For global synchronization, I propose an efficient hardware support to synchronize thread blocks globally within the kernel. A

hardware counter is used to manage a global barrier across thread blocks and partial context switch is leveraged to avoid deadlock. Moreover, we design several techniques to reduce the context switch overhead which is critical to performance improvement. The proposed scheme outperforms repeated kernel launch when 1) data reuse is significant across synchronization points and/or 2) the latency of the API calls is relatively long.

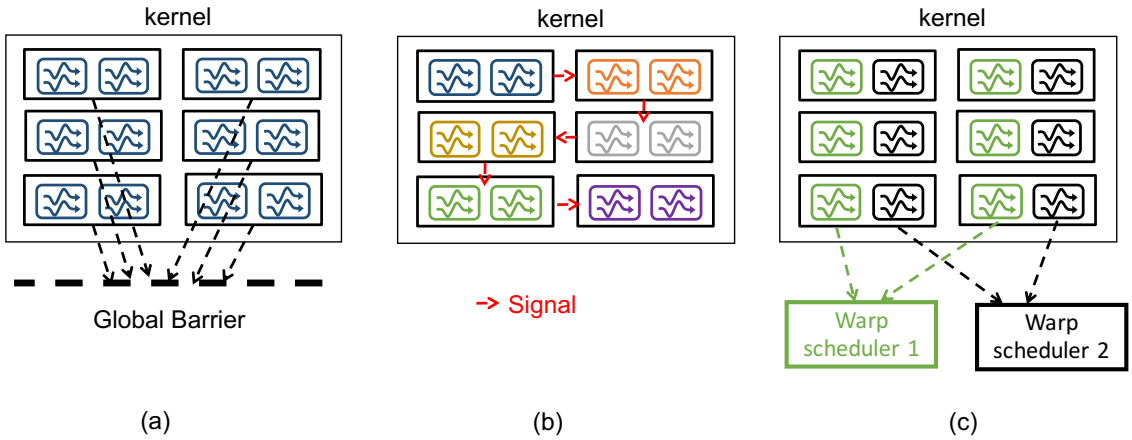

Figure 3: Problem overview: (a) global synchronization (b) wait-signal support (c) Synchronization oblivious scheduling

Table 1: Overview of proposed schemes and challenges

#	**Scheme**	**Challenges**
1	Global Synchronization	Large contexts to save
2	TB-Level Wait-Signal	Complex producer-consumer patterns
3	Synchronization-Aware Warp Scheduler	No cooperation of warp schedulers

For wait-signal support, I propose a hardware-software cooperative framework to achieve Wait-Signal mechanism between thread blocks in GPGPUs to efficiently exploit the inherent parallelism of producer-consumer problems. The proposed scheme guarantees deadlock-free execution and improves load balancing. The global thread block dispatcher is augmented with a wait-signal controller to manage event counters and synchronization of thread blocks.

To avoid busy wait loops, the "Wait" instruction blocks a TB and releases its computing resources for use by other TBs. The happen-before relationships of "Signal" and "Wait" is resolved by allowing both wait and signal instructions to increment an event counter while only wait instructions are blocking. The proposed wait-signal mechanism is flexible to support both static and dynamic producer consumer dependencies. In order to achieve load balancing and avoid unnecessary context switch, we propose a kernel configuration extension to initialize the order of TB dispatching based on the dependencies implied by the applications.

To accelerate RNNs as a representation of producer-consumer problems, we propose a novel wait-signal mechanism at the thread block (TB) level to accelerate multi-layer RNNs. The multi-layer RNN is launched in a single kernel where weights are loaded in registers only once and persist for all the unfolded time steps. The weights of each layer are statically allocated to a specific group of TBs that collaborate to accomplish one computing task for the current time step (namely matrix multiplication). After synchronizng using a local barrier, the TB group assigned to a given layer signals the TB groups assigned to the next layer and waits for the results from the previous layer before proceeding with the computation for the next time step. The local barrier and the wait signals are implemented by lock-free memory flags. With the proposed TB-level wait-signal mechanism, the wavefront parallelism is achieved and unnecessary memory reloading of weights is avoided. We implemented the proposed wait-signal mechanisms on the latest GPUs and evaluated the speedups using a number of widely used RNNs. On average, the proposed scheme achieves 37% speedup over the state-of-the-art persistent RNNs.

To improve performance of within-TB synchronization in the context of multiple warp schedulers, I propose a warp scheduling algorithm that is synchronization aware. The key observation is that excessive stall cycles may be introduced due to synchronizing warps residing in different warp schedulers. We propose that schedulers coordinate with each other to avoid warps from being blocked on a barrier for overly long. Such coordination will dynamically reorder the execution sequence of warps so that they are issued in proximity when a barrier is encountered.

1.3 CONTRIBUTIONS

1.3.1 Efficient GPU Global Synchronization with Light Weight Context Switch

I propose a new global synchronization (Gsync) function at the GPU-side, which applies to both conventional SPMD and persistent programming styles. The existing SM and TB dispatcher is augmented to manage GS and partial context switches of the TBs. I also propose effective techniques for reducing the overhead of context switching to achieve scalability. The experimental results show that the proposed scheme outperforms existing software approaches in three typical type of applications with up to 2x speedup over CPU-driven synchronization. I make the following contributions.

- An extensive study on the effectiveness and trade-offs of the existing software approaches for Gsync in both CUDA and OpenCL
- efficient approach to explicitly support global synchronization using partial context switch to avoid deadlocks. The proposed approach improves both performance and programmability.
- Case studies of representative applications, including scientific computing, graph traversal and machine learning, to demonstrate how to use the proposed Gsync function call and benefit from its performance advantages.

1.3.2 Accelerate RNNs with software Wait-Signal

I propose a novel wait-signal mechanism at the thread block (TB) level to accelerate multi-layer RNNs. First, we implement the *wait*() and *signal*() functionality via lock-free memory flags [25]. A TB that calls a *wait*() function will be stalled until its dependent TBs execute a *signal*() function. With such fine-grained synchronization, the theoretical maximum wavefront parallelism can be achieved. The proposed scheme also improves the computation efficiency and the reuse of shared memory. Specifically, in the proposed scheme, the weights of a certain layer are partitioned among TBs, which is different from the state-of-the-art persistent RNN scheme[33], where the weights of the entire neural network are partitioned among TBs. I make the following contributions:

- A study of the potential for accelerating RNNs with both wavefront parallelisms and on-chip storage of weights.
- The design of an efficient wait-signal functionality at the TB-level using memory flags. This increases both parallelism and hardware occupancy for RNN implementations.
- Experimentation with different representative RNN applications to demonstrate the impact of key hyper parameters such as number of layers, feature dimensions and sequence length on the proposed design.

1.3.3 Thread Block Level Wait and Signal in GPU with Hardware support

I propose a Wait-Signal mechanism with hardware support for local synchronization of thread blocks in GPGPUs. Each Streaming Multiprocessor is augmented with wait and signal buffers and extend the global thread block dispatcher with a wait-signal controller to manage event counters. The proposed scheme can be integrated with context switch engines that were previously proposed for swapping TBs when all the TBs of a kernel cannot fit simultaneously on the GPU. I also propose a mechanism that allows applications to control the order of TB dispatching, thus eliminating the need for context switching in single pass wavefront application kernels as long as the TBs constituting an active front can fit on the GPU.

The proposed mechanism can support one-to-one, one-to-many, many-to-one, and many-to-many producer consumer dependencies. Cycle-accurate simulation shows that, on average, the proposed scheme achieves 2.0x speedup over repeated kernel launch and 31% speedup over the best-known software memory flags techniques. I make the following contributions.

- An extensive study of the effectiveness and trade-offs of the existing approaches for producer-consumer relationships.
- An efficient approach to explicitly support Wait and Signal in hardware to exploit fine-grained parallelism, guarantee deadlock-free execution and improve load balancing.
- Case studies of representative applications executing wavefront and graph algorithms.

1.3.4 Synchronization Aware GPGPU Warp Scheduling

I propose Synchronization Aware GPGPU Warp Scheduling (SAWS) to tackle warp scheduling to mitigate performance loss due to synchronization events in SMs with multiple schedulers. Current designs treat each scheduler independently, which works well without synchronization barriers, but exposes deficiency with barriers. I find that multiple schedulers, when operating independently, will delay the clearance of barriers and create excessive stall cycles to the warps. Hence, I propose to coordinate among multiple schedulers and prioritizes warps as a reaction to synchronization events. In addition, I find that maintaining a uniform barrier-clearing rate during prioritization brings additional performance benefit.

Our proposed scheme SAWS is so simple that integrating it with existing scheduler does not require much additional hardware. The evaluation shows that SAWS can speed up barrier clearance by 15% and total execution by 10% on average, when compared with the state-of-the-art warp scheduler. Finally, SAWS shows its increasing effectiveness when the number of scheduler grows, proving that it is a scalable design. Specifically I make the following contributions.

- A detailed analysis of performance loss due to mishandling of synchronization barriers with multiple warp schedulers.

- A simple synchronization aware warp scheduling algorithm that coordinates multiple schedulers to minimize barrier waiting time and synchronization-induced stalls.

- A design that can be seamlessly integrated with existing warp scheduler so that synchronization-rich kernels can be well handled while not harming synchronization-free kernels.

- A simulation evaluation of performance advantages of the proposed scheduler over existing well-studied warp schedulers.

1.4 THESIS ORGANIZATION

The rest of this thesis is organized as follows. Chapter 2 presents preliminary information of GPGPU. Chapter 3 discusses the related work. The efficient global synchronization is described in Chapter 4. Software implementation of wait-signal to accelerate RNNs are introduced in Chapter 5. Chapter 6 illustrates the thread-block level wait-signal scheme. The synchronization-aware warp scheduling is explained in Chapter 7. Chapter 8 describes the future research directions. Chapter 9 concludes the thesis.

2.0 BASICS OF GPGPU

2.1 GPGPU ARCHITECTURE

General Purpose Graphic Processing Units (GPGPUs) have been evolved actively in recent years with increasing number of cores, higher clock frequency and high-bandwidth high-capacity on-board GPU memory. For example, the state-of-the-art Nvidia Pascal GPU architecture is shown in Figure 4 [2]. The SMs are grouped into Texture Processing Clusters (TPCs) and Graphics Processing Clusters (GPCs). The global GigaThread controller manages TB dispatching, application context switching and also provides a pair of streaming data-transfer engines, each of which can fully saturate GPU's PCI Express host interface.

Each memory controller is attached to 512 KB of L2 cache, and each High Bandwidth Memory (HBM) DRAM stack is controlled by a pair of memory controllers. The full GPU includes a total of 4096 KB of L2 cache. L1 cache can serve as a texture cache depending on workload. The unified L1/texture cache acts as a coalescing buffer for memory accesses, gathering up the data requested by the threads of a warp prior to delivery of that data to the warp.

2.2 STREAMING MULTIPROCESSOR ARCHITECTURE

The most critical component of GPU is the streaming multiprocessor, or SM in short. As shown in Figure 5, Each SM consists of many simple scaler processors (SP) for common simple calculations such as addition and several special function units (SFUs) for complex calculations such as exponential operations. There are abundant of registers on the SM

private to each thread. There is also a L1 data cache which can cache global memory access shared by all threads on the SM. Different from CPU, the capacity of registers (96 KB) on the SM is larger than that of the L1 cache (48 KB). The shared memory is a special scratch memory that is programmable and to be shared by threads from the same TB. In previous GPU architecture, shared memory and L1 cache are physically the same device and can be configured/split by programmers. Recent GPU architecture provides dedicated independent shared memory and L1 cache.

2.3 WARP SCHEDULING

As shown in Figure 5, there are multiple warp schedulers in one SM. The number of warp schedulers range from two to four.

When a kernel is dispatched to a GPGPU, a global TB dispatcher assigns TBs to SMs until all SMs are saturated. An SM may be assigned more than one TB at a time, since the size of a kernel and its TBs may vary. An SM which finishes a TB will get a new one from the global dispatcher, until all TBs of a kernel finish. Hence, the execution latency of a kernel is dependent on how quickly the SMs complete their TBs.

As the core count increases in the latest generations of GPGPUs, each SM may contain one, two or four warp schedulers. When a TB is assigned to an SM, its warps are further assigned to those warp schedulers in a round robin fashion in order to balance the warp distribution, as shown in Figure 30. During execution, each warp scheduler selects a ready warp to execute, allowing one, two or four warps to issue and execute concurrently. A warp can become blocked due to a long latency operation such as a load from the memory or a barrier synchronization. The warp scheduler switches context to another ready warp, if such warp can be found, to mask the stalled cycles due to long latency operations. If no such warps can be found, the scheduler stalls, causing resource underutilization. In fact, how the warps are selected to execute determines the resulting number of stalled cycles. Hence, effort has been put on optimizing the scheduler to mask or reduce stall cycles.

2.4 GPU SYNCHRONIZATION PRIMITIVES

2.4.1 Global Synchronization

Figure 7 shows host (CPU) code examples where kernels are sequentialized by either implicit or explicit CPU-driven synchronizations. As such programming style typically puts kernels and synchronizations in loop bodies, we term this synchronization method "repeated kernel launching". In Figure 7(a), the kernel `stencil` is invoked N times (line 3) within the inner for loop and then the termination condition (convergence in this case) is checked in line 8 of the outer *while* loop[34]. Each kernel launch calculates the new values for all data points based on the results of the previous kernel. Other stencil-like applications, such as solving Laplace differential equations, also follow a similar coding style [34].

Global synchronization is needed between kernel launches, and before the convergence check. In the for loop, since no CUDA streams are specified, all kernels launched belong to a default NULL stream [11]. Kernels in the NULL stream are guaranteed to execute sequentially, which is one of the most common type of implicit global synchronization [11]. Explicit CPU-driven synchronization is also used in this example, by invoking `cudaDeviceSynchronize()` at line 6. This call is necessary because kernel launch is asynchronous with respect to host code [11]. If `cudaDeviceSynchronize()` is not used, the convergence checking at line 8 will be executed immediately after launching N kernels and before the kernels finish execution, thus giving incorrect results.

2.4.2 Within-TB synchronization

GPU programming models also provide explicit within-TB synchronization instructions such as *syncthreads*() in CUDA or *barrier*() in OpenCL. This instruction is used to synchronize all the threads in the same TB.

Many applications, including graphic and general purpose applications, adopt synchronizations to coordinate among multiple threads and guarantee execution correctness when parallelized onto a GPGPU. Within a TB, threads may operate on shared data but make progress at different rate and arrive at different phases, e.g. read or write phases, at differ-

ent times. The synchronization barrier ensures that all threads in the same TB complete one phase before moving on to the next. In the CUDA programming model, for example, shared data may be first loaded into the fast on-chip shared memory of each TB, which is the read phase. Then, computing starts by accessing the shared memory, which is the compute phase. Finally, the shared data is written back to the main memory, which is the write phase. Within-TB synchronization barriers must be used between adjacent phases to guarantee that the read and write sequence of the shared data is correct. Figure 8 illustrates such a common programming paradigm.

Figure 4: GP100 Pascal GPU architecture [2].

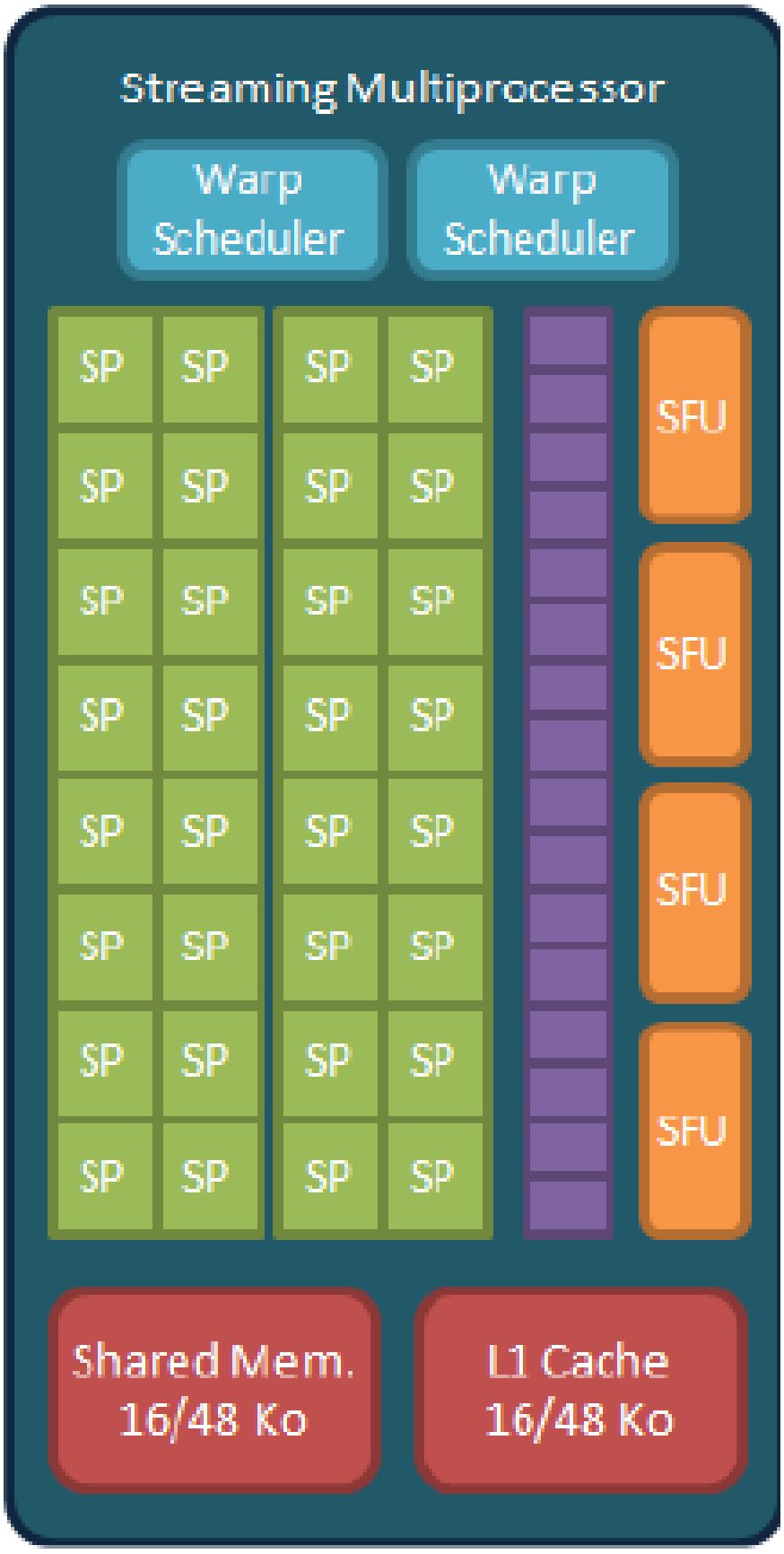

Figure 5: Architecture of Streaming Multiprocessor.

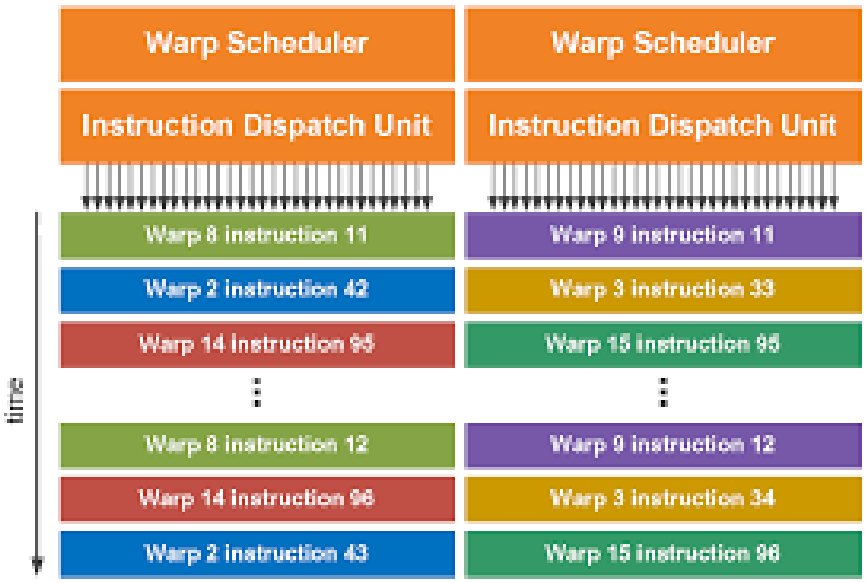

Figure 6: Warp scheduling: warps from the same TB are split.

```
1.    do{
2.        for(int i=0;i<N;i++){
3.            stencil<<<...>>>(...);
4.            // implicit synchronization
5.        }
6.        cudaDeviceSynchronize();
7.        // explicit synchronization
8.        notdone=Check_termination();
9.    } while(notdone)
```

(a)

```
1.  // layer i
2.  im2col <<<...>>>(...);
3.  gemm<<<...>>>(...);
4.  batch_normalization<<<...>>>(...);
5.  scale<<<>>>();
6.  add_bias<<<>>>();
7.  activation<<<...>>>(...);
```

(b)

Figure 7: Host code examples of CPU driven synchronizations through repeated kernel launch. (a) stencil computation. (b) convolutional neural network.

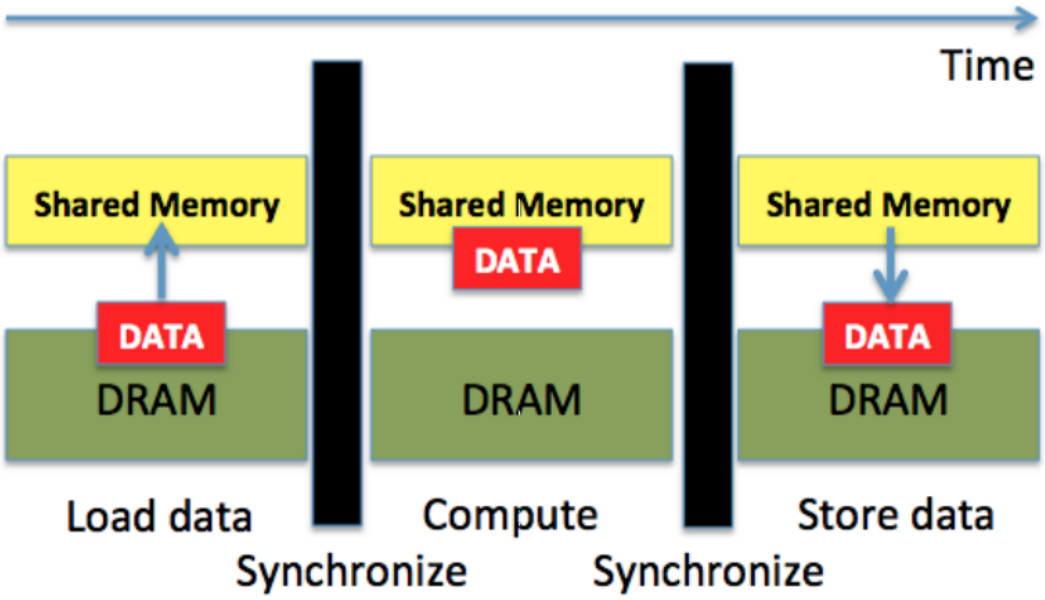

Figure 8: Programming patterns with the within-TB synchronization.

3.0 RELATED WORKS

3.1 ADDRESSING GLOBAL SYNCHRONIZATION

Many applications require global synchronization (Gsync) to coordinate among parallel tasks, which is typically a point (barrier) in a program where a thread may not proceed further until all other parallel threads reach this point. Currently, GPUs do not support explicit implementation of Gsync. There are mainly two workarounds that have been commonly used: 1) split the program into separate kernels where Gsync is needed [11, 25], which we term "repeated kernel launching" as the Gsync and split kernels are typically in loop bodies; 2) use atomic operations or memory flags similar to those used in a CPU [25]. However, both workarounds could degrade performance or lead to deadlock.

3.1.1 Dynamic Parallelism

One variant of repeated kernel launching introduced before is *dynamic parallelism* (DP) [35, 36, 37] which differs from repeated kernel launching in that kernels are launched from within the GPU. A master kernel launches a sequence of children kernels with in-GPU `cudaDeviceSynchronize()` in between. Hence, DP doesn't yield control from the GPU to the CPU. The overheads of DP include kernel queuing, dispatching and context setup when launching thread blocks [35, 36]. Further, when DP launches many small children kernels, serious slowdown occurs due to low hardware utilization and limitation of call stacks, as well as other factors [35, 38]. Such overhead can be mitigated by kernel aggregation [35, 38]. Heterogeneous System Architecture (HSA) with support of OpenCL 2.0 also implements an efficient dynamic parallelism which does not suffer from call stack limitations [39].

3.1.2 Atomic Operations and Memory Flags

Using atomic mutex at the GPU side is one way to avoid splitting a kernel. An atomic function performs a read-modify-write operation on a variable residing in global memory [11, 40]. The basic idea is to use a global mutex variable to count the number of TBs that invoke global synchronization. All TBs busy-check this counter in the memory until it reaches a predefined value. Global memory atomics are long latency operations which are at least 10x slower than atomic operations using on-chip local memory on AMD Radeon GPU [40]. More efficient atomic operations are available. For example, Global Data Share (GDS) of AMD GCN [41] is a special global memory which can utilize atomic operation to implement a fast global barrier. Another option is to implement a lock-free global barrier with memory flag arrays to achieve the same purpose but with better performance than global memory atomics [25].

In either implementation, the kernel is launched once and iterates at the GPU side with one global synchronization at the end of each iteration. Hence, the CPU API call and data thrashing overheads are removed. However, those mechanisms are prone to deadlocks because they require that all TBs of the kernel be dispatched to the GPU. Otherwise, executing TBs that reach the Gsync point are kept busy checking the global counter and never release their resources. In turn, pending TBs never get a chance to execute and progress to the Gsync point. Thus, executing and pending TBs deadlock waiting for each other causing deadlock[42].

3.1.3 Persistent threads

Persistent thread (PT) is a programming approach where the kernel is persistent on the GPU until no work remains. When input data size increases, the number of TBs in a PT kernel stays the same and more workload is processed by each thread. Therefore, PT style code is fundamentally different from the general parallel programming style where one thread maps to certain data points statically. A PT kernel also uses atomic operations or memory flags to implement Gsync. PT relies on programmers to correctly define the kernel dimension to achieve full occupancy, balance load and avoid deadlock [43, 42]. Since occupancy depends on

the available resource, PT code is not portable from device to device unless the programmer uses careful device profiling and kernel configuration routines. PT also prohibits independent progress of kernels from different streams or queues [13].

Advanced PT programming techniques to optimize the task-flow graph are proposed for specific synchronization problems, usually with compiler design to automatically optimize dependency graphs of the workload at thread block level. Free launch is such a compiler design to realize global synchronization for dynamic irregular workloads [38]. Peer Wave proposed a program model to manually map wavefront problem to regular PT tiles [26]. PT RNN [33] also manually optimize the programming by transforming the computing graph with global barrier only. Although achieving great performance improvement, in general these methods are limited to one specific type of applications. The PT programming style is also difficult and time-consuming to implement and understand. We argue that with architectural design and APIs, a comparable level of performance improvement can be achieved with simple conventional CUDA programming style for a more broad group of applications.

3.1.4 Cooperative Kernels

CUDA Cooperative kernels [11] simplifies the profiling by providing auxiliary APIs to calculate occupancy for a given kernel. $cudaLaunchCooperativeKernel$ accepts the number of SMs or maximum number of TBs on an SM as input arguments to guarantee co-residency of TBs. After a kernel is launched, the $grid.sync()$ is called for global synchronization. Furthermore, such synchronization doesn't invalidate registers and shared memory [28]. Although the underlying implementation has not been made public, the documentation implies that context switch is not supported and the number of TBs in the kernel is statically determined.

3.1.5 Occupancy Discovery Protocol

Alternatively, an occupancy discovery protocol is proposed to dynamically discover a safe occupancy for a given GPU and kernel, allowing for a deadlock-free inter-TB barrier by restricting the number of TBs/workgroups according to this estimate [42]. On the one hand, this solution only applies to the kernel which is agnostic to the number of TBs and can

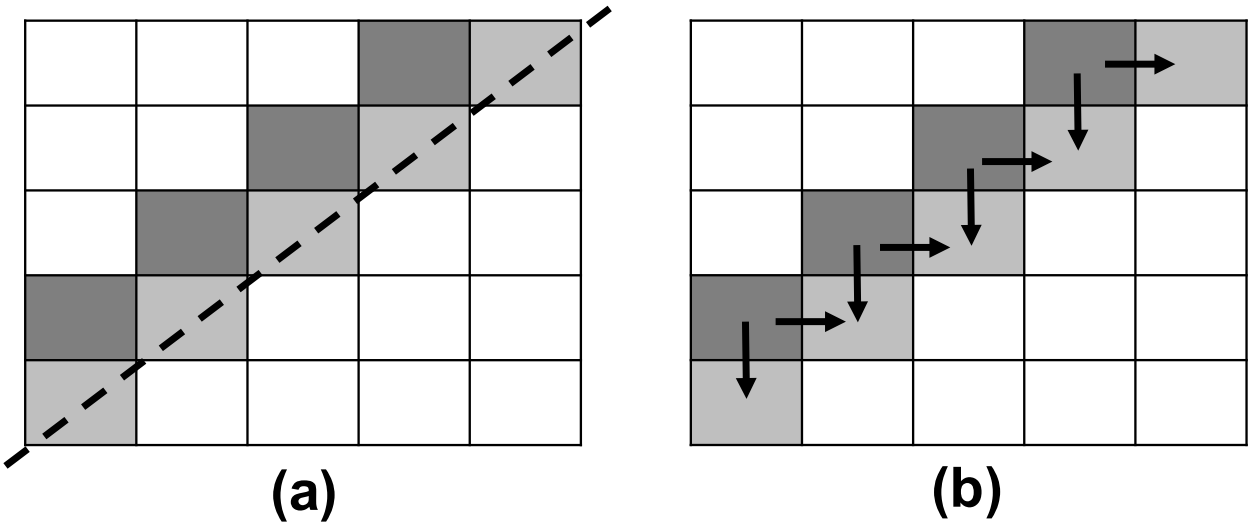

Figure 9: (a) barrier (wavefront) synchronization vs (b) wait-signal synchronization. Computing each tile only depends on the upper and left tiles.

distribute workloads to TBs dynamically like the PT model [42]. On the other hand, it is likely that the occupancy detected is less than the maximum occupancy [42] and underutilizes GPU resources.

3.2 ADDRESSING PRODUCER CONSUMER PROBLEMS

Repeated kernel launch is the most common strategy which enforces dependencies by putting producers and consumers in separate kernels, thus realizing global synchronization at the boundary of kernels [11, 25]. Cooperative groups [28, 11] implements a barrier synchronization for a subset of threads either in the same TB or across different TBs in the same kernel. Both repeated kernel launch and cooperative groups are not flexible enough to express fine-grained producer-consumer dependencies and exploit fine-grained parallelization opportunities like Figure 9.

A barrier synchronizes the current wavefront (darker tiles) before executing any of the lighter tiles. Clearly, a more efficient synchronization should allow each light tile to only wait for its upper and left neighbors. Previous works showed that even an optimized barrier synchronization implementation can result in at least 20% slowdown compared to wait-signal synchronization [26].

3.2.1 Asynchronous Task Management Interface

Heterogeneous system architecture (HSA) defines wait-signal APIs such as *hsa_signal_wait_acquire* [13]. Asynchronous Task Management Interface (ATMI) is a low-level realization on top of HSA for kernel-level wait-signal on HSA-compatible GPUs [29]. ATMI utilizes HSA wait-signal API calls, barrier packets and the underlying hardware support to execute asynchronous tasks of producer-consumer problems. Independent kernels are pushed in separate HSA task queues and executed in parallel. Dependencies are specified by inserting a special barrier packet into the task queue which can have signals associated with it [29]. ATMI achieves 1.6 to 3.3 speedup over repeated kernel launch.

Despite the kernel launch overhead, there are some potential limitations of ATMI. First, Independent kernels have to be in separate task queues to run in parallel. The number of such queues are limited. For instance, there are 24 in AMD FX-8800P APU [29] and 32 in the Nvidia GK110 architecture [36]. This limits the parallelism it could exploit. Second, the number of signals attached to a packet is 5 [29]. To support more than 5 signals, hierarchical barrier packets have to be used which cause additional delay.

3.2.2 Specialized Warps

Specialized warps have been implemented with CUDA named barrier to specify dependencies among warps of a TB [11, 27]. Specifically, two assembly functions are used, bar.sync and bar.arrive, corresponding to wait and signal between a TB's warps. Both functions take two arguments, a barrier_name and a target value (multiple of 32). Each named barrier is associated with a hardware counter in the SM. When a warp issues bar.arrive or bar.sync, the corresponding counter is incremented by 32 (each thread in the warp increment the counter

by 1). Bar.sync is a blocking function while bar.arrive is non-blocking. This design has two important implications: 1) signals are not lost, which means that a signal must be called exactly when it is needed. Issuing more or fewer signals than can be matched with waits will leave the barrier unresolved or incorrectly resolved. 2) happen-before relationships between waits and signals are not necessary. In CUDA, bar.sync and bar.arrive can be issued in any order without affecting correctness as long as the number of issued bar.sync and bar.arrive match. This design is necessary for GPU SIMT execution where warps can be issued in arbitrary orders.

The fine-grained synchronization enabled by named barriers is very expressive for complex dependencies. However, named barriers are only defined within a thread block. Kernel-wide synchronization can only be expressed for a kernel consisting of one TB. This leads to very low SM occupancy since one TB is dispatched to a single SM. In addition, one TB can have no more than 1024 threads which also limits parallelism. In fact, named barrier is mainly used for accelerating sub-tasks such as loading from or storing to memory. They are not meant to be used as the main synchronization mechanism for producer-consumer applications.

3.2.3 Case Study: RNN Acceleration Techniques

Recurrent Neural Networks (RNNs) have shown great performance in many sequential modeling tasks. Different from a traditional neural network, where all inputs and outputs are independent of each other, RNNs are used to model a sequential dependency. For example, if the task is to predict the next word in a sentence, the model must utilize information about words that appeared earlier in the sentence. RNNs are recurrent because they compute every element of a sequence based on the previous computations. Hence RNNs have a "memory" which captures information about what has been calculated so far. The building block of RNN is called a "cell", which is basically a function that takes in an input and a state, and returns an output and the next state [44]. Figure 32 shows how a single RNN cell is unfolded for each time step. The formulas that govern the computation happening in a RNN are described as follows:

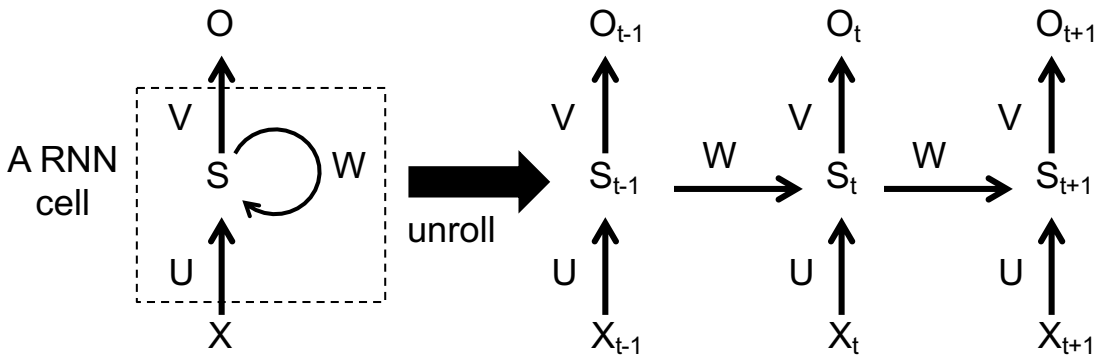

Figure 10: A basic single recurrent neural network cell.

$$S_t = f(\mathbf{S_{t-1}}\mathbf{W} + \mathbf{X_t}\mathbf{U}) \tag{3.1}$$

$$O_t = g(\mathbf{S_t}\mathbf{V}) \tag{3.2}$$

where $\mathbf{X_t} \in \mathbf{R^{B\times P}}$, $\mathbf{S_t} \in \mathbf{R^{B\times H}}$ and $\mathbf{O_t} \in \mathbf{R^{B\times K}}$ are an input matrix, a hidden state matrix and an output matrix at time step t, respectively. B, P, H, K represent batch size, number of input features, number of hidden units and number of output features, respectively. $\mathbf{S_t}$ is the memory of the RNN which is calculated based on the previous hidden state and the input at the current step as shown in formula (3.1). The function f is usually nonlinear, such as *tanh* (the hyperbolic tangent function) or *relu* (the rectified linear unit function). The initial state, $\mathbf{S_{-1}}$, is typically initialized to all zeros or a random number. An important property of the RNN is the reuse of the weights over the time steps. For example, as shown in Figure 32, the weights $\mathbf{W} \in \mathbf{R^{H\times H}}$, $\mathbf{U} \in \mathbf{R^{P\times H}}$ and $\mathbf{V} \in \mathbf{R^{H\times K}}$ are reused multiple times.

We use $\mathbf{U}$ and $\mathbf{W}$ to represent the weights corresponding to inputs and previous hidden states, respectively. The basic RNN has several downsides including vanishing gradients[45] and degraded performance for long sequences [46]. The most successful advancement of

RNN is Long short-term memory (LSTM) networks. LSTM is designed to combat vanishing gradients through a gating mechanism [46] that is governed by the following equations:

$$i_t = \sigma(\mathbf{S_{t-1}}\mathbf{W^i} + \mathbf{X_t}\mathbf{U^i}) \tag{3.3}$$

$$f_t = \sigma(\mathbf{S_{t-1}}\mathbf{W^f} + \mathbf{X_t}\mathbf{U^f}) \tag{3.4}$$

$$o_t = \sigma(\mathbf{S_{t-1}}\mathbf{W^o} + \mathbf{X_t}\mathbf{U^o}) \tag{3.5}$$

$$g_t = tanh(\mathbf{S_{t-1}}\mathbf{W^g} + \mathbf{X_t}\mathbf{U^g}) \tag{3.6}$$

$$S_t = \mathbf{S_{t-1}} \circ \mathbf{f_t} + \mathbf{g_t} \circ \mathbf{i_t} \tag{3.7}$$

$$O_t = \mathbf{S_t} \circ o_t \tag{3.8}$$

where $\circ$ represents element-wise multiplication operations. To improve the performance for modeling long sequences, LSTM utilizes the input gate i_t, forget gate f_t, and output gate o_t to model how much information to incorporate from the current input, how much memory to forget/decay, and how much hidden state to reveal to the output, respectively. All the three gates are applied with sigmoid activation function σ so that the gate outputs work as soft binary masks onto the input X_t and previous hidden state S_{t-1}. Although it looks complicated, LSTM or any other RNN cell is just another way to compute a hidden state $\mathbf{S_t}$ based on the previous state $\mathbf{S_{t-1}}$ and the current input $\mathbf{X_t}$. From this perspective, LSTM is similar to the basic RNN cell with quadruple the number of weight parameters and at least 4 times as many computing steps. Other RNN variants such as gated recurrent unit (GRU) [47] varies in the gate design and number of weight parameters but share the same computing pattern. In summary, RNN cells in general including basic, LSTM and etc can be unified by the following formulas, where different cells differ in the implementation of the parameterized functions F_S and F_O to compute the current hidden state $\mathbf{S_t}$ and output $\mathbf{O_t}$.

$$S_t = F_S(\mathbf{S_{t-1}}\mathbf{W} + \mathbf{X_t}\mathbf{U}) \tag{3.9}$$

$$O_t = F_O(\mathbf{S_t}) \tag{3.10}$$

3.2.3.1 Accelerating Single Layer RNNs As RNNs become larger and deeper, the times for both training and inference rise significantly. Therefore there is a significant incentive to improve the performance and scalability of these networks [33, 48]. While GPUs have become the hardware of choice for training and deploying recurrent models, different implementations often explore a number of basic optimizations. The effect of optimizations may be amplified by exposing parallelism between operations within the network, leading to an order of magnitude speedup over naive implementations across a range of network sizes [49].

There are many ways to naively implement a single propagation step of a recurrent neural network on GPUs. In a basic implementation, each individual computing step (ie. matrix multiplication, sigmoid, point-wise addition, etc.) is implemented as a separate kernel, and kernels are executed sequentially. A widely used optimization is to combine matrix operations sharing the same input into a single larger matrix operation. For example, formulas (3.3)-(3.6) indicate that a LSTM leads to eight matrix matrix multiplications: four operating on the input X_t and four operating on the previous hidden state S_{t-1}. In each group of four, the input is shared. Therefore it is possible to reformulate a group of four matrix multiplications into a single matrix multiplication by concatenating the weights W and U into a single matrix. Because operations involving larger matrices are more parallelisable (and hence are more efficient), this leads to 2x speedup [49]. A similar optimization is possible for other variants of RNNs such as GRU [47]. A very important observation is that the same weights W and U are reused over multiple time steps [33]. In the default implementation described above, the computation at each time step is implemented in a kernel, which does not keep the weights across kernel invocation on-chip (in registers or shared memory). Each kernel has to load these weights again from off-chip memory, resulting in significant latency. Persistent RNN [33] implements the entire network in one kernel and relies on a global barrier, implemented using atomic operations, to synchronize thread blocks. The thread blocks are persistent on the GPU for all time steps achieving up to 10x speedup when the GPU has enough registers to store all the weights of the network.

3.2.3.2 Accelerating Multi-Layer RNNs It is becoming increasingly popular for RNNs to feature multiple recurrent layers for complicated tasks. As shown in Figure 11, layers are stacked such that each recurrent cell feeds its output directly into a recurrent cell in the next layer. In such multi-layer RNNs, it is possible to exploit wavefront parallelism. Specifically, a recurrent network can be considered as a 2D grid of cells, which leads to a diagonal wave of execution propagating from the first cell in the bottom left of Figure 11. Wavefront execution implies that the completion of a recurrent cell not only resolves the dependency on the next iteration of the current layer, but also on the current iteration of the next layer. This allows multiple layers to be computed in parallel, greatly increasing the amount of work that can execute simultaneously on the GPU [49].

To implement such a wavefront parallelism at the kernel level, CUDA streams or HSA task queues can be used to enforce the dependencies and expose as much parallelism as possible. For example, to implement wavefront parallelism for a RNN with L layers and T time steps, at least L separate kernel queues should be used, one queue for each layer. In each kernel queue, a kernel is enqueued and launched T times, one for each time step. Kernel level synchronizations (barrier at kernel boundaries) should be used to enforce the dependencies for different layers and time steps [49]. As indicated earlier, weights used by a kernel will not be saved on-chip when the kernel finishes the computation of the current time step and have to be loaded again by the kernel computing the next time step. Clearly, this implementation leads to significant waste of data movement, causing significant slowdown.

3.2.3.3 Stream implementations of RNNs In these implementations [49], a dedicated kernel is launched for each layer and each time step. The kernel includes matrix multiplication to compute hidden states, fused with all element-wise operations. The kernels in the same layer will be matched into the same kernel queue and launched sequentially. As shown in Figure 12, each block represents a different kernel. To enforce dependencies, the $cudaStreamWaitEvent$ is inserted between any two kernels except the first layer (the "W" block in Figure 12). Such an event will block each kernel in a stream, i until it is signaled. A kernel from the previous layer (stream $i-1$) will signal this wait event when it finishes using $cudaEventRecord$. Hence, wavefront parallelism can be achieved, but the need for event

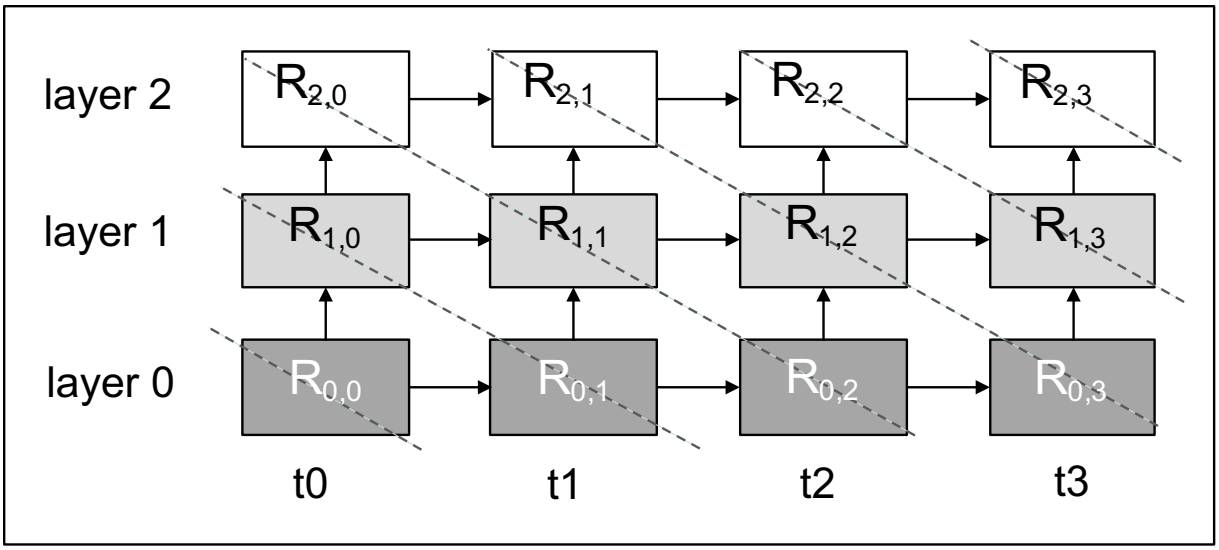

Figure 11: Multi-layer RNNs. The dashed line indicates potential wavefront parallelism. $R_{i,t}$ represents a RNN cell at layer i and time step t. The RNN cells of the same layer (same shades) share the same weights.

signaling splits each layer into multiple kernels across time steps. It should be noted that the kernels in the same stream have exactly the same weights (the same shades in Figure 12). Hence, each kernel in the same stream has to reload the weights at each time step, which causes significant overhead.

The slowdown due to wasted weights reloading depends on the percentage of weights relative to the total memory consumption of the workload. The memory consumption of a RNN includes three parts: weights, inputs and temporal outputs. Both inputs and temporal outputs' memory footprint scales linearly with the batch size. In practice, the batch size could be very small when running inference. As shown in Figure 13, using a batch size of 4 (as in [33]), the percentage of weights varies from 30% to 90% and is 68% on average. This indicates that preserving weights on-chip can greatly reduce off-chip memory traffic. It should be noted that this percentage of weights could stay the same even with larger batch sizes because a large batch could be split among multiple GPUs so that each GPU operates on a small batch size.

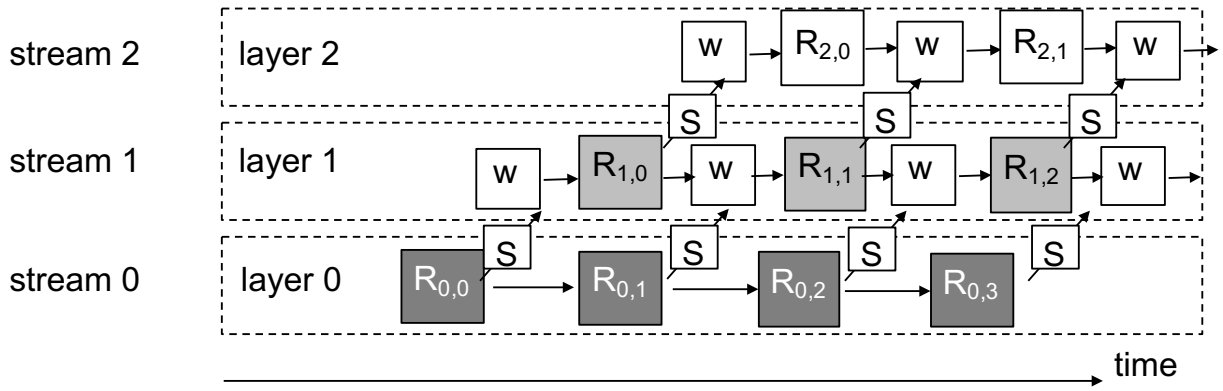

Figure 12: Stream Implementation of wavefront parallelism. W: wait event. S: signal event. $R_{i,t}$ represents a **kernel** to compute a RNN cell at layer i and time step t.

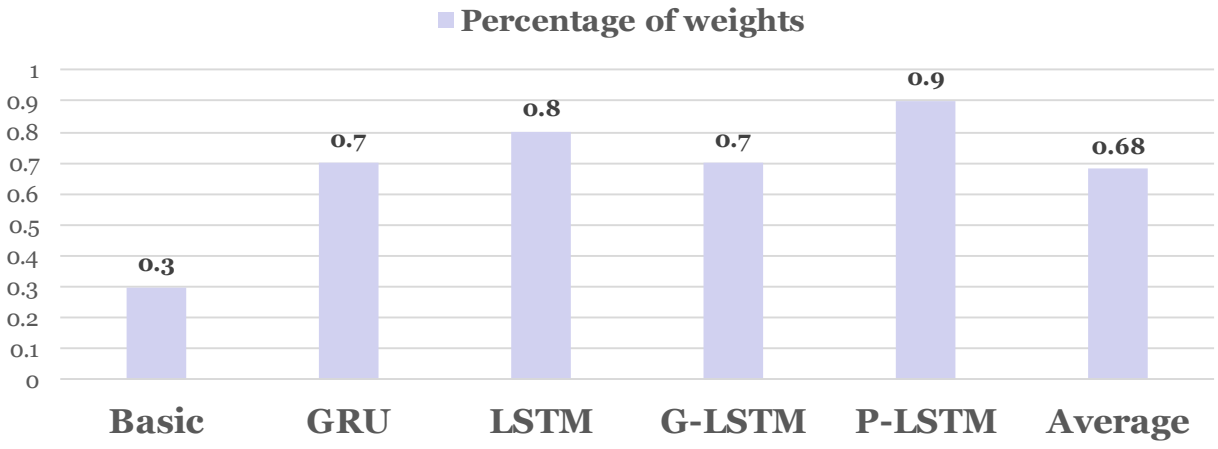

Figure 13: Percentage of reused weights in RNNs with batch size = 4

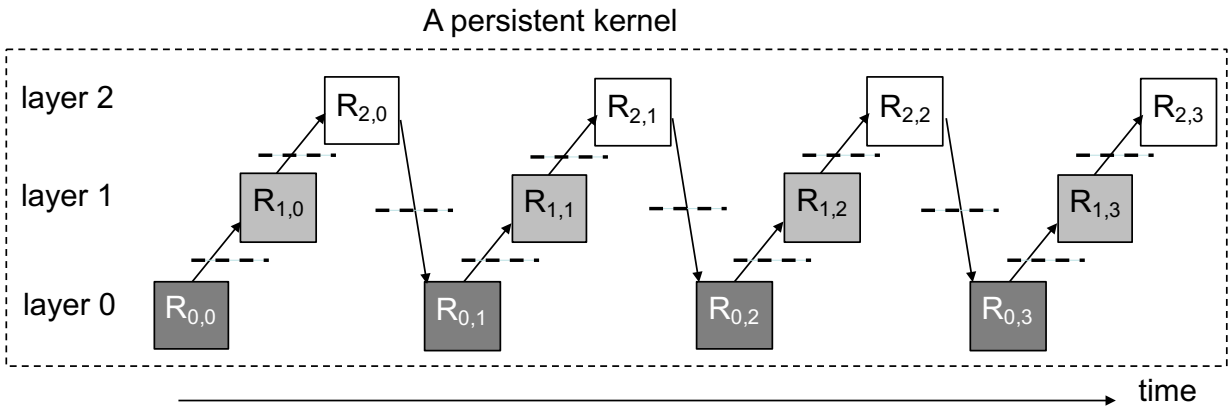

Figure 14: Persistent RNN. The dashed line indicates global barrier. Each $R_{i,t}$ is computed by all TBs sequentially.

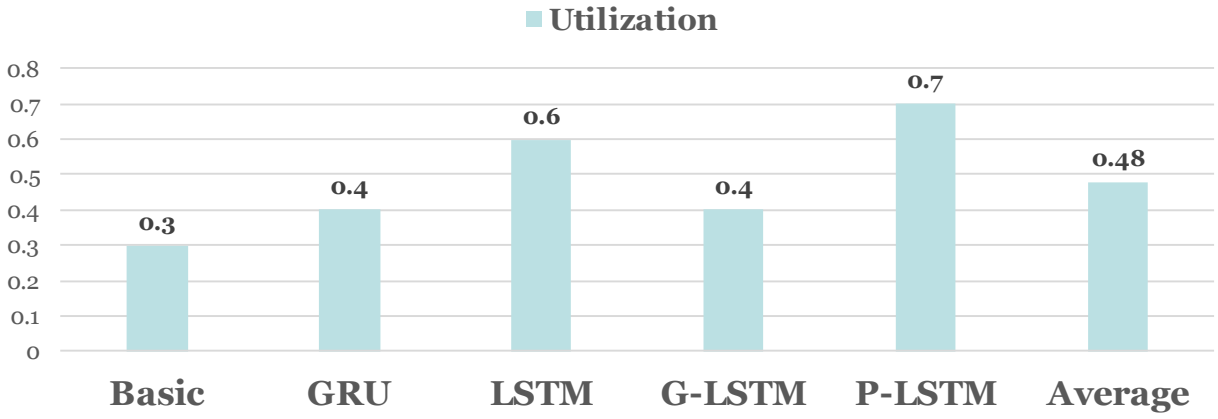

Figure 15: GPU utilization with persistent RNN

3.2.3.4 Persistent implementations of RNNs

Figure 14 shows a persistent implementation of multi-layer RNNs [33]. The block in the figure represents a task that computes a RNN cell at a certain layer and a certain time step. A single kernel is launched and all TBs of that kernel are persistent on the GPU. All the thread blocks accomplish each task together, and the tasks are executed sequentially following the dependency arrows shown in the Figure 14. To preserve the weights on-chip, the weights of all layers have to be partitioned statically to each TB [33]. Specifically, the weights are stored using registers while the inputs and temporal results are stored in shared memory. This design requires that the GPU device have sufficient registers to store the entire weights of all layers. Hence, the batch size has to be carefully adjusted so that the kernel will not exceed the GPU's capacity. The synchronization between thread blocks is global in nature, as shown as the dashed line in the figure. Such a global synchronization is implemented with atomic operations in assembly language [33, 25]. In summary, the kernel executes each block at each layer and each time step sequentially. It should be noted that although persistent RNN takes more time steps than the stream implementation depicted in Figure 12, each time step is much shorter due to reusing on-chip weights. Hence, this design achieves great performance speedup compared to the stream based implementation. Obviously such a design doesn't exploit the wavefront parallelism nature of RNNs. If the single task block in Figure 14 doesn't require a large number of threads to compute, the GPU resources will be significantly underutilized. This actually occurs very frequently in real workloads for inference. As shown in Figure 15, on average, only 50% of the GPU computing resources are utilized for RNN inference. Higher GPU utilization could be achieved by exploiting wavefront parallelism.

3.3 ADDRESSING WARP SCHEDULING

3.3.1 Round Robin Warp Scheduling

Conventionally, a simple round robin (RR) scheduler is used to rotate among ready-to-issue warps on per cycle basis. Studies have shown that there are severe limitations in

such scheduling. RR may easily destroy the intra-warp data locality since a different warp is issued each cycle, slashing the L1 cache hit rates which can be performance critical to cache-sensitive applications.

3.3.2 Greedy Then Oldest Warp Scheduling

The poor locality problem of RR can be overcome by a Greedy-then-Oldest (GTO) algorithm where the scheduler greedily executes a warp until it stalls and then starts from the oldest ready warp in the SM.

3.3.3 CTA-aware two-level warp scheduling

RR gives each warp equal priority so that all warps proceed relatively uniformly. Many warps have fairly symmetric instruction sequences, due to the fact that they belong to the same Cooperative Thread Array (CTA, a block of closely coupled threads, or thread block). Hence, different warps tend to reach long latency operations, such as memory requests, roughly at the same time. Once all warps assigned to an SM are stalled, the SM becomes idle, degrading program performance. To overcome this limitation, a CTA-aware two-level warp scheduling (CATLS) scheme was proposed, in which warps are first divided into isolated groups. The scheduler prioritizes the warps in one group before switching to another group. Hence one group can advance much faster than other groups, creating skewed progress among warps and thus the SM can better hide stalls from a subset of warps.

4.0 EFFICIENT GLOBAL SYNCHRONIZATION

4.1 MOTIVATIONS FOR EFFICIENT GSYNC

Repeated kernel launching is the most widely used practice for implementing Gsync as it is easy to program. The split kernels are launched sequentially on the host side. To allow the CPU process to check convergence, a synchronization API call is used between kernels. Such API calls could account for more than 60% of execution time for short kernels [25]. In addition to the API call overhead, a more prominent problem of repeated kernel launching is data thrashing if there are *data reuses* across the split kernels.

All on-chip data of the kernel preceding the Gsync including registers, shared memories and caches, are flushed upon exit, but have to be reloaded by the kernel following Gsync, as depicted in Figure 16(a). Such data thrashing results in idle time of cores and imposes high pressure on memory bandwidth. In fact, the thrashings could turn a compute-bound task into a memory-bound task [33]. The overhead of data thrashing may be acceptable for long executing kernels that amortize the overhead. However, in many contemporary machine learning applications where real-time response is required, such as real time object detection in self-driving cars, kernels are fairly short. For example, YOLO [50, 51] achieves 150 frames/seconds for object detection where the application running time is 6 ms and the average kernel running time is 200 μs. Due to the lack of hardware support for Gsync, kernel launching overhead plus data thrashing may become the performance bottleneck of an application [33, 52]. If those overheads are removed, the applications turn back to be compute-bound, and performance will scale well with more GPU nodes [33]. In cases where the API call overhead is small since `cudaDeviceSynchronize()` is not needed, data thrashing becomes a prominent problem. In many of these CNN kernels, the output of

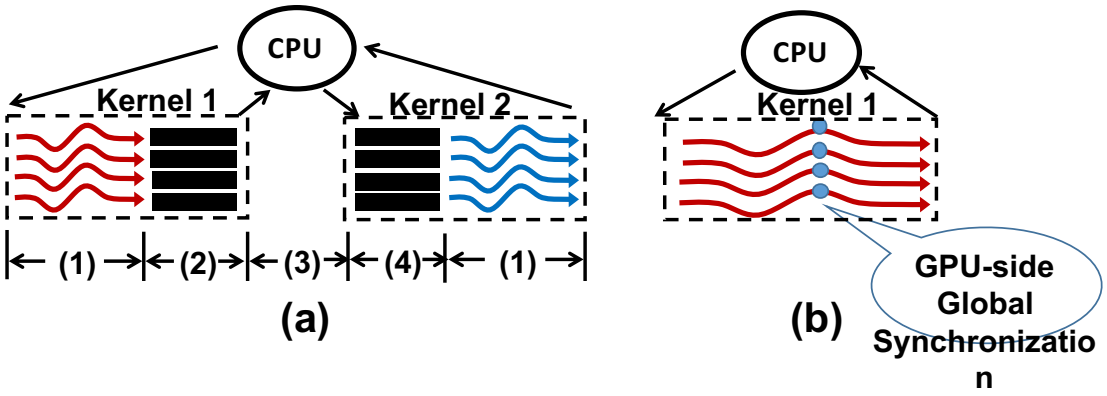

Figure 16: Global synchronization through (a) CPU-side API and repeated kernel launching: (1) kernel execution, (2) data flushing, (3) kernel launch overhead, (4) data reload; (b) the proposed GPU-side synchronization.

one kernel, such as feature maps [52], is the input of the next one. This output is flushed by one kernel but has to be loaded from global memory by the next kernel [52]. In other applications such as recurrent neural networks (RNN), abundant read-only data such as weights (for inference) are reused between kernels or even across many kernels. Repeated kernel launch would thrash those data across consecutive kernel executions, which is a critical performance bottleneck [33, 52]. Combining kernels into one would eliminate such thrashing if no context switching is needed. Even with context switchings at the global barrier, we will show that keeping those reused data on-chip greatly reduces the amount of memory traffic for context switching. Figure 17 quantifies data reuse for two example applications: CNN and RNN. Both are miniature versions with smaller sizes for weights so that they can fit on-chip [33, 53]. As we can see, both have large portions of data reuse, 35% and 80% for CNN and RNN respectively. If this reused data is kept on-chip and no data thrashing occurs across the global barrier, an ideal speedup of 2× and 10× can be achieved over repeated kernel launch for CNN and RNN respectively.

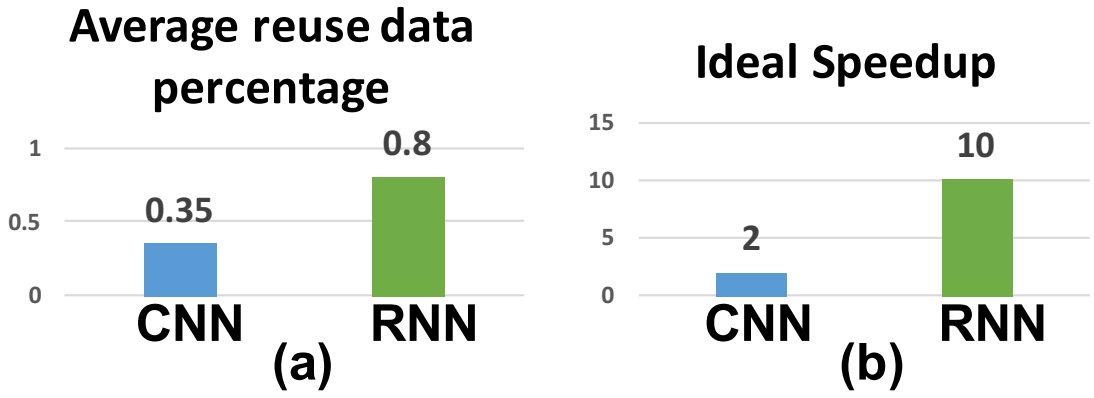

Figure 17: (a) Percentage of reused data across global barrier. (b) ideal speedup over repeated kernel launching if there is no data thrashing.

4.2 GPU-SIDE GLOBAL SYNCHRONIZATION API

Figure 18 shows examples of using $\_\_globalSync()$. Compared with the version in Figure 7(a) using repeated kernel launching, the do-while loop including the termination check is moved into the kernel which is launched only once. Thus, the expensive $\_\_cudaDeviceSynchronize()$ is invoked only once at the very end of the kernel. For neural network applications, an aggressive way is merging the entire network into one kernel [33]. Figure 18(b) shows an implementation where a layer originally composed of 6 kernels is merged into one kernel with 6 functions to exploit data reuse. As will be explained later, a barrier has to be inserted only between the first three functions.

The usage of $\_\_globalSync()$ is very similar to the function call implemented by atomic mutex or memory flags except for one important difference: $\_\_globalSync()$ is a truly parameter-free function call and requires no additional programming effort. For example, programmers are free from the burden of manually setting and resetting global counters or flags. The proposed function is transparent to programmers: synchronize all threads at this barrier so a TB is stalled at this instruction until all TBs reach this global barrier.

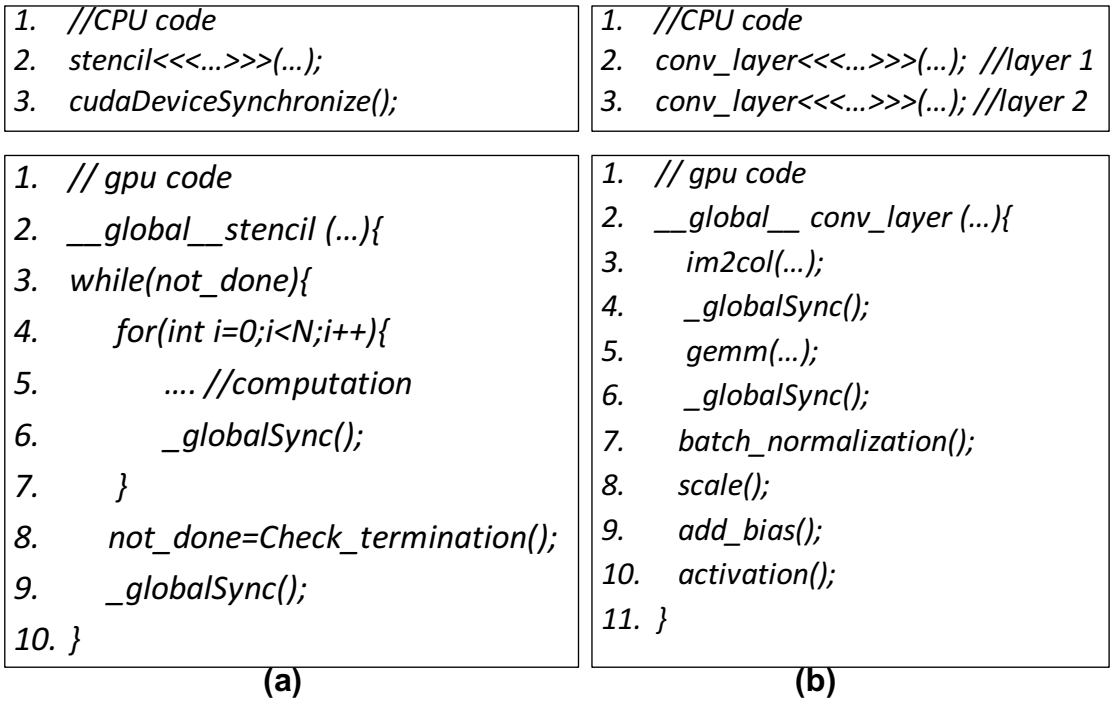

Figure 18: Code examples of using *__globalSync*(): (a) stencil computation; (b) CNN application.

4.3 MICROARCHITECTURE DESIGN FOR GSYNC

Synchronizing all TBs at the global barrier can be implemented using a simple global counter, `Sync_Cnt`, which is initialized to the total number of TBs in the kernel. Typically a 32-bit counter suffices to support the maximum number of TBs in a kernel. Every time a TB in an SM hits the barrier, a signal from this SM is sent to `Sync_Cnt` to decrement it by one, which is done atomically in hardware. The TB is stalled inside the SM until the barrier is cleared. Once `Sync_Cnt` drops to zero, meaning that all TBs have hit this barrier, it is reset to the original TB count of the kernel and all stalled TBs are resumed.

The most important architecture design for global synchronization is to avoid deadlock when there are more TBs in the kernel than the GPU's capacity, as with the atomic operation/memory flag based mechanisms. This problem can be solved by applying one of the GPU context switching techniques proposed in the literature [54, 55, 56]. It should be noted that full context switch is supported y HSA-compatible GPUs [13]. However such functionality targets at context switch at kernel level rather than TB level. For reasons that will be explained in Section 4.4.4, we propose to use Partial Context Switching (PCS) [55] which was originally proposed to enable sharing a GPU among multiple kernels, but can also be used to enable deadlock-free synchronization management. PCS swaps the context of one TB at a time, within or between kernels. Hence, if there are still non-dispatched TBs when some TB invokes *__globalSync*, that TB's context is swapped out and the context of one of the pending TBs is loaded to this SM to assure progress.

Figure 27 shows necessary architecture support for this procedure based on the architecture [55] (new and modified components are shaded). The "Unallocated TB Queue" is the original queue for non-dispatched TBs. A new queue "Global Preempted TB Queue" is used to hold meta data for swapped-out TBs to keep context information, including the global TB ID, the SM id, and the memory addresses to save the contexts, namely registers, shared memory and SIMT stack of a TB, as shown in the lower part of Figure 27. The "Local Preempted TB Queue" has the same purpose except that it keeps swapped TBs to within-SM memories for performance optimization. The "TB Dispatcher" is modified such that it prioritizes the "Unallocated TB Queue" over the "Global Preempted TB queue" since

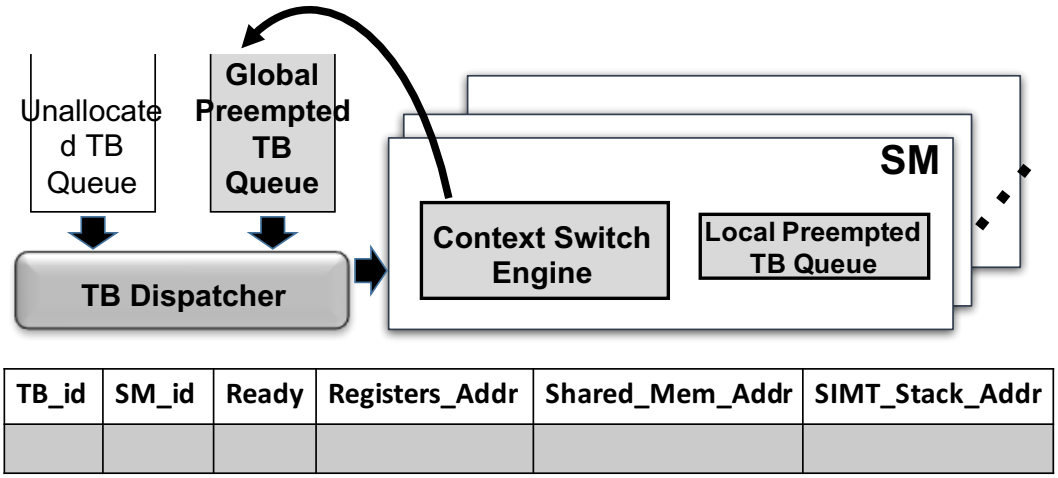

Figure 19: Architecture support for Global Synchronization with Partial Context Switch and meta data for swapped-out TBs.

TBs in the former have not started execution yet and TBs in the latter are to be dispatched after a barrier is cleared. The "Context Switch Engine" is also slightly modified so that saving context to the "Local Preempted TB Queue" is preferred over the global queue for performance improvement. The size of the meta data in each queue entry is 20 bytes. The number of entries in the preempted TB queues plus the number of TBs executing on all SMs should be at least equal to the total number of TBs in a kernel. This can be specified as an architecture constraint, similar to other constraints such as the maximum amount of shared memory available per SM. From our experiments, 1K TBs is sufficiently large to handle all kernels running with large input data, in contrast to a limit of 80 TBs per kernel in the memory flag implementation proposed previously [25].

The pseudo code of the global synchronization is sketched in Algorithm 1. When a warp calls $\_\_globalSync()$, it is stalled by its local warp scheduler (using the same mechanism currently used for synchronizing threads within a TB). When all warps of a TB are stalled on this call, the SM signals `Sync_Cnt` which is decremented by one. If `Sync_Cnt` becomes zero, then the TB is the last TB to reach the barrier and to clear it. At this time, no more

context switch is necessary, all stalled TBs in global and local preempted TB queues should be flagged as ready-to-execute, and `Sync_Cnt` should be reset to the original TB count of the kernel. If `Sync_Cnt` is still greater than zero, then as long as there are pending TBs in any of the TB queues, a TB swap is triggered.

Algorithm 1 Algorithm of global synchronization with partial context switch

```
1: if all warps of TB_i issued __globalSync() then
2:   Decrement Sync_Cnt
3:   if Sync_Cnt==0 then
4:     Mark "ready" all TBs in global preempted queue
5:     Broadcast to all SMs to mark all TBs "ready"
6:     Reinitialize Sync_Cnt
7:   else
8:     if there are pending TBs then
9:       Swap_out(TB_i)
10:      Swap_in()
11:    end if
12:  end if
13: end if
```

For performance advantage, a swap-out always tries to store the context first in the local "preempted TB queue" and then the global "preempted TB queue". A swap-in of a TB also follows the same order: unallocated, then local then global TB queues. When the kernel hits the first global barrier, there might be unallocated TBs waiting for resources to execute. Hence, they should be given the highest priority in the swap-in process. Once the kernel passes the first barrier and proceeds to the next barrier, there might be TBs at the head of the local and/or global preempted TB queue that are ready to execute (set to ready at the clearance of the previous barrier). The engine then first finishes the local queue and then moves on to the global queue.

4.4 MANAGING CONTEXT SWITCH

The main challenges in our design are context switch overhead and the memory contention during context switches. The context of a TB consists of registers, shared memory and SIMT stack [54, 55, 56, 57]. According to [57], the SIMT stack per thread is calculated to be at most 640 Bytes for a 16×16 TB, which includes program counter and divergence information for each warp in the TB. Hence, the amount of registers and shared memory are dominant in the context. We measured from our benchmarks that the amount of data being swapped per barrier ranges from 1MB to 9MB. How to allocate memory space for them, the overhead of moving them and the bandwidth contention from them are the challenges we address in this section.

4.4.1 Memory Allocation for Context Saving

Memory space for swapped-out TBs can be allocated either statically or dynamically. Static allocation sets apart a continuous memory space separate from kernel's data region. Loads/Store instructions for context switching can thus be perfectly coalesced [55]. However, static space cannot adapt to the changing context sizes across different kernels, resulting inefficient usage of memory capacity.

An alternative solution is to allocate memory space on-demand dynamically. We use GPU's device memory rather than the main memory to store context as they need to be alive at the GPU side for continuous execution. Modern GPUs support unified virtual memory [11, 13] where GPU memory manage unit (MMU) is able to migrate allocated pages [58] and modify page tables accordingly. The MMU also keeps a list of free pages. Pages can be allocated by the context switch engine for saving context. The allocated pages are the swap space in memory for those TBs that are in progress but cannot fit into the GPU at a time. Pages are deallocated upon the completion of the kernel. The page number and offset can be saved as the meta data in preemption queues shown in Figure 27. If a TB's context is larger than the size of a base page (4KB) [58], MMU looks for a large page (2MB) and allocate a continuous memory space for the TB's context.

4.4.2 Reducing Context Switch Overhead.

We propose three approaches to reduce context switch overhead: 1) leveraging unallocated registers and shared memory; 2) reducing context sizes; 3) eliminating unnecessary context switches.

Leveraging unallocated registers and shared memory. As shown in [55, 56, 59], some kernels do not fully use registers or shared memory. For example, if the maximum number of threads/TBs per SM is reached, the GPU cannot dispatch more TBs to the SM even though there are unallocated registers and shared memory on that SM. Hence, the unallocated on-chip storage capacity can be utilized to save contexts.

When unallocated registers and shared memory are enough for one more TB context, there is no need to perform swapping out to global memory. Only the meta data including the register address and shared memory address is pushed to the local preempted TB queue. After swapping out, the SM has space to take one new TB. Moreover, contexts (mainly registers) are moved to shared memory if shared memory is not used. Meanwhile, the metadata (Figure 27) of the swapped out TB is pushed to the local preempted TB queue. Since the register usage is the same for all TBs in a kernel, we only need to store the starting address in the meta data. Similarly we can use unallocated registers to store contexts.

Reducing the size of context swapping. Due to the unique position of global barriers, which are usually between two different computing phases, a significant portion of the context need not be saved. The first opportunity to reduce the context size is the read-only data. In some applications, a significant amount of shared memory is used to store read-only data. For example, in the inference of deep learning applications, a weight matrix is fixed and reused across global barriers. At context switching, instead of saving the read-only weights to global memory, we can simply save the addresses of the read-only shared memory as part of the contexts and load them from global memory at the next swapping in. Read-only shared memory can be tagged by the compiler before execution.

The second opportunity to reduce the context size is the abundant registers that are used just for computing intermediate and temporary data. Those data are no longer live after the barrier, and hence, do not need to be saved. This opportunity was exploited for

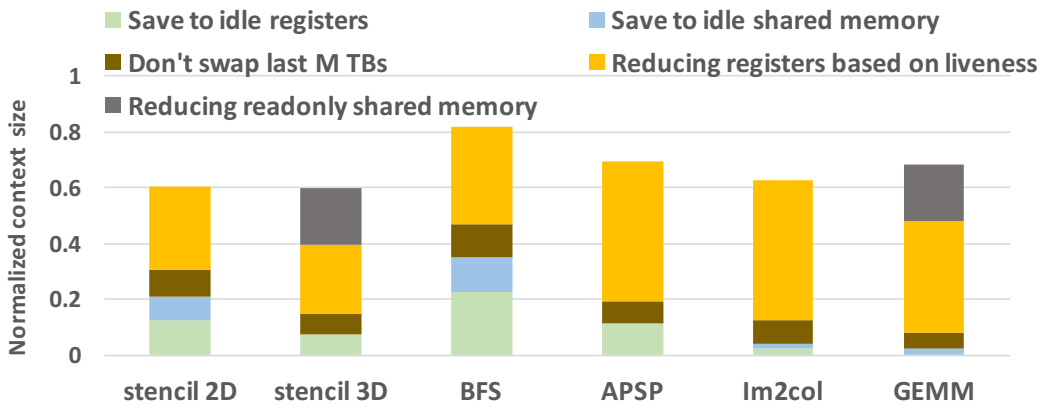

Figure 20: Breakdown of context size reduction.

general preemption points [59], but we found it particularly effective for global barrier. For the applications evaluated, there are 60% registers on average storing temporary data before the Gsync point and do not need to be saved during context switch. The compiler can tag those registers when processing the *sass* file [59, 60].

Avoiding unnecessary context switches. Finally, let M be the maximum number of TBs that can be dispatched to the GPU at a time. Then the last M TBs of the kernel reaching a barrier do not need to be swapped out since there are no pending TBs to swap in. This is easy to determine since all TB queues in Figure 27 do not have ready TBs at this time.

The effectiveness of the proposed techniques are quantified in Figure 20 which plots the percentages of context that can be reduced via each technique. As we can see, the size of the context of all tested kernels is reduced by at least 60%, and by up to 80% in BFS. The most effective technique is the elimination of saving short-lived temporary registers, due to the position in the program where context switch happens. Reducing read-only shared memory is also significant when there is such opportunity. We remark that the reductions of temporary registers and read-only data in shared memory scale proportionally with increasing number of TBs, and are not limited by the available hardware resources which the other techniques depend on.

4.4.3 Reducing Memory Congestion

Even when significant amount of context is saved on-chip, the off-chip context switching is still a major threat to memory bandwidth and performance. We observe that the actual swapping of different TBs from the same or different SMs could interleave, as depicted in Figure 21(a), which delays the start of the swap-in instructions. Hence, we propose a prioritized swapping inspired by the two-level warp scheduling [61]. As shown in Figure 21(b), the warp scheduler gives fixed priority to the swap-out instructions of one TB on a first come first serve basis. For example, if TB0 is the first TB to reach a global barrier, then the warp schedulers will issue only its swap-out memory instructions. The scheduler starts to issue the swap instructions for TB1 only after all TB0's swap instructions are issued. As a result, the TB to be swapped in, TB2 in this example, will start loading its context after TB0 has completely vacated its space, which is earlier than in Figure 21 (a).

The memory controller can follow a common scheduling policy such as the First-Ready, First-Come-First-Serve (FR-FCFS) policy [62]. The prioritized memory requests from one SM may interleave with those from another SM at the memory controller, but FR-FCFS can schedule them to preserve memory access locality, which also helps to keep their order since memory requests for context switching have strong locality. In addition, throttling is used when the number of outstanding memory requests is above a threshold so that context switching behaves at worst like a memory-intensive kernel. This is effective in controlling the performance impact of heavy memory traffic [55].

4.4.4 Using Other Context Switch Techniques

Full context switch techniques, rather than PCS, can also be used to prevent deadlock when implementing global barriers. However, using full-SM context switch [56] will incur higher context switching overhead than using PCS because all TBs in one SM have to reach the barrier before the context switch is triggered, causing all TBs to swap simultaneously. In contrast, using PCS, TB swaps are staggered[55]. Moreover, memory latency can be mostly hidden because as soon as one TB reaches the barrier, swaps begin and overlap with the execution of other TBs. The Chimera scheme [54] differs from [56] in that it either flushes

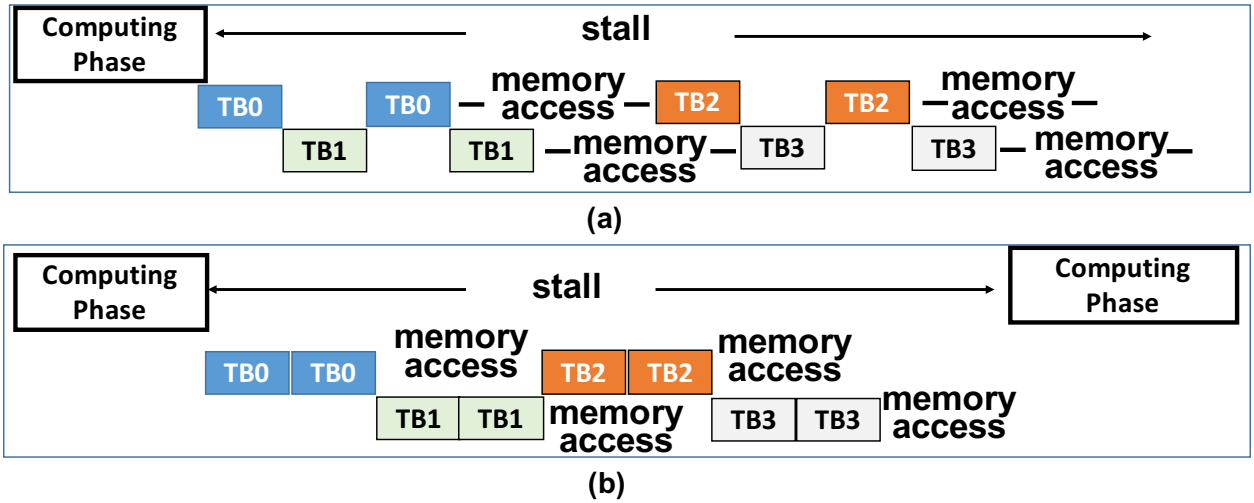

Figure 21: Scheduling swap instructions when TB0 & TB1 are swapped out and TB2 & TB3 are swapped in (to/from global memory). (a) Baseline. (b) Proposed.

or drains the SM before swapping. Flushing would cause deadlocks at the barrier as no TBs can make forward progress, and draining cannot be achieved as all TBs will be blocked at the barrier.

4.5 EXPERIMENTAL RESULTS FOR GLOBAL SYNCHRONIZATION

4.5.1 Experiment Setup and Methodology

We extended GPGPU-Sim [63] version 3.2.2 to model both kernel execution in Gsync-enhanced architecture and kernel launching overhead. The simulator is configured to closely model a Maxwell architecture, and then enhanced with proposed microarchitectural modifications. Kernel launching overhead is measured from a Maxwell GPU processor, and simulated kernel execution time is converted to wall-clock time through a regression model so that the two timing can be combined with reasonable accuracy. Configuration details are listed in Table 3. Proposed global synchronization scheme is denoted as **GS** in the following

discussion. The same experiment setup will also be used for wait-signal and warp scheduling evaluations. We compare with previous approaches including repeated kernel launching, atomic operation, memory flags, and PT model based synchronization mechanisms. We do not compare with dynamic occupancy discovery [42] because the best performance it can achieve is the same as PT if a tight lower bound of occupancy is detected [42]. To collect measurements accurately, we observe that the end-to-end execution time of kernels consists of CPU execution and GPU execution portions. The CPU portion includes kernel launching time and API call overhead such as `cudaDeviceSynchronize()`. The GPU portion consists of kernel execution time including function calls such as atomic functions and the proposed *_globalSync*(). The GPU portion of time can be collected via simulation since it has the proposed GS. Whereas the CPU portion can only be collected using real devices. Hence, when summing up those two portions of time, we convert all simulated timings to their corresponding real device execution times so that it can be added to the CPU portion of time.

Table 2: Configurations of GPGPU-Sim for evaluation of global synchronization.

warp schedulers per SM	4
SM	13
warp size	32
warp scheduling policy	greedy-then-oldest
scratchpad memory	96KB per SM
maximum threads per SM	2048
L2 cache	2MB
processor/memory clock	1050/7010 MHz

The wall-clock time of API calls for GS based implementation should be close to that of the implementation using memory flags because: 1) both GS and memory flags launch the kernel once; 2) they have the same kernel codes except for the *_globalSync*() function so they have the same resource requirement; 3) API latency does not change drastically with different input sizes. Figure 22(a) shows the latency of API calls for different input data size for stencil 2D application and Figure 22(b) shows the average latency of API calls in different applications over different data sizes. These measured latencies are added to the simulation latency of kernel execution converted to real time as shown below to obtain the

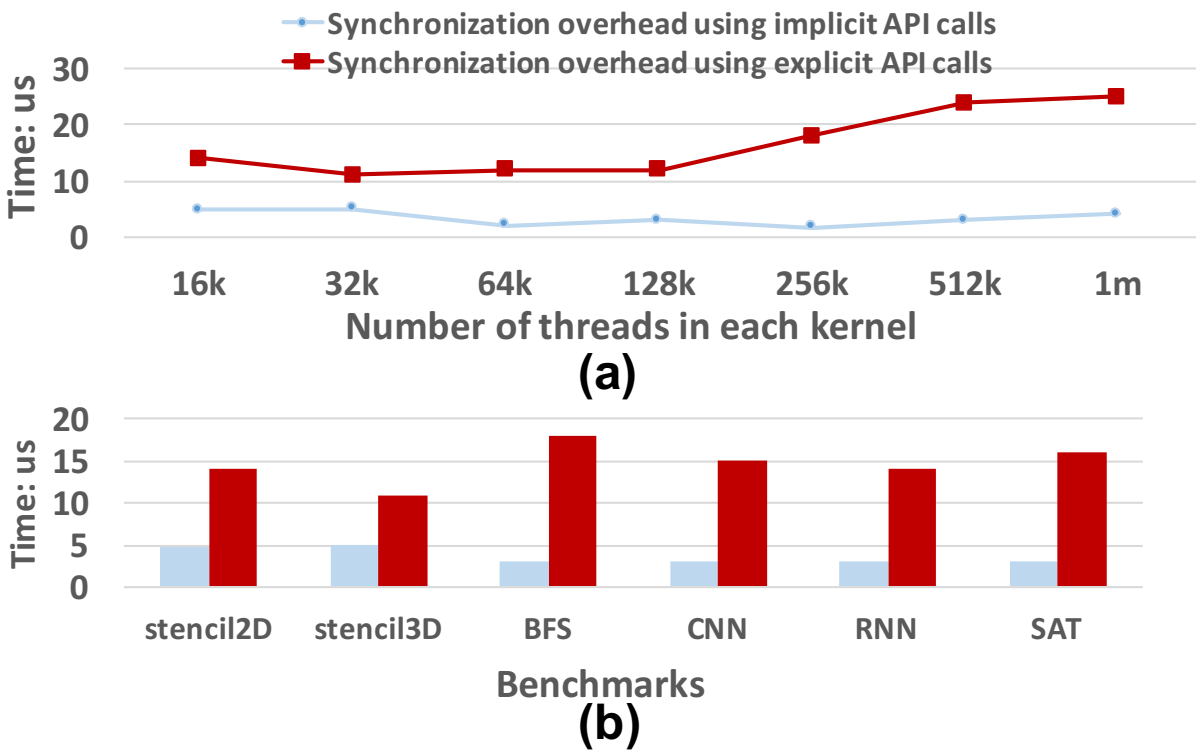

Figure 22: API call latency for different data size and applications.

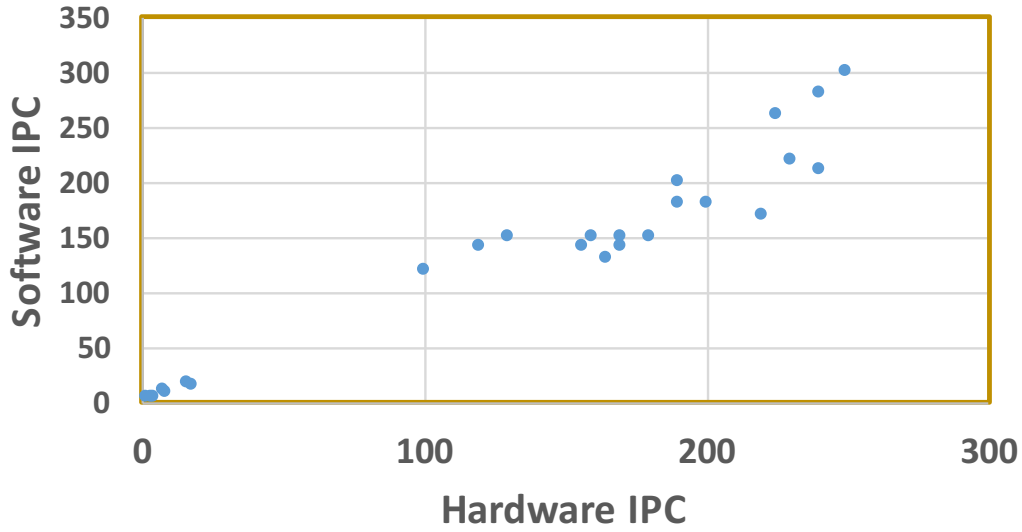

Figure 23: Correlation between simulated and real device execution on GTX 970.

end-to-end execution time.To convert the kernel simulation time to real time, we compare the kernel execution time (excluding kernel launching overhead) of the memory flags-based synchronization mechanism in both simulation and a particular GPU device. Then, we use a linear regression model to map simulated kernel execution times, for all schemes including the proposed GS, to real time. For example for GTX 970, each point in Figure 23 represents one benchmark of a particular data size. From this figure, we observe a strong correlation with a correlation coefficient of 0.95 (1 is perfect correlation). Similar results are observed for GTX 960 and 1070. We enhanced GPGPU-Sim by a module which automates this process and estimates the cycle count based on the regression model. We remark that this model can be replaced by a more accurate simulation of the kernel launching overhead when more details about the architecture are made available to the public.

4.5.1.1 Stencil applications Stencil computations are widely used in important classes of applications, such as solving partial differential equations in discrete domains [64, 65, 66]. Each iteration of stencil presents large amount of parallelism since calculating one point is independent of all the other points. Global synchronization is needed between iterations.

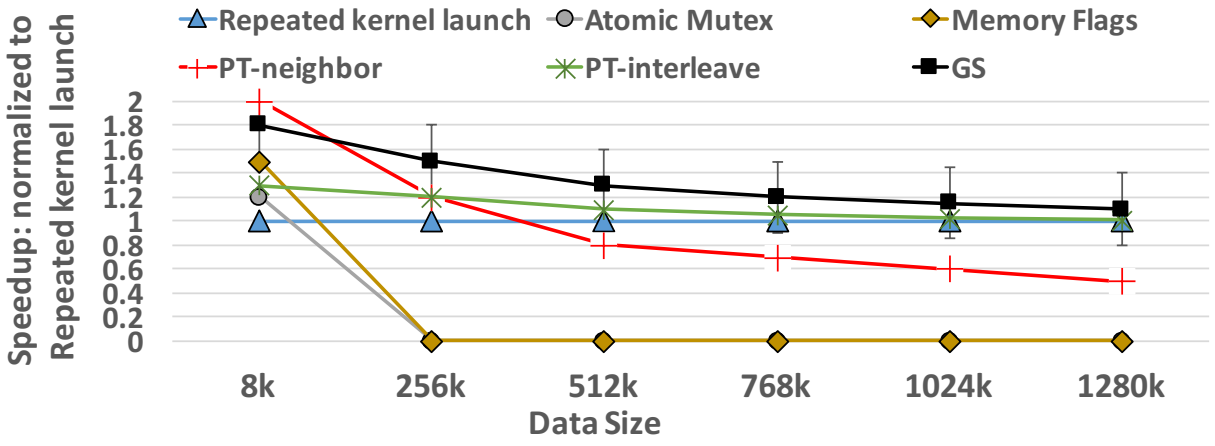

Figure 24: Speedup for stencil 2D.

In the repeated kernel launching method, the kernel is launched every iteration. In all the other models, the kernel is only launched once, but the global synchronization is invoked for each iteration at GPU side. We also implemented two PT models with memory

flags: PT-neighbor and PT-interleave [43, 67]. In PT-neighbor, each thread calculates a continuous region of equal number of points. In PT-interleave, one thread calculates a group of points with an interval stride equal to the total number of threads in the kernel. The performance result is shown in Figure 24 normalized to the repeated kernel launching with CPU-side synchronization. On average, GS achieves 1.1x to 1.8x speedup over the repeat kernel launch baseline. We make several observations:

The proposed GS achieves the best overall speedup. The repeated kernel launching and DP incurs significant overhead. The memory flags array and atomic mutex deadlocks with only 100 TBs while the other schemes can scale to 1000+ TBs. The proposed GS achieves the best performance due to reduced kernel launching overhead and the techniques of saving context switch overhead.

PT-neighbor achieves the best performance for small data size due to good data locality. However with large data size, the coalescing degree of memory accesses starts to degrade until fully diverged which leads to significant slowdown. PT-interleaved achieves good performance with large data size since its memory accesses are fully coalesced and it requires fewer TBs to synchronize at global barriers. However, PT-interleaved suffers from unbalanced loads meaning that different threads and SMs have different loads when the data size is small [26].

We remark that in this and all following experiments, there is diminishing gain for GS over repeated kernel launch when the kernel's input data size increases. This is normal as when data size increases, the computation per kernel increases and the overhead of repeated launches becomes relatively small. Further, the number of TBs increases which increases context switching in GS.

4.5.1.2 Graph Traversal

BFS is an algorithm for traversing a tree or graph data structures in an order that prioritizes sibling nodes over children nodes. One node is assigned to one thread during processing [63, 68]. A frontier corresponds to all sibling nodes being processed at the current level. Global synchronization is needed to ensure one frontier is traversed before the next frontier. In traversing each level, the threads for nodes in the current frontier are active and can work in parallel while all other threads are idle. Each active thread is responsible for traversing its children. It is crucial to enforce synchronization before

traversing each frontiers/levels, since we must assign the smallest level number to each node. The traversal within a single frontier is usually parallelized into a kernel which is repeatedly launched to traverse the graph level by level until all nodes are traversed [63, 68].

The repeated kernel launching must use explicit CPU-driven synchronization to check the termination condition as shown in Figure 7(b). We implement the proposed GS, atomic mutex and memory flags using the same algorithm but move the termination checks into the kernel. We also implemented a PT version with a global job queue which has to be locked by a thread before retrieving or posting jobs[43]. Finally, we implement a naive version of DP where CPU repeatedly launches a sequence of parent kernels and each parent kernel will dynamically launch children kernels for vertex expansion. This mechanism [38] tends to overly launch kernels. As shown in Figure 25, GS achieves significant performance improvement over all other solutions. On average, GS achieves 2.1x speedup over the repeat kernel launch and DP1. GS only launches the kernel once so the expensive explicit CPU-driven synchronization is invoked only once. There is no need to do memory copying since the check for termination is moved into the kernel. The PT model is known to have very bad performance for irregular workload [43] due to loss of parallelism. When many threads try to access the global work queue concurrently, the sequential locking serializes the available threads that could have run in parallel [43]. The DP has bad performance due to large run time overhead, which involves not only saving contexts of parent threads but also creating children kernels [38]. In addition, it also invokes the same explicit synchronization API, *cudaDeviceSynchronize*, at the device side between parent and child kernels.

4.5.2 Performance Overhead and Scalability

The performance overhead of the proposed GS design are mainly context switches. However, our proposed techniques to reduce such overhead are quite effective, as illustrated in Figure 26 for stencil 2D as an example. The figure also demonstrates that the proposed techniques scale well with increasing data sizes. The vertical axis represents execution time normalized to an ideal GS (the proposed GS with context switch latencies set to zero). When using naïve implementation of our GS function call and PCS, the total execution time, the "Partial

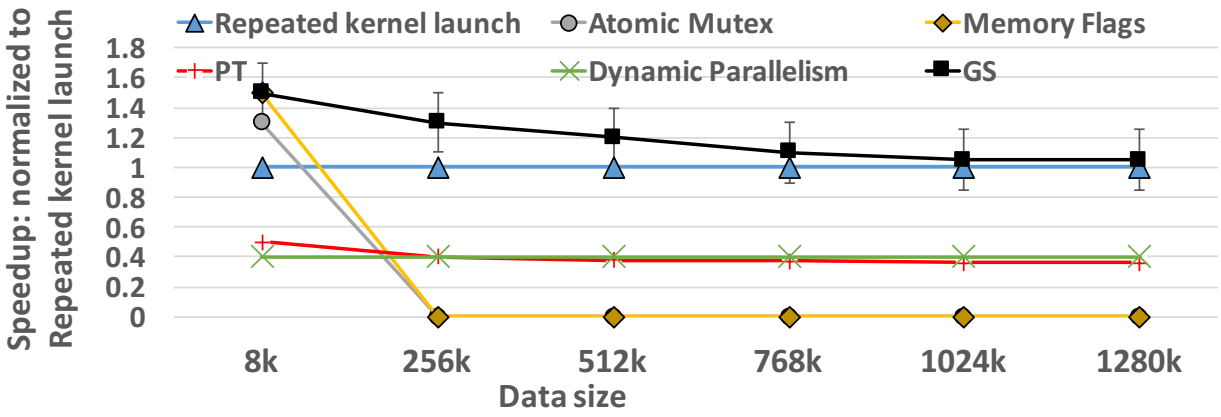

Figure 25: Speedup for BFS.

Context Switch" curve, is only lower than repeated kernel launch when data size is small. The latter has API call overhead while the former does not and there are no context switches. When data size increases, the naïve implementation quickly surpasses repeated kernel launch as the context switch overhead outweighs the benefit of reduced API call overhead. However, after applying all reduction techniques for context switches and promoting data reuses, the final performance indicated by the 'All' curve, is clearly better than repeated kernel launch.

More importantly, when data size increases, data thrashing becomes dominant. For repeated kernel launch, this is due to data flushing from exiting TBs and data loading from new TBs. Such traffic is equivalent to performing full context switches across TBs. In our GS, more context switches occur but the context reduction ratio achieved by our proposed techniques remains fairly constant with increased data size. Hence, no matter how large the data size grows, the 'All' curve will always stay below that of 'Repeated kernel launch', demonstrating the good scalability of our design. On the other hand, we remark that when data size grows too large, it is inefficient to keep the entire application on a single GPU. Using a cluster of GPU nodes and partitioning the data such that each node runs a reasonably small and independent portion of the data will yield a more scalable and efficient execution [33]. In this case, our design becomes more favorable since it works most effectively on relatively small data sizes.

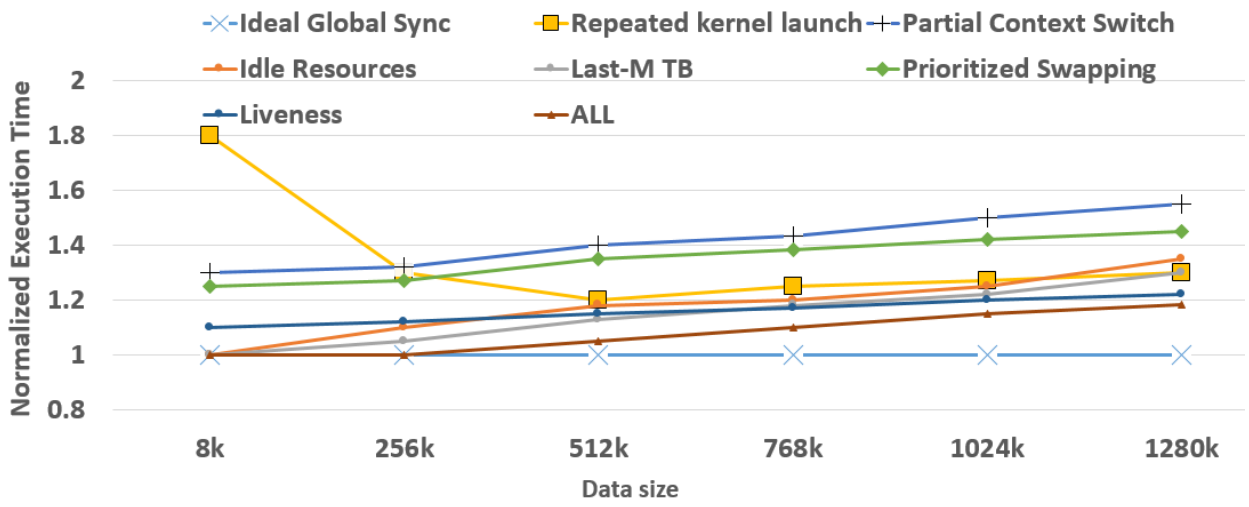

Figure 26: Performance overhead of partial context switch normalized to an ideal GS.

4.5.3 Hardware Overhead Analysis

In our design, we extensively reused existing on-chip resources. The global preempted TB queue (Figure 27) already exists in the original PCS design [55]. The number of entries in this queue can be defined as a programming constraint of the number of TBs in a kernel. We concluded experimentally that 1K entries are sufficient for the tested applications. We added a local preempted TB queue for performance optimization which contains no more TBs than one SM can host. In addition, the meta data storage stores the context information for all swapped TBs. The number of entries is equal to the total number of entries in the global and local preempted TB queues. Each entry is 20 Bytes for storing the addresses of registers, shared memory and SIMT stack of a TB, assuming a Maxwell architecture. The context switch engine also includes the logic for a finite state machine implementing the algorithm and an instruction generator for global load/store instructions implementing context switch. We limit the local preempted TB queue size to 64 TBs per SM.

5.0 ACCELERATING RNN WITH SOFTWARE WAIT-SIGNAL

5.1 ALGORITHM

Similar to the persistent RNN, we also propose to launch one kernel for the entire RNN. The overall dependency graph is shown in Figure 27. Each layer is computed by a dedicated group of TBs. This group of TBs will compute $R_{i,t}$ of layer i sequentially for all time steps $t = 0, 1, ...$

TBs will wait if they depend on the results from other TBs to start computing. The waiting mechanism is implemented using a lock-free barrier synchronization [25], where a single thread of a TB periodically checks a memory flag while other threads in the same TB are blocked by within-TB synchronization $\_syncthreads()$ or $barrier()$. This memory flag array is a simple fixed size array of boolean variables where "zero" represents "not ready". The detailed operation of memory will be introduced in section 5.1.2.

The memory flag arrays indicated by the dark blue arrays in Figure 27 are termed as "read memory flags". The RNN cells have two sets of read memory flags because they depend on both S_{t-1} and O_t. The TBs for cell $R_{i,t}$ can start computing when both read memory flag arrays are ready. Fine-grain parallelism is also possible where some TBs can start as soon as one of the read memory flag array is ready. This will be detailed in section 5.1.1.

The signal mechanism is implemented by the "write memory flag array", light blue arrays in Figure 27. Each TB writes to its own entry in the write memory flag arrays after it finishes the computation for the current time step. If all entries of the write memory flag array are "one", the current cell $R_{i,t}$ is finished, and it signals two groups of TBs to start computing, as indicated by the two arrows between light blue array and dark blue arrays in Figure 27.

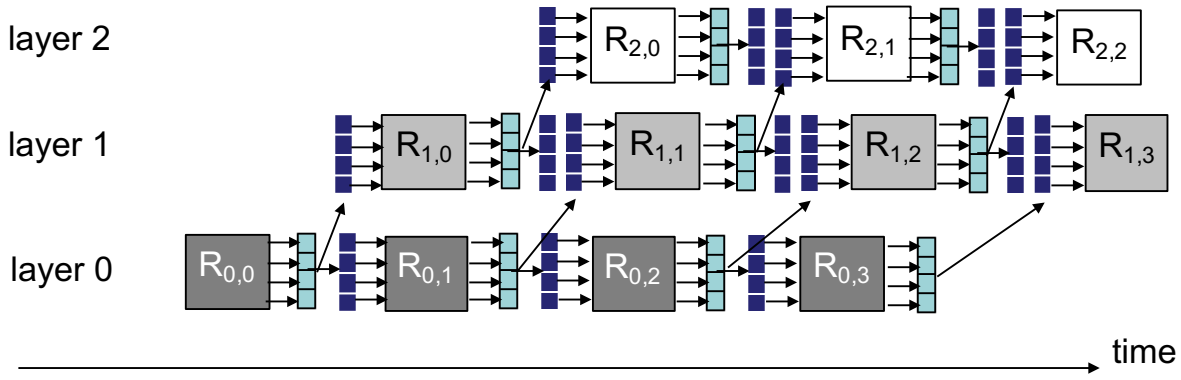

Figure 27: The proposed TB-level wait-signal. The light/dark blue arrays are write/read memory flag arrays. The entire network is computed by one kernel and for each i, the cells $R_{i,t}$, $t = 0, 1, ...$ are computed by a subset of TBs.

5.1.1 Fine-grained Parallelism within a layer

As indicated by formulas (9) and (10), there exists parallelism within a layer. Specifically, computing $\mathbf{S_{i,t-1}W}$ and $\mathbf{O_{i-1,t}U}$ are independent and could run in parallel by different threads. As shown in Figure 28, a barrier synchronization is necessary before computing the functions F_O and F_S. This parallelism is particularly important for the first layer where the input to the cell, $\mathbf{O_{-1,t}}$, is actually the input data $\mathbf{X_t}$ which is ready for all the time steps. Hence the computation for $\mathbf{X_tU}$ can start in parallel immediately. In the stream implementation [49], this is achieved by launching two kernels for $\mathbf{S_{i,t-1}W}$ and $\mathbf{O_{i-1,t}U}$ in two different streams so that they can run concurrently. The barrier synchronization between kernels from different streams is also implemented using stream events. Hence, two kernel streams are used for each layer to exploit this fine-grained parallelism [49]. In our proposed scheme, we utilize two groups of TBs, TB_{SW} and TB_{OU}, for each layer to compute $\mathbf{S_{i,t-1}W}$ and $\mathbf{O_{i-1,t}U}$ in parallel, respectively. The synchronization in Figure 28 can also be implemented as a wait-signal event between these two TB groups, TB_{SW} and TB_{OU}. Specifically, TB_{SW} will first calculate the matrix multiplication $\mathbf{S_{i,t-1}W}$ and then wait. TB_{OU} will calculate the matrix multiplication $\mathbf{O_{i-1,t}U}$ in parallel.

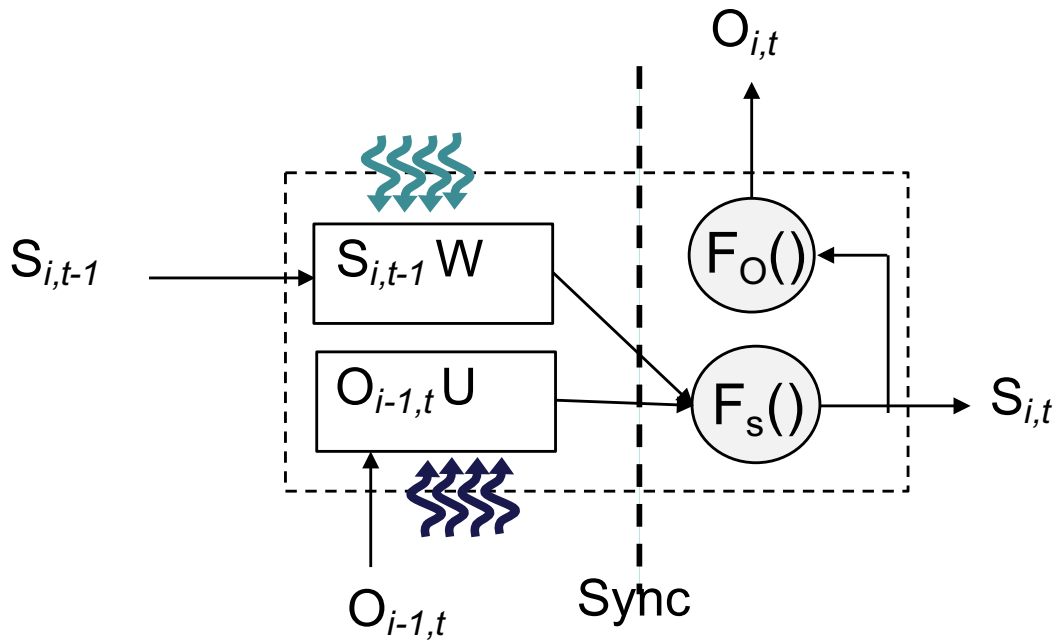

Figure 28: Fine-grained Parallelism within an RNN cell. $\mathbf{S_{i,t-1}W}$ and $\mathbf{O_{i-1,t}U}$ run in parallel by different threads.

5.1.2 Implementing Wait Signal with memory flags

We utilize memory flags to implement the wait-signal mechanism at the thread block level. Memory flags are the most common software workarounds to implement synchronization among TBs in GPU [25]. For each wait-signal event pair, two dedicated memory flag arrays are used to coordinate the synchronization requests.

As shown in Figure 29(a), in these two arrays, each flag is mapped to a TB. When a producer, TB i, finishes its execution in the current step, its leading thread (thread with id = 0) sets element i in *writearray* to "True". A dedicated TB monitors *writearray*, and when all the producers write their memory flags, it copies the values from *writearray* to *readarray*. The leading thread of the consumer, TB j, repeatedly reads element j of *readarray* in a while loop until its value becomes "True". Other threads of TB j are blocked on a within-TB barrier. The memory flags are implemented using volatile global variables to avoid data hazard due to cache incoherence. Such a communication mechanism is lock-free [25], as opposed to the atomic operation which incurs significantly larger delay if too many TBs are associated with the same event.

The software implementation of memory flags have some downsides: 1) it introduces a busy wait loop, which occupy computing units but are not making any progress and 2) accessing the memory flag arrays is a long latency operation. A hardware support of wait-signal at architecture level, it available, will resolve these inefficiencies. This may be the subject of future work.

Clearly it will inefficient for a naive design which assigns one set of memory flag arrays to each wait-signal pair. If the RNN is unrolled for many time steps or if the RNN has many layers, the required number of memory flags also increases. We propose a simple recycling algorithm for memory flags. First, we observe that after being copied, the *writearray* is safe to be reset by the dedicated thread that copies the flags to *readarray*. Second, the *readarray* is safe to reset after being read by the consumer TB. With these two observations, we could simply use one write memory flag array and two read memory flag arrays for each layer and recycle them for consecutive time step. An alternative design of memory flags is shown in Figure 29(b), where all waiting TBs can read a single entry in the memory read array to

indicate that they are ready to execute. With this design, the read memory flag array only needs one entry which is more efficient than the previous design. However, resetting the read memory array in the second design will be complicated, since it can only be reset after all TBs finish reading this entry. The solution is to allocate two such write-read memory array pairs for each layer and use them alternatively. In this way, when the group of TBs start to access the write array of the second pair, the read array of the first pair can be reset. In conclusion, the design in Figure 29(a) is simpler and more efficient since it avoids contention when multiple TBs access the same location in the read array.

5.1.3 Workload Assignment

It is critical to split and assign appropriate portion of the workload to each TB. The most dominant computing task of RNN is matrix multiplication. Hence we study how the matrix multiplication is mapped to each TB. Assume the task is to compute $Y = XW$ where X, W and Y are the inputs, weights and outputs matrices, respectively. As discussed above, a fundamental optimization is to partition the weights to each TB since the weights will be reused over many time steps. Therefore the weights will be partitioned column-wise to each TB while the X matrix will be shared by all the TBs. In figure 30, we compared the workload assignment of persistent RNN and the proposed wait-signal scheme. In persistent RNN, all TBs collaborate to accomplish all computing tasks sequentially. Hence, in this case, all 3 TBs of persistent RNN will do three matrix multiplications in a row. Each TB will be allocated one-third of the weights of each layer. To fully utilize the on-chip storage, weights can be loaded onto registers while the shared memory is used to load tiles of input data X and store partial result. Since shared memory is private to each TB, the input data X will be replicated and loaded 3 times by TB 1, 2 and 3. In contrast, in the proposed wait-signal scheme, each TB is dedicated to one layer and hence the weights of one layer are loaded to only one TB without any replication. It should be noted that in real applications, more than one TB are used to finish one matrix multiplication with the proposed wait-signal mechanism, which implies that X will be inevitably replicated. However, the proposed scheme will have way fewer replication than the persistent RNN design. In fact, it can be proved that the proposed

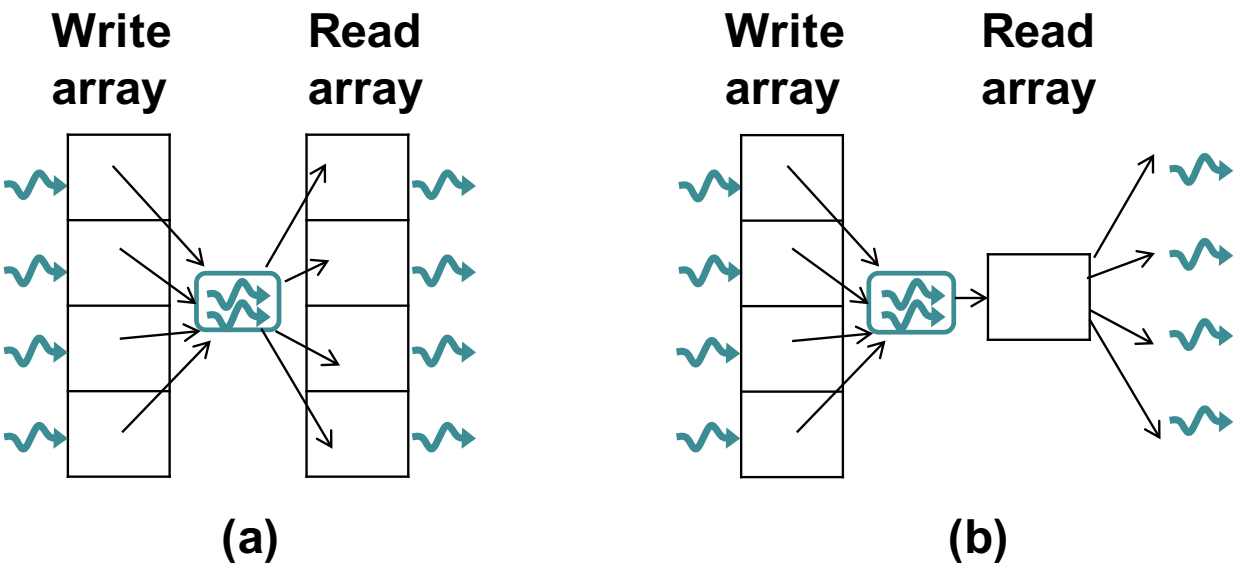

Figure 29: Memory flag arrays to implement TB-level wait-signal. (a) Each TB reads its own entry in the read memory flag array. (b) All TBs read the same entry in the read memory flag array

scheme will always have N times fewer replicated copies of X than persistent RNN, where N is the number of layers. A detailed TB layout for partitioning weights in the proposed scheme is shown in Figure 31, assuming that it takes three TBs to complete one matrix multiplication. As discussed earlier, the weight matrices W and U are partitioned column wise to each TB so that there is no shared weights among TBs. The width of a TB should be a multiple of 32 so that loading the weights will be fully coalesced. The height of a TB can be calculated as $H/REGS$, where H is the height of the weight matrix and $REGS$ is the number of registers that are allocated to each thread to store weights.

We apply the tiling structure to implement the matrix multiplication efficiently for GPUs by decomposing the computation into a hierarchy of thread block tiles, warp tiles, and thread tiles and applying the strategy of accumulating matrix products. This hierarchy closely mirrors the GPU programming model. The data movement from global memory to shared memory (matrix to thread block tile), from shared memory to the register file (thread block tile to warp tile), and from the register file to the CUDA cores for computation (warp tile to thread tile) [69]. We apply the outer product formulation to each tile, which leads to the most efficient data usage [17].

Although TBs are mapped to different layers, the code executed by each TB is exactly the same. Each TB can infer its corresponding layer index based on its TB index. For simplicity, we assume that all the layers have the same number of hidden units [49]. When each layer has a different size, we allocate registers and shared memory based on the most resource demanding layer of the network.

5.2 EXPERIMENTAL RESULTS FOR EVALUATION RNN

We implemented the proposed TB-level wait-signal scheme using CUDA 8.0 and measured the performance on Nvidia TiTan GPU. The GPU configuration is shown in Table 3. The proposed scheme is denoted as **wait-signal** in the following discussion. It is critical that the GPU have sufficient on-chip registers (larger than 6.1 MB) to store the weights to run persistent RNN [33].

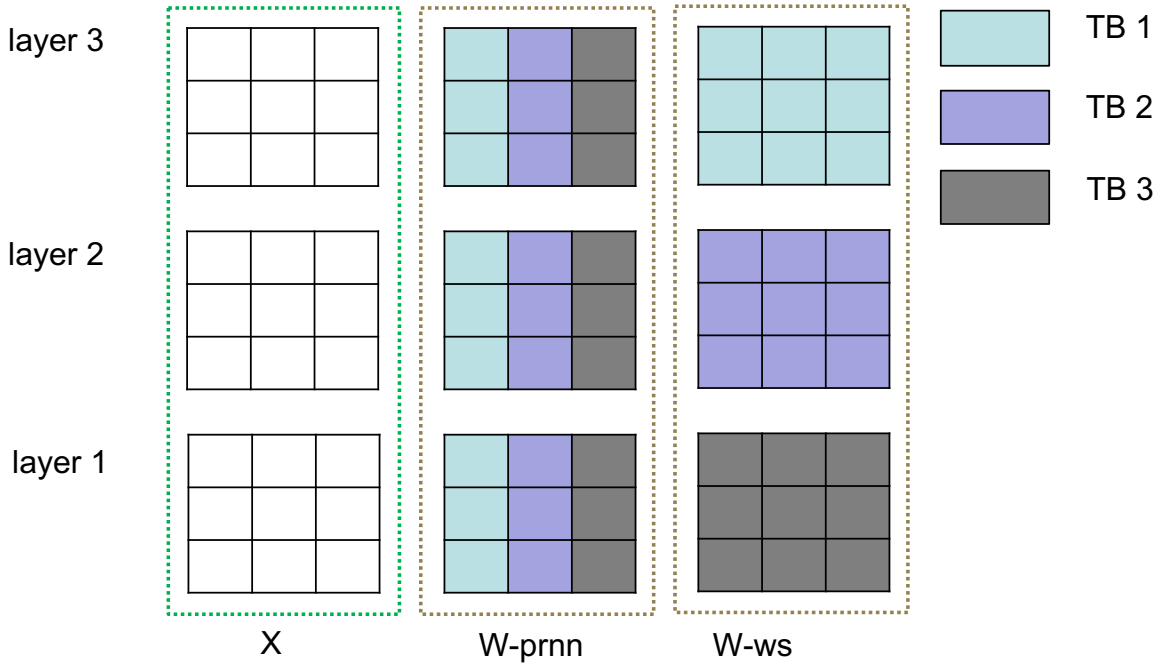

Figure 30: Comparing the TB layout of persistent RNN (W-prnn) and the proposed wait-signal mechanism (W-ws).

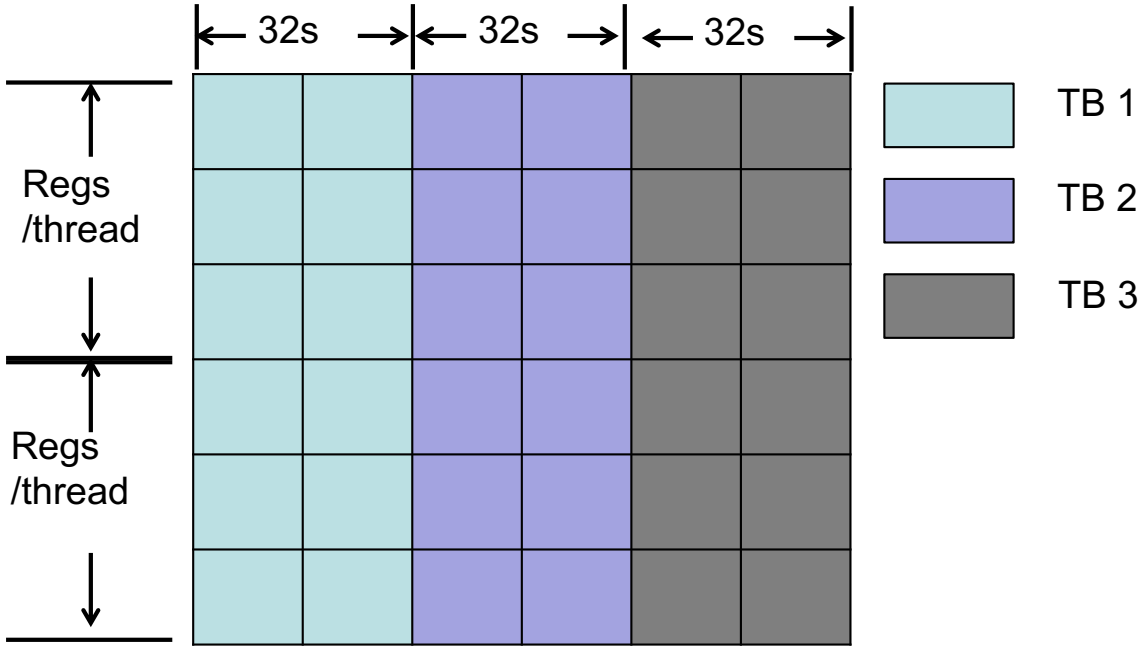

Figure 31: The layout of TB dimensions of the proposed scheme. The TB's width is multiples of 32 threads (32s).

Table 3: Configurations of GPU devices for RNN speedup comparison

GPU	Nvidia Titan
Cuda Computing Capability	5.x
warp schedulers per SM	4
SM	25
warp size	32
warp scheduling policy	round-robin
scratchpad shared memory	96KB per SM
registers	256KB per SM
maximum threads per SM	2048
L2 cache	2MB
processor/memory clock	1050/7010 MHz

5.2.1 Benchmarks

We studied 5 applications of RNNs: basic RNN, long short-term memory (LSTM), gated recurrent unit (GRU), grouped long short-term memory (g-LSTM) and phased long short-term memory (p-LSTM). The basic RNN is introduced in section 2.A which are described by formulas (1) and (2). The basic RNN is the simplest RNN which is also the fastest due to the small size of weight parameters. The basic RNN is used extensively in simple tasks such as char-level language generation, and time series.

LSTM is introduced in section 2.A with formulas 3 to 8 and is built to learn long-term dependencies. LSTM achieves this goal by removing or adding information to the cell state, via carefully regulated structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a point-wise multiplication operation. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. LSTM is at least 4 times more computing intensive than the basic RNN.

GRU is a simpler variant of LSTMs that share many of the same properties such as forget gates and input gates. GRU is less computing intensive than LSTM by combining forget gate with output gate. Hence GRU is about 3x more computing intensive than the basic RNN.

G-LSTM is another simplification of LSTM. It reduces the number of parameters and accelerates LSTM by using two techniques: 1) it applies matrix factorization of LSTM matrix into the product of two smaller matrices. 2) it partitions the LSTM matrix, as well as the inputs and states into the independent groups. Both approaches enable running large LSTM networks significantly faster to the near state-of the art perplexity while using significantly less RNN parameters.

P-LSTM extends the LSTM unit by adding a new time gate. This gate is controlled by a parametrized oscillation with a frequency range that produces updates of the memory cell only during a small percentage of the cycle. P-LSTM network achieves faster convergence than regular LSTMs on tasks which require learning of long sequences. However, it requires more weight parameters than any other RNNs mentioned.

5.2.2 Speedup Comparison

We implement the kernel stream implementation for RNN [49] and the persistent RNN [33], termed as "stream" and "PT-rnn" in the following discussion. The stream implementation is based on the source code [70]. For PT-rnn, we could not configure and reproduce the original PT-rnn implementation [33, 71] on our local machine, which is also an open issue for other users [72]. Another implementation of PT-rnn is to use the Cudnn's APIs such as $cudnnCreatePersistentRNNPlan$[17] based on the code reported in [73].

Matrix Multiplication Kernels Both baseline implementations rely on closed source libraries of *cublas* for the matrix multiplications, which are the most time consuming computing kernels [74]. Our proposed design requires modifying the matrix multiplication kernel with customized TB dimensions and local arrays to store weights. Hence we implement the proposed scheme based on the best-known open source implementation of matrix multiplications from *cutlass* [69]. For fair comparison, we also replace the matrix multiplication kernels in stream and PT-rnn baselines with cutlass kernels [69].

Stream Baseline The stream baseline [70] for wavefront parallelism is fully open sourced except for the matrix multiplication kernel. We don't need to modify other parts of this baseline. The stream baseline implemented 4 optimizations [49]: 1) combined matrix multiplications. For example, the 8 matrix multiplications of LSTM in formula (3)-(6) can be combined into 2 matrix multiplications, $\mathbf{S_{t-1}W}$ and $\mathbf{X_tU}$ where W and U are combined matrices. 2) Fusing Point-wise operations which are fully parallel. For example, the element-wise matrix multiplication can be merged into the matrix multiplication kernel. 3) combined matrix multiplication with input data in the first layer. 4) implementation of wavefront parallelisms using stream synchronization. The first three optimizations are general and are also used in the persistent RNN baseline and our proposed wait-signal RNN.

PT-rnn Baseline PT-rnn with Cudnn's APIs [17] are high-level end-to-end APIs for RNNs. For example, a single API, $cudnnRNNForwardInference$, can be invoked to run the inference of a multi-layer RNN. Because modifying the undelying matrix multiplication kernel is not possible with the given API, we implemented our own version of PT-rnn and

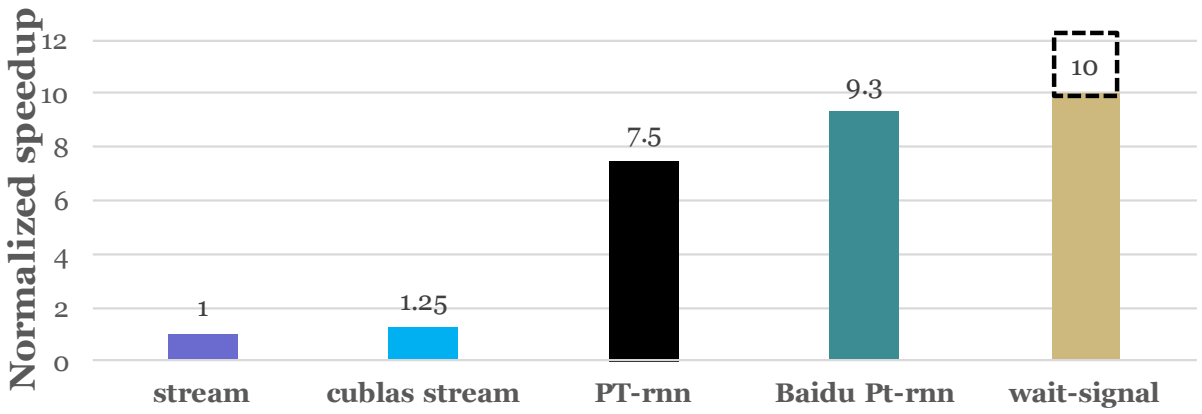

Figure 32: Speedup comparison for basic RNN.

included the key optimizations: stashing weights using on-chip registers, global barrier based on atomic operations, and best open source matrix multiplications. The optimizations we haven't implemented are: 1) assembly implementation of atomic operation and parameter loading. 2) the probably better optimized closed source matrix multiplication.

We compare our stream and PT-rnn baseline implementations with the original stream and PT-rnn implementations. We only use basic, LSTM and GRU applications in this comparison because the cudnn APIs do not support the other applications [17]. In Figure 32, 33 and 34, "stream" is our modified stream implementation with open source matrix multiplications. "cublas stream" is the original stream implementation [49, 70] with cublas closed source matrix multiplication. "PT-rnn" is our modified implementation with open source matrix multiplication. "Baidu PT-rnn" is the cudnn implementation [17, 73] of the Baidu's persistent RNN with closed source matrix multiplication and assembly optimizations. Finally, "wait-signal" is our proposed scheme. For all three applications, we observe that our baseline implementations are slower than the original baselines, which is expected. Specifically, the slowdown due to less efficient matrix multiplication is about 15% to 25%

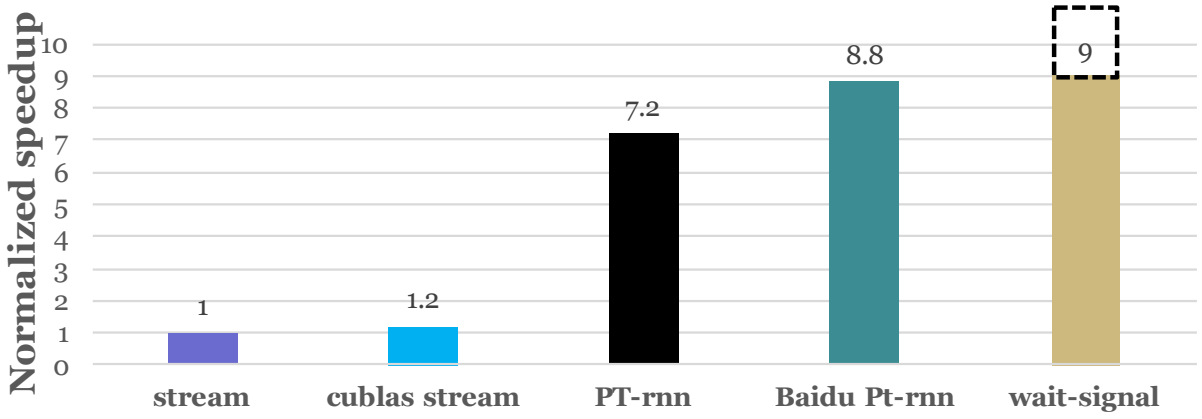

Figure 33: Speedup comparison for GRU RNN.

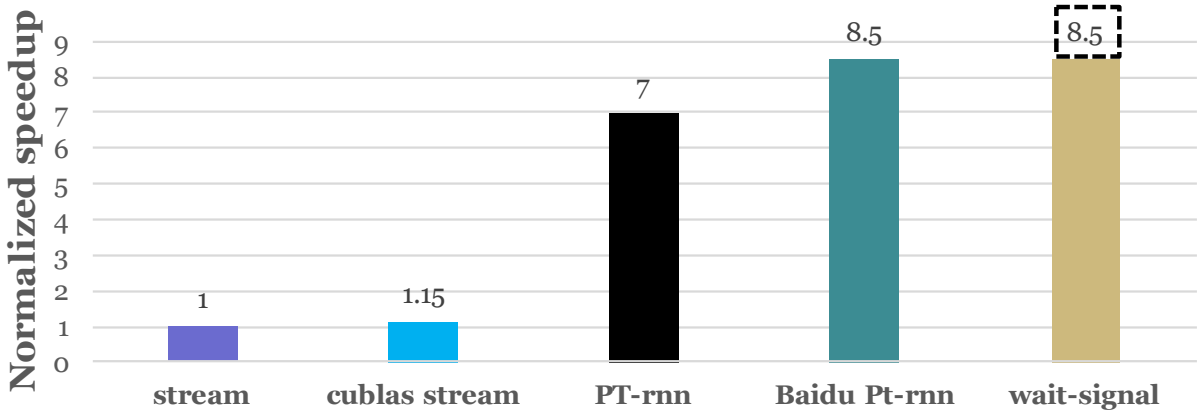

Figure 34: Speedup comparison for LSTM RNN.

and the slowdown due to lack of assembly optimization is about 5% to 10%. We argue that these two optimizations are orthogonal to the TB-level wait-signal synchronization we proposed. If we adopt the same optimizations, we conservatively estimate that there could be 20% improvement as shown by the dashed block on top of the "wait-signal" bar in all three figures.

6.0 THREAD-BLOCK LEVEL WAIT SIGNAL WITH HARDWARE SUPPORT

6.1 SYNTAX AND USAGE OF WAIT AND SIGNAL IN GPU

I propose to implement producer-consumer type synchronization among TBs using two functions; `TB_wait(int event_id, int target)`, and `TB_signal( int event_id, int target)`. The first input argument to these functions, `event_id`, is a unique event identifier associated with a hardware counter. The producer and consumer TBs are synchronized using this event identifier. The second input argument, `target`, is set to the total number of producers and consumers that are synchronizing using `event_id`.

When all threads in a TB reach `TB_wait()`, a signal is generated to atomically increment the event counter by one and put the TB to a sleep mode. `TB_signal()` is called by a producer TB to also increment the counter by one. This call is non-blocking, meaning that the producer TB will not be blocked. When the target value is reached, a signal will be broadcast to wake up all waiting consumer TBs associated with this event. This is why the `target` is set to the total number of producers and consumers. The advantages of such design is that we do not require a strict wait-before-signal order and yet, no signals are dropped. It also correctly supports the many-to-many semantics, where a group of TBs are waiting for signals from another group of TBs. Note that, as soon as a waiting TB is put to sleep, it is made eligible for a context switch to make room for pending TBs that have not made progress towards this synchronization point, including those that have not been dispatched yet. Note that in some applications, the values of the two arguments, `event_id` and `target`, may not be deterministically known at programming time and are only known at run time. Further, the correct use of `TB_wait()` and `TB_signal()` depends

on the type of the desired producer-consumer dependencies. Specifically, we observed four types of dependencies among producers and consumers: one-to-one (one producer and one consumer), one-to-many (one producer and many consumers), many-to-one (many producers and one consumer), and many-to-many (many producers and many consumers). We next consider the use of `TB_wait()` and `TB_signal()` for each of these types. We first examine dependencies that are known statically and then we examine dynamic dependencies.

6.1.1 Developing Usage for Static Dependencies

As an example of static dependencies, Figure 35 shows the computation flow of the Summed Area Table (SAT) application, which is also known as image integral [26]. The application calculates an integral image of an input image so that the pixel in the integral image equals to the sum of pixels in the upper-left region of that pixel in the original input image. In Figure 35 (a), it is assumed that SAT is implemented in conventional CUDA style where one thread maps to one point in the output image and one TB maps to one tile of the output image. Obviously the TB depends on the output of its west and north neighboring TBs. Hence there are one-to-many and many-to-one dependencies in this case. It is worth mentioning that there is a more efficient PT version [26] of SAT as shown in Figure 35 (b). In this case, each thread calculates a whole row of the output image in a "for" loop so that a TB maps to contiguous rows. Therefore each TB only depends on the TB above it, resulting in a one-to-one dependency.

The one-to-many, many-to-one and one-to-one static dependencies in the SAT application can be implemented as sketched in Figure 36. Initially, all TBs that can be dispatched to the GPU will start executing, but only TB0 at the upper left corner can actually proceed with its computation because it does not depend on anything. All remaining TBs will be put to sleep via the `TB_wait()` call. For the implementation of the diagonal wavefront propagation algorithm shown in Figure 35(a), once TB0 finishes its computing part, it wakes up two TBs to its east and south via the `TB_signal()` call, so that they both can start computing. This wave of dependencies will propagate along the lower right direction in the matrix until the last TB completes its computation. In both the Wait and Signal function calls, the first

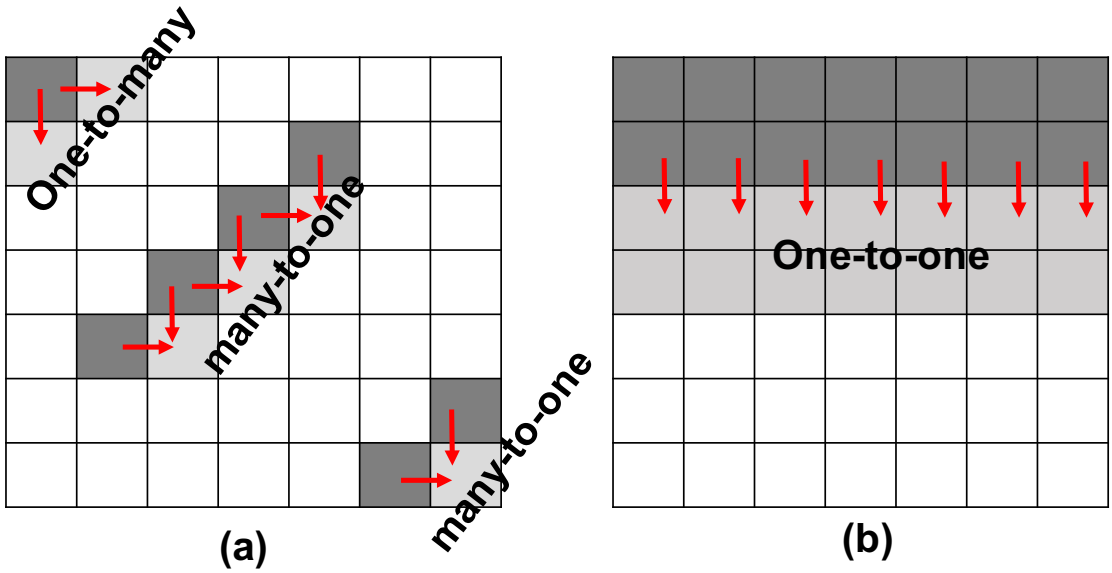

Figure 35: (a) Summed area table (SAT) with static one-to-many and many-to-one dependencies. (b) SAT using PT produces one-to-one dependencies.

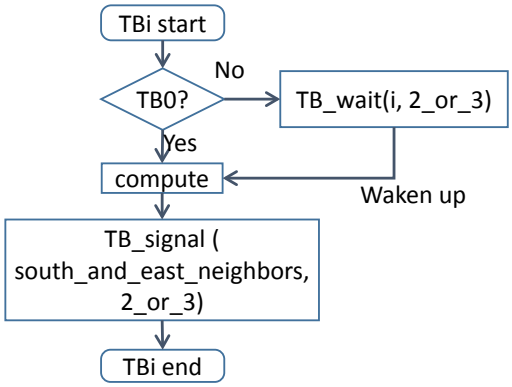

Figure 36: The use of local synchronization in one-pass SAT.

argument is the event ID that a TB is waiting on, or will signal. In this example, it is the global ID of the waiting TB since every TB is waiting on its producers to wake it up. The second argument is the total number of producers and consumers for each event. As we can see, if the consumer TB is on the top-left edge of the matrix, then the argument should be two, otherwise, it should be three. Note that the implementation of the vertical wavefront propagation algorithm of Figure 35(b) is similar except that the value of "2_or_3" is always set to two.

This SAT example illustrates how static dependencies in one wavefront sweep can be handled[1]. Other problems can be derived using the same principle in practice.

Another example that demonstrates the use of different types of dependencies is the blocked version of the All Pairs Shortest Path (APSP) of a graph. This application executes N iterations where N is the graph's diameter. In each iteration, TBs work in three steps which we explain using the adjacency matrix shown in Figure 37(a). In step 1, only the TB on the diagonal, called pivot TB is working. In step 2, the TBs in the same row and column as the pivot TB start working. In step 3, all the remaining TBs start working. So from step 1 to 2, a one-to-many synchronization is needed, from step 2 to step 3, a many-to-many synchronization is needed (all type-3 TBs are waiting until all type-2 TBs finish). A many-to-one synchronization is needed from step 3 of an iteration to step 1 of the next iteration. All the dependences are static and the number of waiting and signaling TBs for

[1]Other applications, such as Successive Over-Relaxation and Smith Waterman perform multiple sweeps.

each dependency can be determined in terms of the specific iteration number and the total number of iterations. In each iteration of the ASPS algorithm, there are only 3 steps to finish the calculation so conceptually, only 3 events are necessary. For example, in Figure 37(a), the pivot TB is waiting for the event specifying the termination of the last iteration. The TBs numbered '2' are waiting for the event specifying the completion of the pivot TB. The remaining TBs, numbered '3', are waiting for an event specifying the completion of all TB-2. Since this algorithm has N iterations, we need at most $3N$ events to enforce all dependencies.

In some applications, even when the dependencies are statically known among TBs, the number of events that correspond to those dependencies may be overwhelmingly high. Given that an event ID is implemented as a counter, having a large number of events is clearly not a scalable solution, especially if hardware counters are used. We will discuss this problem and outline potential solutions in the context of the SAT and APSP applications. As described earlier, the number of event IDs in the SAT application is $O(N^2)$, where N^2 is the total number of blocks in the adjacency matrix, which can be very high. However, because of the diagonal nature of the wavefronts, it is possible to reuse the event identifiers after each wavefront, which reduces the total number of events to N, the maximum number of blocks in a diagonal. Similarly, in the ASPS application, the $3N$ events may be reduced to a total of just 6 events by observing that the pivot TB will become a type '3' TB in the next iteration. Hence, once the pivot TB finishes the calculation in the current iteration, it signals type '2' TBs and goes to sleep waiting for an event that will wake it up as a type '3' TB in the next iteration. This event must be a new one as the current type '3' TBs have not completed yet. Analogously, the other two events in the next iteration must also be new. However, after two iterations, all 6 events can be recycled safely, reducing the event count to $O(1)$.

The above "event recycling" solution is dependent on the specific algorithm. Another solution, "event grouping", may also be used to reduce the number of events (or reduce further the number of events). For example, in the SAT algorithm, assuming that the events ids for each wavefront are $1, ..., n$, $n \leq N$, it is possible to group every k consecutive events, $i, ..., i + k$, into one event, thus reducing the number of events by a factor of k. The downside of event grouping is that a false dependency is formed and performance loss is incurred. Of course, the larger the number, k, of grouped events, the larger the performance

loss. Moreover, events grouping should be carefully done in a way that does not cause deadlock. Clearly, event grouping and event recycling need careful programming efforts to avoid deadlock and to minimize the potential performance loss. To mitigate these drawbacks, we introduce a hardware based solution to virtualize the hardware event counters by storing them in global memory while caching the currently active counters on-chip. For instance, if N^2 events are used for the SAT application, the events counters corresponding to the current wavefront (at most N event counters) will be cached on-chip. We will discuss this hardware solution further in section 6.3.1.

6.1.2 Developing Usage for Dynamic Dependencies

Very often, the dependencies among TBs are only revealed at run time. Hence the number of producers and consumers associated with a particular event are only known at run time. Consequently, the second argument, `target`, in the wait and signal function calls cannot be determined statically. Figure 37(b) will be used to explain this behavior during the execution of the Breadth First Search (BFS) algorithm. At any given iteration, nodes that have the same depth from the root form a frontier (three frontiers are shown in Figure 37(b) with each node labeled with the ID of the TB processing that node). Each node is assigned to a thread randomly, so a TB may have nodes from different frontiers. For example, TB1 contains nodes in both top and middle frontiers of the graph.

Any TB that is responsible for one or more node in the current frontier traverses the neighbors of these nodes to decide which TBs should be in the next frontier. All TBs in the current frontier must finish computing before the TBs in the next frontier can start. Hence, the TBs in the next frontier are only generated at run time, forming a dynamic dependency between the current and next frontier in a graph. As we can see, the synchronization is not global, because frontiers are mostly a subset of TBs. Hence, a local synchronization is necessary with the TBs in the current frontier being the producers and the TBs in the next frontiers being the consumers, which are only known dynamically. In the example in Figure 37(b), suppose TB1 and TB2 have nodes in the first frontier generated by the root. All TBs are initially put to sleep except for the root. Suppose now that the root has waken

up the top frontier consisting of TB1 and TB2. They perform some computing, then traverse their neighbors and search for the TBs in the next frontier, in this case TB1, TB2 and TB3, and sets the corresponding bits of a global flag array that is used to mark the TBs in the next frontier. These steps are all done in parallel between TB1 and TB2, so the last one that completes the steps will wake up all TBs found to be in the next frontier and reset the flag array. The producers will then go back to sleep, until they are waken up again. As TB1 is in both the top and middle frontier, it will first signal itself due to it being in the next frontier and its event counter increases to one. After that, it will call a `TB_wait()` to find out that its event counter now reaches two, so it should wake up to do the computation associated with the middle frontier.

This BFS algorithm is sketched in Figure 38, where the dynamic dependency is handled in the two highlighted steps, one for marking TBs in the next frontier and one for signaling them. The next frontier is determined by traversing all the neighbors of the current frontier. Note that, although the dependencies are many-to-many between TBs, the dynamic nature of the frontiers forces us to transform each many-to-many dependency to multiple one-to-one dependencies (with the help of a flag array). Specifically, only the last TB in the current frontier signals the TBs in the next frontier. Moreover, it signals each TB, j, separately using an event whose id is j. Hence, each TB has one event associated with it and the target value of the wait and signal calls for that event is two.

Finally, we note that it is not possible to group or recycle events due to the dynamic nature of the TB frontiers, unless we group all events into one, which is equivalent to using a global synchronization after each frontier. Yet, the event virtualization solution, which is transparent to the user, can be readily applied.

6.2 COMBINING KERNELS TO AVOID GLOBAL SYNCHRONIZATION

In some applications, kernels for different tasks are launched sequentially to enforce dependencies. By combining the sequence of kernels into one kernel, the TBs can through the functionality of the sequence of kernels with dependencies enforced through Wait and Sig-

nal. We illustrate this idea using the LU decomposition application which computes two triangular matrices L and U from a given matrix A [29].

We examine a blocked version where LUD is computed for a block recursively from the top left corner to the bottom right corner as shown in Figure 39. In each iteration, there are three types of blocks and each block is computed by a TB. All the perimeter TBs depend on the diagonal TB and each internal TB only depends on two perimeter TBs as indicated by the black arrows. As shown in Figure 40(a), with repeated kernel launching, three different kernels are launched sequentially for each iteration to compute *diagonal*, *perimeter* and *internal* blocks. Consequently, an internal TB has to wait for all the perimeter TBs, when in fact it only depends on the two perimeter TBs affecting it [29].

Figure 40(b) shows how to utilize wait signal to combine the functionalities of the three kernels into one kernel, $lud_{waitsignal}$.We will show that the resource requirements of the combined kernel are determined by the most demanding kernel for each resource.

6.3 ARCHITECTURE DESIGN

We propose an architecture design to implement TB-level wait and signal. As shown in Figure 41, a global wait signal controller is implemented with input/output queues and event counters. We walk through the process of a wait-signal event to explain the design.

(1) When a warp scheduler issues a `TB_wait()` instruction in a warp, it calls `__syncthreads()` which sets the bit of the warp in a "bitset" for the TB. When all the bits in the bitset for that TB are set, the entry corresponding to that TB in a "TB status array" is set to 1, representing a "sleep" mode (none of its warps are eligible for issue). Each TB also has an entry to store the *event_id* that this TB is waiting for.

(2) When a warp issues a `TB_signal()` instruction, a message is generated directly without waiting for the other warps in the same TB. The message is enqueued into the Signals TX buffers (the same one used by the wait instruction). The message includes both *event_id* and the *target* value.

(3) The message at the head of the queue in the SM is sent to the input queue of the global wait-signal controller through the on-chip networks. We assume the delay of network communication is 20 cycles. If multiple messages arrive at the same cycle, they are enqueued in a round robin fashion.

(4) Each cycle, the controller fetches one message at the head of the input queue and adds one to the counter corresponding to the *event_id*. The counter value is compared with the value of *target*. If equal, the event counter is reset to 0 and a broadcast message is generated and enqueued to the output queue. This message includes just the *event_id*.

(5) Each cycle, the controller extracts the head message of the output queue and broadcasts it to all the SMs. The message is enqueued to the Wait Rx buffer of each SM.

(6) Each cycle one message is dequeued from the wait Rx buffer. The $event_{id}$ in the message is compared with the event ids of all the waiting TBs in the "TB status array". If a match is found, the waiting TB's status entry is reset to 0, the TB is waken up and the warp schedulers are notified that the TB's warps can be issued. If no match is found, then the event is not meant for any TB on this SM and the message is deleted.

6.3.1 Virtualizing Event Counters

Clearly, the number of hardware counters are fixed for a given implementation, and it may not be possible to provide sufficient hardware counters for each event using the scarce on-chip resources only. As discussed in Section 6.1, the number of event counters required can be as many as the number of TBs. Since a GPU kernel can have hundreds of thousands of TBs, the size of all event counters could put a large burden on the on-chip storage. Rather than storing all the event counters on-chip, we propose to store them in the global memory and only cache a small portion of these counters in the on-chip wait-signal controller. Hence virtually, programmers can use an unlimited number of events. Like a cache, the on-chip event counter buffers are initially empty, and a global memory load is issued to fetch a counter from global memory to on-chip buffers when an event counter is required. When the buffer is full, an event counter is evicted based on the least recently used policy, with priority for eviction given to events that are reset (have been already signaled).

6.3.2 Integration with Partial Context Switching

In cases that the kernel has more TBs than the SMs can concurrently execute, existing TB-level wait-signal approaches, such as memory flag implementations, as well as the hardware approach discussed above, can potentially deadlock. Specifically, the producer TBs may not be dispatched if all SMs are fully occupied by consumer TBs waiting for the target values of events to be reached. At least some of the waiting TBs have to be swapped out of their SMs to release their resources and allow the producer TBs to be swapped in.

The Full SM context switch scheme proposed in [56] performs context switch at the SM level. It can solve the deadlock problem described above but would be very inefficient. Specifically, to avoid deadlock, we have to wait for all TBs on an SM to be blocked (waiting) before swapping them out. This will underutilize the SM since many TBs may be blocked while waiting for all TBs on the SM to be blocked. Moreover, the simultaneous swap out of all the TBs on an SM creates a extensive load on the memory bandwidth.

A more efficient solution to avoid Deadlock is to integrate the partial context switch (PCS) technique proposed in [55] into our design. This technique performs context switch at the TB granularity. As shown in Figure 42, we augment the context switch hardware proposed in [55] by adding three queues to each SM: ready, wait and swapped-out queues. A TB can be in only one of the three queues, with all TBs initially put in the ready queue. When a TB executes a `TB_wait()`, its entry is moved from the ready to the wait queue (transition 1 in Figure 42) but its context remains on the SM and its warps are marked as "sleep" in the "TB status array" as described in the previous section.

As in the design proposed in [55], the global TB dispatcher is responsible for dispatching TBs to SMs. If there are TBs in the unallocated TB queue that have not been dispatched yet, the dispatcher will try to find an SM that has free space for a new TB. Typically this is limited by 4 conditions: the maximum number of TBs per SM, the maximum number of threads per SM, the available registers and the available shared memory. To support Wait and Signal, we modify the dispatcher such that, if all SMs are full while there are still unallocated TBs, it will look for an SM that has a non-empty wait queue and swap out a TB from that wait queue to make room for an unallocated TB. The TB that is swapped out

is moved to the swapped-out queue (transition 2 in Figure 42). The context switch engine is responsible for generating store instructions for storing the context (registers, shared memory and SIMT stack) of the swapped out TB into the global memory to make room for a yet unallocated TB to be dispatched. If all SMs have empty wait queues, which means that all TBs are active, the unallocated TBs stay in the unallocated queue until one TB is moved to the wait queue.

The entry corresponding to a TB in the ready/wait/swapped-out queues stores the information of where the TB's context resides. Moreover, the entry in the swap-out queue will store the global memory address and offset that stores the context. The TB's SIMT stack information are also stored in the entries of the wait and swapped out queues. It should be noted that when a TB's entry is moved to the wait queue, that TB does not count as an active TB on the SM so that we can relax the maximum number of TBs constraint. The price is that the TB's SIMT stack needs to be stored on-chip. This technique is similar to Chimera [54] except that we only consider the maximum number of TB constraint. It is worth mentioning that even when a TB's context is swapped out, its status entry continue to be stored on the TB status array.

When the waiting event of a TB in the wait queue is cleared by a signal, that TB is moved to the ready queue (transition 3 in Figure 42). However, when the waiting event of a TB in the swapped-out queue is cleared by a signal, it is marked as ready in the TB status array but remains in the swapped-out queue. It is moved to the ready queue only after it is swapped in as described next.

The functionality of the context switch engine proposed in [55] is also augmented to monitor the status of the TBs in the swapped-out queue. If that queue contains a TB that is ready (according to the TB status array) and the wait queue is not empty, then it swaps out the TB at the end of the waiting queue (and moves it to the swapped out queue) and swaps in the ready TB (and moves it to the ready queue). The intuition for picking the last TB in the wait queue is that this TB is least likely to be waken up in the near future among all the TBs in the waiting queue. Many techniques to reduce context switch overhead can be applied. Firstly, there might be idle registers and shared memory when a SM cannot host more TBs because the number of threads and TBs reach the limit. If such idle resource is

large enough to save a TB's context, we can keep that TB's context on-chip (and mark it as such in the swapped out queue entry). Secondly, not all the register values need to be saved; some are only used for intermediate values which do not need to be stored/restored at context switch. Liveness analysis is proposed in [59] to identify such registers.

6.3.3 Specifying the order of TB Dispatching

A challenge with wait-signal synchronization is imbalanced load in GPU. Empirically, TB dispatcher distributes TBs to SMs evenly for utilization and load balancing [75]. However, when there is producer-consumer relationship among TBs, the even distribution of TBs does not imply even workload among SMs. For example, it is preferred that independent producer TBs are distributed to SMs evenly. However, in Figure 39, the leftmost column perimeter TBs are producers of internal TBs, but the perimeter TBs could be dispatched to the same SM if the dispatcher follows a row major policy. Consequently, one SM is loaded with heavy producers while other SMs are mostly idle because the consumers are sleeping.

Inspired by software techniques on specifying TB_{ID} with SMs and dynamic job allocation [76], we propose to extend kernel launching configuration syntax with an optional argument, TB_order. The default parameters of kernel configuration includes kernel dimensions, TB dimensions, dynamic shared memory size, stream ID and so on [11, 40]. TB_order is a pointer to an integer array that specifies the order of TBs to be dispatched to SMs in a round robin way to achieve load balancing. This option of defining the TB dispatch order will not be used for applications with dynamic dependencies, e.g., BFS, but will be useful for static dependencies known to the user. The array is initialized by user and directly defines the order in which TBs should be dispatched. For example, the perimeter TBs in Figure 39 can have the same value in TB_order, and be evenly distributed among SMs. The array can be passed to the TB dispatcher through the driver which sets up kernel during a launch.

In addition to load balancing, another advantage of defining TB dispatch order is reducing the cost of context switch. Unnecessary context switch could happen when a consumer TB is dispatched before its producer TB, but quickly put to sleep and swapped to memory. If its produced TB is dispatched first, even if the consumer TB is put to sleep, it could be woken

up much sooner because its producer started earlier. This is particularly effective in single-pass wavefront applications. We observed that context switches are completely removed if TBs are launched in wavefront order and the TBs in each front fit on the GPU (see section 6.4.3). Furthermore, with the capability of load balancing, all SMs will have active TBs running which can hide the latency of context switching. Our evaluation will show such significant improvement in performance due to load balancing and reduced context switches. The feasibility of the proposed specified TB dispatching order is based on the correct understanding of the mechanism of the existing TB dispatcher of GPU. TB dispatcher is known as the Gigathread scheduler for NVIDIA GPUs [2] and workgroup dispatcher for AMD GPUs [77]. Unfortunately, there is no public disclosure about the architecture details of the TB dispatcher as stated in prior works [76, 78, 79, 75]. In both Nvidia's Pascal [2] and AMD's Vega [77] architecture whitepapers, the description of the TB dispatcher's functionality is limited to two points: 1) it is responsible for dispatching TBs/Workgroups to SMs/CUs and 2) it is efficient at managing context switch between applications. In fact, the TB dispatcher is capable to dispatch TBs in arbitrary orders [67, 80]. Several attempts have been made to reverse engineer the default TB dispatching policies and the results reveal that TBs are dispatched in an almost/approximate round robin order to each SM [75, 78, 76, 81]. It is also shown that the TB dispatcher has the liberty to change this round-robin dispatch order to optimize the performance based on kernel and TB dimensions [81]. More sophisticated TB scheduling is proposed to exploit the inter-TB locality and alleviate memory congestion without assuming the architecture details of the TB dispatcher [79]. In conclusion, based on these prior works, we believe that the TB dispatcher has enough flexibility to implement the proposed TB-dispatch order scheduling.

6.4 EXPERIMENTAL RESULTS OF THE PROPOSED WAIT-SIGNAL SCHEME

6.4.1 Experimental Methodology

In the following experiments, we measure the GPU kernel execution time including function calls such as atomic functions and the proposed $TB\_wait()$ and $TB\_signal()$. We compare the Wait-Signal scheme with three kinds of baselines: 1) the original implementation of ReKL which essentially implements a global synchronization; 2) memory flags which implements a TB-level synchronization as the Wait-Signal scheme but without hardware support; 3) dynamic parallelism (DP) where kernels are launched at the GPU side and only kernel-level synchronization is used for dynamic dependency applications such as graph applications.

6.4.2 Results for Graph Applications

We first examine the total execution time of three graph applications, BFS, SSSP and APSP, using real data sets obtained from the parboil benchmark [82]. BFS and APSP are introduced in Section 6.1. Single Source Shortest Path (SSSP) [] is another shortest path problem which requires finding the path from a source node to all other nodes in a weighted graph such that the weight of the minimum-weight edge of the path is maximized. The results presented in this section are for these four graphs: 1M, NY, SF and UT. The number of nodes for these graphs ranges from 240K to 1M. Since the graphs are large, the schemes that do not use context switching will deadlock when the number of nodes increases. Hence, we ran two types of experiments: 1) We ran the algorithms on a reduced sample of each graph containing 40k nodes and 2) we ran the algorithms on the original graphs with all the nodes. Each TB has 512 threads.

As shown in Figure 43, with sampled small graphs, the proposed Wait-Signal scheme outperforms CPU ReKL and GPU DP by an average of 1.95x and 1.65x respectively. This is mainly due to the removal of over-synchronization to obtain more parallelism among independent TBs. The average speedup over memory flags is 22%, which is mainly due to removing the busy wait and global memory access for flags.

Figure 44 shows the speedup with the original inputs with up to 1M nodes. The proposed Wait-Signal scheme outperforms CPU ReKL and GPU DP by 1.21x on average. Due to the large data size, partial context switching is deployed. The improvement is smaller compared with smaller inputs because: 1) the kernel execution time is significantly longer so the API call overhead is less critical for ReKL and DP. 2) For the proposed Wait-Signal, the context switch overhead increased with large data size. It should be noted that 1) memory flags deadlocks with this data size because busy-waiting prevents context switching, so they are not shown in the figure. 2) PT can solve the deadlock problem but could cause significantly slowdown due to sequentialized job distribution [43]. 3) TB ordering is not used for the proposed Wait-Signal since the dependency is dynamic.

6.4.3 Results for Wavefront Applications

We compare EffiSync with ReKL and memory flags-PT for three wavefront applications, SAT [26], SOR [64] and LUD [83]. We vary the data size in our experiments to better understand the scalability of each scheme. For these applications with static dependencies, the TB dispatch order can be determined based on the wavefront ID. We denote such implementation as "wait-signal TB-order". The "wait-signal" curve represents the default wait-signal without specifying dispatch order but with context switch support. We also implement the PT model with memory flags (Peerwave) with optimal synchronization interval [26]. We used Row-to-SM Peerwave without hyperplanes. The input dimension of tile varies with each application. The number of TBs equals to the number of SMs. LUD was described in Section 6.2. The SAT, described in Section 6.1.1, and SOR are defined by formulas (1) and (2) respectively, where A is the output array, $\hat{A}$ is the value in previous iteration, and P[i,j] is the input array.

$$A[i,j] = P[i,j] + A[i-1,j] + A[i,j-1] - A[i-1,j-1] \tag{6.1}$$

$$A[i,j] = A[i-1,j] + A[i,j-1] + \hat{A}[i,j] + \hat{A}[i+1,j] + \hat{A}[i,j+1] \tag{6.2}$$

Both SAT and SOR applications exhibit many-to-one and one-to-many dependencies. The SOR and SAT are similar except that 1) SAT uses three values to compute a point whereas the SOR uses 5 values to compute a point. 2) SAT is a one-pass application while

SOR is multi-pass which iterates until it converges. In our experiments, we ran only 10 iterations[2]. The comparison of speedups for the wavefront applications are shown in Figures 45, 46 and 47 respectively. The default Wait-Signal scheme achieves 1.2x∼2.1x speedup over the baseline ReKL due to higher TB parallelism."Wait-Signal TB-order" achieves the highest speedups, 2x∼2.5x across varying data sizes. These results demonstrate the importance of removing over-synchronization and improving parallelism among TBs. The gap between Wait-Signal and "memory flags+PT" shows the effect of removing memory busy-wait. The gap between "Wait-Signal TB-order" and other schemes proves the significance of improving load balance and reducing the cost of context switching.

The memory flag implementation deadlocks for large data sizes. Its PT version resolves the deadlock problem and achieves good performance except for small data size because PT code uses fixed number of TBs [26]. However, for small data size, fewer TBs can be used than that of PT so that fast intra-TB communication can be used with shared memory than inter-TB communication. The Wait-Signal scheme allows the majority of TBs that are not in the current wavefront to be in sleep mode. In contrast, all the TBs in the memory flag implementation are active despite the fact that only the TBs in the wavefront are making progress.

Figure 47 shows the performance comparison of the different schemes for LUD. The proposed Wait-Signal achieves more improvement over the baseline than SOR or SAT because LUD can take advantage of prefetching opportunities of some ready data while waiting for other data that is not ready. Specifically, while the perimeter TBs (see Figure 39) are executing in some iteration, an internal TB can start to load the value of its own block from the previous iteration to shared memory. It should be noted that such technique works best when no context switch is triggered. Otherwise the prefetched data may instead cause pollution.

[2]A large number of iterations is needed for convergence, which increases the simulation time without affecting the main execution characteristics.

6.4.4 Overhead Analysis

The hardware overhead of EffiSync mainly includes the centralized wait signal controller and additional hardware at each SM as shown in Figure 41. We use 20 bits to represent the event ID which correspond to up to one million event counters. For the experiment, we use 4,000 on-chip event counter buffers[3]. We allocate 8 bits for each event counter since recent GPU support as small as 8-bit precision data access [84]. As a result, the event counter buffer for virtualization is 14KB. For each SM, the number of entries of the wait signal buffers should not be more than the maximum number of TBs per SM which is 32 in modern GPU. Hence the input and output queue of the centralized controller has 960 entries, assuming the GPU has 30 SMs. In conclusion, the total size of all the event ID queues/buffers and the on-chip event counters are 78 KB and 9 KB respectively. The overhead of the TB order array is estimated to be 1.3KB. In summary the total overhead is less than 1% of the total on-chip storage.

We developed software event grouping (Section 6.1.1) and hardware event virtualization (Section 6.3.1) which aim to reduce the number of events and on-chip overhead to store them. Grouping event counters will create false dependencies among the signaling and waiting TBs, while virtualizing event counters may generate off-chip memory traffic upon a counter miss. In Figure 48, we compare the two approaches using LUD with a large data size that requires 4000 event counters but vary the number of on-chip counters along the x-axis. Clearly, the more on-chip counters we provide, the higher speedups can be achieved. Further, if we were to restrict the number of on-chip counters, event virtualization always outperforms event grouping indicating that false dependencies are more critical than off-chip memory fetch due to count misses.

[3]Current GPUs support 20480 hardware locks. We ran a simple kernel to measure the number of locks used by atomic operations.

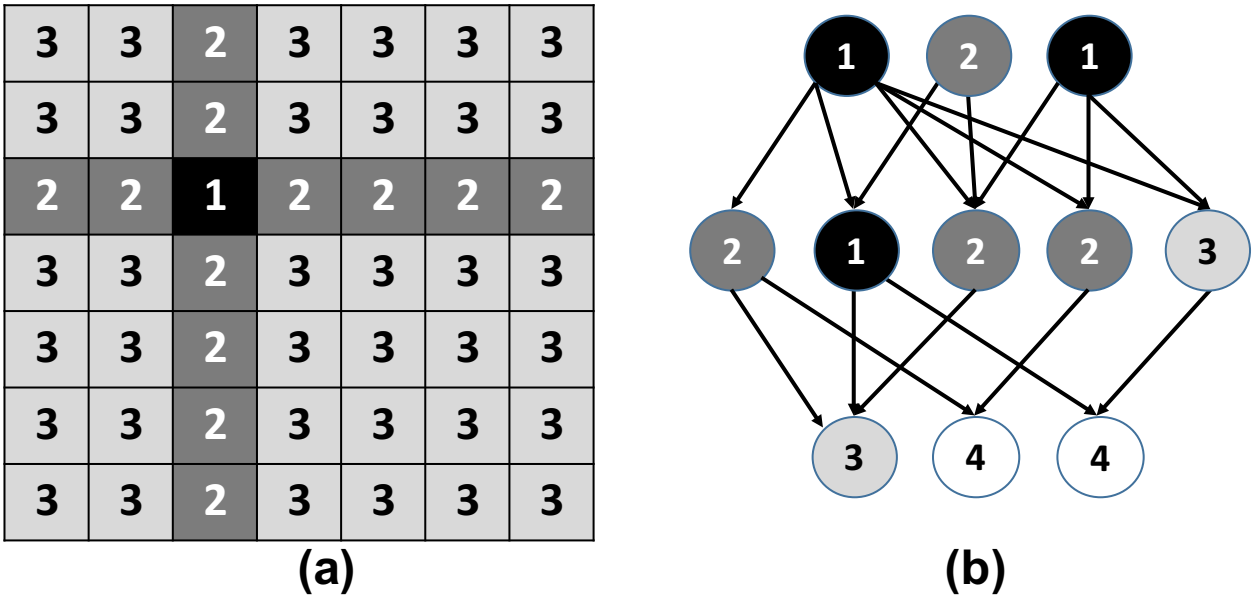

Figure 37: (a) All pair shortest path (APSP) with static many-to-many dependencies. (b) Breadth first search (BFS) with dynamic dependencies (the number in each node indicates the TB that is processing that node).

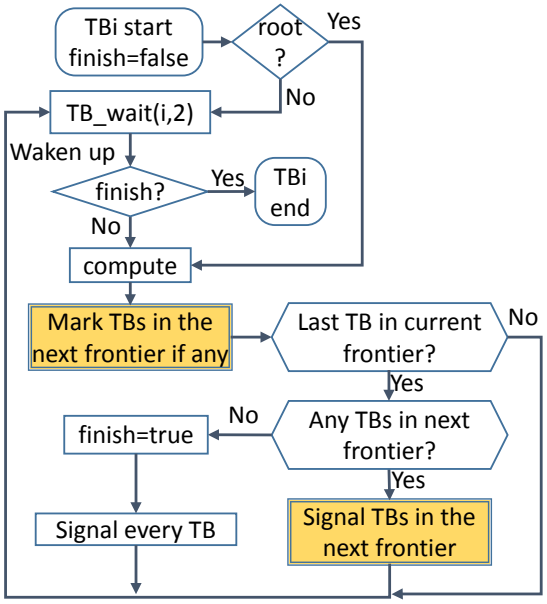

Figure 38: The use of wait and signal primitives to enforce dynamic dependenciesin the BFS algorithm.

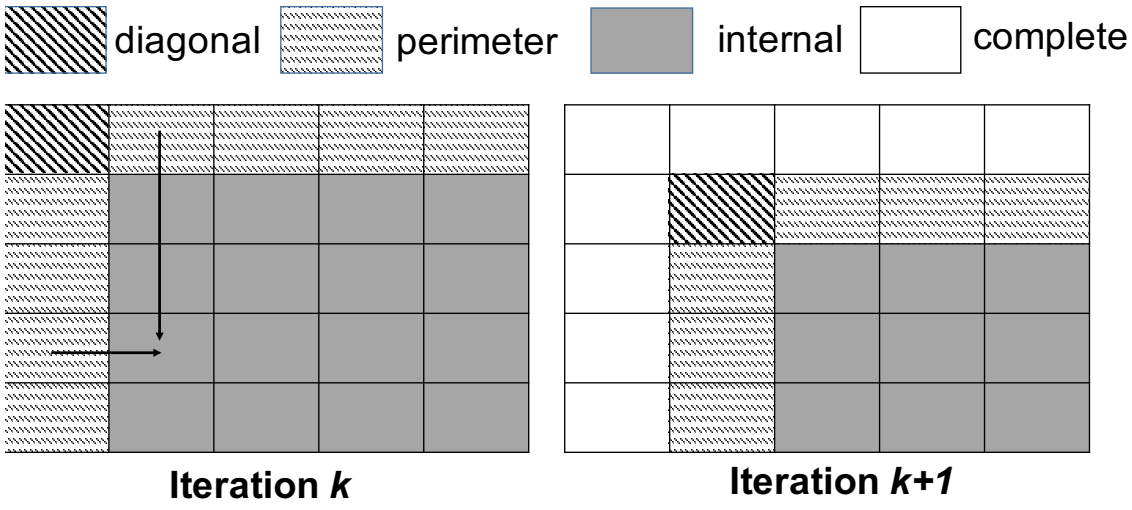

Figure 39: TB dependencies in the LUD application.

```
1.  void lud(....){
2.      for(int i=0;i<N;i++){
3.          lud_diagonal<<<...>>>(...);
4.          lud_perimeter <<<...>>>(...);
5.          lud_internal <<<...>>>(...);
6.      }
7.  }
```

(a)

```
1.  __global__ lud_waitsignal (...){
2.      for(int i=0;i<N;i++){
3.          if diagonal TB:
4.              if i!=0:  TB_wait(...);
5.              lud_diagonal(...); TB_signal(...);
6.          if perimeter TB:
7.              TB_wait(...); lud_perimeter(...); TB_signal(...);
8.          if internal TB:
9.              TB_wait(...); lud_internal(...); TB_signal(...);
10.     }
11. }
```

(b)

Figure 40: Skeleton of LUD (a) Global synchronization through repeated kernel launch. (b) wait-signal using a combined kernel

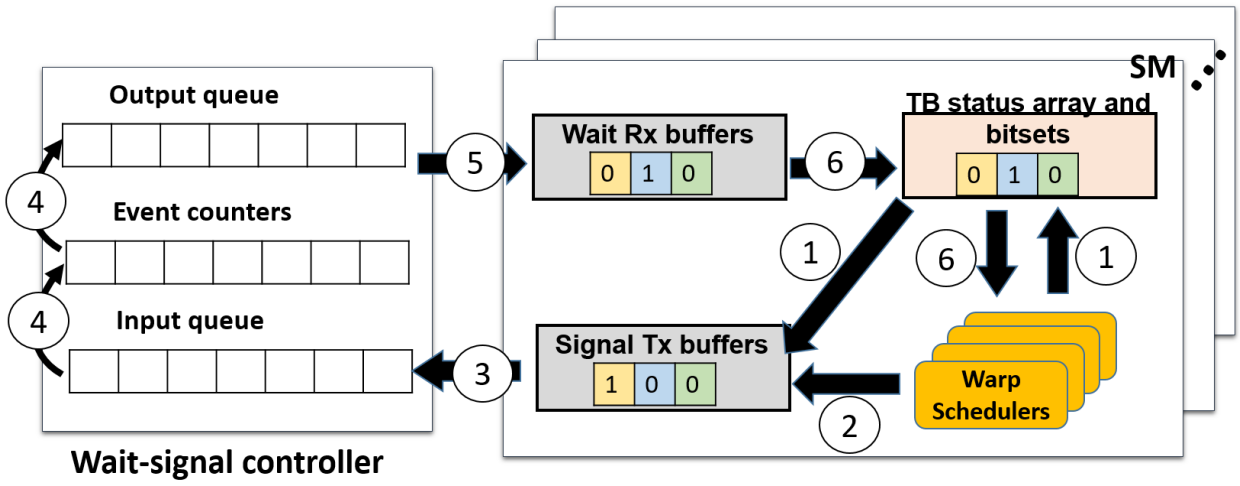

Figure 41: Microarchitecture support for Wait and Signal

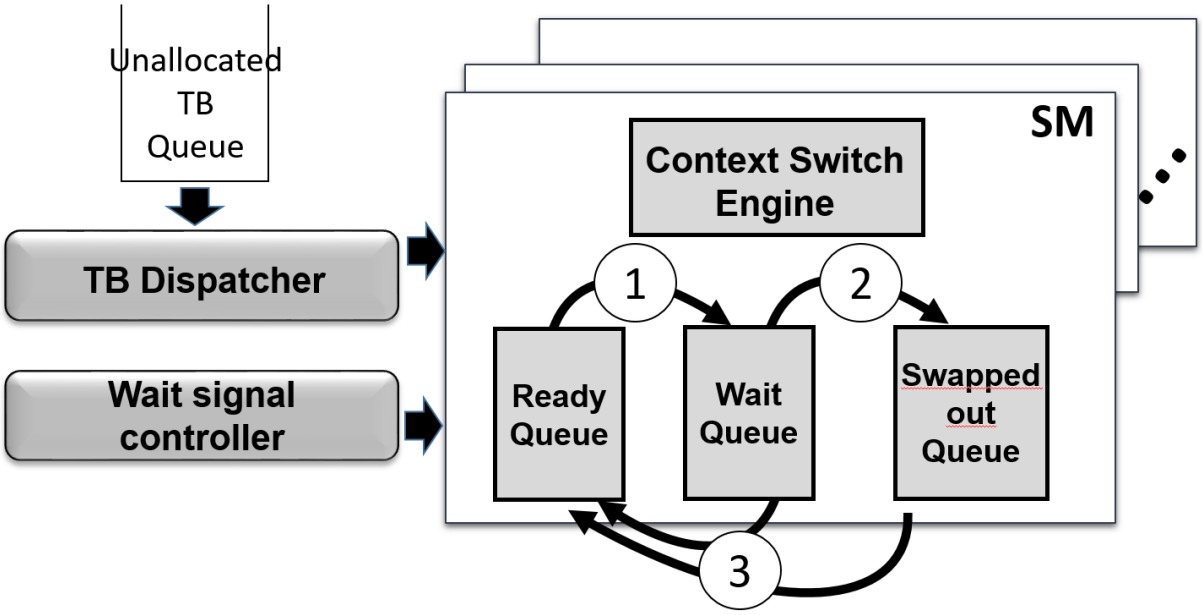

Figure 42: Integration with context switch engines.

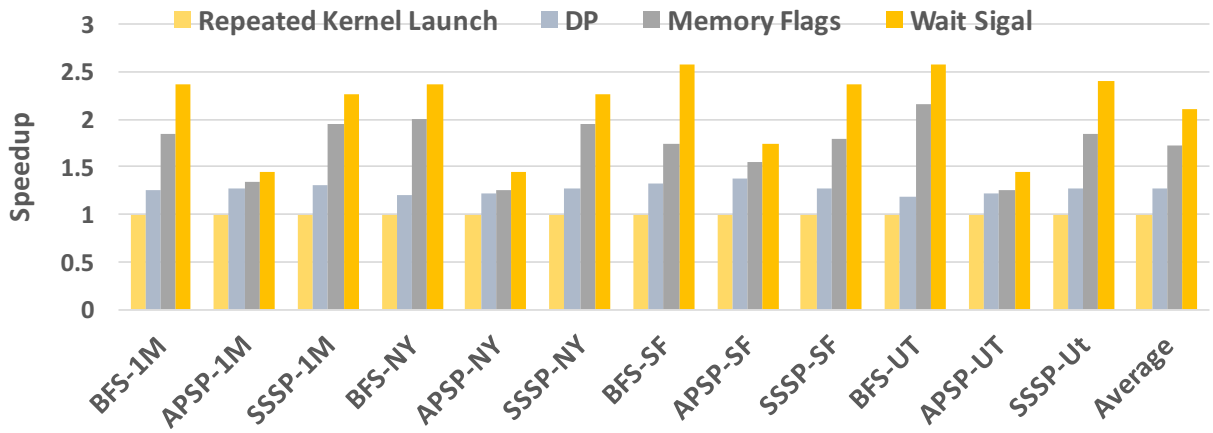

Figure 43: Comparing graph applications with sampled data sets (40k nodes).

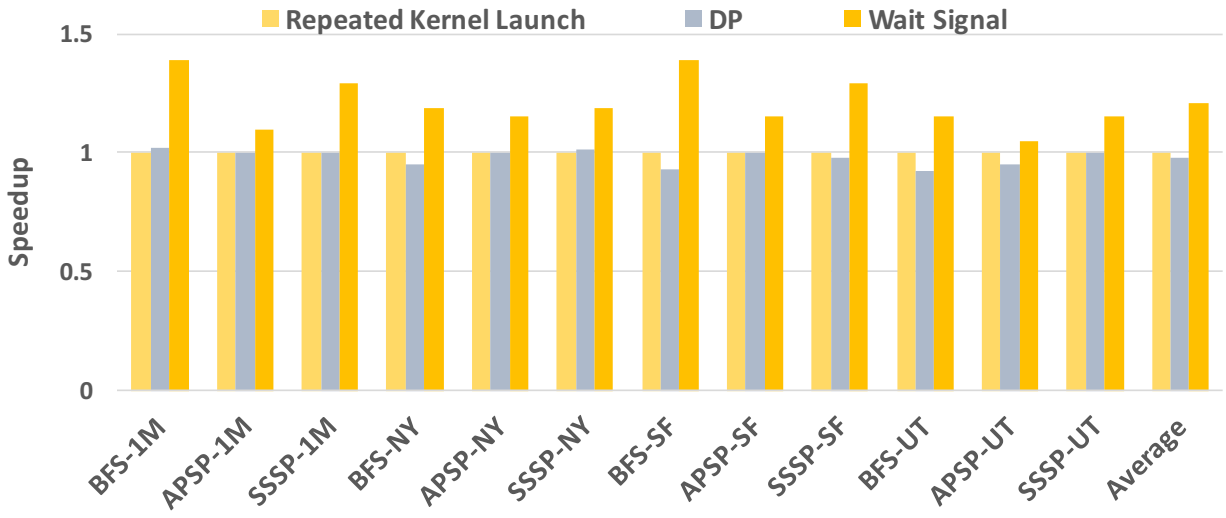

Figure 44: Comparing graph applications with the original data sets (1M nodes).

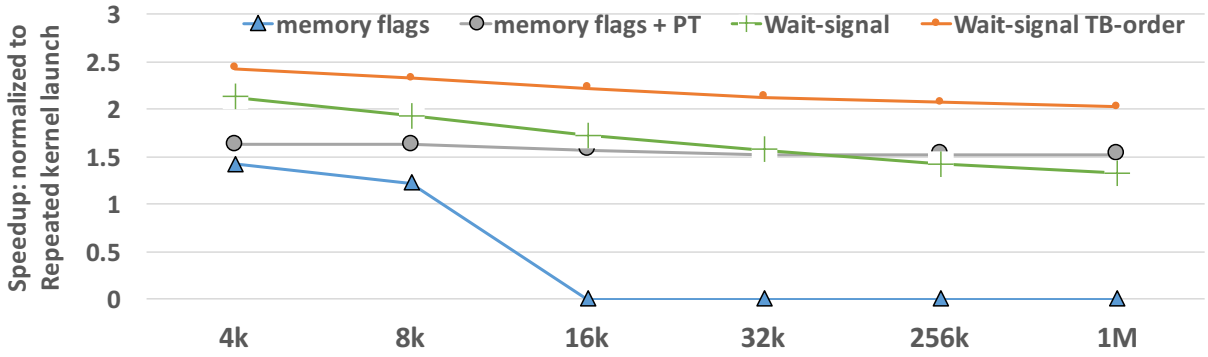

Figure 45: Comparing SAT with varied data size.

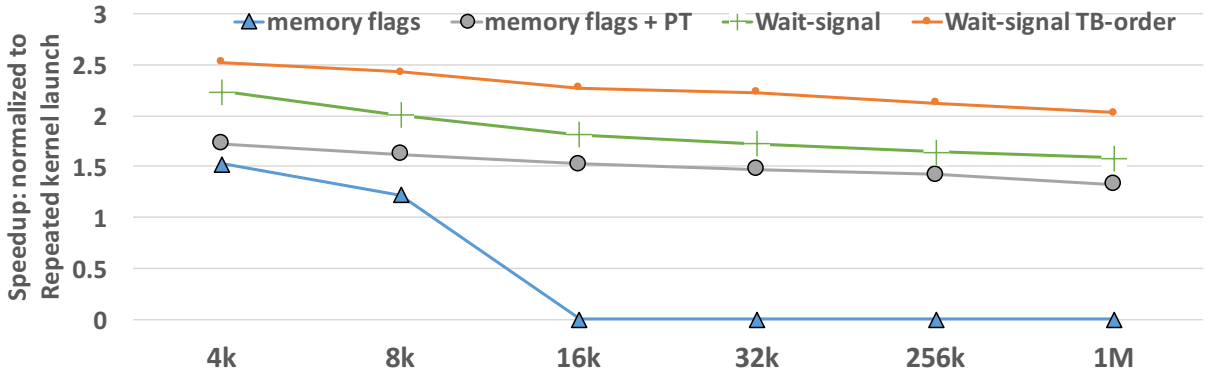

Figure 46: Comparing SOR with varied data size.

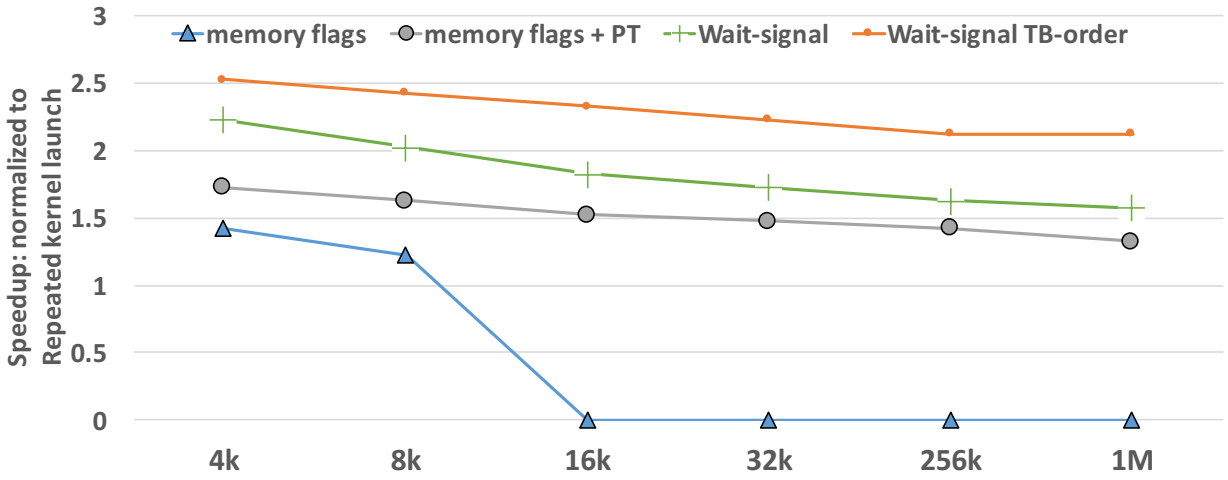

Figure 47: Comparing LUD with varied data size.

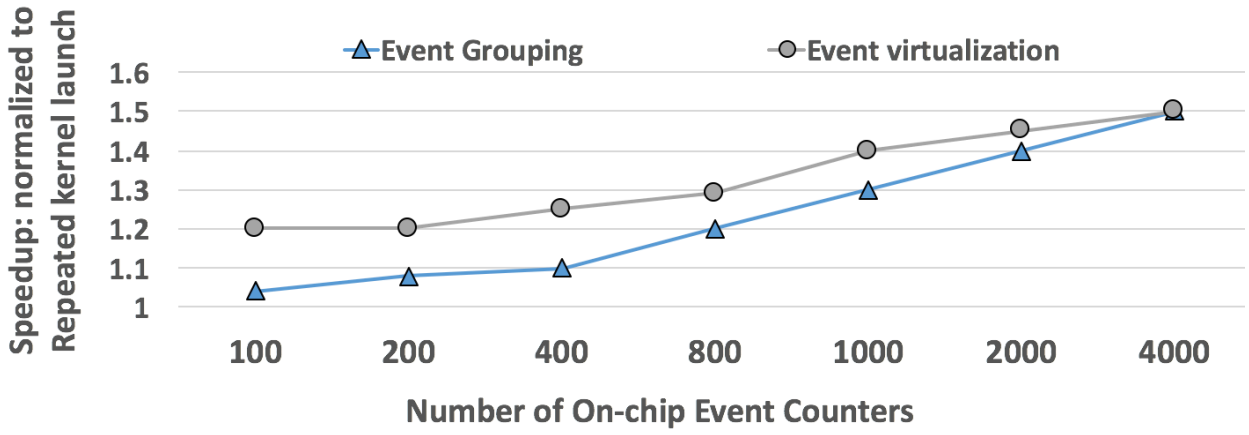

Figure 48: Measuring the effect of event grouping and counter virtualization using the LUD application.

7.0 SYNCHRONIZATION AWARE GPGPU WARP SCHEDULING

The objective of scheduling warps is to hide long latency operations (LLO) of a warp through switching to a different warp to keep the cores busy. The traditional RR policy achieves this through rotating among all warps assigned to a scheduler. Unfortunately, RR can often lead to stalling of all warps because threads have a similar programming structure and make roughly the same progress and reach a stalling stage at about the same time.

Among several existing scheduling mechanisms, CATLS aims to mitigate the problem of RR through creating more execution skew among different warps. The LLO considered was only memory accesses, i.e. L2 misses. This is achieved through a three-step procedure. First, all CTAs assigned to a scheduler are formed into groups, with one group being active at any time and all remaining groups put in a pending pool. Second, in every cycle, the scheduler executes ready warps from only the active group. All other groups are pending, even if they have ready warps. The warps in the active group are scheduled in a RR or custom defined order. Third, if the active group has no ready warps, the scheduler switches to another group, in a RR order. Through such scheduling, warps in the active group can advance more than those in pending groups such that they are in different execution stages and the latency hiding capability of the schedule is improved. Also, the intra-CTA (or inter-warp) locality is improved since warps belonging to one CTA are not divided into different groups, hence the term CTA-aware.

However, with multiple schedulers per SM, both CATLS and GTO suffer performance loss when synchronization barriers are present. To see this, we compare side-by-side the performance of single-scheduler versus two-scheduler designs for both CATLS and GTO. Results1 in Figure 2 show that on average, having two schedulers degrades performance noticeably for kernels with extensive synchronizations. For example, 18%(11%) and 13%(13%)

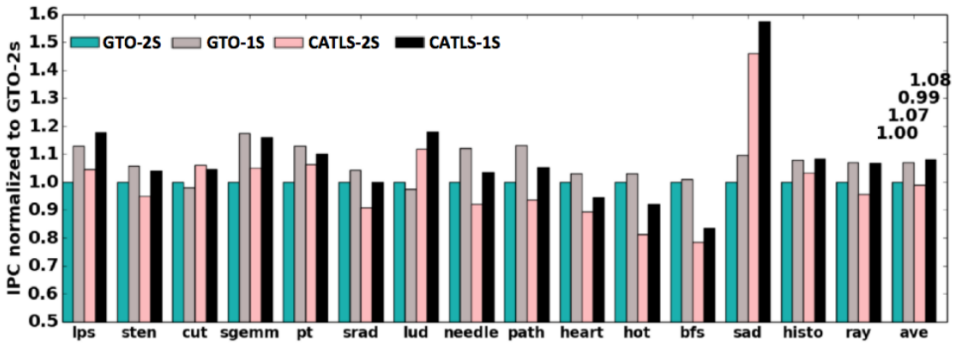

Figure 49: Performance of single vs. twoscheduler SMs for synchronization-rich kernels.

losses are seen in sgemm and lps for GTO (CATLS). To confirm that the loss in performance is due to synchronization we repeat the comparison for synchronization free kernels and show the results in Figure 49 and Figure 50 It is interesting to see that in the absence of synchronization GTO generally does better with two schedulers, by 2% on average. This is an indication that the harm caused by running synchronization barriers on two schedulers well surpasses their benefit, revealing a cumulative reduction of 9% (7+ 2%) from one-scheduler to two-scheduler design with GTO. The performance of CATLS running on one scheduler vs. two schedulers without synchronization are almost identical, and hence, the average slowdown shown of 9% from one- to two-scheduler are due to the presence of synchronizations.

7.1 KEY OBSERVATIONS OF SAWS

We advocate that multiple warp schedulers should coordinate with each other in the presence of synchronization barriers. In this section, we propose a Synchronization Aware Warp Scheduling, or SAWS, to form a coordinated warp scheduling and minimize synchronization induced stalls. The objectives of SAWS are:

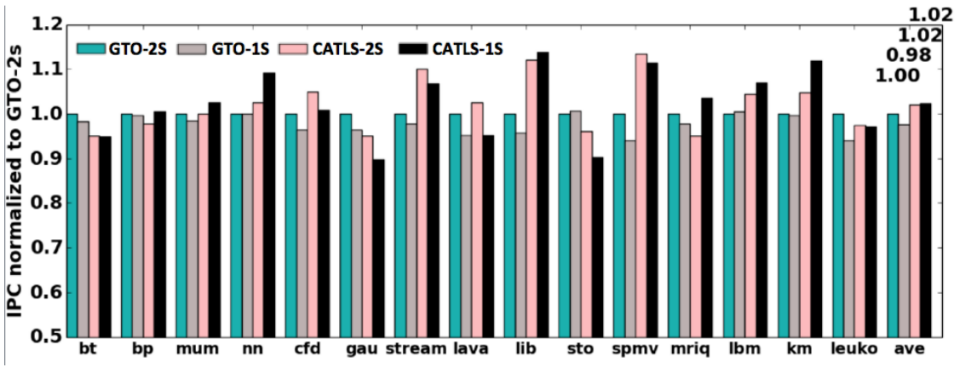

Figure 50: Performance of single vs. twoscheduler SMs for synchronization-free kernels.

- Shorten the time span between first hitting and clearing a barrier. These events can be in different schedulers and could be very far apart due to different scheduling decisions which results in large number of stall cycles.
- Achieve a uniform barrier-clearing rate among different CTAs when coordinating among different schedulers.
- Achieve the same performance as prior schedulers on synchronization-free kernels.

Our approach is to prioritize such CTAs so that their warps are executed sooner and arrive at the barrier sooner. Meantime, we also ensure that such prioritization will not cause the execution of any CTA to be heavily penalized, delaying the execution of the whole kernel. To develop SAWS, we will first examine the anatomy of synchronization events in multiple schedulers with scheduling policies such as GTO and CATLS. We will extract the commonality of those policies to understand why their scheduling could produce extra stalls due to synchronization.

7.1.1 The Anatomy of Synchronization Events

In an SM with single scheduler design, warps within a CTA will arrive at a barrier sequentially, as illustrated in Figure 51(a). We will use the terms 1st hit and clear to indicate the events of the first warp and last warp arriving at the same barrier. With multiple schedulers

(we use two schedulers for ease of illustration), warps are split and scheduled independently, resulting in different progress of warps on different schedulers. Warps that execute earlier are likely to hit a new barrier first, and warps having slower progress are likely to clear that barrier. Hence, in most cases, the first hit and clearance of a barrier occur on different schedulers, as illustrated in Figure 51(b). Due to this reason, the elapsed time between the two events could be very long due to the independence of different schedulers.

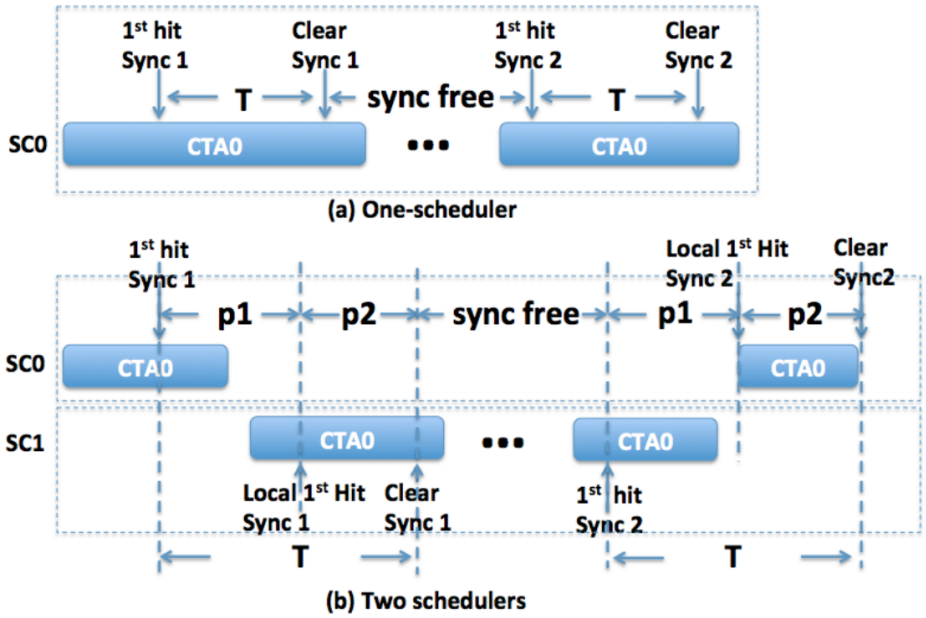

Figure 51: Timing of synchronization events

7.1.2 Reducing Intra-CTA Interference

Once C2 in SC1 encounters a local 1st hit, our next goal is to minimize the number of other CTAs that could interfere with the execution of C2 before its clear, i.e. to minimized the intra-CTA interference. The execution of C2 could be interrupted due to stalls, both short-term and long-term stalls. Short-term stalls are usually a couple of cycles, and long-term stalls could be hundreds of cycles. If all warps of C2 are stalled, then the scheduler identifies another high-priority ready warp through searching from head to tail in the list, to execute and cover the stalls produced by C2. However, the most critical scheduling step is that once

a C2 warp becomes ready again, the scheduler should resume its execution immediately. This is particularly challenging for resuming from short-term stalls which could be just one or two cycles. Hence, to ensure that the scheduler can timely pick up a ready warp from C2 to issue, we let the scheduler scan from the head of the list on every cycle to look for a ready warp. Since C2 has been promoted and is close to the head, it is highly likely that a warp is immediately identified once it becomes ready. The only exception, is when C2 yields to a ready warp with even higher priority, e.g. the list head. As we can see, even though C2 can be interfered by CTAs with either higher or lower priorities due to intrinsic stalls, the scheduler minimizes such interference by performing a prioritized scan on every cycle. C2 enters p2 when C0 is entirely stalled. During this time, if a warp arrives at the barrier and becomes blocked, it is moved to the tail of the list. Otherwise, a warp could be stalled.

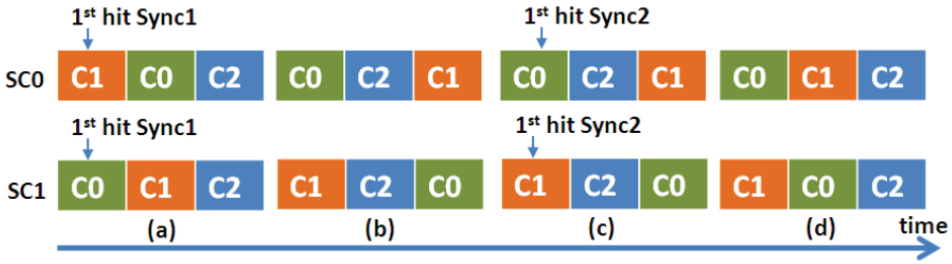

Figure 52: CTA promotion could cause nonuniform barrier-clearing rate.

7.1.3 Removing Barriers at the Same Rate

One potential problem with promoting CTAs to the front of the priority list is the nonuniform rate of removing barriers of each CTA. A CTA may incur barriers repeatedly and every occurrence introduces a promotion. When this happens frequently, other CTAs may suffer from excess delay. This is particular worsened if two CTAs keep promoting themselves alternatively. This may starve CTAs that have not yet reached a barrier and thus do not get an opportunity to be promoted. The example in Figure 52 illustrates such a problem. In this example, SC0 and SC1 are initially executing C1 and C0 respectively. The priority

order on SC0 and SC1 are C1, C0, C2 and C0, C1, C2 as shown in Figure 52 (a). When C1 in SC0 incurs a 1st hit, it promotes its sibling warps in SC1 and demotes itself to the tail of the list. The sibling warps of C1 in SC1 are not moved since they are already the second CTA in the list. Similarly when, C0 in SC1 incurs a 1st hit it promotes its sibling warps in SC0 and demotes itself to the tail of the list. At this time, the priority order on SC0 and SC1 become C0, C2, C1 and C1, C2, C0 respectively as shown in Figure 52 (b). Hence, the list heads of SC0 and SC1 may clear the first barrier in their own CTA and then progress towards the next barrier. When they have a 1st hits on this new barrier, they will promote each others sibling warps in the opposite scheduler, and those sibling warps will move from tail to the second CTA position again. The priority order on SC0 and SC1 become C0, C1, C2 and C1, C0, C2 respectively as shown in Figure 52 (d). Up to this point, C2 has never had a chance to become the head of list. If C0 and C1 are not stalled due to other reasons, C2 will never execute before C0 or C1 passes all its barriers. Such a starvation of C2 is indeed quite often as observed in our study.

7.2 SAWS ALGORITHM

The SAWS algorithm is formalized as follows. As stated earlier, the SAWS algorithm in each scheduler uses a priority list which includes all the warps assigned to the scheduler sorted in priority order, and in every cycle, the list is scanned from head to tail to find the first ready warp to issue. The order of the warps in the priority list changes only in response to two types of events. The first is the occurrence of a first hit on a barrier in some CTA. In this case, each scheduler updates its priority list according to the following algorithm. The algorithm uses a promotion register which is initially set to 1 (indicating that promoted warps are to be inserted after the first CTA) and a toggle bit which is initially set to true. Warp Promotion Algorithm :

7.3 INTEGRATING WITH PRIOR WARP SCHEDULER

One important virtue of the SAWS design is that its hardware data structure is very similar to GTO such that the two can be easily integrated. The philosophy is that the scheduler performs GTO when no synchronization barrier has occurred. Upon seeing the first barrier, the scheduler switches to SAWS, and stay in SAWS till the end. In this way, we have a GTO scheduler for synchronization-free applications, and a GTO-then-SAWS for synchronization-rich applications. The scheduler hardware is only slightly different from the original GTO design. We remark that it is also possible to integrate SAWS into CATLS, but it would require much more changes and higher hardware cost. In addition, the types of benchmarks GTO and CATLS are suitable for are quite disjoint. We observed that most synchronization-rich benchmarks are GTO-friendly. And CATLS-friendly benchmarks do not typically involve barriers. Hence, we will discuss the integration with GTO in this paper but still compare with the authentic CATLS in our evaluations.

The warp scheduler in GTO uses two arrays: a warp array and a timestamp array with one-to-one mapping between them. Each cell of the timestamp array stores the timestamp when the corresponding warp(CTA) is assigned to this SM. The timestamp of the warp defines the age of the warp. GTO issues one warp greedily until it is stalled and then it will issue the oldest ready warp. Therefore GTO requires a pointer to identify the current warp that is being greedily issued.

To implement SAWS, we use the timestamp arrays as priority arrays and use the greedy register as the promotion register. By default, the timestamp arrays store the time stamps at the beginning, where 0 means the oldest age. As soon as a warp hits the synchronization barrier for the first time, the global synchronization control unit (SCU) sends a signal to both schedulers to inform them to switch to the SAWS policy, where each cell in the timestamp array stands for the priority of the corresponding warp with 0 being the highest priority. The promotion register in each scheduler is set to 1, meaning a warp can be promoted the position with timestamp 1, which is the second CTA position in the array. After each promotion the register will be increased by 1. When a CTA that has a timestamp value smaller than the promotion register is blocked by a barrier, the promotion register is decreased by 1. Every

cycle, the scheduler will sweep the warp array in descending priorities, and issue the first ready warp it finds. Through updating timestamp arrays and the promotion register, SAWS algorithm can be correctly implemented.

7.4 WARP SCHEDULING EXPERIMENTAL RESULTS

The experiment setup is the same as global synchronization evaluation as presented in section 4.5.1 and Table 4.1. We implement SAWS, CATLS and GTO in GPGPU-Sim (version 3.2.2) [63]. The configuration is shown in Table 3. We evaluate 30 applications collected from Parboil [82], Rodinia [83] and CUDA SDK [63], 15 of which have CTA-level synchronizations. Further, we implemented both a dual-scheduler and a quad-scheduler design per SM to test the scalability of SAWS. In all sets of experiment, we assume that only one kernel can be launched to SMs at any time.

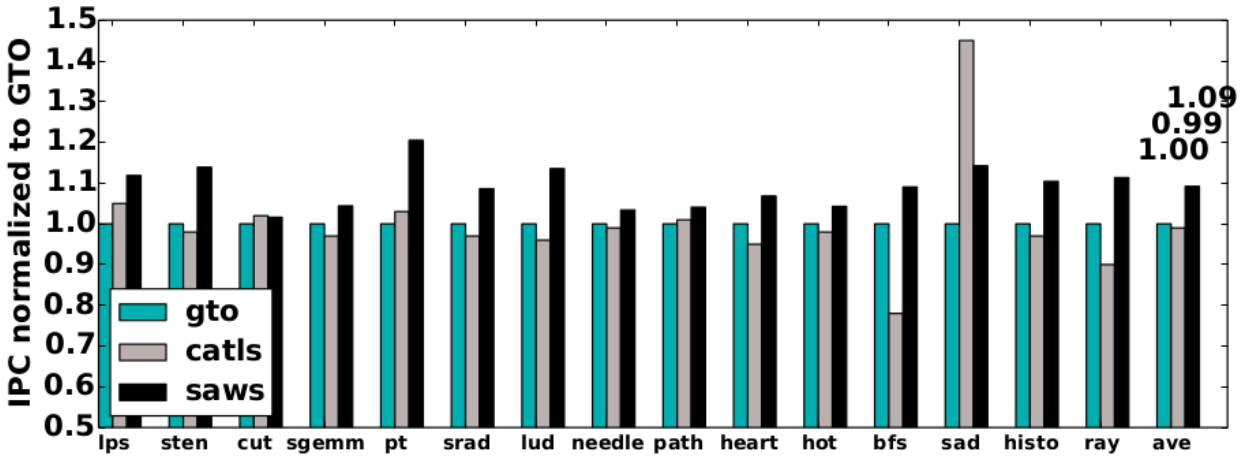

Figure 53: Performance comparisons for synchronization-rich benchmarks.

7.4.1 Performance Improvement, SAWS vs. GTO and CATLS

Through coordinating among different schedulers, SAWS reduces synchronization-induced stalls and improves performance. Figure 53 compares the performance among SAWS, GTO and CATLS for synchronization-rich benchmarks. The SAWS here is the GTO-integrated version, meaning that before seeing any barriers, the scheduler is GTO and once the first barrier is hit, the scheduler converts into pure SAWS. The results are relative to a pure GTO scheduler. As we can see, GTO and CATLS have similar average performances, because they do not coordinate among different schedulers on synchronizations. GTO sometimes performs significantly inferior to other schedulers, e.g. *sad* which is memory intensive, largely due to its weaker capability in hiding memory latencies. In other occasions, e.g. bfs and ray, GTO is noticeably better than CATLS because those benchmarks are cache sensitive and GTO can preserve strongly the intra-warp locality. On average, SAWS with coordination among schedulers outperforms both GTO and CATLS by 9% and 10% respectively. The only exception is *sad*, which is highly memory intensive. This benchmark has strong inter-warp and intra-CTA locality. It is most beneficial to have warps from the same CTA running concurrently to benefit from each other's memory accesses. CATLS, by grouping warps, can very well create such execution. We observed that the long latency L2 accesses in CATLS is 10× fewer than other two schedulers.

As discussed previously, such improvement is mainly due to two factors: (1) reduced barrier waiting time which consists of `p1`, the duration that presents inter-CTA interference, and `p2`, the duration that presents intra-CTA interference; and (2) ensuring uniform rate of clearing barriers among CTAs. Next, we show measurements to demonstrate each factor quantitatively.

7.4.2 SAWS Reduces Barrier Waiting Time

Figure 54 plots the breakdown of barrier waiting time for GTO, CATLS and SAWS, normalized to the total barrier waiting time of GTO. As we have discussed earlier, `p1` dominates over `p2`, accounting for 85%, 90%, and 92% of total barrier waiting time for GTO, CATLS and SAWS respectively. CATLS has slightly more barrier waiting time than GTO, 2% on

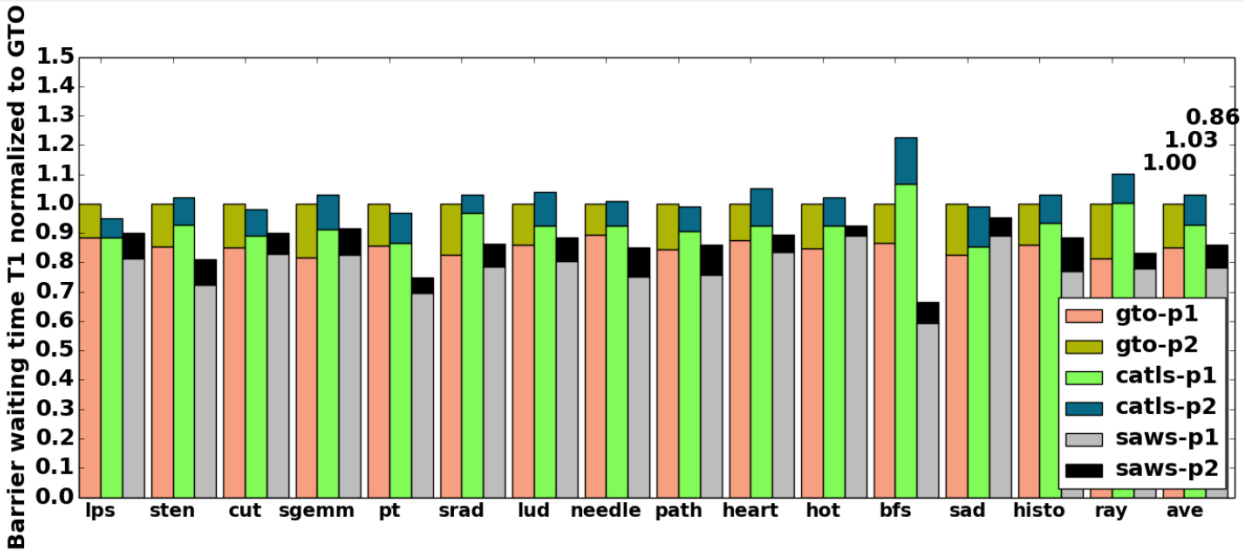

Figure 54: Breakdown of barrier waiting time.

average, mainly due to a larger `p1`. The longest barrier waiting times are seen in bfs and ray, which are consistent with their low performance shown in Figure 53. Clearing a barrier requires the scheduler to move quickly to the sibling warps but warps in CATLS are constrained in groups, and switching from group to group is rather slow. GTO also has a similar problem but is not constrained by group boundaries. SAWS effectively reduces the barrier waiting time by 15% and 17% for GTO and CATLS mainly due to prioritizing CTAs rather than following a fairly fixed scheduling order. Moreover, SAWS maintain a uniform rate of removing barriers among CTAs to further improve performance, as demonstrated next.

7.4.3 SAWS Ensures Uniform Barrier Clearing Rate among CTAs

A critical design in SAWS is to maintain a uniform rate of removing barriers among all CTAS, which is achieved through toggling CTA promotion on every other barrier of this CTA. Such toggling also improves performance effectively. Figure 55 compares performance of SAWS with and without promotion toggle. The results are normalized to GTO. As we can see, "with toggle" achieves 7% performance improvement over GTO. Having toggle will increase

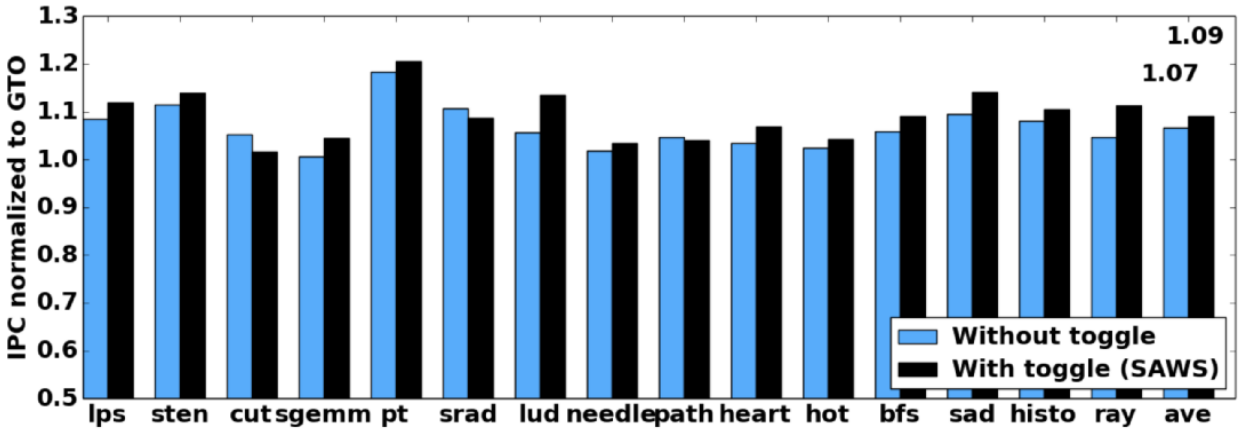

Figure 55: SAWS with and without promotion toggling.

the improvement to 9%, and larger improvements are seen in *lud*, *ray*, *sad* and *lps*. This is because an unfair scheduling such as "with toggle", though benefit some CTAs on clearing their barriers fast, will force other CTAs to linger for a long time and eventually become the bottleneck of the entire kernel. After all, the performance of a kernel is determined by the slowest CTA. Hence, being fair to all CTA means to reduce the latency of the slowest CTAs, and consequently improve the overall performance.

8.0 FUTURE RESEARCH DIRECTIONS

In this work, we proposed software and hardware designs for efficient GPU synchronization in the context of a single GPU. To further speedup computation and alleviate the memory capacity constraint of a single GPU, multi-GPU computing has lately emerged in the super-computing landscape. Specifically, multiple GPUs could be within a single network node or across network nodes. Therefore, we propose to extend the global synchronization and wait signal synchronization to single-node multi-gpu and multi-node multi-gpu contexts as future research directions.

In general, the existing communication patterns for multi-gpu is shown in table 4. For single-node multiple-gpu context, if the GPU devices are connected to the single CPU node via the PCIE bus, GPUs can be synchronized by the host CPU. If the GPU devices are connected via NVLINKs, GPUs can be synchronized by peer-to-peer memory access with memory flags [11, 1]. For multiple-node multiple-gpu, synchronization relies on the message passing of the host nodes. A critical design challenge of multi-GPU synchronization is to overlap communication with computing since not only synchronization signal needs but also a significant amount of data need to be shared among multiple GPUs. Correspondingly

Table 4: Communication for multiple GPUs. P2P: peer-to-peer GPU communication. [1]

	single process	multiple processes
single node	P2P or shared host memory	P2P or shared host memory
multiple nodes	Not apply	host-side message passing

the synchronization overhead is much higher than the single GPU synchronization. For simplicity, in the following discussion, we only consider global synchronization and the most common CPU-GPU setup where the CPU node connects to multiple GPUs through a PCIE bus.

8.1 SINGLE-NODE MULTI-GPU SYNCHRONIZATION

A conventional single-node multi-gpu synchronization pattern is to launch a kernel to each GPU in a separate stream so that the kernels in parallel on each GPU which processes a separate piece of data. Taking the stencil application as an example, a kernel is launched for each iteration at each GPU. At the end of an iteration, a global barrier is necessary for all the GPUs which is achieved by calling *cudaDeviceSynchronize* for each GPU sequentially [1, 11]. To integrate the Gsync scheme we proposed in this context, only one kernel is launched for all iterations on each GPU. Hence, the challenge is to allow multiple GPUs to communicate with each other in the middle of a kernel via the host CPU. Without any hardware modification in addition to the Gsync scheme, this can be achieved by synchronizing using global memory flags as shown in Figure 56. Basically, for a given GPU, if all its TBs execute the $_{g}ysnc()$ functions, a special memory flag is set in its global memory. Next, the GPU reads the special memory flags from all other GPUs and writes/copies it to its local memory flag array. This operation is carried out repeatedly in a busy-wait loop until all the memory flags are set. Reading the other GPU device's memory flags is enabled by universal virtual memory space and demand paging [11]. It should be noted that the memory flag goes through the CPU host memory in the PCIE setting. In case of stencil application, each GPU also needs to exchange the boundary data points which are also known as the halo region [1] with its neighbor GPUs. To further improve the performance, a GPU can also initiates prefetching the halo from its neighbor GPU if the corresponding special memory flag is set.

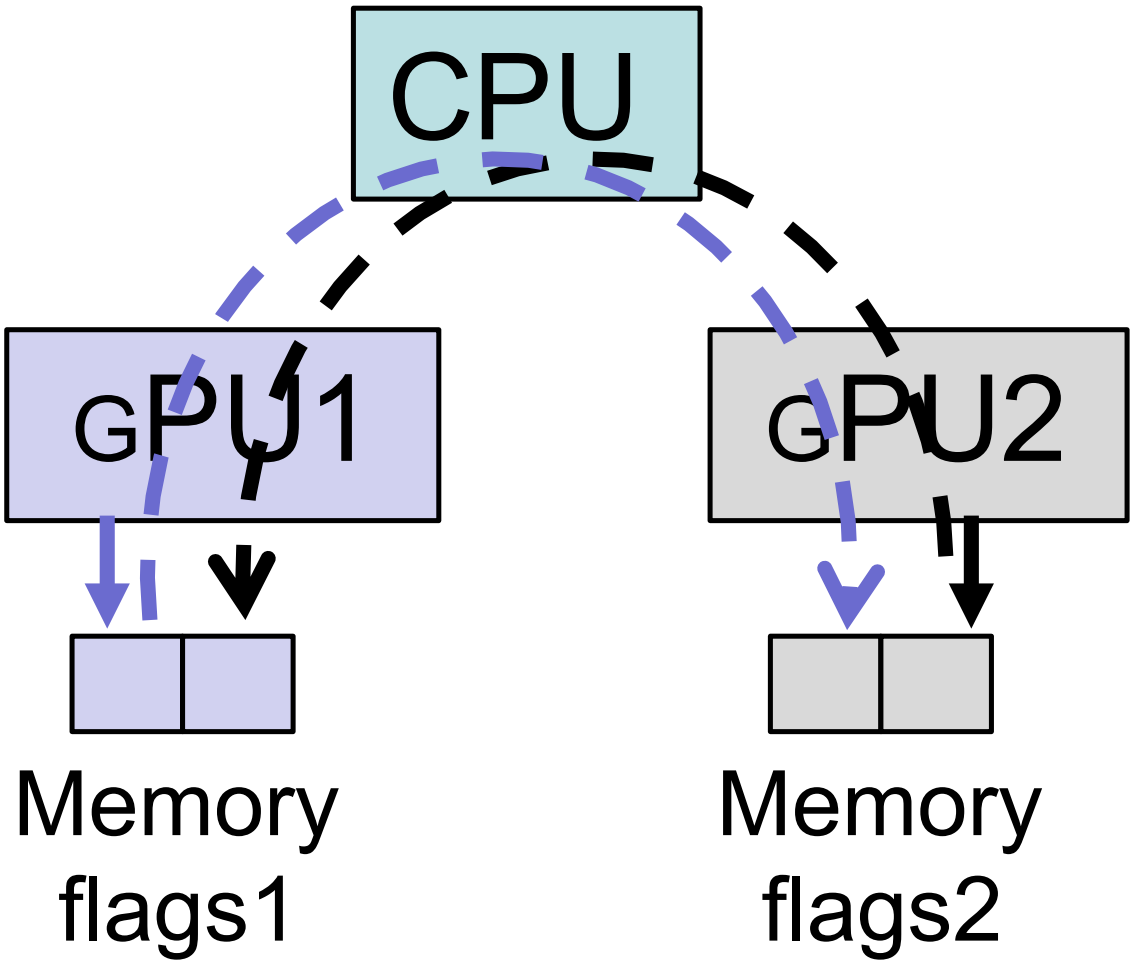

Figure 56: Single node Multiple GPU global synchronization via memory flags enabled by unified virtual memory and demand paging. Dashed arrow: read and write. Solid arrow: write.

8.2 MULTI-NODE MULTI-GPU SYNCHRONIZATION

The Message Passing Interface (MPI) is a standard API for communicating data via messages between distributed processes that is the natural choice for inter-node communication. CUDA-aware MPI is capable to pass pointers to host (system) memory to the MPI calls [85]. In this case the pointer points to the special memory flag on one GPU device. However, this is achieved by dedicated APIs such as MPI_Send and MPI_Recv which can only be invoked by the host CPUs. Remember to integrate the Gsync we proposed, we need a mechanism to synchronize multiple nodes and multiple GPUs in the middle of the kernel on each GPU. The existing APIs MPI_Send and MPI_Recv cannot be initiated by the GPU. Hence a new unified memory space across multiple host CPU nodes is required.

9.0 CONCLUSION

In this thesis, first I proposed a new global synchronization (Gsync) function at the GPU-side, which applies to both conventional SPMD and PT programming styles. We augment the existing SM and TB dispatcher to manage GS and partial context switches of the TBs. We also propose effective techniques for reducing the overhead of context switching to achieve scalability. The experimental results show that the proposed scheme outperforms existing software approaches in three typical type of applications with up to 2x speedup over CPU-driven synchronization.

Second, I proposed a software implementation of wait-signal synchronization mechanisms at the thread block level to accelerate multi-layer recurrent neural networks (RNN). This implementation is able to exploit fine-grained wavefront parallelism among layers, which was not possible in the state-of-the-art persistent RNN design. Moreover, by statically mapping thread blocks to each layer, the proposed scheme improves utilization of shared memory and reduces global memory accesses. Experiments using representative RNN applications demonstrate that the proposed scheme achieves up to 37% speedup and 18% bandwidth improvement over the state-of-the-art RNNs.

Third, I proposed wait signal with hardware support to enable efficient synchronization for producer-consumer problems. We found that the prevalent methodology, Repeated kernel launch, presents over-synchronization which unnecessarily serializes independent threads and restricts inherent parallelism. Alternative approaches such as PT using memory flags are general but incur inefficient memory spin and are prone to deadlocking. The proposed scheme uses wait-signal to remove over-synchronization and memory spin. Deadlocks are resolved leveraging context switching supported in today's GPU. Although context switching is expensive, the proposed scheme is able to reduce its overhead and still achieves signif-

icant speedup due to better parallelism and balanced on-chip load. Lastly, I proposed a synchronization-aware warp scheduling to mitigate performance loss due to synchronization events in SMs with multiple schedulers. Current designs treat each scheduler independently, which works well without synchronization barriers, but exposes deficiency with barriers. We find that multiple schedulers, when operating independently, will delay the clearance of barriers and create excessive stall cycles to the warps. Hence, we propose to coordinate among multiple schedulers and prioritizes warps as a reaction to synchronization events. In addition, we find that maintaining a uniform barrier-clearing rate during prioritization brings additional performance benefit. The proposed scheme SAWS is so simple that integrating it with existing scheduler does not require much additional hardware. Our evaluation shows that SAWS can speed up barrier clearance by 15% and total execution by 10% on average, when compared with the state-of-the-art warp scheduler. Finally, SAWS shows its increasing effectiveness when the number of scheduler grows, proving that it is a scalable design

BIBLIOGRAPHY

[1] Paulius Micikevicius. Multi-gpu programming.

[2] Nvidia. Gp100 pascal white paper. *NVIDIA Corporation, Santa Clara, CA*, 2017.

[3] Paul S Heckbert. Survey of texture mapping. *IEEE computer graphics and applications*, 6(11):56–67, 1986.

[4] Andrew S Glassner. *An introduction to ray tracing*. Elsevier, 1989.

[5] Chris Maughan and Matthias Wloka. Vertex shader introduction. *NVIDIA Technical Brief*, 2001.

[6] Richard Vuduc. *A Brief History and Introduction to GPGPU*. Springer, 2013.

[7] Graham Singer. The history of the modern graphics processor. *Techspot Blogs*, 2013.

[8] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[9] Frank Luna. *Introduction to 3D game programming with DirectX 10*. Jones & Bartlett Publishers, 2008.

[10] 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 (TOG)*, volume 22, pages 896–907. ACM, 2003.

[11] CUDA NVidia. Cuda c programming guide version 9.0. *NVIDIA Corporation, Santa Clara, CA*, 2017.

[12] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[13] AMD. Hsa platform system architecture specification version 1.0.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[15] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[16] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[17] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[18] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative study of caffe, neon, theano, and torch for deep learning. 2016.

[19] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[20] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[21] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[22] Fermi NVidia. Nvidia next generation cuda compute architecture. *NVidia, Santa Clara, Calif, USA*, 2009.

[23] AMD. Amd graphics cores next (gcn) architecture. *AMD Corporation, Sunnyvale, CA*, 2012.

[24] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[25] Shucai Xiao and Wu-chun Feng. Inter-block gpu communication via fast barrier synchronization. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[26] 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*, pages 25–35. ACM, 2015.

[27] Michael Bauer, Henry Cook, and Brucek Khailany. Cudadma: optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 12. ACM, 2011.

[28] Mark Harris. Cooperative groups: Flexible cuda thread programming.

[29] Sooraj Puthoor, Ashwin M Aji, Shuai Che, Mayank Daga, Wei Wu, Bradford M Beckmann, and Gregory Rodgers. Implementing directed acyclic graphs with the heterogeneous system architecture. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 53–62. ACM, 2016.

[30] Jaekyu Lee, Nagesh B Lakshminarayana, Hyesoon Kim, and Richard Vuduc. Many-thread aware prefetching mechanisms for gpgpu applications. In *Proceedings of the 2010 43rd Annual IEEE/ACM international symposium on Microarchitecture*, pages 213–224. IEEE Computer Society, 2010.

[31] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83. IEEE Computer Society, 2012.

[32] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *ACM SIGARCH Computer Architecture News*, 38(3):235–246, 2010.

[33] Greg 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 *Proceedings of The 33rd International Conference on Machine Learning*, pages 2024–2033, 2016.

[34] José María Cecilia, José Manuel García, and Manuel Ujaldón. Cuda 2d stencil computations for the jacobi method. In *Applied Parallel and Scientific Computing*, pages 173–183. Springer, 2010.

[35] Izzat El Hajj, Juan Gómez-Luna, Cheng Li, Li-Wen Chang, Dejan Milojicic, and Wenmei Hwu. Klap: Kernel launch aggregation and promotion for optimizing dynamic parallelism. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[36] 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*, volume 43, pages 528–540. ACM, 2015.

[37] Hancheng Wu and Michela Becchi Da Li. Compiler-assisted workload consolidation for efficient dynamic parallelism on gpu.

[38] Guoyang Chen and Xipeng Shen. Free launch: optimizing gpu dynamic kernel launches through thread reuse. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 407–419. ACM, 2015.

[39] Saoni Mukherjee, Yifan Sun, Paul Blinzer, Amir Kavyan Ziabari, and David Kaeli. A comprehensive performance analysis of hsa and opencl 2.0. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 183–193. IEEE, 2016.

[40] OpenCL blogs: Performance of atomics. http://simpleopencl.blogspot.com/2013/04/performance-of-atomics-atomics-in.html.

[41] AMD. Amd gcn3 isa architecture manual.

[42] Tyler Sorensen, Alastair F Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Portable inter-workgroup barrier synchronisation for gpus. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 39–58. ACM, 2016.

[43] Kunal Gupta, Jeff A Stuart, and John D Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14. IEEE, 2012.

[44] Zachary C Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[45] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[46] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[47] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[48] Viacheslav Khomenko, Oleg Shyshkov, Olga Radyvonenko, and Kostiantyn Bokhan. Accelerating recurrent neural network training using sequence bucketing and multi-gpu data parallelization. In *Data Stream Mining & Processing (DSMP), IEEE First International Conference on*, pages 100–103. IEEE, 2016.

[49] Jeremy Appleyard, Tomas Kocisky, and Phil Blunsom. Optimizing performance of recurrent neural networks on gpus. *arXiv preprint arXiv:1604.01946*, 2016.

[50] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. *arXiv preprint arXiv:1612.08242*, 2016.

[51] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[52] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[53] Joseph Redmon. Tiny darknet. https://pjreddie.com/darknet/tiny-darknet/.

[54] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 593–606. ACM, 2015.

[55] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous multikernel: Fine-grained sharing of gpgpus.

[56] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 193–204. IEEE Press, 2014.

[57] Myung Kuk Yoon, Keunsoo Kim, Sangpil Lee, Won Woo Ro, and Murali Annavaram. Virtual thread: Maximizing thread-level parallelism beyond gpu scheduling limit. In *ACM SIGARCH Computer Architecture News*, volume 44. IEEE Press, 2016.

[58] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. Mosaic: a gpu memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 136–150. ACM, 2017.

[59] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. Effisha: A software framework for enabling effficient preemptive scheduling of gpu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–16. ACM, 2017.

[60] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. Gpu register file virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 420–432. ACM, 2015.

[61] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. Improving gpu performance via large warps and two-level warp

scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317. ACM, 2011.

[62] Nagesh B Lakshminarayana and Hyesoon Kim. Effect of instruction fetch and memory scheduling on gpu performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU*, pages 1–10, 2010.

[63] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.

[64] Peng Di, Hui Wu, Jingling Xue, Feng Wang, and Canqun Yang. Parallelizing sor for gpgpus using alternate loop tiling. *Parallel Computing*, 38(6):310–328, 2012.

[65] Elias Konstantinidis and Yiannis Cotronis. Accelerating the red/black sor method using gpus with cuda. In *Parallel Processing and Applied Mathematics*, pages 589–598. Springer, 2011.

[66] Didem Unat, Xing Cai, and Scott B Baden. Mint: realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.

[67] John Cheng, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[68] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *High performance computing–HiPC 2007*, pages 197–208. Springer, 2007.

[69] NVIDIA. Cuda templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass.

[70] Nvidia. Cudnn stream implementation of rnn. https://github.com/parallel-forall/code-samples/tree/master/posts/rnn.

[71] Baidu. baidu-research persistent-rnn on github. https://github.com/baidu-research/persistent-rnn.

[72] Baidu. Reproduction issues of pt-rnn. https://github.com/baidu-research/persistent-rnn/issues.

[73] Motoki Sato. An pt-rnn example using cudnn. https://gist.github.com/aonotas/c431f5bc55d1e9c3b9b7201f1ffbb2ab.

[74] NVIDIA. Cutlass: Fast linear algebra in cuda c++. https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/.

[75] Hoda Naghibijouybari, Khaled N Khasawneh, and Nael Abu-Ghazaleh. Constructing and characterizing covert channels on gpgpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 354–366. ACM, 2017.

[76] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 119–130. ACM, 2015.

[77] AMD. Radeons next-generation vega architecture. *Advanced Micro Devices, Santa Clara, CA*, 2017.

[78] Abdulaziz Tabbakh, Murali Annavaram, and Xuehai Qian. Power efficient sharing-aware gpu data management. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 698–707. IEEE, 2017.

[79] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving gpgpu resource utilization through alternative thread block scheduling. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 260–271. IEEE, 2014.

[80] Cuda thread scheduling. https://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture5.pdf.

[81] Sreepathi Pai. How the fermi thread block scheduler works (illustrated).

[82] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[83] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[84] Nvidia. Mixed-precision programming with cuda 8.

[85] Jiri Kraus and Peter Messmer. Multi gpu programming with mpi.