Understanding the Overheads of Launching CUDA Kernels

Motivation

- Nvidia GPUs can run 10,000s of threads on independent SMs (Streaming Multi-processors)
  - Not ideal for device-wide barriers
- Method for device-wide barriers in GPUs
  - Software barriers (example in [1])
  - Implicit barriers: launching separate kernels (impacts performance)
- Alternative ways to achieve the same goal
  - Grid synchronization or multi-grid synchronization [2]
  - Higher performance might come from lower occupancy [3]
- Implicit barrier (additional kernels) vs. single kernel
- **Question:**
  - When not to launch an additional kernel?
  - What is the penalty of using different kinds of barriers in CUDA?

Background

- Different kinds of kernel launch methods.
  - **Traditional Launch**
  - **Cooperative Launch** (CUDA 9)
    Introduced to support grid synchronization
  - **Cooperative Multi-Device Launch** (CUDA 9)
    Introduced to support multi-grid synchronization
- Sleep instruction: wait specific nanosecond in GPU kernel.

Micro-benchmark

- Definition
  - **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 have 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.

```
__global__ void null_kernel_DEP()
{
    repeat10(asm volatile("nanosleep.u32 1000;"););
}
...
//example of launchfunction: traditional, cooperative, multi_device_cooperative
start(timer);
repeat5(launchfunction(null_kernel_DEP,launchparameters));
cudaDeviceSynchronize();
stop(timer);
```

Latency that related to kernel execution here is 10 us (wait unit 1000 ns here)
time=5X(CUP Launch Overhead)
CPU thread wait for GPU kernel
Kernel Latency

Figure 1: Sample code of micro-benchmark that call launch function 5 times, and repeats a wait unit (sleep 1000 ns) 10 times.

- Additional wait unit (sleep 1000 ns) do not increase any kernel overhead (Considering System Error)

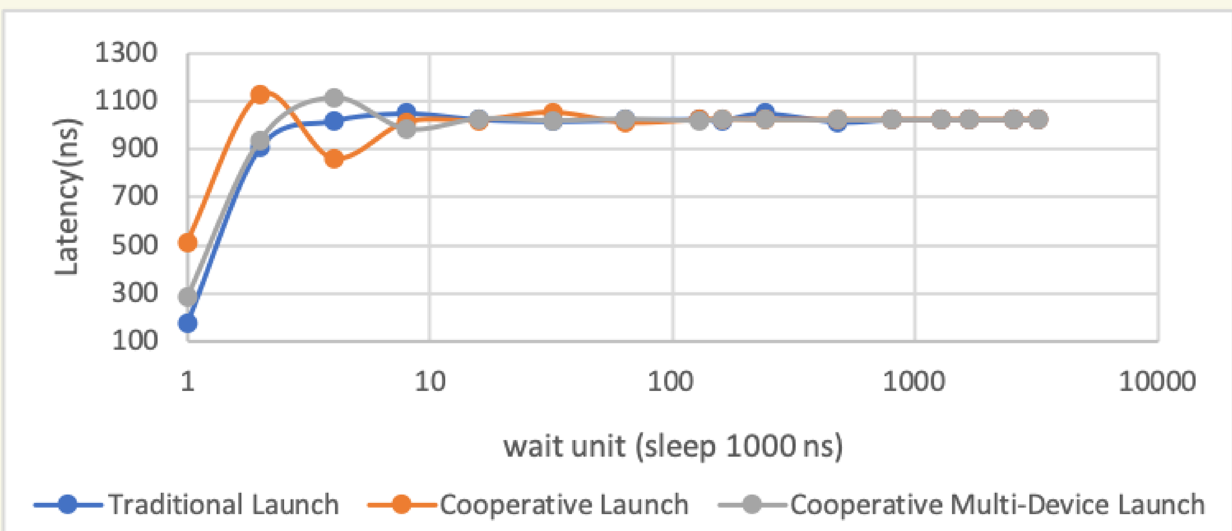

Figure 2: Gradient of latency per wait unit (sleep 1000 ns) in a single kernel

- Test overhead in small kernels
  Method: Using null kernel (no code inside) to represent a Small Kernel
- Test overhead in large kernels
  Method: Using kernel fusion to unveil the overhead.

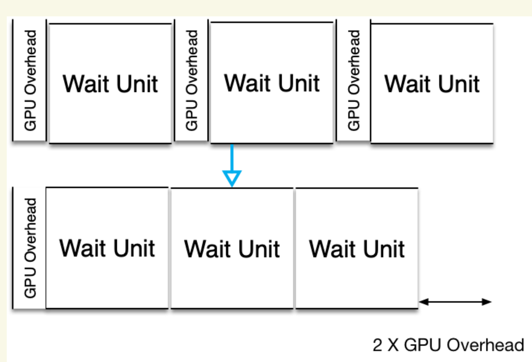

Figure 3: Using kernel fusion to test overhead hidden in kernel execution

Launch Overhead in Small Kernels

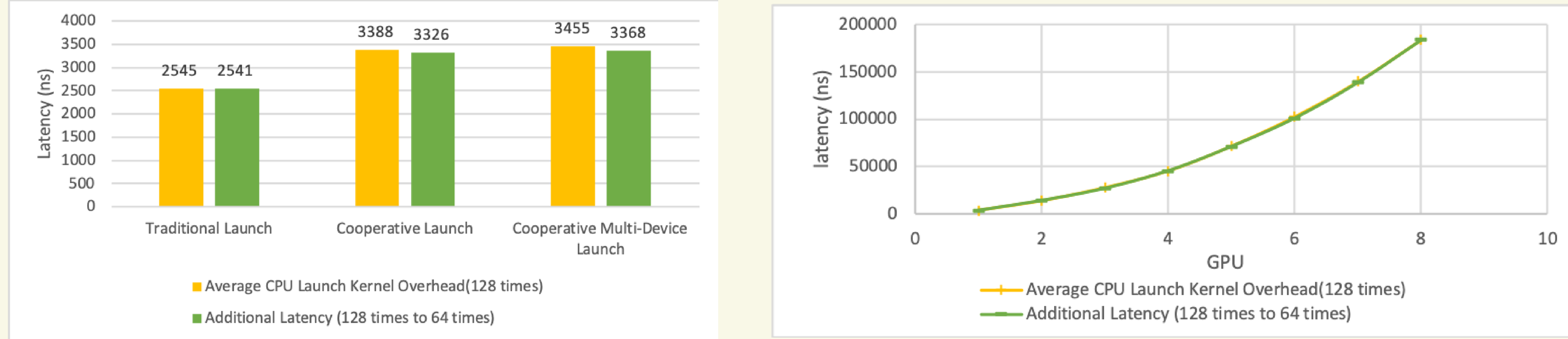

Figure 4: Comparison of null kernel overhead using three different launch functions that employ different types of barriers (left) , Cooperative Multi-Device Launch among different devices (right).

- **CPU Launch Overhead** is the main overhead in Small Kernel.

Launch Overhead in Large Kernels

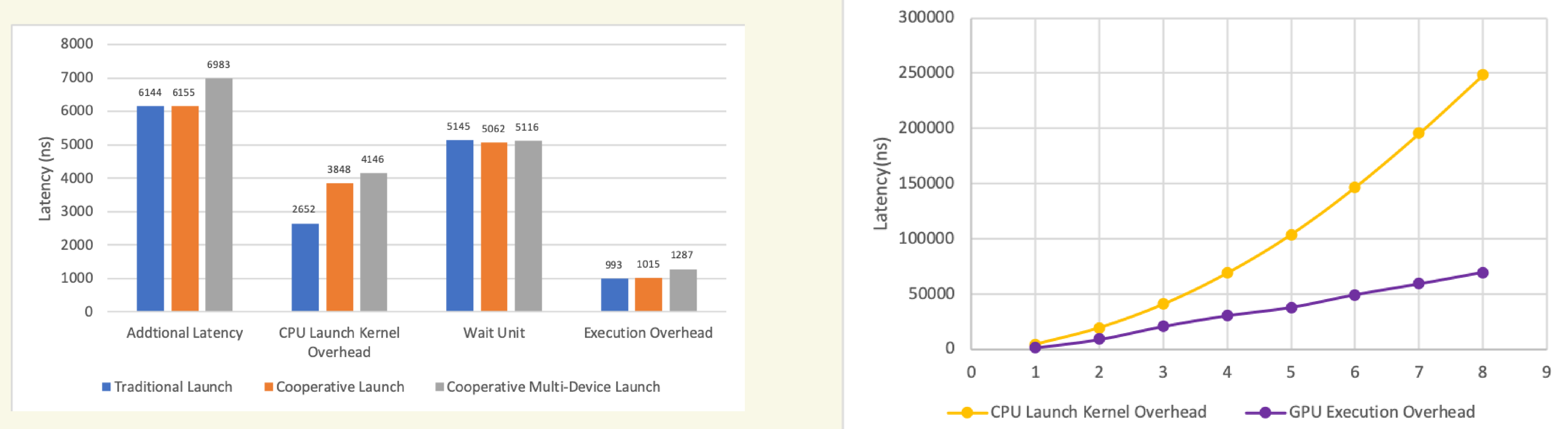

Figure 5: Comparison of Large Kernel Overhead among different launch functions (left), Cooperative Multi-Device Launch among different devices (right).

- CPU launch overhead is recorded to prove that it is not distinctive here. (the result is not as precise as the one in "Small Kernel" section)
- GPU execution overhead does exist.

Other Overheads

- Empty kernel lasts about 8 us, still longer than the overheads we reported.

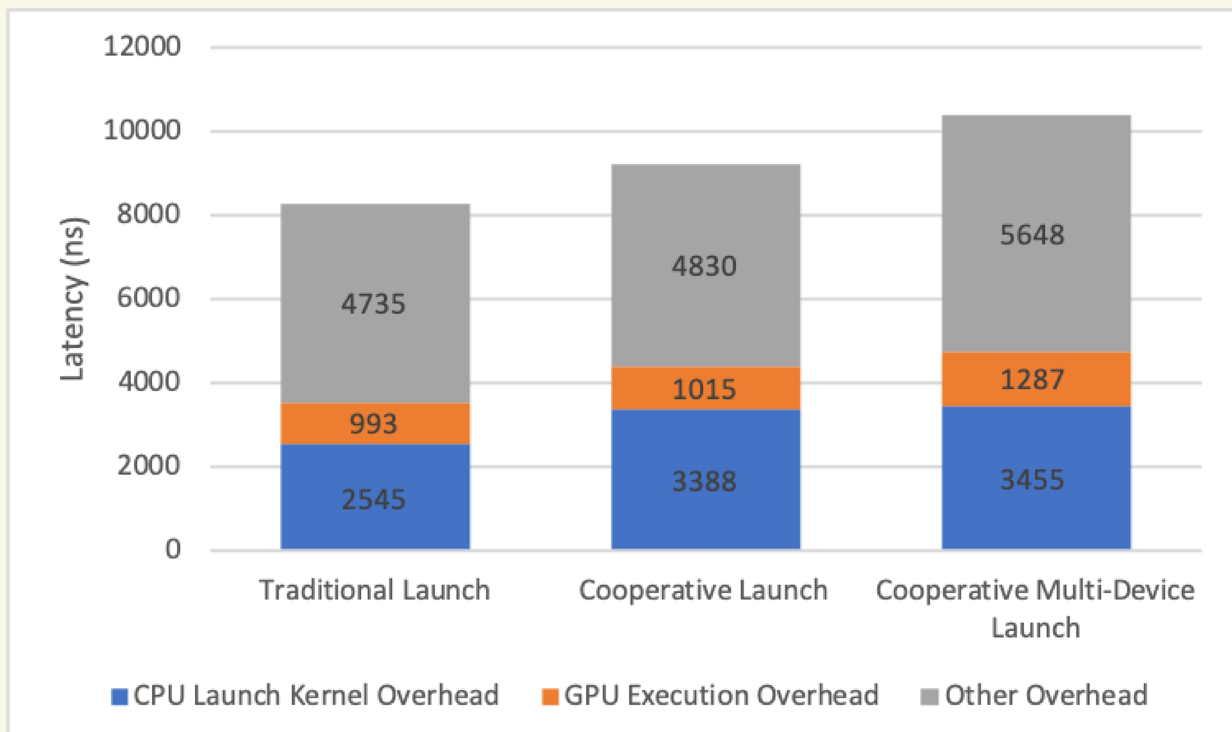

Figure 6: Comparison of different overheads in different launch functions

- **Other Overhead** is distinctive in single kernel. (Larger than the two kinds of overhead we reported)

Conclusion

- Main overheads:
  - **Small Kernels:** CPU Launch Overhead
  - **Large Kernels:** GPU Execution Overhead
  - **Single Kernel:** Other Overhead
- Overhead of different launch functions
  - **Cooperative Multi-Device Launch > Cooperative Launch > Traditional Launch**
- Launch a new kernel when the performance improvement surpasses the overhead of a new kernel.

References

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

Cuda c programming guide, May 2019.

Vasily Volkov.
Better performance at lower occupancy.
In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA, 2010.

Lingqi Zhang[1], Mohamed Wahib[2], Satoshi Matsuoka[1 3]
zhang.l.ai@m.titech.ac.jp, mohamed.attia@aist.go.jp, matsu@is.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