Multithreaded Double Queuing for Balanced CPU-GPU Memory Copying*

ABSTRACT

Memory transfers between CPU host and GPU devices are known to be heavy overhead for GPU applications. On GPU applications with large data inputs, memory transfers frequently take orders of magnitude larger times than the kernel execution times[5]. Current implementation of CPU-GPU memory copy is throttling back the performance of memory transfer. The memory bandwidth difference between CPU memory and GPU memory is not appropriately considered in CUDA library. In this work, we propose a multithreaded memory copy technique with double queuing to fully utilize PCIe bandwidth to GPU memory while transferring data between host CPU and GPU devices. This technique allows different devices with disparate bandwidths to work in balance, doubling the memory transfer rate of current CUDA implementation of *cudaMemcpy()*.

CCS CONCEPTS

• **Computing methodologies** → *Graphics processors*; • **Software and its engineering** → *Main memory*; *Multithreading*; Buffering;

KEYWORDS

Memory Copying, Multithreading, Double Queueing, GPU Computing

ACM Reference Format:

. 2019. Multithreaded Double Queuing for, Balanced CPU-GPU Memory Copying. In *Proceedings of ACM SAC Conference (SAC'19).* ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/xx.xxx/xxx_x

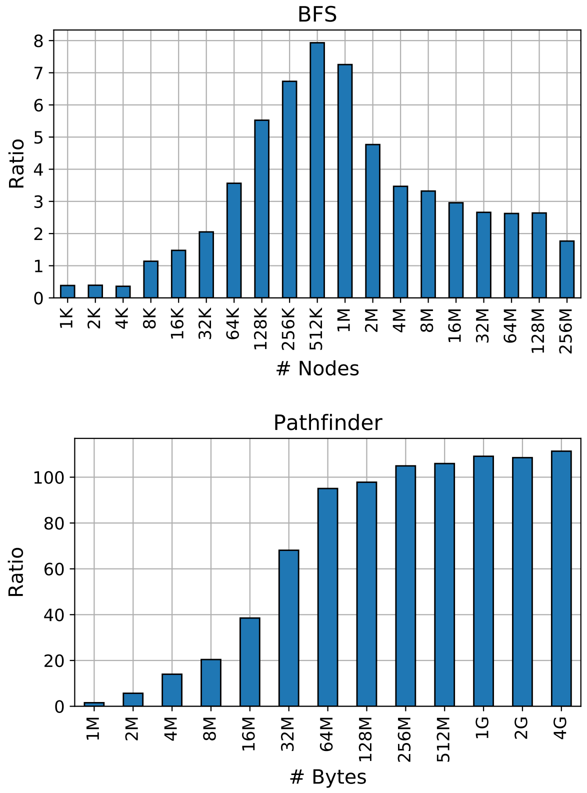


Figure 1: Memory Transfer Time/Kernel Execution Time

1 INTRODUCTION

While memory transfer overhead is a well-known issue in the field of GPGPU computing, memory transfers were considered as hardware vendor's subjects rather than software's. Researchers have focused on improving kernel computation rather than actual memory transfer rate between host and devices. Software bottleneck is being an obstacle in inter memory transfer between host and devices unlike conventional hardware bottleneck problem[6]. CUDA adopted a double buffering system for memory transfer in NVIDIA GPUs. This double buffering technique involves memory pinning of 1MB buffers each for stable remote DMA[11]. Pinned memory differs from memory locked memory in Linux as memory locked memory may migrate in memory, pinned memory is fixed to a physical address. This scheme is used universally in every system using NVIDIA GPU.

We have extracted the CUDA driver and discovered that NVIDIA pins memory by using *os_lock_user_pages()*. CUDA driver calls this function twice at its launch phase which is aimed to pin buffers for memory transfer to GPU device. Buffers' virtual addresses are always fixed, stipulated by CUDA. Thus the driver neglects system configuration of host memory and PCIe and deploys monochromatic scheme for memory transfer between host and devices. This double buffering scheme has stayed the same since the beginning of CUDA. Nevertheless, the fact that buffers are implemented in pinned memory indicate that the buffers be allocated at the fixed location all the time, CUDA does not guarantee the buffers to be located at a constant physical address. CUDA driver utilizes *kmalloc()* or *vmalloc()* for memory allocation of buffers and calls *NV_GET_USER_PAGES()* which eventually invokes a kernel function *get_user_pages()* to pin the allocated memory[2].

Most systems have a mismatch between PCIe bandwidth and host memory bandwidth in real memory copy transfer rate. Inter

*First author and second author contribute equally to this work.

memory transfer between host and devices are hampered by the disparateness of devices' bandwidth[3]. CUDA's negligent approach of universal scheme on every system needs to be changed. The recent explosive epidemic of new applications in the areas of artificial intelligence and augmented reality are driving stricter requirements around memory performance along with GPU computing. Following the trend, over the past decade, hardware vendors reserved enhanced memory products a prominent place in graphics involving technologies such as industry standard GDDR5. Furthermore, hardware vendors have been introducing new PCIe standards to the market nearly triennial interval. Frequent hardware enhancements cause fragmented memory bandwidth within a system. Despite this fragmented memory bandwidth, CUDA insists on static memory transfer technique in its *cudaMemcpy()*. Even though double buffering is used to fill the PCIe link full, there are gaping holes in between transferring as memory copy in host memory cannot follow up with PCIe memory copy. We propose Multithreaded Memory Copy with Double Queuing to prevent host memory copy rate from falling out with PCIe memory transfer rate, promising the link to be full while transferring. With this technique, no matter what system configuration one has and how the system changes over time, *cudaMemcpy()* will fully utilize PCIe bandwidth.

We have tested our implementation of *cudaMemcpy()* by library-interpositioning on Rodinia BFS and Pathfinder as these two are malleable with dataset size that we can monitor how memory transfer rate differs over different data sizes. We have observed about 250% improvement in memory transfer rate with a large data set. Technical background, details of Multithreaded Double Queuing, experiment result, and conclusion constitute the rest of this paper.

2 BACKGROUND

2.1 Pinned Memory

Pinned memory is the page-locked memory that does not cause a soft page fault[7]. Pinned memory is allocated and freed with specific functions implemented by CUDA. These functions also map allocated page-locked memory for Direct Memory Access(DMA) by devices[11]. The host operating system uses the kernel function *get_user_pages()* for pinned memory in Linux[2]. CUDA keeps track of pinned memory of itself and accelerates memory copies that involve host addresses pointing at pinned memory[11]. PCIe does not need to wait for memory to be retrieved from secondary memory as memory pinning prevents hard and soft page faults. Furthermore, devices can remotely perform DMA[11].

However, allocating pinned memory is expensive because it involves a considerable amount of work for the host operating system[11]. What's more, pinned memory reduces the amount of physical memory available to other processes[7].

2.2 Double Buffering

Because the device operates autonomously and access the host memory without the host operating system, only pinned memory enables asynchronous memory copy between hosts and devices. The device cannot access pageable memory directly, so the driver implements pageable memory copy with a pair of pinned buffers that are allocated with the CUDA context. To perform a host to device memory copy, the driver first copies the data of the host to

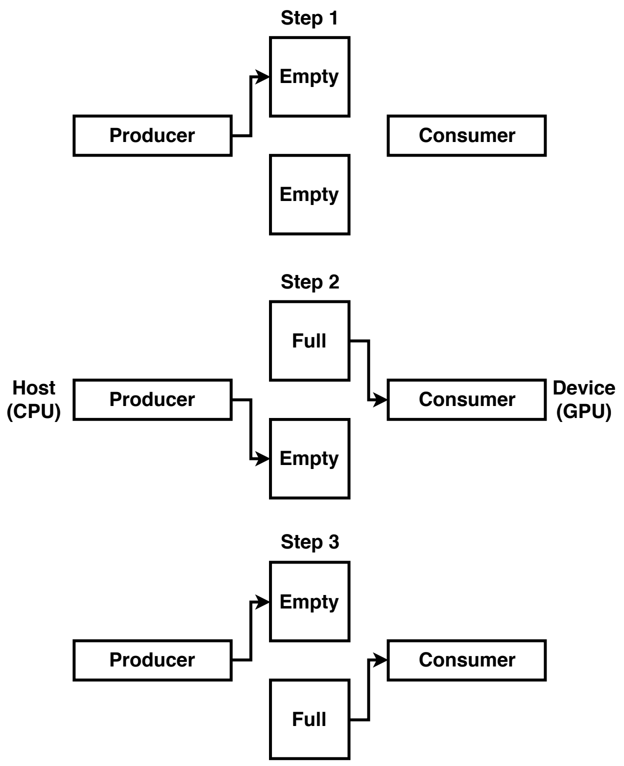


Figure 2: Logic of Double Buffering

one pinned buffer(Step 1 in Figure 2), then sends the request of a DMA operation to read that data with the device. While the device starts processing that request, the driver copies another data in the other pinned buffer(Step 2 in Figure 2). The host keeps flip-flopping with the device between pinned buffers, with synchronization, until the device performs the last memory copy(Step 3 in Figure 2)[11]. The host also naturally pages in any cached pages while the data is being copied.

But this method causes relatively slow data transfer rate if the speed of memory copy from the host to the buffer is slower than the speed of memory copy from the buffer to the device because there are significant delays among memory copies from the buffer to the device shown in Figure 3.

2.3 Write-Combined Memory

Write-combined memory or uncacheable write-combining memory was created to allow the host to write to device frame buffers more quickly while not polluting the host cache[1]. For CUDA, write-combined memory can be allocated by calling *cudaHostAlloc()* with *cudaHostWriteCombined* as a parameter[11]. Besides setting the page table entries to bypass the host caches, write-combined memory is ensured not to be snooped during PCIe bus transfers[1].

However reading write-combined memory from the host is 6x slower than regular memory. This deteriorates the condition as disparity between CPU memory bandwidth and PCIe bandwidth gets wider. Such memory copy slow down can be circumvented by using *MOVNTDQA* instruction that is brand new with *SSE4.1*[11].

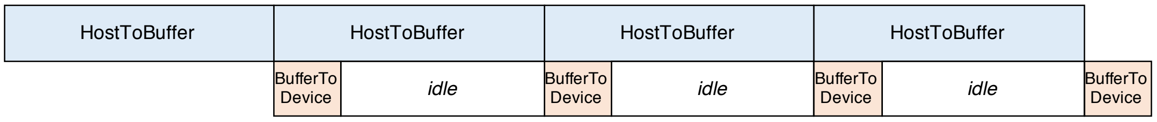


Figure 3: Timeline of Memory Transfers in Double Buffering

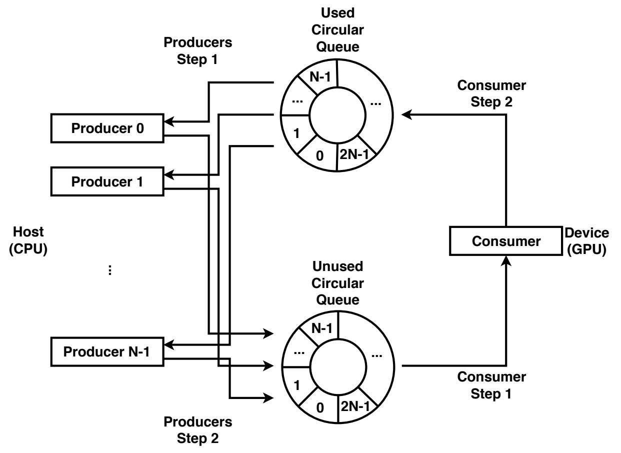


Figure 4: Multithreaded Double Queuing: N producers and 1 consumer

NVIDIA implements memory copy with *MOVNIDQA* instruction to accelerate memory copy from write-combined memory to regular memory for the device host transfer of *gdrcopy*(A low-latency GPU memory copy library based on NVIDIA GPUDirect RDMA technology)[10]. We used this function to implement device to host transfer due to the slow rate of standard memory copy from the buffer to the host.

3 MULTITHREADED DOUBLE QUEUING

Transferring data from the host to devices can be transposed to a producer-consumer problem[4]. Multiple producers copy the data of the host to the buffer concurrently by multithreading while one consumer copies the data of the buffer to the device. Numerous producers are required over one consumer to fill the chasm between the bandwidth of PCIe and the bandwidth of memory copy between the host and the buffer. The number of buffers is set twice the number of producers. All buffers are write-combined pinned memories to ensure cache checking time to be eliminated. Used circular queue contains addresses of empty buffers and unused circular queue contains addresses of full buffers and index of data. The size of the used and unused queue equates the number of buffers. Operating logic of double circular queuing method are shown in Figure 5 and Figure 6.

Multiple producers execute the steps of producers and one consumer executes the steps of a consumer as shown in Figure 4. Because producers copy the data of the host to the buffer concurrently and queues guarantee first in first out[9], the consumer can copy the data of the buffer to the device without any delay expressed in Figure 7.

```
while(true){

    lock(producer_index_lock)
    if(producer_index < #_of_data_copy){
        target_index = producer_index
        ++producer_index
        is_end = false
    }
    else
        is_end = true
    unlock(producer_index_lock)
    if(is_end == true)
        break

    // Producers Step 1
    // Get an empty buffer from used
       circular queue
    // Memory copy the data of the host to
       the empty buffer
    semaphore_wait(used_queue_full)
    lock(used_queue_lock)
    target_buffer = get_from_used_queue()
    unlock(used_queue_lock)
    semaphore_post(used_queue_empty)
    memcpy(target_buffer,
        host_src + target_index *
            buffer_size,
        buffer_size)

    // Producers Step 2
    // Put the full buffer from Step 1 in
       unused circular queue
    semaphore_wait(unused_queue_empty)
    lock(unused_queue_lock)
    put_in_unused_queue(target_buffer,
       target_index)
    unlock(unused_queue_lock)
    semaphore_post(unused_queue_full)

}
```

Figure 5: Algorithm of Producer

There are two conditions expressed in Figure 8 to reach maximum utilization level of PCIe bandwidth. S_{B2D} is the speed of copying data of buffer to device. S_{H2B} is the speed of copying data of host to buffer. N_P is the number of producers. B_{MEM} is the maximum bandwidth of host's main memory. The reason of second

```
for(i = 0; i < #_of_data_copy; i++){

    // Consumer Step 1
    // Get a full buffer with the index of
        the data that is in the full buffer
    // from unused circular queue
    // Memory copy the data of the  buffer
        to the device using the index
    semaphore_wait(unused_queue_full)
    lock(unused_queue_lock)
    target_buffer , target_index =
        get_from_unused_queue()
    unlock(unused_queue_lock)
    semaphore_post(unused_queue_empty)
    memcpy(device_dst + target_index *
        buffer_size ,
        target_buffer ,
        buffer_size)

    // Consumer Step 2
    // Put the empty buffer from Step 1 in
        used circular queue
    semaphore_wait(used_queue_empty)
    lock(used_queue_lock)
    put_in_used_queue(target_buffer)
    unlock(used_queue_lock)
    semaphore_post(used_queue_full)

}
```

Figure 6: Algorithm of Consumer

condition is that copying data of buffer to device shares the bandwidth of host's main memory with copying data of host to buffer. These two conditions are also applied to double buffering.

In case of transferring data from the device to the host, the same algorithm is applied, but the speed of copying data of the buffer to the host is slower than the speed of normal memory copy even when using SSE4.1 memory copy so the number of producers should be increased.

4 EXPERIMENTAL RESULT

To evaluate our technique, we used Intel server with two CPU nodes, each of which has 16 cores (32 hardware threads) Table 1. For CUDA applications, an NVIDIA Titan V GPU is equipped on the server. NUMA hinders memory copy to perform its ordinary speed. We have observed that NUMA machine on our server slows down the memory copy speed nearly half than UMA machines[8]. Multithreaded Double Queuing threads are pinned on same CPU node to ensure all memory used to be on the same NUMA node. BFS and Pathfinder from Rodinia are two benchmarks that are apt to test with various input data sizes. No matter on which benchmark Multithreaded Double Queuing is tested, memory transfer decrease rate show about the same with same dataset size. Thus, it is better to show how transfer rate differs by different dataset sizes within same application. In addition, active state power management (ASPM)

Table 1: Specification of Experiment Environment

Category	Specification
CPU	Intel Xeon Silver 4110 2.10GHz 16 Threads * 2 Sockets
RAM	8 * 8GB DDR4-2400 Triple Channels
GPU	1 * TITAN V
NUMA	Available: 2 Nodes (0-1) Node 0 CPUs : 0 1 2 3 4 5 6 7 16 17 18 19 29 21 22 23 Node 0 Size : 64065 MB Node 1 CPUs : 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31 Node 1 Size : 64480MB Node Distances : 0-0, 1-1: 10 0-1, 1-0: 21
CUDA	9.0
Driver	390.87
Benchmark	Rodinia 3.1
Dataset	BFS : # of nodes Pathfinder : # of bytes Both use max runnable size
Extra	numactl –membind=0 –cpubind=0 aspm off intel_idle.max_cstate=0

protocol is set to off to maximize the PCIe link performance without increasing its latency.

4.1 Memory Transfer Times

The actual transfer time measured by *nvprof* were impressively increased with a sufficiently large dataset. In BFS datasets below 32K show slower transfer time than the current CUDA transfer time Figure 9. This is because 16K nodes have 600KB of data transfer which is smaller than the 1MB buffer size. Our implementation imitates buffer with pinned memory allocated in a program. This is slightly slower than CUDA implementation of buffers. This is inevitable due to the nature of our implementation in user space. If we use existing *cudaMemcpy()* for our implementation for data smaller than 1MB instead then, this problem is solved. However, if Multithreaded Double Queuing is embedded in CUDA, then such performance degrading will not be a concern. Buffering system from current implementation will be used, promising the same performance in data under 1MB.

In BFS, from the dataset of 32K nodes and larger show performance increase in data transfer rates and converge to 270% performance increase. Figure 9 shows normalized memory transfer time compared to plain CUDA version. The rate converges to a specific value as the transfer rate reaches PCIe link bandwidth maximum. This increased rate will be different in other system configurations. Results from pathfinder clearly show that data transfer that involves data larger than 1MB always gets faster in our implementation of *cudaMemcpy()* Figure 10. The dataset that larger than 1MB involves

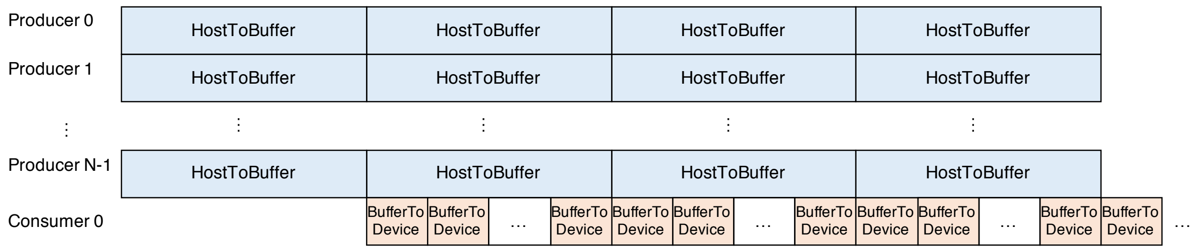


Figure 7: Timeline of Memory Transfers in Multithreaded Double Queuing

$$S_{B2D} <= N_P * S_{H2B}$$
$$S_{B2D} + 2 * N_P * S_{H2B} <= B_{MEM}$$

Figure 8: Two Conditions Determining The Number of Producers

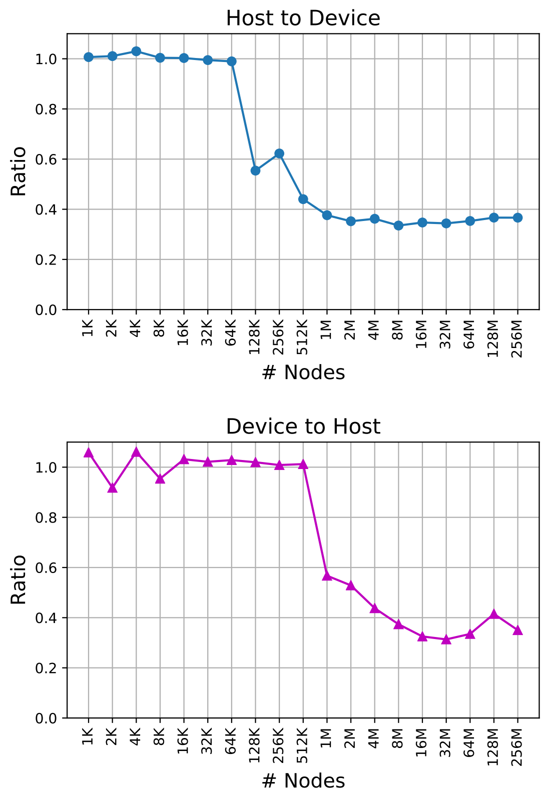


Figure 9: BFS Normalized Memory Transfer

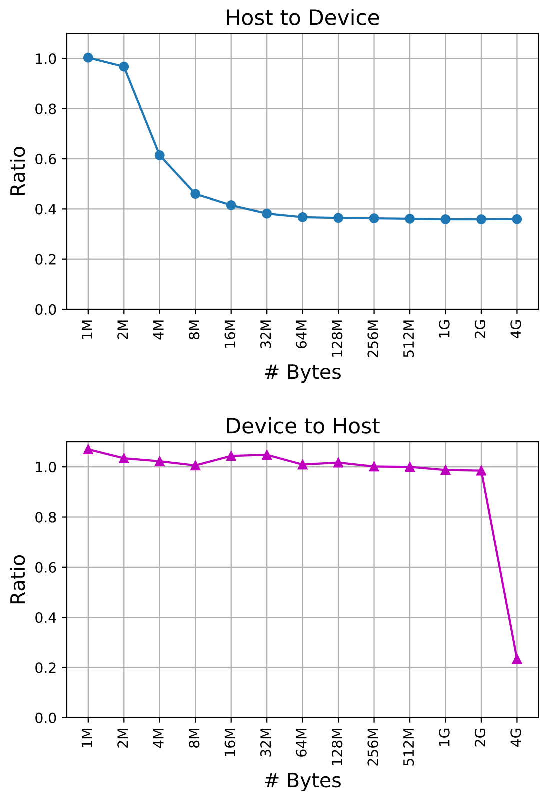
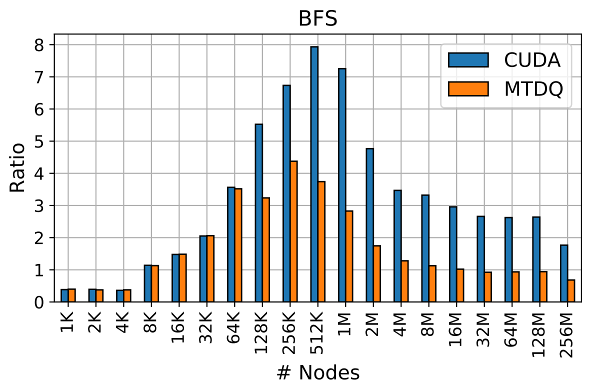


Figure 10: Pathfinder Normalized Memory Transfer

memory copy at least twice. This will make PCIe suffer starvation. Transfers from devices to host are usually relatively small than the host to devices.

Result in pathfinder device to host abruptly increases in a 4GB dataset as this is the point where data is sufficiently large enough to utilize Multithreaded Double Queueing effectively. The 4GB dataset in pathfinder moves 4MB of data from the device to host. As shown in host to device results, 4MB is where memory transfer performance gets striking. 2GB dataset has 2MB result which does not have much increase in inter CPU memory and GPU memory transfer performance according to host to device results. By the way, overall results in memory transfer in the device to host are unstable as they move relatively small data and *cudaMemcpy()* from device to host in CUDA is unstable. Memory transfer time by *cudaMemcpy()* from device to host fluctuates wildly which led to the experiment unstable results in device to host memory movement.



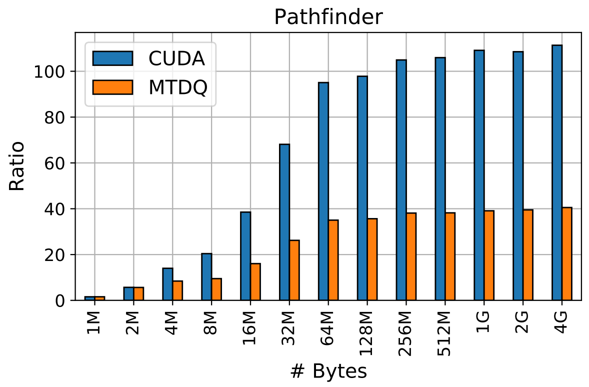


Figure 11: Memory Transfer Time/Kernel Execution Time of Plain CUDA and Multithreaded Double Queuing

4.2 GPU Activity Times

Memory transfer time/kernel execution time shows what portion memory transfer takes in GPU activities. Memory transfer and kernel execution comprise GPU activities. The time taken by memory transfer compared to kernel execution time is gigantic as shown in Figure 1. This portion is recalculated with Multithreaded Double Queueing and compared with plain CUDA version in Figure 11. The portion memory transfer takes dramatically reduces with Multithreaded Double Queuing. With a large dataset, the portion is halved compared to the plain benchmark portion. As more data are transferred over PCIe, more time is saved.

5 CONCLUSIONS

Multithreaded Double Queuing has been proven that it deserves to take place in CUDA to balance mismatching bandwidth between CPU memory and PCIe. Any data larger than 1MB creates a mismatch in CPU memory transfer time and PCIe transfer time. Even 2MB of data transfer causes a gaping hole in link utilization. Multithreaded Double Queueing can fill up the hole caused by the tiny dataset 2MB and lead to the performance increase. This improvement will be more prominent if it is embedded in CUDA as our implementation takes place in process level. Furthermore, runtime improvement gets conspicuous as more data are transferred. This improvement can extend further with a thread pool. In future usage of GPU related computation will continuously involve data transfer such as autonomous driving. Autonomous driving requires the constant participation of GPU computation acceleration. Multithreaded Double Queueing will save a tremendous amount of time with endless involvement of memory transfer in such applications. Moreover, such life-critical tasks demand ultra-low latency. Hardware vendors incessantly improve physical transfer rate. Underutilization of this enhancement in hardware is deplorable. Multithreaded Double Queueing ensures to make the most of flourishing PCIe bandwidth.

REFERENCES

[1] 1998. *Write Combining Memory Implementation Guidelines.* Technical Report. Intel Corp. Order number 244422-001.

[2] 2018. Developing a Linux Kernel Module using GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html

[3] Jaehyung Ahn, Dongup Kwon, Youngsok Kim, Mohammadamin Ajdari, Jaewon Lee, and Jangwoo Kim. 2015. DCS: a fast and scalable device-centric server architecture. In *Proceedings of the 48th International Symposium on Microarchitecture.* ACM, 559–571.

[4] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. 2014. *Operating systems: Three easy pieces.* Vol. 151. Arpaci-Dusseau Books Wisconsin.

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, and Kevin Skadron. 2008. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of parallel and distributed computing* 68, 10 (2008), 1370–1380.

[6] Jungsik Choi, Jiwon Kim, and Hwansoo Han. 2017. Efficient memory mapped file I/O for in-memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17).* USENIX Association.

[7] Jonathan Corbet. 2014. Locking and pinning. https://lwn.net/Articles/600502/

[8] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13).* ACM, New York, NY, USA, 381–394. https://doi.org/10.1145/2451116.2451157

[9] Donald Ervin Knuth. 1997. *The art of computer programming: sorting and searching.* Vol. 3. Pearson Education.

[10] NVIDIA. 2018. NVIDIA/gdrcopy. https://github.com/NVIDIA/gdrcopy

[11] Nicholas Wilt. 2013. *The cuda handbook: A comprehensive guide to gpu programming.* Pearson Education.