A Study of Single and Multi-device Synchronization Methods in Nvidia GPUs

Lingqi Zhang*, Mohamed Wahib†§, Haoyu Zhang‡, Satoshi Matsuoka§*,
* Tokyo Institute of Technology, zhang.l.ai@m.titech.ac.jp
† National Institute of Advanced Industrial Science and Technology, mohamed.attia@aist.go.jp
‡ miHoYo Inc, (This work was done while the co-author worked in Tokyo Institute of Technology) lynkzhang@gmail.com
§ RIKEN Center for Computational Science, matsu@acm.org

***Abstract*—GPUs are playing an increasingly important role in general-purpose computing. Many algorithms require synchronizations at different levels of granularity in a single GPU. Additionally, the emergence of dense GPU nodes also calls for multi-GPU synchronization. Nvidia's latest CUDA provides a variety of synchronization methods. Until now, there is no full understanding of the characteristics of those synchronization methods. This work explores important undocumented features and provides an in-depth analysis of the performance considerations and pitfalls of the state-of-art synchronization methods for Nvidia GPUs. The provided analysis would be useful when making design choices for applications, libraries, and frameworks running on single and/or multi-GPU environments. We provide a case study of the commonly used reduction operator to illustrate how the knowledge gained in our analysis can be useful. We also describe our micro-benchmarks and measurement methods.**

***Index Terms*—CUDA Barrier, Synchronization, GPUs**

I. Introduction

GPUs have been playing an increasingly important role in general-purpose computing. Different scientific areas exploit the power of GPUs to accelerate science and engineering applications. Many complex algorithms require different levels of synchronizations, through the use of barriers. Until recently [1], developers used two methods of synchronization in CUDA. First, developers made use of CUDA thread block synchronization to develop complex algorithms [2]. Second, for applications like Deep Learning (DL), the CPU-side implicit barrier occurring after the kernel launch function is used for device-wide synchronization [3].

Due to the importance of device-wide synchronization, several researchers attempted to develop software device-wide barriers [4], [5]. Liu et al. [6] also proposed a hardware-software cooperative framework for synchronization. Yet the increase in complexity and density of GPUs in GPU-based systems, e.g. Nvidia DGX-2 includes 16 GPUs, call for a general and high-performance method for devices-wide and multi-GPU synchronization. Recently Nvidia proposed methods for synchronizations that spans all levels of granularity from a small group of threads in a GPU to a multi-GPU device: warp level, thread block level, and grid level. The grid level synchronization can be a productive way to perform device-wide and multi-device level synchronization. This hierarchy of synchronization methods can make GPU programming more productive. Thus, it is important to study the performance characteristics of different levels of synchronization methods.

In this paper, we characterize the synchronization methods in Nvidia GPUs. Specifically, in this work:

- We identify the performance characteristics of different synchronization methods in Nvidia GPUs.
- We use different implementations of the reduction operator as a motivating example to demonstrate how to use the knowledge gained in this study to optimize the reduction kernel.
- We explore the pitfalls of using several synchronization instructions.
- We provide our micro-benchmarks used in measurements [2].

II. Background

A. CUDA Programming Model

CUDA is a C-like programming model for Nvidia GPUs. It offers three levels of programming abstractions: thread, thread block, and grid. Among them, thread is the most basic programming abstraction. At the hardware side, there is a hierarchy that maps to the CUDA programming model. Three different levels of hardware resources exist: ALU, Stream Multi-Processor (SM), and the GPU. Take the Volta V100 [7] as an example, a V100 GPU consists of 80 SMs; an SM is partitioned into 4 processing blocks, each consists of several ALUs, e.g. 16 FP32 Cores.

A *warp* in CUDA is a small number of threads executed together as a working unit in a SIMT fashion. A warp in all Nvidia GPU generations consists of 32 threads. Inside an SM in V100 there are 4 warp schedulers corresponding to the 4 partitions inside one SM. CUDA's runtime will schedule one thread block to only one SM, and one grid to only one GPU, though it may occupy several SMs.

Figure 1 shows the details of CUDA programming model, its corresponding hardware abstraction, and the mapping relationship between them.

[1] Nvidia introduced a hierarchy of synchronization methods (based on Cooperative Groups(CG)) since CUDA 9.0 [1]

[2] The source code is available at:
https://github.com/neozhang307/SyncMicrobenchmark

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00057

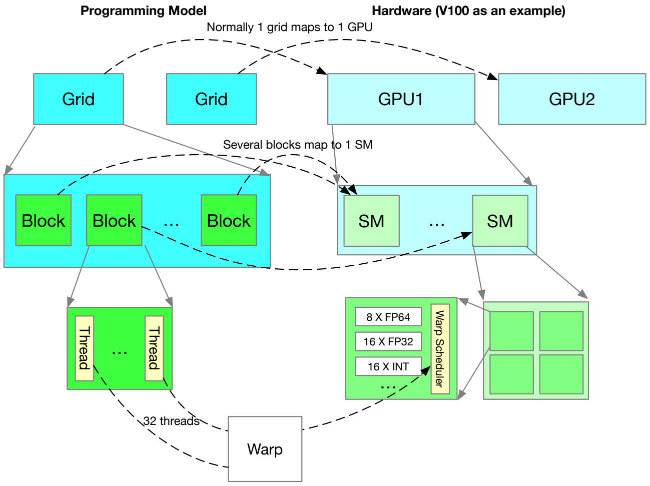


Fig. 1. CUDA programming model and corresponding hardware structure

B. Related Work

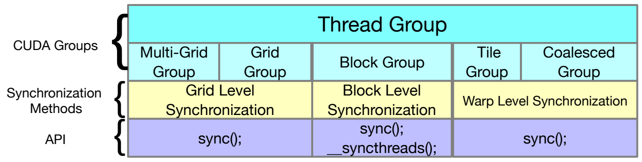
Many efforts have been done to micro-benchmark GPUs. Volkov et al. [8] benchmarks were partially used to study kernel launch overhead, manual device-wide barriers, data transfer, pipeline latency, instructions throughput, and metrics related to GPU memory system. This knowledge discovered was then used to tune several dense linear algebra algorithms. Wong et al. [9] proposed the use of more fine-grained micro-benchmarks to understand the performance of GPUs, including the behavior of instructions and memory structure of GPUs. Zhang et al. [10] introduced assembly-level micro-benchmarks. Recently, Jia et al. use ASM code to run micro-benchmarks on new Nvidia Tesla GPUs, i.e. V100 and P100 [11]. Several other works mainly focused on the memory hierarchy of GPUs, e.g. [12]–[14]. Among them, Mei et al. [14] discovered some cache patterns that were missed by previous researches. To the authors' knowledge, none of the GPU micro-benchmarking efforts focus on CUDA's hierarchy of synchronizations.

Volkov et al. [15] also compared kernel launch overhead and a manually implemented software barrier. Yet they only tested the overhead of light kernels, which is not practical for most of the applications. Other efforts analyzed software synchronization methods by comparing the performance of implementations of several algorithms with and without their software synchronization methods [4], [5], [16]. The analysis works on case-by-case bases and can not be generalized to different kernels.

III. Overview of Synchronization Methods in Nvidia GPUs

A. Primitive Synchronization Methods in Nvidia GPUs

Starting from CUDA 9.0, Nvidia added the feature of *Cooperative Groups (CG)*. This feature is planned to allow scalable cooperation among groups of threads and provide flexible parallel decomposition. Coalesced groups and tile groups can be used as a method to decompose thread blocks. Beyond the level of thread blocks, grid synchronization is proposed for inter-block synchronization. Multi-grid synchronization is proposed for inter-GPU synchronization.



CUDA Groups	Thread Group				
	Multi-Grid Group	Grid Group	Block Group	Tile Group	Coalesced Group
Synchronization Methods	Grid Level Synchronization		Block Level Synchronization	Warp Level Synchronization	

API	sync();		sync();
			__syncthreads();
				sync();	

Fig. 2. Hierarchy of synchronizations in CUDA

In the current version of CUDA (10.0), tile group and coalesced group only work correctly inside a warp. Analysis of PTX code shows that those two instructions are transformed to the *warp.sync* instruction. Hence, as it stands, we consider the synchronization capability of those methods to be only applicable to the warp level.

Figure 2 shows the granularity of cooperative groups and synchronization in the current version of CUDA.

1) Warp-level Synchronization (Synchronization Inside a Single GPU): Current CUDA supports two intra-warp synchronization methods, i.e. tile synchronization and the coalesced group synchronization corresponding respectively to the tile group and coalesced group in Figure 2. Previous versions of CUDA guarantee that all threads inside a warp process the same instruction at a time. Yet the introduction of synchronization methods inside a warp plus the fact that each thread now has its own Program Counter (PC) implies a future possibility of removing this feature.

2) Block-level Synchronization (Synchronization Inside a Single GPU): Block-level synchronization corresponds to the thread block in the programming model. According to CUDA's programming guide [1], its function is the same as the classical synchronization primitive __syncthreads().

3) Grid-level Synchronization (Single GPU Synchronization): Starting from CUDA 9.0, Nvidia introduced grid group grid-level synchronization. Grid-level synchronization is a method to do single GPU synchronization. In order to use a grid group, cudaLaunchCooperativeKernel() API call is necessary, in comparison to the traditional kernel launch (<<<>>>).

4) Multi-Grid Level Synchronization (Multi-GPU Synchronization): CUDA 9.0 also introduced the concept of multi-grid group. This group is initialized by a kernel launch API: cudaLaunchCooperativeKernelMultiDevice(). Synchronizing this group can act as a way to do multi-GPU synchronization in a single node.

B. Non-primitive Synchronization

1) Software Barrier for Synchronization: Li etc. [16] researched fine-grained synchronization. Beyond it, Xiao, etc. [5] introduced a software device-level synchronization. The authors limit the number of blocks per SM to only one in order to avoid deadlocks. Sorensen et al. extended this work by adding an automatic occupancy discovery protocol to discover activate warps [4].

2) Implicit Barrier for Synchronization: Before the introduction of grid-level synchronization, the typical way to introduce a device-wide barrier to a program was to use several kernels in a single CUDA stream. A stream is a logical queue that enforces an execution order on the CUDA kernels in the stream, i.e. the kernels and data movement commands are executed in the order by which they appeared in the stream. For example, many DL frameworks, e.g., Chainer [3], use this method to enforce execution order.

3) Multi-GPU Synchronization: The common way to do multi-GPU synchronization is to synchronize CPU threads orchestrating the GPUs. The basic idea is to use one CPU thread per device (or one MPI rank per device). Additionally, with the help of the *GPUDirect* CUDA technology, it is also possible to implement multi-GPU software barriers using GPUDirect APIs.

Since we are concerned in this paper with studying general and intrinsic barrier methods, we would not discuss manually implementation barriers, including software barriers and GPUDirect based manually implementations.

IV. Synchronization via CPU-side Implicit Barriers

Launching new kernels in a single stream can act as a device-wide implicit barrier to maintain the order of the program. Yet launching an additional kernel is not a free lunch: it will also introduce overheads. This section will inspect the overhead of traditional launch function, i.e., the <<<>>> kernel invocation method, and the new launch functions, i.e. cudaLaunchCooperativeKernel() and cudaLaunchCooperativeKernelMultiDevice() Nvidia introduced from CUDA 9.0 for CG.

To simplify our discussion, this section does not consider the extra overhead of launching the first kernel. Instead, in all our measurements we assume a warm-up kernel was already launched, and we focus our analysis on the behavior of kernels launched after the warm-up kernel.

Before further discussion in this section, we introduce the following terms:

- **Kernel Execution Latency:** Total time spent in executing the kernel, excluding any overhead for launching the kernel.
- **Launch Overhead:** Latency that is not related to kernel execution.
- **Kernel Total Latency:** Total latency to run kernels. $T_{Kernel\ Total\ Latency} = T_{Kernel\ Execution\ Latency} + T_{Launch\ Overhead}$

Figure 3 is our sample code for micro-benchmarks. It also shows the concept of kernel execution latency and kernel total latency. Kernel execution latency is controlled by the sleep instruction. $T_{kerne\ totall\ latency} = ((timer3 - timer2) - (timer2 - timer1))/(5-1)$ here; Elaborate details on the bench-marking methods are discussed in Section IX-B.

```
__global__ void null_kernel(){
    //kernel execution latency is 10 us here.
    repeat10(asm volatile("nanosleep.u32 1000;");)
}
...
record(timer1);
repeat1(launch(null_kernel, launchparameters););
cudaDeviceSynchronize();
record(timer2);
repeat5(launch(null_kernel, launchparameters););
cudaDeviceSynchronize();
record(timer3);
...
```

Fig. 3. Sample code to micro-benchmark implicit barriers for a null (empty) kernel

TABLE I
Launch Overhead and Null kernel Latency of Different Launch Functions

Launch Type	Launch Overhead (ns)	Null Kernel Kernel Total Latency (ns)
Traditional	1081	8888
Cooperative	1063	10248
Cooperative Multi-Device	1258	10874

In this way, We measured the launch overhead by using the kernel fusion method. We also test the kernel total latency of a null kernel for comparison. Table I shows the result.

V. Single GPU Synchronization

In this section, we characterize the performance of warp, thread block, and grid level synchronization. Warp and block abstractions exist inside an SM. For warp and block, we used the micro-benchmark discussed in Section IX-C. Grid is an inter-SM abstraction, for that, we used the micro-benchmark discussed in Section IX-D.

For the warp shuffle operation and block synchronization operation, the throughput is reported by CUDA programming guide [1] at the granularity of warps and blocks, respectively. Yet it is possible that the size of a group that performs synchronization or shuffle would influence the performance itself. Hence in this work, we consider the group size when experimenting with warp shuffle and block synchronization.

A. Warp-Level Synchronization

The current CUDA (10.0) supports two kinds of warp level synchronization: tile group based and coalesced group based (as seen in Figure 2). Additionally, the CUDA shuffle operation, which exchanges a register value among threads in a warp, is an operation that implies a synchronization after it. We also include the results of the shuffle operation.

Since the size of a synchronization group might influence the result, we tested every possible group size for both tile group and coalesced group. The possible tile group sizes are: 1, 2, 4, 8, 16, and 32. The possible coalesced group size ranges from 1 to 32. Latency is tested by using only 32 threads (a warp) in a CUDA kernel with one block. The throughput is tested by iterating every possibility pair of up to 1024 threads

TABLE II
PERFORMANCE OF WARP SYNCHRONIZATION IN A BLOCK

Type (group size)	**Latency** **cycle**		**Throughput** **(sync/cycle)**		**Reference** [1] thread op/cycle	
	V100	**P100**	**V100**	**P100**	**V100**	**P100**
Tile(*)	14	1	0.812	1.774	-	-
Shuffle(Tile)(*)	22	31	0.928	0.642	32	32
Coalesced(1-31)	108	1	0.167	1.791	-	-
Coalesced(32)	14	1	1.306	1.821	-	-
Shuffle(COA)(*)	77	50	0.121	0.166	-	-
Block(warp))	22	218	0.475	0.091	16	32

and up to 64 blocks per SM and recording only the highest result. Table II shows the result of warp level synchronization.

For tile group synchronization the size of the group influence neither latency nor throughput. A possible explanation is that CUDA could be merging all the concurrent tile group synchronization instructions into a single instruction. For coalesced group synchronization, the group size does not influence the performance of P100. The group size does, however, influence the performance of coalesced group in V100. The performance is the highest when all the threads inside a warp belong to a single coalesced group. For convenience, because the group size doesn't influence the total latency of tile group synchronization, we only record the throughput in the case of a group size of 32 in tile group synchronization.

We use the reference throughput of shuffle operation mentioned in the CUDA programming guide [1] in Table II. Apparently, the performance of V100 is closer to the theoretical result in the programming guide. On the other hand, there seem to be some overheads that influence the throughput of the shuffle operation in P100.

B. Block-Level Synchronization

We tested every possible group size at the block level, i.e. starting from 32 to 1024. We find that the throughput of block-level synchronization is related to the number of active warps per SM.

Figure 4 shows the relationship between the throughput of block synchronization divided by warp count (warp sync per cycle) and the maximum number of activate warps per SM (as calculated by [7]). When the warp count exceeds the size of max activate warp per SM, the device is saturated and the throughput of block synchronization reaches its maximum.

With this observation, we conclude that the performance of block-level synchronization is related to the warp count per SM. We further summarize the performance of block synchronization from a warp's perspective in Table II.

CUDA's programming guide [1] reports that the throughput for __syncthreads() (or block-level synchronization) is 16 operations per clock cycle for capability 7.x (V100) and 32 for capability 6.0 (P100). The throughput of V100 is relatively close to 16 op/cycle. But the result of P100 is far away from 32 op/cycle. To further support our result, the inverse of the gradient of the points in the up part of Figure 4 can represent throughput. Obviously, the gradient of block synchronization in P100 is larger than V100. So, the throughput of P100 should not be larger than V100's.

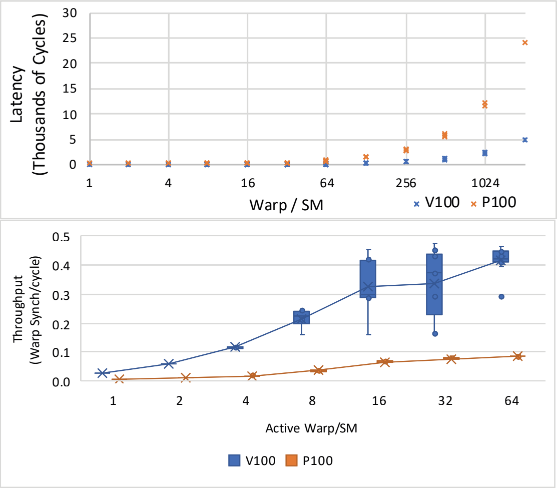
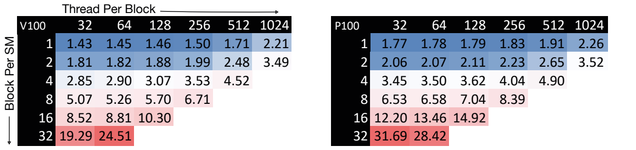


Fig. 4. Relationship between throughput of block sync (per warp perspective) (up) and active warp/SM perspective (down)



Thread Per Block
Block Per SM

V100	32	64	128	256	512	1024
1	1.43	1.45	1.46	1.50	1.71	2.21
2	1.81	1.82	1.88	1.99	2.48	3.49
4	2.85	2.90	3.07	3.53	4.52	
8	5.07	5.26	5.70	6.71		
16	8.52	8.81	10.30			
32	19.29	24.51				

P100	32	64	128	256	512	1024
1	1.77	1.78	1.79	1.83	1.91	2.26
2	2.06	2.07	2.11	2.23	2.65	3.52
4	3.45	3.50	3.62	4.04	4.90	
8	6.53	6.58	7.04	8.39		
16	12.20	13.46	14.92			
32	31.69	28.42				

Fig. 5. Latency (us) of grid synchronization in V100 (left) and P100 (right)

Admittedly, it is also possible that the performance of block synchronization in P100 is not ideal due to over-subscription. Yet the latency of block synchronization in P100 is so large that it is nearly impossible to find a point at which the instruction pipeline is saturated while the overhead of over-subscription is not so severe.

C. Grid-Level Synchronization

Figure 5 shows the heat map of grid synchronization. It shows that in both V100 and P100 the latency of grid synchronization is more related to the grid dimension (specifically, block count per SM) than to the block dimension.

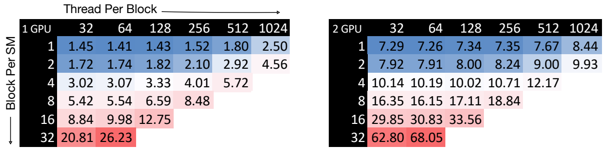
No matter how small the grid is, it seems that it is still slower than the overhead of kernel launch we measured in Section IV. Single GPU grid synchronization might not bring about any benefit in performance, in comparison to implicit barrier methods. Yet we argue that this performance difference is negligible (at most $2.5us$ with two blocks/SM) in real applications. In addition, using the implicit barrier instead would eliminate the possibility of data reuse in shared memory and registers.

VI. MULTI-GPU SYNCHRONIZATION METHODS

We consider three ways to do multi-GPU synchronization:

```
#pragma omp parallel num)threads(GPU_count){
  unit gid=omp_get_thread_num();
  cudaSetDevice(gid);
  ...
  kernel<<<>>>();
  cudaDeviceSynchronize();
  #pragma omp barrier
  ...
}
```

Fig. 6. Code example of using CPU threads for synchronization



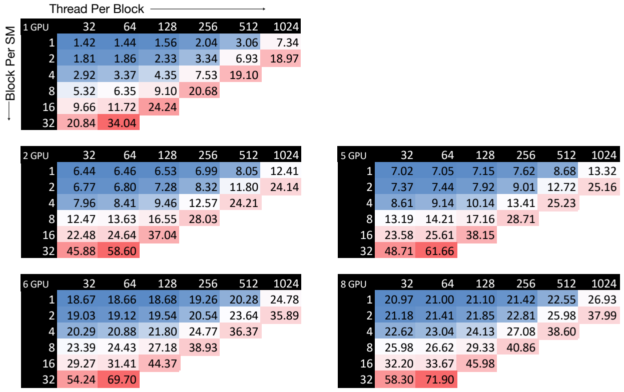
Thread Per Block →

Block Per SM ↓

1 GPU	32	64	128	256	512	1024
1	1.45	1.41	1.43	1.52	1.80	2.50
2	1.72	1.74	1.82	2.10	2.92	4.56
4	3.02	3.07	3.33	4.01	5.72	
8	5.42	5.54	6.59	8.48		
16	8.84	9.98	12.75			
32	20.81	26.23				

2 GPU	32	64	128	256	512	1024
1	7.29	7.26	7.34	7.35	7.67	8.44
2	7.92	7.91	8.00	8.24	9.00	9.93
4	10.14	10.19	10.02	10.71	12.17	
8	16.35	16.15	17.11	18.84		
16	29.85	30.83	33.56			
32	62.80	68.05				

Fig. 7. Latency (us) of multi-grid synchronization in P100 platform for one GPU (left) and two GPUs (right)



Thread Per Block →

Block Per SM ↓

1 GPU	32	64	128	256	512	1024
1	1.42	1.44	1.56	2.04	3.06	7.34
2	1.81	1.86	2.33	3.34	6.93	18.97
4	2.92	3.37	4.35	7.53	19.10	
8	5.32	6.35	9.10	20.68		
16	9.66	11.72	24.24			
32	20.84	34.04				

2 GPU	32	64	128	256	512	1024
1	6.44	6.46	6.53	6.99	8.05	12.41
2	6.77	6.80	7.28	8.32	11.80	24.14
4	7.96	8.41	9.46	12.57	24.21	
8	12.47	13.63	16.55	28.03		
16	22.48	24.64	37.04			
32	45.88	58.60				

5 GPU	32	64	128	256	512	1024
1	7.02	7.05	7.15	7.62	8.68	13.32
2	7.37	7.44	7.92	9.01	12.72	25.16
4	8.61	9.14	10.14	13.41	25.23	
8	13.19	14.21	17.16	28.71		
16	23.58	25.61	38.15			
32	48.71	61.66				

6 GPU	32	64	128	256	512	1024
1	18.67	18.66	18.68	19.26	20.28	24.78
2	19.03	19.12	19.54	20.54	23.64	35.89
4	20.29	20.88	21.80	24.77	36.37	
8	23.39	24.43	27.18	38.93		
16	29.27	31.41	44.37			
32	54.24	69.70				

8 GPU	32	64	128	256	512	1024
1	20.97	21.00	21.10	21.42	22.55	26.93
2	21.18	21.41	21.85	22.81	25.98	37.99
4	22.62	23.04	24.13	27.08	38.60	
8	25.98	26.62	29.33	40.86		
16	32.20	33.67	45.98			
32	58.30	71.90				

Fig. 8. Latency (us) of multi-grid synchronization in V100 platform

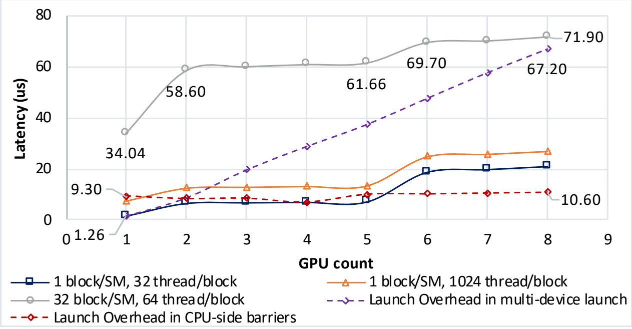


Fig. 9. Comparison of implicit barriers performance: multi-device launch vs. CPU-side barriers and multi-grid synchronization across 8 GPUs in DGX-1

A. Using Multi-device Launch Function as an Implicit Barrier

When using the multi-device launch function with the default flag, kernels will not execute until all the previous operations in all the GPU streams involved have finished execution [17]. Although this implicit barrier method is not commonly used, we nonetheless evaluate it to assess if this method is a valuable alternative. Section IX-B discusses in detail micro-benchmark we use in this subsection.

B. Using CPU-side Barriers

A common way to make a barrier between GPUs is to use CPU threads or processes to synchronize different GPUs. We use openMP to measure the overhead in this case. Each thread calls the cudaDeviceSynchronize() API to ensure the asynchronously launched GPU kernels are executed till their end. In addition, the threads use the openMP barrier API to synchronize. Figure 6 shows the code example for this kind of barrier. Finally, we appropriately pin the CPU threads. We applied the same micro-benchmark discussed in Section IX-B for this subsection.

C. Using Multi-grid Synchronization

Section IX-D discusses in detail micro-benchmark we use in this subsection. Figure 7 and Figure 8 show the heat maps of the latency of multi-grid synchronization in V100 and P100. Because the inter-connection in the P100 system is PCIe, the performance is worse than the V100 system that is equipped with NVLink connection between devices.

We experimented with all 8 GPUs in the DGX-1, we found that the performance of multi-grid synchronization among 2-5 GPUs is similar to each other, and the performance of multi-grid synchronization among 6-8 GPUs are similar to each other. This behaviour is likely related to the internal NVLink network structure of DGX-1. From Figures 7 and 8, we can see that the performance of multi-grid synchronization is influenced by both the grid dimension and number of active warps per SM. With $block/SM <= 8$ and $warp/SM <= 32$, the performance is acceptable. Apart from the case of one GPU, latency in all cases is no more than 2x slower than the fastest case (1 block/SM, 32 threads/block) and 2x faster than the slowest case (32 blocks/SM, 64 threads/block).

D. Comparison

Figure 9 shows the results of all three multi-GPU synchronization methods across 8 GPUs in DGX-1. For simplification, we only plot the data of three cases of multi-grid synchronization in Figure 9: a) one block/SM, 32 threads/block as the fastest case, b) 32 blocks/SM, 64 threads/block as the slowest case, and c) one block/SM, 1024 threads/block as a general case, which is within the parameters we recommended in the previous paragraph.

The CPU-side barrier relying on openMP barriers outperform implicit barriers in multi-device launch when the GPU count is larger than two. Also, the overhead of the CPU-side barrier is relatively steady w.r.t. GPU count. It is worth mention that this result is relatively close to the kernel total latency of a null kernel as shown in Table I.

Figure 9 shows two performance drops in multi-grid synchronization. We anticipated that the second drop would be between 4 GPUs and 5 GPUs, based on the internal network structure of DGX-1 that groups 4 GPU together. However, we find no reasons for the performance drop between 5 GPU and 6 GPU.

The figure shows that multi-grid synchronization outperforms the multi-device kernel launch function as an implicit barrier. On the other hand, as long as the program is not oversubscribed, i.e., no more than 1024 threads per SM, the performance of multi-grid synchronization is at most 3x slower than CPU-side barriers. Yet the difference is around $16us$, which is practically not an issue in the situation of 8 GPUs.We argue that this minor cost should not discourage programmers from considering the use of multi-grid synchronization in their algorithms, given the utility provided in terms of simplicity of programming, and avoiding reliance on third-party libraries such as openMP or MPI.

VII. Case Study: Reduction Operator

We use the reduction operator (summing the elements of an array) as a case study. Harris et al. [2] is a notable work that focused on optimizing the reduction operator in CUDA. They studied several optimization methods and optimized the operator by optimizing for maximum memory bandwidth utilization. Additionally, Luitjens et al. [18] introduced the use of the shuffle primitive in reduction. The optimized reduction kernels can be found in CUDA SDK samples [19]. There are other similar optimization strategies [20], [21]. To the best of the authors' knowledge, all of the previous strategies didn't quantitatively compare different synchronization methods in different implementations. In this section, we will demonstrate how to capitalize on the analysis in previous sections to make a decision between different reduction implementations, depending on the input size and number of workers involved. This approach can be applied to optimize any of the previous reduction implementations and many other code generation frameworks [22].

In addition to using single GPU synchronization methods in optimizing for input size, there is a programmability benefit in using multi-grid synchronization for multi-GPU systems. In dense system, such as Nvidia DGX-1 and DGX-2, the peer access feature enables one GPU to access the memory of another GPU. In this case, multi-grid synchronization provides an easy way to ensure sequential consistency. We explain this in detail in section VII-E.

It is important to mention another potential benefit that does not appear in the case of the reduction kernel. There is a potential of improving data reuse by the means of replacing several kernel invocations with a single persistent kernel that uses multi-grid synchronization. An example of that would be replacing kernel invocations in iterative stencil methods with a persistent kernel that includes the time loop inside the kernel.

A. Performance Model

TABLE III
Projected concurrency of the two configurations in Section VII-B

scenery		bandwidth B/cycle		latency cycle		concurrency B	
		V100	P100	V100	P100	V100	P100
1	1 thrd.	0.62	0.43	13.0	18.5	8	8
	1 warp	19.6	13.8	13.0	18.5	256	256
2	32 thrd.	19.6	13.8	13.0	18.5	256	256
	1024 thrd	215	141	13.0	18.5	2796	2615

```
while(i<n){sum+=g_idata[i];i+=groupsize;}
```

Fig. 10. Code example of the main instruction in the memory bandwidth micro-benchmark for proxying the reduction operation

We assume that the throughput is indifferent to the size of the problem (for any problem size that fully utilizes the device). We also assume that the cost of synchronization is the main cost of multi-threading. We can use Equation 2 to know when to use fewer threads. In this equation, "basic" might refer to single thread, single warp, single block, or single GPU, and "more" corresponds to more threads, more warps, more blocks, or multi-GPU. We use Little's Law [23] to compute concurrency (Equation 1). To simplify the problem, we consider T_{basic} as the latency in Little's Law, and T_{more} includes the overhead of synchronization as Equation 3 shows. From this equation we can imagine three different scenarios:

1) If the input size is not larger than the concurrency of "basic" threads, using fewer threads would always be more profitable.
2) If the input size is larger than the concurrency of "basic" threads and no larger than the concurrency of "more", we can use Equation 4 to compute the switching point.
3) If the input size is larger than the concurrency of "more" threads. We can use Equation 5 to know at which point we should use fewer threads.

$$C = T * Thr \tag{1}$$

$$T_{basic} + \frac{Max(0, N-C_{basic})}{Thr_{basic}} < T_{more} + \frac{Max(0, N-C_{more})}{Thr_{more}} \tag{2}$$

$$T_{more} = T_{basic} + T_{sync} = T + T_{sync} \tag{3}$$

$$N_m < (T + T_{sync}) * Thr_{basic} \tag{4}$$

$$N_l < \frac{(T_{sync})*Thr_{more}*Thr_{basic}}{Thr_{more} - Thr_{basic}} \tag{5}$$

$*(T\ represent\ Latency; Thr\ represent\ Throughput;$
$C\ represent\ concurrency)$

B. Micro-benchmark and Basic Prediction

In the case of the GPUs we examine in this paper when the input size is large enough, the bottleneck of reduction algorithm is device memory bandwidth. Hence we use a memory bandwidth micro-benchmark to proxy the performance of reduction. To make this micro-benchmark an accurate representation, we add two add instructions to imitate the real computation in the reduction operation. Figure 10 shows the main instruction in the micro-benchmark.

Our objective is to identify when to use a single thread, a single warp barrier, and until when would it be more efficient

TABLE IV
PREDICTING THE SWITCHING POINT BETWEEN TWO CONFIGURATIONS

scenery		sync ltc* cycle		switch point B	
		V100	P100	V100	P100
1	1 warp N_l	110	155	70	70
	1 warp N_m	-	-	76	75
2	1024 thrd N_l	420	2135	9076	32681
	1024 thrd N_m	-	-	8501	29737

*: 5 times synchronization

```
//assume the data resides in shared memory
for(step = 16; step >=1; step/=2){
  //or use the shuffle operation here
  if(tid+step <32)sm[tid]+=sm[tid+step];
  synchronize();
}
```

Fig. 11. Code example of warp level reduction with synchronization

to use a multi-GPU barrier. Instead of enumerating every possible case, we only consider two configurations here (and it can be extended to other cases):

- To use a single thread or single warp barrier
- To use a single block with 1024 threads or with 32 threads

Normally in the two configurations we mentioned, the data is usually kept in shared memory or cache, so we only measure shared memory for the following part. Table III shows the results of bandwidth (throughput), latency and concurrency.

Take the double type as an example (8 Bytes). In this case, in both configurations, the input size exceeds the concurrency of both ”basic” and ”more” settings, hence we only need to take N_l in Equation 5 into consideration. Table IV shows the results.

Table IV shows that: first, it is better to compute 32 data points with a warp; second, there would be no benefit to compute 1024 data points with 1024 threads per block. Our further experiments show that those predictions are correct.

In addition, another potential overhead caused by synchronization would be that the synchronization would possibly clear the instruction pipeline. Threads might need additional time to saturate the pipeline. So the real switching point would likely be larger than this.

C. Warp Level Reduction

In this subsection, we compare different warp level synchronization methods in the reduction kernel by observing their behaviour in the current generations of GPUs. Figure 11 shows our sample code, and Table V shows the result.

As shown in Table V, when using the *volatile* qualifier for the input data, the performance of warp level synchronization is no worse than in the case without the volatile qualifier (shown as ”tile” in the table). Accordingly, the warp level synchronization does not have much overhead other than to ensure memory consistency. We can conclude that warp level synchronization is no more than a memory fence in the current version of CUDA. We also observe that the results for using the shuffle operation with the tile group have the lowest latency.

TABLE V
LATENCY (CYCLES) TO COMPUTE SUM OF 32 VALUES (DOUBLE PRECISION)

	serial	nosync *	volatile & tile	tile	coa	tile shuffle	coa shuffle
V100	299	89	237	237	237	164	1261
P100	383	112	282	281	251	212	1423

*result of no synchronization version is incorrect

```
__device__ REAL summing(...){...
  uint i = threadid + blockid * blockdim;
  sum=0;
  while(i<n){
    sum+=g_idata[i];
    i+=blockdim*griddim;
  }
  return sum;
}
__device__ REAL block_reduce(...){...
  i = threadid;
  sum=0;
  while(i<n){sum+=td[i]; i+=blockdim;}
  //n is the pre-computed switch point
  td[threadid]=sum;
  sum=0;
  block.sync();
  if(warpid==0)
  {
    i = threadid;
    while(i<blockDim){sum+=td[i];i+=32;}
    sum = shuffle_reduce_warp(sum);
  }
  return sum;
}
```

Fig. 12. Basic function of device wide reduction

D. Single GPU Reduction

In this Subsection, we directly apply the knowledge in Section VII-B in implementing device-wide reduction. Figure 13 shows the code of reduction with explicit synchronization and Figure 14 shows the code of reduction with implicit synchronization for a single GPU.

The widely used GPU C++ library CUB [24] and CUDA SDK samples [19] include single GPU reduction implementations, we compare the performance of those implementations with our implementation.

Figure 15 and Table VI show the results. Our implementation is comparable to state of art implementations on V100 and is noticeably better on P100. We can learn from Figure 15 that using a CPU-side barrier (”implicit” in the figure) always outperforms using grid synchronization (”grid sync” in the figure), though the performance difference is not so decisive.

E. Multi-GPU Reduction

In this section, we use the code in Figure 13 and implicit-MultiGPU code in Figure 14. Figure 16 shows the results. Though it is hard to notice, an implicit barrier is always slightly better than the multi-grid synchronization method.

```
//works in both single and multi GPU
__global__ void ExplicitGPU(...){...
  while(step.notfinish()){
    //directly store data in the target GPU
    dest[step][threadid]
        = summing(src[step][threadid], ...);
    grid.sync();//explicit synchronize;
  }
  if(gpu_id==0)
  {
    sum=block_reduce(src[0][0], ...);
    if(threadid==0)
      output[threadid]=sum;
  }
}
```

Fig. 13. Code example of reduction with explicit device synchronization

```

__global__ void Kernel1(...){...
  uint i = threadid + blockid * blockdim;
  sum=summing(...);
  output[i]=sum;
...}
__global__ void Kernel2(...){...
  sum=block_reduce(...);
  if(threadid==0)
    output[threadid]=sum;
...}

//following parts are CPU functions
void implicitSingleGPU(...){...
  Kernel1<<<...>>>(...);//implicit synchronization
  Kernel2<<<...>>>(...);
  ...}

void implicitMultiGPU(){...
#pragma omp for num_threads(gpucount){...
  cudaDeviceSet(tid);
  Kernel1<<<...>>>(...);
  //gather data to one GPU that would do the
    remaining computation.
  while(step.notfinish()){
    cudaDeviceSynchronize();
    #pragma omp barrier;
    //transfer data from current GPU to another GPU
    transferdata(src[step][tid],dst[step][tid]);
  }
  cudaDeviceSynchronize();
  #pragma omp barrier;
  if(tid==0)Kernel2<<<...>>>(...);
      }
  ...}
```

Fig. 14. Code example of reduction with implicit device synchronization

As section IV mentioned, the overhead in cooperative multi-launch might be the cause of this performance difference.

On the other hand, we want to emphasize here the benefit of programming. We can easily rewrite implicit barrier code (Figure 14) into the explicit barrier one (Figure 13), i.e. a single persistent kernel is required in grid synchronization, and eliminate the complexity of managing several GPUs with CPU threads or processes. More importantly, the kernel function requires no knowledge of the hardware structure.

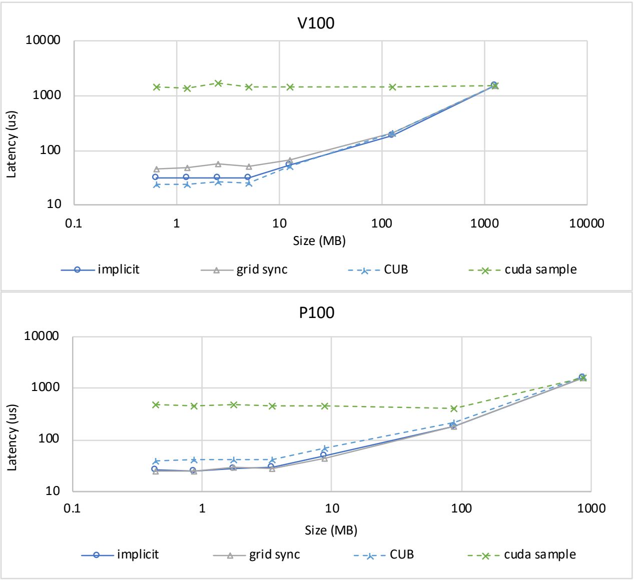


Fig. 15. Comparison of the performance of single reduction in V100 (up) and in P100 (down)

TABLE VI
BANDWIDTH (GB/S) OF DIFFERENT REDUCTION METHODS

	implicit	grid sync	CUB	CUDA sample	theory
V100	865.40	855.59	849.39	852.98	898.05
P100	592.40	590.85	543.96	590.65	732.16

VIII. Considerations of Using CUDA Synchronization Instructions

In this study, we identified several cases at which the synchronization instructions might not work as intended. In this section, we summarize some of those cases.

A. Synchronization Inside a Warp

In this section, we examine synchronization at the warp level. To see if a barrier inside a warp is effective on all threads in the barrier, we run the code in Figure 17. In the ideal case, the timers in all threads in the warp before the barrier are smaller than the timers after the sync in every thread. We test all the synchronization methods. Results show that P100 does not assure all threads inside a warp are blocked at the barrier (also the shuffle operation does not work correctly in this code either), which we believe explains why the latency of warp level synchronization in P100 is as fast as Table II shows. On the other hand, in V100, we observed the anticipated behavior (likely due to the fact that in V100 each thread has its own program counter). Figure 18 shows our observation when calling tile synchronization. We observed the same phenomenon when running all other synchronization instructions in both V100 and P100.

B. Deadlocks in Synchronization of Parts of Thread Groups

In this section, we examine the behaviour of synchronization with a subset of a thread group: would synchronizing a subset

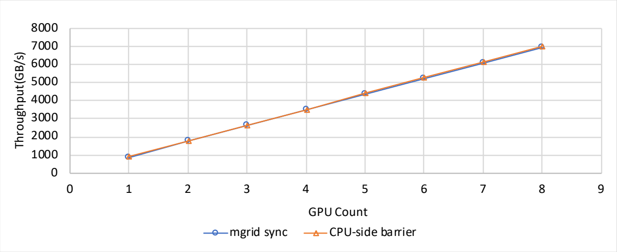


Fig. 16. The throughput of reduction on DGX-1

```
if(tid==0){timer(start);sync;timer(end);}
else if(tid==1){timer(start);sync;timer(end);}
...
else if(tid==30){timer(start);sync;timer(end);}
else{timer(start);sync;timer(end);}
```

Fig. 17. Code example to verify synchronization inside a warp

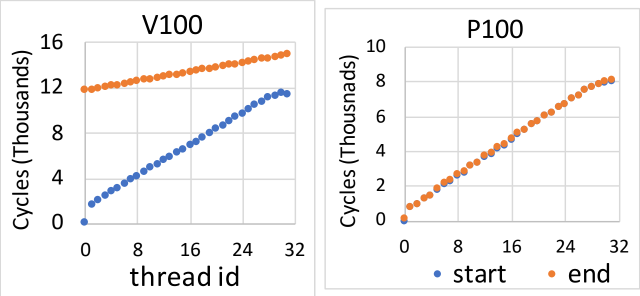


Fig. 18. Timer of threads inside a warp when calling tile synchronization in V100 (left), and in P100 (right) in code sample of Figure 17

TABLE VII
ENVIRONMENT INFORMATION

Platform	Default Freq.	Driver	CUDA
P100 x 2	1189MHz	418.40.04	V10.0.130
V100 x 8(DGX-1)	1312MHz	410.129	V10.0.130

of a group cause a deadlock or not? We implement a test suite to see what happens when part of a thread group calls the synchronization function. We test through every granularity including threads, warps, blocks, and GPUs. As a result, we observed deadlocks when we synchronize parts of blocks in grid group, multi-grid group, and when we synchronize parts of GPUs in a multi-grid group. In summary, one should be careful, after initializing a grid group or a multi-grid group, since current CUDA does not support synchronizing sub-groups inside a grid group.

IX. BENCHMARKING CUDA SYNCHRONIZATION METHODS

A. Experiments Environment

We use Pascal P100 and Volta V100 cards to conduct our experiments. We set the application frequency of both platforms to default. We use the latest stable driver. Table VII shows the details of the environment.

B. Micro-benchmark for Implicit Barriers

We use the terminologies in Section IV. We do a warm-up kernel call before every measurement that we don't report the results for.

We found that directly using a null kernel would not give a correct result here. Because at this point the stream pipeline is not saturated enough: the overhead tested would be larger than usual. The kernel execution latency needs to be larger than a certain number. This value is around $5us$ for a single GPU and around $250us$ for 8 GPUs in DGX-1. In order to control the kernel latency, we use the sleep instruction introduced in CUDA for Volta platform. We use kernel fusion to unveil the overhead hidden in kernel latency. The basic assumption here is that merging the work of multiple argument-less kernels into one single kernel does not introduce additional launch overhead, and then the time saved when using kernel fusion should be equal to the overhead of launching an additional kernel. From our previous observations, the sleep instruction has insignificant overhead and fits well into this assumption. In this situation, we can compute the overhead with Equation 6.

Since we use the sleep instruction as a tool to analyze launch overhead, which is only available in Volta Platform in CUDA, we only conduct experiments on the V100 GPU for this experiment.

$$O = \frac{Latency_{ij} - Latency_{ji}}{i-j} \tag{6}$$

$*(O\ represents\ Overhead;\ In\ Latency_{ij}\ (the\ left\ one),$
$i\ represents\ call\ launch\ function\ i\ times,$
$j\ represents\ launch\ kernels\ with\ j\ wait\ unit)$

To the best of the authors' knowledge, Volkov et al. [8] was the first one measured the overhead of implicit barrier, i.e. CUDA kernel launch overhead. Xiao et al. [5] additionally build a model for implicit and explicit barriers. They both neglect the fact that the launch overhead is far smaller when kernel execution latency is long enough. When using null kernels, we tested a launch overhead of around $3us$ for traditional launch, which is the same as the best case reported by Volkov et al. [8].

C. Micro-benchmark for Intra-SM Instructions

We directly use Wong's [9] method for instruction micro-benchmarking. Wong's method relies on the GPU clock. The basic methodology is to build a chain of dependent operations to repeat a single instruction enough times to saturate the instruction pipeline. By using the clock register to record the begin and end timestamps of the series of operations, it is possible to average the repetitions to infer the latency of that instruction. Figure 19 shows an example sample code to measure the latency of an add instruction.

D. Micro-benchmark for Inter SM Instructions

Jia's work [11] can work correctly only inside a single thread, Wong's work [9] can work correctly only in a single

```
__global__ void kernel1(){
  start=clock();
  repeat256(p=p+q;q=p+q); //repeat=512
  end=clock();
  return q;
  }
```

```
__global__ void kernel2(){
  start=clock();
  repeat512(p=p+q;q=p+q); //repeat=1024
  end=clock();
  return q;
  }
```

```
cpuclock();
kernel();
syncdevice();
cpuclock();
```

Fig. 19. Sample code to measure the latency of the add instruction in GPU

SM. Yet current synchronization instructions might involve cooperation across different threads, different SMs, and even different GPUs. As we move to grid level synchronization and beyond, we need a new method.

In order to test the performance of synchronization beyond a single SM, a global clock is necessary. In CUDA's execution model, a CPU thread launches a kernel and it can call the DeviceSynchronize() function to block the CPU thread until the GPU kernel finishes execution. So it is possible to use the clock in that CPU thread as a global clock to test GPU instructions. Yet we need to fix two issues before we can use the CPU clock:

- We need to eliminate any latency not related to the target instruction
- Account for the relative inaccuracy in the CPU clock measurement, in comparison to the GPU's clock measurement.

In order to solve those issues, we need to additionally introduce two assumptions:

- The measurement of the latency of every instruction becomes more accurate when the pipeline is saturated
- Additional instructions in a kernel do not increase the launch overhead of kernel launch

$$T_{instruction} = \frac{L_{k_1} - L_{k_2}}{r_1 - r_2} \tag{7}$$

$$\begin{aligned}
\sigma_{\frac{k_1-k_2}{r_1-r_2}} &= \sqrt{\frac{\sum_{n=1}^{N}(\frac{L_{k_1}-L_{k_2}}{r_1-r_2})^2 - \sum_{n=1}^{N}\overline{(\frac{L_{k_1}-L_{k_2}}{r_1-r_2})}^2}{N-1}} \\
&= \frac{1}{r_1-r_2}\sqrt{\frac{\sum L_{k_1}^2 - \overline{L_{k_1}}^2}{N-1} + \frac{\sum L_{k_2}^2 - \overline{L_{k_2}}^2}{N-1}} \\
&= \frac{1}{r_1-r_2}\sqrt{\sigma_{k_1}^2 + \sigma_{k_2}^2}
\end{aligned} \tag{8}$$

*(L_{k_i} *represents kernel total latency of kernel* i;
r_i *represents repeat times in kernel* i)

Under those assumptions, if we increase the repetitions of instructions in the GPU kernel (in Figure 19), the additional kernel latency is only related to the additional repeat times of instructions. In this manner, we are able to avoid unrelated latency that might come from kernel launch (to get more accurate measurements). Equation 7 shows how to measure the instruction latency with this method. (First issue solved)

Standard deviation can be used to represent the uncertainty in a single measurement [25]. Equation 8 shows the standard deviation of the instruction tested, and its deduction (the measurement of kernel 1 and kernel 2 is independent to each other). And by deduction, if the difference in repeat times is large enough, the standard deviation of the instruction latency we seek to measure will be small. (Second issue solved)

In order to verify that the method we proposed in Section IX-D matches our assumptions, we use both Wong's method and our method to test the single precision add instruction. Both results show that float-add costs 6 cycles in P100 and 4 cycles in V100. Those results match the result in [11]. We can conclude that the inter SM micro-benchmark method we propose is a reliable measurement tool that approaches the accuracy of the GPU clock.

We additionally verify that the repeat times of a synchronization instruction itself would not influence the performance itself in block and grid level. Tile shuffle at warp level also works as we anticipated. Other warp level synchronization can be unstable: the latency of the synchronization instruction might increase suddenly when increasing repeat times. It could be the case that this warp synchronization relies on a software implementation. So when repeating an instruction too many times, instruction cache overflow can occur. We only record the fastest result for warp level synchronization instructions.

X. Conclusion

In this paper, we conduct a detailed study of different synchronization methods in Nvidia GPUs, ranging from warp to grid, and from single GPU to multi-GPU.

We find that the performance of block synchronization is related to the number of warps involved, and the performance of grid level synchronization is mainly affected by the number of blocks involved. In addition, the performance of multi-grid level synchronization depends on the network structure connecting the GPUs, and the number of active blocks and warps.

CPU-side implicit barriers generally perform better than grid level and multi-grid level synchronization. But if the program size is large enough, the performance difference would not be so severe, with the added benefit that multi-grid synchronization simplifies multi-GPU programming.

We use the reduction operator as an example to use the knowledge we gain from micro-benchmark. We build a performance model to predict where would be the point that using fewer threads is more profitable. Additionally, using code samples, we show a possible simple way to do multi-GPU programming without much performance degradation. Moreover, with more multi-grid barriers in a kernel, the launch

TABLE VIII
SUMMARY OF OBSERVATIONS

Warp Level Sync	Does not work on Pascal; Shuffle performs better in real code.
Block Sync	The number of active warps per SM affects performance
Grid Sync	The number of blocks per SM mainly affects performance; Generally, the performance is acceptable if $block/SM <= 2$; Currently, only parts of blocks inside a grid calling grid level synchronization would cause deadlock.
Multi-Grid Sync	Both the number of blocks per SM and active warps per SM affect performance; If $thread/SM <= 1024$ and $block/SM <= 8$ the performance is relatively acceptable; Currently, only parts of grids inside a grid calling grid level synchronization would cause deadlock.
Implicit Sync & CPU Based Sync	Generally, their performance is slightly better than explicit synchronization when in single GPU or when the GPU count is large, or when there is no much synchronization steps; The issue for CPU Based Sync is programmability, especially in the situation of multi-GPUs.

overhead in multi-device kernel launch would become more insignificant. Table VIII summarizes the knowledge we gained from this study.

XI. ACKNOWLEDGMENTS

This work was partially supported by JST-CREST under Grant Number JPMJCR19F5.

REFERENCES

[1] Nvidia, "Programming guide," 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2] M. Harris *et al.*, "Optimizing parallel reduction in cuda," *Nvidia developer technology*, vol. 2, no. 4, p. 70, 2007.

[3] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," in *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, vol. 5, 2015, pp. 1–6.

[4] T. Sorensen, A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakamarić, "Portable inter-workgroup barrier synchronisation for gpus," in *ACM SIGPLAN Notices*, vol. 51, no. 10. ACM, 2016, pp. 39–58.

[5] S. Xiao and W.-c. Feng, "Inter-block gpu communication via fast barrier synchronization," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–12.

[6] J. Liu, "Efficient synchronization for gpgpu," Ph.D. dissertation, University of Pittsburgh, 2018.

[7] NVIDIA, "V100 gpu architecture," 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[8] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 2008, pp. 1–11.

[9] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2010, pp. 235–246.

[10] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, "Understanding the gpu microarchitecture to achieve bare-metal performance tuning," *ACM SIGPLAN Notices*, vol. 52, no. 8, pp. 31–43, 2017.

[11] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[12] S. S. Baghsorkhi, I. Gelado, M. Delahaye, and W.-m. W. Hwu, "Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 23–34.

[13] X. Mei, K. Zhao, C. Liu, and X. Chu, "Benchmarking the memory hierarchy of modern gpus," in *IFIP International Conference on Network and Parallel Computing*. Springer, 2014, pp. 144–156.

[14] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2016.

[15] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU technology conference, GTC*, vol. 10. San Jose, CA, 2010, p. 16.

[16] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar, "Fine-grained synchronizations and dataflow programming on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 109–118.

[17] Nvidia, "Nvidia cuda runtime api," 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[18] J. Luitjens, "Faster parallel reductions on kepler," *Parallel Forall. NVIDIA Corporation. Available at: https://devblogs. nvidia. com/parallelforall/faster-parallel-reductions-kepler*, 2014.

[19] Nvidia, "Nvidia cuda sample," 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-samples/index.html

[20] P. J. Martín, L. F. Ayuso, R. Torres, and A. Gavilanes, "Algorithmic strategies for optimizing the parallel reduction primitive in cuda," in *2012 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2012, pp. 511–519.

[21] W. A. R. Jradi, H. Nascimento, and W. S. Martins, "A fast and generic gpu-based parallel reduction implementation," in *2018 Symposium on High Performance Computing Systems (WSCAD)*. IEEE, 2018, pp. 16–22.

[22] S. G. De Gonzalo, S. Huang, J. Gómez-Luna, S. Hammond, O. Mutlu, and W.-m. Hwu, "Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on gpus," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Press, 2019, pp. 73–84.

[23] J. D. Little and S. C. Graves, "Little's law," in *Building intuition*. Springer, 2008, pp. 81–100.

[24] Nvidia, "Cub library," 2019. [Online]. Available: https://nvlabs.github.io/cub

[25] J. Taylor, *Introduction to error analysis, the study of uncertainties in physical measurements*. University Science Books, 1997.