# A Study of Synchronization Methods in Modern GPUs

Lingqi Zhang[1,a)] Mohamed Wahib[2] Haoyu Zhang[1] Satoshi Matsuoka[3]

***Abstract:*** GPUs are playing an increasingly important role in general-purpose computing. Various scientific fields utilize the power of GPUs. Many complex algorithms require different levels of synchronizations, through the use of barriers. Many researchers struggle to implement communication methods for GPUs, and thus implementations tend to under-perform for specific algorithms that require device-wide barriers. This work approaches this problem by using micro-benchmarks to study Nvidia's new synchronization methods, as well as the performance they yield.

***Keywords:*** GPU, synchronization, micro-benchmark

## 1. Introduction

GPUs have been playing an increasingly important role in general-purpose computing. Different scientific areas exploit the power of GPUs to solve various questions. Many complex algorithms require different levels of synchronizations, through the use of barriers. Before Nvidia proposed a hierarchy of synchronization methods [1], developers made use of block synchronization and the implicit barrier inside a warp to develop complex algorithm [2]. Besides, for applications like deep neural network, an implicit barrier produced by kernel launch function is playing a role of device-wide synchronization [3].

Previous researchers also struggled to develop software device-wide barriers [4], [5]. But the upcoming GPU dense systems, e.g. DGX, call for a general way for devices-wide synchronization. Recently Nvidia proposed different levels of synchronizations, including warp level, block level, and grid level. The grid level synchronization can be a productive way to perform device-wide and multi-device level synchronization. In other words, this hierarchy of synchronization methods can make GPUs programming more productive and help developers to construct a more complicated software structure. Thus, it is valuable to study the performance of utilizing different levels of synchronization methods.

There are many successful methods to conduct GPUs specific micro-benchmark. Wong etc. [6] was the first research using micro-benchmark to understand the performance of GPUs. Their research was thorough and firm. Mei etc. [7] focus on memory hierarchy of GPUs, they discovered some cache patterns that was dismissed by [6]. Recently, Jia, etc. [8] proposed to use asm code to conduct micro-benchmark. And conduct research on new Nvidia Platforms, i.e. V100 and P100 GPUs. This result is guaranteed to be more accurate, but it is possible to achieve the same result with the methods proposed by [6]. Nonetheless, we see no researches focus on Nvidia's hierarchy synchronization structures.

In this research, we use micro-benchmark to demystify the performance feature of different level of synchronization methods in Nvidia GPUs.

The remainder of this paper is organized as follows. Section 2 reviews the CUDA programming model and CUDA synchronization hierarchy. Section 3 describes our measurement methodology, and Section 4 presents the results. Section 5 summarizes our ndings and Section 6 introduces our future plan.

## 2. Background

### 2.1 CUDA Programming Model

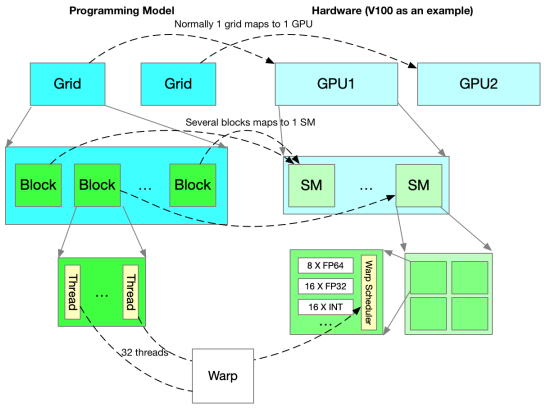


**Fig. 1** Programming model and corresponding hardware structure in CUDA

CUDA provides a C-like programming model to utilize Nvidia GPUs. It offers three levels of programming abstractions: thread, block and grid. Among them, Thread is the most basic programming unit.

From hardware perspective, there is a hierarchy structure similar to CUDA programming model. Three different kinds of hardware structure exists: Mathematics Unit, Stream Multi-Processor (SM) and GPU. Take the structure V100 [9] as an example, a

[1] Tokyo Institute of Technology, Dept. of Mathematical and Computing Science, Tokyo, Japan
[2] AIST-Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory
[3] RIKEN Center for Computational Science,Hyogo,Japan
[a)] zhang.l.ai@m.titech.ac.jp

V100 GPU consists of 84 SMs; an SM is partitioned into 4 processing blocks, each consists of several Mathematics Units, e.g. 16 FP32 Cores.

There is a special abstraction, named Warp, to bridge the gap between the programming model and the hardware SMs. Currently a warp consists of 32 threads. And inside an SM in V100, there are 4 Warp schedulers corresponding to the 4 partitions inside a SM. Beyond that, the CUDA runtime will schedule 1 block to only one SM, and 1 grid to only one GPU, though it might occupy several SMs.

Figure 1 shows the detail of CUDA programming model, its corresponding hardware abstraction, and the mapping relationship between both structures.

### 2.2 Different Levels of Synchronization in CUDA

As Figure 2 shows, CUDA support different kinds of groups and provide different levels of synchronization methods for these groups, including warp level, block level and grid level.

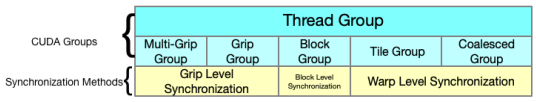


**Fig. 2** Synchronization Hierarchy in CUDA

#### 2.2.1 Warp Level Synchronization

Current CUDA support 2 intra-warp synchronization methods, i.e. tile synchronization and the coalesced group synchronization, corresponding to the tile group and coalesced group in Figure 2

Tile synchronization is based on the tile group built inside a warp. For example, set tile size to 4 can create 8 independent tile groups inside a Warp currently, each group consist of 4 threads. Coalesced group synchronization is based on the coalesced group, which is built from a single branch of an if-statement inside a warp. For example, the if statement (tid%4==1) can create a coalesced group with 8 members inside a Warp currently. Both of them can not possess a group size larger than 32 (size of a warp). And the synchronization only happened inside a single group.

Additionally, in order to better understand the difference between tile synchronization and coalesced synchronization, we can take the reduction algorithm as an example. Both methods can be used to implement the reduction algorithm inside a warp.

Figure 3 shows the case of using tile synchronization. At the step of reduction moves forward, more tiny tile group is created. When synchronization happens, each small tile groups synchronize all threads inside.

Figure 4 shows the case of using coalesced synchronization. At the step of reduction moves forward, a smaller group is created. When synchronization happens, that group synchronizes all threads inside.

Previous versions of CUDA guarantee that all threads inside a warp process the same instruction at a time. But the proposing of synchronization methods inside a warp implies a potential future plan of removing this feature.

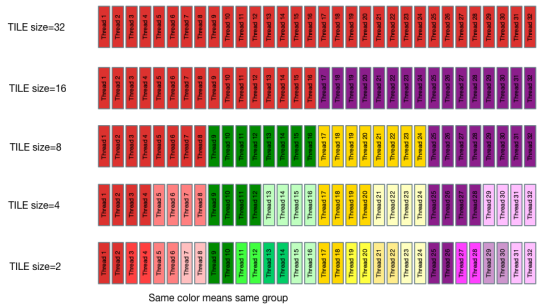


**Fig. 3** Use tile group to implement reduction


**Fig. 4** Use coalesced group to implement reduction

#### 2.2.2 Block Level Synchronization

Block level synchronization corresponds to the block abstract in the programming model. It is a rename of the instruction __syncthreads in previous CUDA versions.

#### 2.2.3 Grid Level Synchronization

Start from CUDA 9.0, Nvidia proposed grid synchronization and multi-grid synchronization. Grid synchronization synchronizes every block inside a grid abstraction in CUDA programming model. Multi-grid synchronization synchronizes all the grid that launched by cudaLaunchCooperativeKernelMultiDevice() function.

## 3. Micro-benchmark

Both Wong [6] and Jia [8] proposed methods to measure a instruction inside a GPU. Jia's work needs to modify the asm code. But according to our experiments it achieves the same result as Wong's work while requiring additional knowledge of Disassemble.

Additionally, Jia's work can work correctly only inside a single thread, Wong's work can work correctly only in single SM. But Synchronization might involve cooperation across different threads, different SMs and even different GPUs.

We utilize Wong's work to measure the performance of synchronization instruction inside an SM (Section 3.1). And we further propose a method to measure the latency of GPU instructions in CPU (Section 3.2).

### 3.1 Measuring GPU Instructions with GPU-based Methods

Wong's [6] method is a GPU based measurement. The basic methodology is to build a chain of dependent operations to repeat a single instruction several times to saturate the instruction pipeline. By utilizing clock register to record the begin and end time stamp of these serials of operations, it is possible to use aver-

age latency to infer the latency of that instruction. Figure 5 takes measuring add instruction as an example.

```
start=clock();
repeat256(p=p+q;q=p+q);
//an add instruction is repeat 512 times here
end=clock();
return q;
```

**Fig. 5** Sample code to measure the latency of add instruction inside a GPU kernel

### 3.2 Measuring GPU Instructions With CPU-based Methods

In order to test the performance of synchronization beyond a single SM, a global clock is necessary. In CUDA execution model, a CPU thread launches a kernel and it can call DeviceSynchronize() function to wait for the finish of the kernel. So it is possible to use the clock used by that CPU thread as a global clock to test GPU instructions.

But there still remain two issues:

- CPU function is more unstable than GPU's
- Need to wipe off the latency not related to the aimed instruction

By assuming that the latency of every instruction is stable when the pipeline is saturated, and additional instruction does not increase the launch overhead of kernel launch, we propose a CPU based measurement. If we increase the repeat times in the GPU kernel in Figure 5, the additional time consumption is only related to the additional repeat times. In this way, we are able to wipe off unnecessary latency. Formula 1 shows how to compute instruction overhead with this method.

$$T_{instruction} = \frac{Latency_{kernel1} - Latency_{kernel2}}{Repeat_{kernel1} - Repeat_{kernel2}} \quad (1)$$

Standard Deviation shows how far every experiment spread out over mean value. We can use the Standard Deviation to indicate the stability of the test result. Formula 2 shows that, given that Standard Deviation is not related to the repeat time of the instruction, if the difference in repeat times is large enough, the Standard Deviation of the result of the instruction we want to test will be small. And small Standard Deviation means that the mean of the result is stable. Because the stability of CPU function is independent to the kernel function, it can be reduced to a low level if we choose a suitable difference in repeat time.

$$\sigma_{instruction} = \frac{\sqrt{\sigma^2_{kernel1} + \sigma^2_{kernel2}}}{Repeat_{kernel1} - Repeat_{kernel2}} \quad (2)$$

### 3.3 Verification

In order to verify the method we proposed in section 3.2 is feasible, we use both Wong's method and our method to test float add instruction in both V100 and P100. Both results show that float-add costs 6 cycles in P100 and 4 cycles in V100. These results match the result in [8].

Table 1 shows the details of the verification result in the V100 GPU. It is easy to see that CPU inferred latency is equal to GPU result within the margin of error. We can also observe that when the repeat times is larger than 8192, the average latency of a single instruction starts to increase. We guess this is caused by either the overflow of instruction pipeline or the cache miss of instruction cache. The experiment in P100 GPU also gave a similar result.

So, in conclusion the CPU based method we proposed gives exactly the same result as Wong's [6] and Jia's [8] method.

**Table 1** Comparison of measuring latency of float add instruction with CPU based measurement and GPU based measurement in V100(average in 20 experiments)

| Repeat Difference | CPU Inferred Latency(cycle) | GPU Tested Latency(cycle) |
|---|---|---|
| (512) | - | **4.150** |
| 2056 | **4.045** | **4.025** |
| 5120 | **4.034** | **4.025** |
| 7680 | 5.278 | 5.067 |
| 10240 | 7.259 | 6.952 |
| 20480 | 7.080 | 6.950 |
| 25600 | 7.066 | 6.958 |
| 40960 | 7.014 | 6.954 |
| 51200 | 7.005 | 6.958 |

### 3.4 Metrics

We used two metrics to measure the performance of synchronization instruction, i.e. latency and throughput. Latency is the time required to finish a certain operation. Throughput is the max number of operations finished per time unit. The choice of these metrics follows the Little's Law [10].

Additionally, synchronization is not steady by its nature. We only record the average value here among 20-time experiments.

Table 2 shows the test plan for the following experiments. As Table 2 shows, we only use GPU based method to measure warp level synchronizations and latency of block level synchronization, while use CPU based method to measure others. This is because GPU based measurement is easy to implement with higher accuracy. But GPU based measurement can only measure the latency in a single SM, which is not the case in testing the throughput of block and testing grid level synchronizations.

For GPU based measurement, we use a basic repeat time of 512 times, which is the same as [6]. But we observe a suddenly latency increase in warp-level synchronization, before the average latency is stable to a certain value. Seems that some overhead happened before warp-level synchronization reaches its pipeline saturation. Because the faster result might be the closest to the value when it reaches its saturation point, we only record the faster result here.

For CPU based measurement, we keep the basic repeat time to 512 times and repeat difference to 5120 times. We also did additional test to prove that the latency of block and grid synchronization instruction is stable when the pipeline is saturated, and additional instruction does not increase the launch overhead of kernel launch.

We record both latency and throughput for block level and warp level synchronizations, while only record latency for grid level synchronizations. Because we can hardly imagine a scenery that several kernels run simultaneously in a single GPU and need to run synchronization instruction at the same point, but it is easy to imagine similar cases in both warp level and block level synchronizations.

**Table 2** Methods and metrics to test different synchronization levels

| Level | Method | Metric |
|---|---|---|
| **Warp** | GPU based | Latency and Throughput |
| **Block** | GPU based and CPU based | Latency and Throughput |
| **Grid** | CPU based | Latency |

### 3.5 Experiment environment

We use both P100 and V100 card to conduct this experiment. Both are an up-and-coming platform in CUDA. The feature of these platforms suggests the trend of Nvidia GPUs in the near future.

In order to get a more accurate result, we set the application frequency of both platforms to default. We used the latest driver and CUDA version. But because V100 card located in a DGX machine, the driver of the machine is relatively out of data then P100, but relatively stable. Table 3 shows the details of the experiment environment.

Because synchronization operation my introduce random error, the results in Section 4 is the average value of 20 times experiments.

**Table 3** Environment information

| Platform | Default Freq | Driver | CUDA |
|---|---|---|---|
| P100 X 2 | 1189MHz | 418.40.04 | V10.0.130 |
| V100 X 8(DGX1) | 1312MHz | 410.104 | V10.0.130 |

## 4. Results

### 4.1 Warp Level Synchronization

**Table 4** Performance of Warp Synchronization

| Type | Group Size | Latency(cycle) V100 | Latency(cycle) P100 | Throughput(OP/cycle) V100 | Throughput(OP/cycle) P100 |
|---|---|---|---|---|---|
| **TILE** | * | 54 | 198 | $\frac{8.353}{(GroupSize)}$ | $\frac{2.89}{GroupSize}$ |
| **Coalesced** | 1-31 | 108 | 1 | 0.166 | 1.882 |
| **Coalesced** | 32 | 14 | 1 | 0.769 | 1.882 |

Under a believe that 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 size is: 1, 2, 4, 8, 16 and 32. The possible coalesced group size is 1-32. Latency is tested by using only 32 threads (a warp) in a CUDA kernel with 1 block. Throughput is tested by using 1024 threads (upper limit of a block) in a CUDA kernel with 1 block.

Table 4 shows the result of warp level synchronization.

For tile group synchronization the size of the group participated do not influence the performance, both latency and throughput. A possible explanation is that current CUDA might merge all the concurrent tile group synchronization instruction into one single instruction.

For coalesced group synchronization, the group size does not influence the performance of P100 GPU. But the group size does influence the performance of V100 GPU. When all the threads inside a warp belong to a single coalesced group, the performance is the highest.

### 4.2 Block Level Synchronization

Again, we tested every possible group size in the block level, i.e. start from 32 to 1024. Latency is tested with only one block.

**Table 5** Performance of Block Synchronization

| Thread Size | Latency(cycle) V100 | Latency(cycle) P100 | Throughput(block sync/us) V100 | Throughput(block sync/us) P100 |
|---|---|---|---|---|
| 32 | 22 | 220 | 219.296 | 87.329 |
| 64 | 24 | 220 | 200.602 | 51.67 |
| 128 | 28 | 224 | 143.136 | 27.674 |
| 256 | 36 | 235 | 81.241 | 13.731 |
| 512 | 52 | 317 | 40.812 | 6.598 |
| 1024 | 84 | 428 | 19.469 | 3.130 |

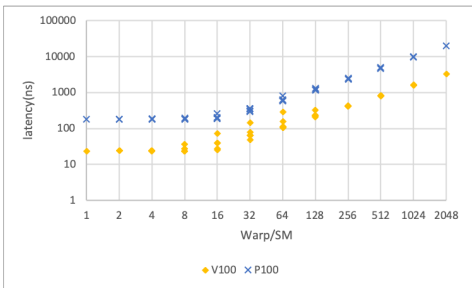


**Fig. 6** Relationship between latency and warp/SM

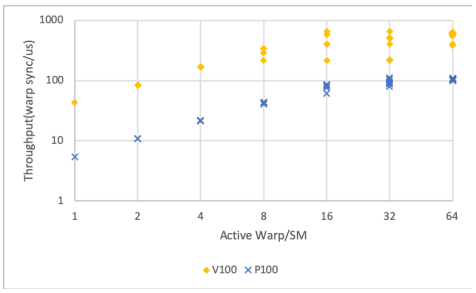


**Fig. 7** Relationship between throughput of block sync (per warp perspective) and active warp/SM

Throughput is tested by increasing the block size from 1 block per SM to 64 blocks per SM and record the highest throughput among all results.

Table 5 shows the result. We observe a performance decrease both in V100 and P100 platform as block size increases.

Additionally, Figure 6 shows the relationship between total block synchronization latency and warp/SM. There are around 42 points in the figure, and most of the points are close to a line. Thus, we can deduce that the performance of block synchronization is related to the warp count per SM.

In order to prove that, we additionally draw a figure to show the relationship between the throughput of block synchronization divided by warp count (warp sync per us) and the activate warp per SM. Figure 7 shows the result. Maximum active warp is computed according to [9], maximum block count is 32 and maximum warp count is 64 per SM in both V100 and P100 platform. When warp count exceeds the size of max activate warp per SM, the device is saturated and the throughput of block synchronization reaches its maximum.

### 4.3 Grid Level Synchronization

In order to utilize grid synchronization and multi-grid synchronization, cooperative launch related methods are necessary. These launch methods will make some limitation to both grid-dim and block-dim.

Our observation is that, when the total warp is larger then the

maximum active warp in current kernel setting, the launch function will reject to launch a kernel.

### 4.3.1 Grid Synchronization

| V100 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|
| 1 | 1.43 | 1.44 | 1.46 | 1.50 | 1.72 | 2.20 |
| 2 | 1.84 | 1.85 | 1.90 | 1.99 | 2.49 | 3.49 |
| 4 | 2.85 | 2.89 | 3.07 | 3.52 | 4.54 | |
| 8 | 5.06 | 5.25 | 5.70 | 6.65 | | |
| 16 | 9.21 | 9.49 | 10.39 | | | |
| 32 | 21.06 | 24.78 | | | | |

| P100 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|
| 1 | 1.76 | 1.76 | 1.78 | 1.82 | 1.90 | 2.25 |
| 2 | 2.05 | 2.07 | 2.08 | 2.21 | 2.65 | 3.49 |
| 4 | 3.42 | 3.46 | 3.57 | 4.01 | 4.85 | |
| 8 | 6.43 | 6.51 | 7.03 | 8.48 | | |
| 16 | 13.23 | 14.75 | 16.63 | | | |
| 32 | 35.13 | 31.34 | | | | |

**Fig. 8** Latency (us) of grid synchronization in V100 (left) and P100 (right)

Figure 8 shows the heat map of grid synchronization. It shows that that in both V100 and P100 the latency of grid synchronization is more related to grid dim than to block dim.

So, in order to use grid synchronization, it is better to control the number of blocks resided in a single SM.

### 4.3.2 Multi-Grid Synchronization

| 1 GPU | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|
| 1 | 1.44 | 1.45 | 1.47 | 1.51 | 1.78 | 2.49 |
| 2 | 1.71 | 1.74 | 1.81 | 2.08 | 2.91 | 4.52 |
| 4 | 2.99 | 3.04 | 3.27 | 3.98 | 5.61 | |
| 8 | 5.43 | 5.55 | 6.46 | 8.50 | | |
| 16 | 9.72 | 10.97 | 14.13 | | | |
| 32 | 22.94 | 29.06 | | | | |

| 2 GPU | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|
| 1 | 7.26 | 7.25 | 7.32 | 7.35 | 7.68 | 8.41 |
| 2 | 7.90 | 7.93 | 7.95 | 8.30 | 9.08 | 10.82 |
| 4 | 10.73 | 10.75 | 11.05 | 11.80 | 13.28 | |
| 8 | 18.26 | 17.84 | 19.01 | 20.96 | | |
| 16 | 33.18 | 34.13 | 36.92 | | | |
| 32 | 69.08 | 74.89 | | | | |

**Fig. 9** Latency (us) of multi-grid synchronization in P100 platform 1 GPU (left) and 2 GPUs (right)

| 1 GPU | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|
| 1 | 1.43 | 1.45 | 1.57 | 2.06 | 3.09 | 7.38 |
| 2 | 1.84 | 1.88 | 2.35 | 3.37 | 6.97 | 19.07 |
| 4 | 2.90 | 3.36 | 4.36 | 7.53 | 19.09 | |
| 8 | 5.31 | 6.35 | 9.10 | 20.58 | | |
| 16 | 9.67 | 11.68 | 23.89 | | | |
| 32 | 20.63 | 33.41 | | | | |

| 2 GPU | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|
| 1 | 6.43 | 6.45 | 6.54 | 6.99 | 8.08 | 12.45 |
| 2 | 6.77 | 6.80 | 7.28 | 8.34 | 11.80 | 24.09 |
| 4 | 7.97 | 8.42 | 9.47 | 12.54 | 24.13 | |
| 8 | 12.48 | 13.64 | 16.54 | 27.97 | | |
| 16 | 22.48 | 24.66 | 36.99 | | | |
| 32 | 45.80 | 58.50 | | | | |

| 5 GPU | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|
| 1 | 7.05 | 7.07 | 7.17 | 7.64 | 8.71 | 13.26 |
| 2 | 7.36 | 7.42 | 7.88 | 8.98 | 12.71 | 25.11 |
| 4 | 8.67 | 9.14 | 10.17 | 13.37 | 25.13 | |
| 8 | 13.20 | 14.23 | 17.16 | 28.64 | | |
| 16 | 23.57 | 25.63 | 38.09 | | | |
| 32 | 48.76 | 61.42 | | | | |

| 6 GPU | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|
| 1 | 18.70 | 18.74 | 18.76 | 19.29 | 20.26 | 24.63 |
| 2 | 19.01 | 19.08 | 19.59 | 20.53 | 23.76 | 35.85 |
| 4 | 20.44 | 20.89 | 21.84 | 24.70 | 36.32 | |
| 8 | 23.49 | 24.93 | 27.15 | 38.49 | | |
| 16 | 29.78 | 31.55 | 43.79 | | | |
| 32 | 54.44 | 67.64 | | | | |

| 8 GPU | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|
| 1 | 20.97 | 20.92 | 21.03 | 21.64 | 22.67 | 27.05 |
| 2 | 21.33 | 21.44 | 21.79 | 22.91 | 26.07 | 38.00 |
| 4 | 22.57 | 23.27 | 24.09 | 26.90 | 38.50 | |
| 8 | 25.79 | 26.71 | 29.44 | 41.22 | | |
| 16 | 31.55 | 33.73 | 46.24 | | | |
| 32 | 58.72 | 72.09 | | | | |

**Fig. 10** Latency (us) of multi-grid synchronization in V100 platform

Figure 9 and Figure 10 show the heat maps of the latency of multi-grid synchronization in V100 and P100. Though we tested through all 8 GPUs in DGX1, we found that the performance multi-grid synchronization among 2-5 GPUs are similar to each other, and the performance multi-grid synchronization among 6-8 GPUs are similar to each other. The reason might more or less related to the network structure of DGX1. From Figure 9 and Figure 10, it is easy to see that the performance of multi-grid synchronization is influenced by both grid dim and active warp per SM. With $griddim < 8$ and $activewarp < 32$, the performance is acceptable to our perspective. Apart from the case of 1 GPU, its latency is no more than 2 times slower than the fastest case (1 block per SM, 32 threads per block) and 2 times faster than the slowest case (32 blocks per SM, 64 threads per block).

Figure 11 shows the latency of multi-grid synchronization across 8 GPUs in DGX1. We take three cases for this experiment: 1. 1 block/SM, 32 thread/block as the fastest case; 2. 32 block/SM, 64 thread/block as the slowest case; 3. 4 block/SM, 256 thread/block as a general case, which is within the parameter

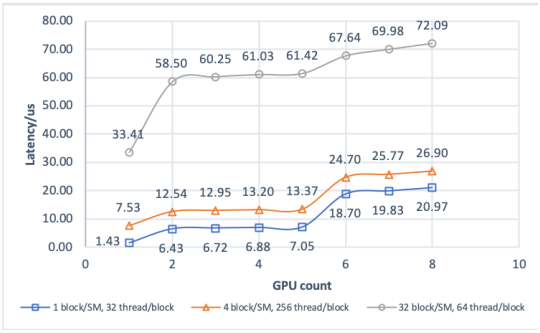


**Fig. 11** Latency of multi-grid synchronization across 8 GPUs

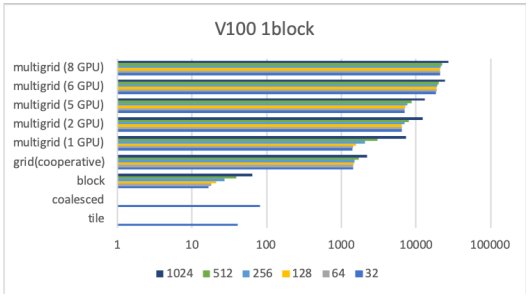


**Fig. 12** Latency(ns) of synchronization through all levels (1 V100 GPU, 1 block)

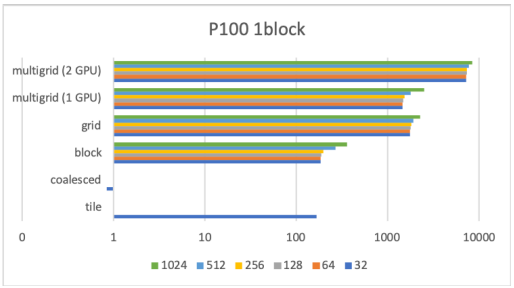


**Fig. 13** Latency(ns) of synchronization through all levels (1 P100 GPU, 1 block)

we recommend in the previous paragraph. In addition to proving that the parameter setting we gave is practical, Figure 11 also shows two performance gap: one between 1-GPU and 2-GPU and one between 5-GPU and 6-GPU. It is easy to understand the first performance gap, because 1-GPU case might not require any network involvement. We thought that the second gap should between 4-GPU and 5-GPU, based on the network structure of DGX1 that 4 GPU groups together, if there are any kind of such gap. This performance gap between 5-GPU and 6-GPU might come from bad implementation.

### 4.4 Comparison through all level

To better understand the performance differences of all synchronization methods through all levels, we plot two figure with all the 1 block data from both V100 and P100 platform. Figure 12 and Figure 13 show the results of V100 and P100 respectively. We list our finding as follow:

- **Grid-level synchronization's performance is similar in single GPU.**
- **1024-thread always means a large performance degrade**
- **Performance gap between different synchronization level does not always exist** The performance of warp level synchronization is not so difference as block level synchroniza-

**Table 6** Summarize of latency results in 1 GPU, warp level and block level

| Type | Case | V100 | P100 |
|---|---|---|---|
| **Tile**(cycle) | *groupsize* = 1 − 32 | 54 | 198 |
| **Coalesced**(cycle) | *groupsize* = 1 − 31 | 108 | 1 |
| **Coalesced**(cycle) | *groupsize* = 32 | 14 | 1 |
| **Block**(cycle) | *blockdim* = 32 (min) | 22 | 220 |
| **Block**(cycle) | *blockdim* = 256 (acceptable) | 36 | 235 |
| **Block**(cycle) | *blockdim* = 1024 (max) | 84 | 428 |

**Table 7** Summarize of throughput results in 1 GPU, warp level and block level

| Type | Case | V100 | P100 |
|---|---|---|---|
| **Tile**(sync/sync) | *group size* = 1 | 8.353 | 2.89 |
| **Tile**(sync/sync) | ... | ... | ... |
| **Tile**(sync/sync) | *group size* = 32 | 0.261 | 0.090 |
| **Coalesced**(sync/sync) | *group size* = 1 | 0.090 | 1.882 |
| **Coalesced**(sync/sync) | *group size* = 32 | 2.89 | 1.882 |
| **Block**(warp-sync/ns) | *blockdim* = 32 | 0.219 | 0.087 |
| **Block**(warp-sync/ns) | *blockdim* = 64 | 0.402 | 0.103 |
| **Block**(warp-sync/ns) | *blockdim* > 64 | 0.626-0.658 | 0.102-0.111 |

**Table 8** Summarize of latency (us) results in 1 GPU, grid sync

| | V100 | | P100 | |
|---|---|---|---|---|
| Case | Min | Max | Min | Max |
| *griddim* = 1 | 1.435 | 2.199 | 1.762 | 2.254 |
| *griddim* = 2 | 1.838 | 3.485 | 2.050 | 3.493 |
| *griddim* = 4 | 2.847 | 4.536 | 3.421 | 4.852 |
| *griddim* = 8 | 5.055 | 6.649 | 6.420 | 8.478 |
| *griddim* = 16 | 9.207 | 10.393 | 13.228 | 16.627 |
| *griddim* = 32 | 21.061 | 24.785 | 31.345 | 35.134 |

tion.

Lastly, we feel a little suspicious about two results:

- **Block synchronization seems faster than warp level synchronization in V100.** We tend to believe that a higher level of synchronization requires more resources and thus slower.
- **Coalesced group synchronization seems not work in P100.** It is not so possible that a synchronization instruction lasts only 1 cycle.

## 5. Conclusion

In this research, we present our measurement techniques and micro-benchmarks for the synchronization primitives in Nvidia V100 and P100 GPUs.

The technique we proposed can not only measure the latency of basic instructions in high accuracy, but also measure the latency of synchronization instructions that require the involvement of several different SMs. We believe this method will be useful for measuring and analyzing the performance of GPU-like architectures.

We use this measurement to analyze the performance of both V100 and P100 GPUs, which we think can represent the up coming trend of new Nvidia GPUs. Table 6, Table 7, Table 8 and Table 9 summarize the results of our experiments. Table 10 shows the lessons we can learn from these experiments. In general, controlling the active warp per SM to a reasonable range can improve performance when using synchronization methods. Considering the possibility of achieving higher performance with lower occupancy [11], future trend in CUDA programming might be using smaller griddim and blockdim to control the total active warp per SM.

**Table 9** Summarize of latency (us) results in 2 GPU, multi-grid sync

| V100 | | | P100 | | |
|---|---|---|---|---|---|
| Case | Min | Max | Case | Min | Max |
| *block/sm* <= 4 *and* *warp/sm* <= 16 | 6.02 | 9.12 | *block/sm* <= 2 *and* *warp/sm* <= 32 | 7.26 | 9.08 |
| *in between* | 11.39 | 16.40 | *in between* | 10.73 | 20.96 |
| *block/sm* >= 16 *or* *warp/sm* = 64 | 22.00 | 58.32 | *block/sm* > 8 | 33.18 | 74.89 |

**Table 10** Lessons learned

| | |
|---|---|
| **Block Sync** | Controlling the total warp/sm controls the performance of block sync |
| **Grid Sync** | Controlling the total block/sm controls the performance of gird sync |
| **Multi-Grid Sync** | Both block/sm and active warp/sm influence the performance of multi-grid sync |

## 6. Future Work

Previous researchers developed software device-wide barriers [4], [5]. The comparison of these barriers and the barriers provided by CUDA might be our future work.

And we want to understand the surprising results we mentioned in section 4.4. We believe some interesting new code implementation method can be proposed if the block synchronization is faster than warp level synchronization in V100.

Thirdly, as we mentioned in section 4.3.2, seems that the implementation of multi-grid synchronization does not take the network structure into consideration, the optimization of this synchronization method can be our future work.

Lastly, we believe grid level synchronization can make multi-GPU programming more productive. We plan to study the penalty or performance advance of using this method over other counterparts in the near future.

### References

[1] C. Nvidia, "Programming guide," 2019.
[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, pp. 1–6, 2015.
[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, pp. 39–58, ACM, 2016.
[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)*, pp. 1–12, IEEE, 2010.
[6] 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)*, pp. 235–246, IEEE, 2010.
[7] 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.
[8] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.
[9] T. NVIDIA, "V100 gpu architecture," 2017.
[10] J. D. Little and S. C. Graves, "Little's law," in *Building intuition*, pp. 81–100, Springer, 2008.
[11] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU technology conference, GTC*, vol. 10, p. 16, San Jose, CA, 2010.