Understanding the Overheads of Launching CUDA Kernels

Lingqi Zhang[1], Mohamed Wahib[2], Satoshi Matsuoka[1 3]
zhang.l.ai@m.titech.ac.jp
[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

1 INTRODUCTION

GPU computing is becoming more and more important in the field of general computing. Many scientific areas utilize the performance of GPUs. Several classes of algorithms require device-wide synchronization, through the use of barriers. However, thousands of threads running on independent SMs (Streaming Multi-Processors) impede this task. Previous research [3] proposed two kinds of device-wide barriers: software barriers or implicit barriers. Recently, Nvidia proposed new methods to do device-wide barriers, i.e. grid synchronization and multi-grid synchronization [1]. Based on the possibility of achieving high performance from lower occupancy [2], we envision using a single kernel with several barriers instead of using multiple kernels as an implicit barrier. But we need to understand the penalty of using different kinds of barriers, i.e. new explicit barrier functions and implicit barrier.

Additionally, Nvidia has proposed new launch functions (e.g. cooperative launch and multi-cooperative launch). These functions are used to support grid synchronization and multi-grid synchronizations [1], i.e. the new explicit barrier functions. In order to utilize the new features, programmers need to turn to the new launch functions. But there is no research try to study the penalty of turning into these new launch functions.

In this research we will use micro-benchmark to understand the overheads hidden in launch functions. And try to identify the cases when it is not profitable to launch additional kernels. We will also try to make a better understanding of differences in the different launch functions in CUDA.

2 MICRO-BENCHMARKS USED IN OUR STUDY

Throughout this abstract, we use the following terminologies:

- **Kernel Latency:** Total latency to run kernels, start from CPU thread launching a thread, end at CPU thread noticing that the kernel is finished.
- **Kernel Overhead:** Latency that is not related to kernel execution.
- **Additional Latency:** Considering that CPU thread has just called a kernel launch function, additional latency is the additional latency to launch an additional kernel.
- **CPU Launch Overhead:** Latency of CPU calling a launch function.
- **Small Kernel:** Kernel execution time is not the main reason for additional latency.
- **Larger Kernel:** Kernel execution time is the main reason for additional latency.

Currently, researchers tend to either use the execution time of empty kernels or the execution time of a CPU kernel launch function as an overhead of launching a kernel. Although those methods might work correctly when considering a single GPU kernel, this is not enough in the case of multi-kernels. Under this circumstance, we mainly focus on the overhead for launching an additional Kernel.

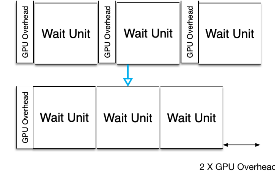

Figure 1: Using kernel fusion to test the execution overhead

We use the sleep instruction to control the kernel latency. Sleep instruction is only available in Volta architecture. This instruction is very light, and according to our experiments, no matter how many times we repeat this instruction, the overhead of the kernel remains unaffected.

We use several sleep instructions to compose a wait unit. Different wait unit inside a single kernel represent a valid kernel execution latency.

This micro-benchmark consist of two different kinds of variable:

- The times to launch a kernel
- the numbers of wait units inside a single kernel. In a single experiment, wait unit should be settled.

To test the overhead of small kernels, we propose to use a null kernel (no code inside) as an example of a small kernel. In this situation, the overhead can be computed with the formula 1

$$O = \frac{Latency_{i0} - Latency_{j0}}{i - j} \quad (1)$$

$*(O \textit{ represents Overhead};\ i,\ j \textit{ represents call launch function times};$
$*0 \textit{ represents 0 wait unit inside a kernel})$

To test the overhead of a large kernel, we propose to use kernel fusion to unveil the overhead hidden in kernel latency. The details of this method is shown in Figure 1. In this situation, the overhead can be computed with the formula 2

$$O = \frac{Latency_{ij} - Latency_{ji}}{i - j} \quad (2)$$

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

Table 1: Environment Information

Platform	Driver	CUDA
DGX1	410.104	V10.0.130

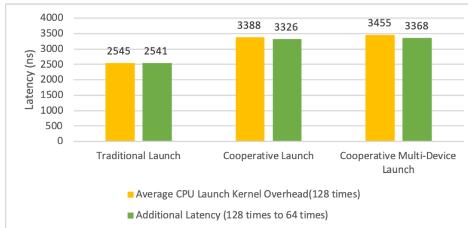

Figure 2: Comparison of null kernel overhead for different launch functions

3 OVERHEAD OF LAUNCHING KERNELS

3.1 Experimental Environment

Since we utilize the sleep instruction as a tool to analyze launch overhead, which is only available in Volta Platform in CUDA, we only conduct experiments in the V100 GPU. Table 1 shows the environment information. Each result presented is the average result of 100 experiments.

3.2 Launch Overhead in Small Kernels

We found that latency of CPU Launch Overhead to be nearly equal to the latency of the additional kernel. We hereby additionally plot the latency of the launch function in Figure 2.

Considering the system error, it is relatively safe to assume that the time consumed when the CPU launches a kernel is the main source of latency among all other steps in kernel launch.

3.3 Launch Overhead in Large Kernels

In a single node, we use 5 workload units (sleep 5000 ns). Figure 3 shows that the additional latency is larger than the CPU launch overhead, which means CPU launch overhead do not influence additional latency. And using the kernel fusion method, we found that the execution overhead does exist.

We only prove that this kind of overhead exists in this work. The relation between the execution overhead and how complex the kernel is as well as the launch parameters might is future work. In real-world workloads, the actual execution overhead might be larger than what we are reporting now.

3.4 Other Launch Overheads

We observe that apart from the overhead of CPU launching kernel and GPU execution overhead, there are remaining overheads.

We use formula 3 to compute that kind of overheads.

$$O_{Other} = O_{Total} - (O_{CPU\ Launch\ Kernel} + O_{Execution}) \quad (3)$$

$*(O\ represents\ Overhead;)$

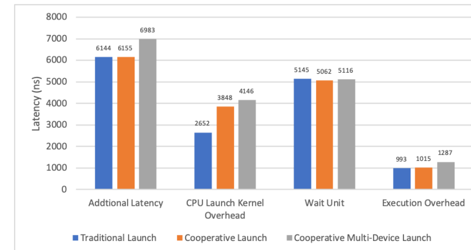

Figure 3: Large kernel launch overhead of different launch methods

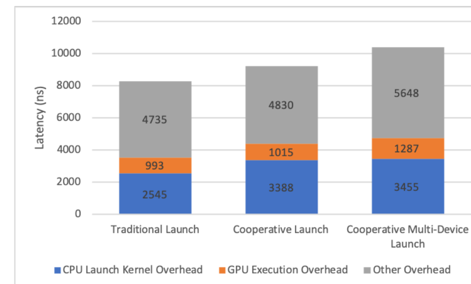

Figure 4: Comparison of different overheads in different launch functions

The result is shown in figure 4. Although the overheads seem large, it does not play an important role when launching a large number of kernels.

3.5 Conclusion

In this work, we use micro-benchmarks to analyze the launch overhead behaviours of different launch functions, in the case of both small kernels and large kernels. The result reveals two different kinds of kernel overheads and some unknown overhead only distinctive in the situation of a single kernel. The overhead of CPU launching kernel mainly has impacts in the situation of small kernels, while the execution overhead mainly has impacts in the situation of large kernels. We conclude that launching a new kernel is only profitable in the situation when the performance improvement surpasses the overhead of a new kernel. Additionally, we observed that Cooperative Multi-Device Launch is slightly slower than Cooperative Launch, and Cooperative Launch is slightly slower than Traditional Launch. This additional latency is trivial considering the benefit of using grid lebel synchronization. This research is mainly focused on the V100 GPUs in DGX1. But we also observe similar behaviors in P100 platform.

REFERENCES

[1] 2019. CUDA C Programming Guide. Retrieved June 3, 2019 from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[2] Vasily Volkov. 2010. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, Vol. 10. San Jose, CA, 16.
[3] Shucai Xiao and Wu-chun Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–12.