Multithreaded Double Queuing for Balanced CPU-GPU Memory Copying

Sanghun Cho*
Sungkyunkwan University
Suwon, Republic of Korea
shcho@arcs.skku.edu

Jaewan Hong*
Sungkyunkwan University
Suwon, Republic of Korea
jwhong@arcs.skku.edu

Jungsik Choi
Sungkyunkwan University
Suwon, Republic of Korea
jhchoi@arcs.skku.edu

Hwansoo Han†
Sungkyunkwan University
Suwon, Republic of Korea
hhan@skku.edu

ABSTRACT

Memory transfers between CPU host and GPU devices are known to impose heavy overhead on GPU applications. For GPU applications with large data inputs, memory transfers frequently take much longer than kernel execution time. For memory transfer mechanism for unpinned pages, current mechanism in CUDA library fails to achieve the maximum memory transfer potential. Memory bandwidth discrepancy between CPU and GPU are not properly considered in the current CUDA library. In our work, we propose a multithreaded memory copy technique that uses double queuing to fully utilize the PCIe bandwidth in GPU memory during the data transfers between host CPU and GPU device. Our technique doubles the memory transfer rate of current CUDA memory transfer in *cudaMemcpy()* by allowing multiple devices with different bandwidths to operate in a balanced manner.

CCS CONCEPTS

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

KEYWORDS

Memory Copying, Multithreading, Double Queuing, GPU Computing

ACM Reference Format:
Sanghun Cho, Jaewan Hong[1], Jungsik Choi, and Hwansoo Han. 2019. Multithreaded Double Queuing for, Balanced CPU-GPU Memory Copying. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3297280.3297426

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

1 INTRODUCTION

In computer architecture, *memory wall* [11] has been coined to describe the long struggle of speed discrepancy of CPUs with off-chip memories. Similar phenomenon appears, when GPU processes data on DRAM in host. As shown in Algorithm 1, a typical GPGPU program starts with preparing input data in host side memory, and CPU schedules a memory copy from host memory to device memory. Then, a kernel execution is launched on GPU, and results are copied back to CPU memory. GPU has to stay idle for the data to be completely brought in to GPU memory. Asynchronous execution is devised to hide this latency [5], but often times GPU execution outperforms the memory transfer speed from host memory. Data transfer time can overwhelm the kernel execution time. As shown in graphs of Figure 7, memory transfer takes 8 times longer than kernel execution in *BFS* with 512K nodes as an input, and 96 times longer in *Pathfinder* with 64MB as an input. This speed gap will be further exacerbated by faster GPUs, as recent GPU architectures execute a kernel more rapidly than older ones. Kernel execution times are reduced with advanced architectures with a large number of processor cores. As more advances are made to speed up GPUs, memory transfer overheads will be more critical to overall performance. While such memory transfer overhead is well-known in the field of GPGPU computing, memory transfers have been less focused by software optimization than actual kernel executions.

A typical optimization is double buffering adopted by CUDA for memory transfer in NVIDIA GPUs. Double buffering technique in CUDA involves memory pinning of 1MB buffers each for stable remote DMA [10]. Since this scheme is a typical mechanism in other RDMA systems, it is widely in every system which uses NVIDIA GPUs. Another scheme is to pin the pages that hold host data to transfer, instead of using pinned memory buffers. The later

Algorithm 1 Typical Sequence of CUDA Application

```
$initialize\_data(host\_data, device\_data)$
$*device\_data \leftarrow *host\_data$
// Memory Transfer Host to Device
$launch\_kernel(device\_data)$
$*host\_data \leftarrow *device\_data$
// Memory Transfer Device to Host
$host\_works(host\_data)$
```

scheme may require to pin too many pages for a large input data. Nonetheless, both schemes require pinned pages for DMA transfer from CPU to GPU. Pinned pages are different from memory-locked pages in Linux. Memory-locked pages may migrate in memory, whereas pinned pages are unmovable, fixed to physical addresses. Such memory-locked pages reduce the only available memory space on the system for other processes to use. Pinned pages, however, additionally hinder page allocator to efficiently manage memory space, as they are fixed to certain physical memory addresses. The first scheme with pinned buffers often takes longer than the second, since it requires memory copies from host data to buffers, but it can limit the number of pinned pages as many as the buffer size. This is one of the reasons why our technique for *cudaMemcpy()* is preferable over pinning all the pages and still increases the memory transfer rate.

We have extracted the CUDA driver and discovered that CUDA implementation 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. The virtual addresses of buffers are always fixed, stipulated by CUDA. Thus, the driver does not take account of system configuration of host memory and PCIe and deploys a simple scheme for memory transfer between host and devices. This double buffering scheme has been unchanged from the first version of CUDA. Nevertheless, the fact that buffers are implemented in pinned pages indicates that the buffers should be allocated at the fixed location all the time, but CUDA does not guarantee the buffers to be located at a fixed physical address. Thus, 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]. This implies that CUDA utilized Linux's strategy of pinning memory, which will inherit its problems of shrinking available memory of other processes and being an overhead in Linux memory management system [6].

Most systems have a mismatch between PCIe bandwidth and host memory bandwidth in real memory copy transfer rate. Inter-memory transfer between host and devices are hampered by the disparateness of device bandwidths [3]. Due to the imbalanced bandwidth between devices, the original CUDA scheme needs to be changed. The recent explosive growth of new applications in the areas of artificial intelligence and augmented reality are driving strict requirements on memory performance as well as GPU computing performance. 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 caused imbalanced memory bandwidths within a system. Despite this imbalanced memory bandwidth, CUDA library does not update its static memory transfer technique in *cudaMemcpy()*. Even though double buffering is used to fill the PCIe link full, there are gaping holes in between transferring, as memory copy speed in host memory cannot follow up with PCIe memory copy speed. Figure 1 illustrates how memory transfer is done in heterogeneous environment. The host memory and PCIe often does not work congruently. Copying memory to buffer and moving the data to the device usually have speed imbalance that forces one of the copying to stay idle.

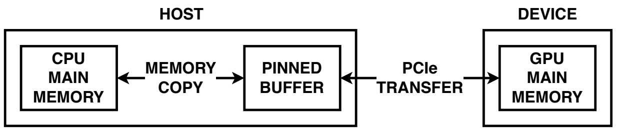

Figure 1: Buffer Model in CUDA

To overcome this cacophony in memory copying, we propose *multithreaded memory copy with double queuing*. This technique prevents 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 two Rodinia benchmark, BFS and Pathfinder, as these two are malleable with input size and 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[6]. These functions also map allocated page-locked memory for Direct Memory Access(DMA) by devices[10]. 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[10]. We have injected a code that leaves log in CUDA driver *NV_GET_USER_PAGES()* and checked that CUDA calls this function twice at initializing step which is for double buffering that is introduced in detail in next section. To make our implementation of *cudaMemcpy()* resemble the original implementation by NVIDIA, we also exploited pinned memory technique for our buffers. Using extant pinned buffer to transfer data is more preferable than pinning all the memory that will be transferred as pinning overhead can be reduced. However, this scheme brings another problem.

2.2 Double Buffering

The host keeps flip-flopping with the device between pinned buffers, with synchronization, until the device performs the last memory copy[10]. 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 2.

We extended this double buffering technique to Multithreaded Double Queuing that takes advantages of double buffering while complementing mismatches of different memory bandwidths.

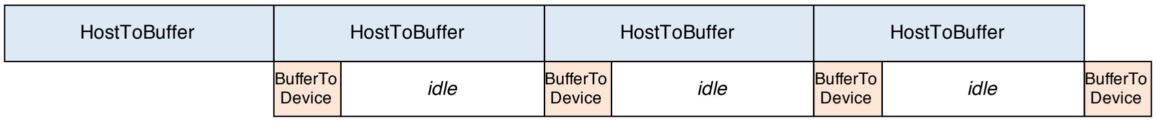

Figure 2: Timeline of Memory Transfers in Double Buffering

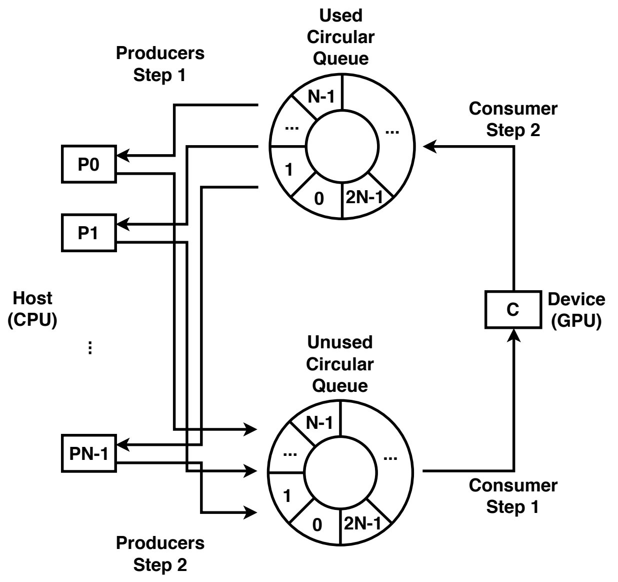

Figure 3: Multithreaded Double Queuing: N producers and 1 consumer

Algorithm 2 Algorithm of Producers

```
// $producer\_index$ is initialized as zero
// and shared by all producers
while $producer\_index < \#\_of\_data$ do
    $target\_index \leftarrow producer\_index$
    $producer\_index \leftarrow producer\_index + 1$
    // Producers Step 1
    $target\_buffer \leftarrow get\_from\_used\_queue()$
    $*target\_buffer \leftarrow *(src + target\_index \times buffer\_size)$
    // Producers Step 2
    $put\_in\_unused\_queue(target\_buffer, target\_index)$
end while
```

Algorithm 3 Algorithm of Consumer

```
for $i = 0$ to $\#\_of\_data$ do
    // Consumer Step 1
    $target\_buffer, target\_index \leftarrow get\_from\_unused\_queue()$
    $*(dst + target\_index \times buffer\_size) \leftarrow *target\_buffer$
    // Consumer Step 2
    $put\_in\_used\_queue(target\_buffer)$
end for
```

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[10]. 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*[10]. 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)[9]. 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 reason of using two circular queues is to manage empty buffers and full buffers separately. The size of the used and unused queue equates the number of buffers. Multiple producers execute the steps of producers and one consumer executes the steps of a consumer as shown in Figure 3.

Algorithm 2 describes the step of producers. The first step of producers is getting an empty buffer from used circular queue and filling the buffer with a data. The second step of producers is putting the full buffer from the first step with the index of the data in unused circular queue.

Algorithm 3 describes the step of a consumer. The first step of a consumer is getting a full buffer with the corresponding index from unused circular queue and copying the data of the buffer to the device using the index. The second step of a consumer is putting the empty buffer from the first step in used circular queue.

Because producers copy the data of the host to the buffer concurrently and queues guarantee first in first out[8], the consumer

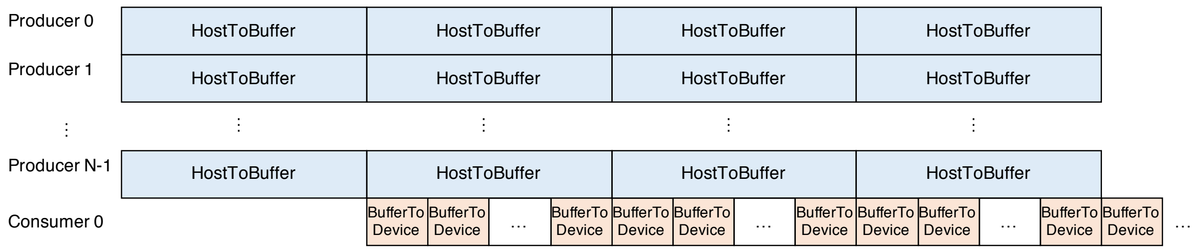

Figure 4: Timeline of Memory Transfers in Multithreaded Double Queuing

Table 1: Specification of Hardware Environment

	Specification
CPU	Intel Xeon Silver 4110 2.10GHz 16 Threads * 2 Sockets
RAM	8 * 8GB DDR4-2400 Triple Channels
GPU	1 * TITAN V
NUMA Configuration	numactl –membind=0 –cpubind=0
Power Management Configuration	aspm off intel_idle.max_cstate=0

Table 2: Specification of Software Environment

	Specification
CUDA	Library Version: 9.0 Driver Version: 390.87
Benchmark	Rodinia 3.1 BFS, Pathfinder

can copy the data of the buffer to the device without any delay expressed in Figure 4.

There are equation 1 and 2 to reach maximum utilization level of PCIe bandwidth.

$$S_{B2D} <= N_P * S_{H2B} \quad (1)$$

$$S_{B2D} + 2 * N_P * S_{H2B} <= B_{MEM} \quad (2)$$

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 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 did experiments under the specific conditions described in Table 1 and 2. "NUMA Configuration" in

Table 3: Kernel Execution Time (milliseconds) in Different Architectures

	BFS Kernel1	BFS Kernel2	Pathfinder Kernel
Maxwell	10.8032	1.12909	1.09822
Pascal	1.62578	0.321606	0.187015
Volta	0.568653	0.265402	0.094213

Table 1 is for using only one node of NUMA machine. The first line of "Power Management Configuration" is for setting Active State Power Management (ASPM) to off to maximize the PCIe link performance without increasing its latency and the second line is for preventing the performance decrease caused by power saving.

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[7]. 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 are two benchmarks that are apt to test with various input sizes. No matter on which benchmark Multithreaded Double Queuing is tested, memory transfer decrease rate show about the same with same input size. Thus, it is better to show how transfer rate differs by different input sizes within same application. Table 3 is the kernel execution time measured by nvprof in gtx-750ti, P100, and V100 GPUs. They were run on the same server with CUDA version 9.1. Kernel execution time doubles in Pascal over Volta's. Maxwell reported even 10times slower execution time than Pascal while the memory transfer time from host to device stayed almost the same over all three devices as they have the same CPU memory configuration, and PCIe link. This emphasizes the importance of our work. Current CUDA cannot utilize the full potential of advanced GPU.

4.1 Memory Transfer Times

The actual transfer time measured by *nvprof* were impressively increased with a sufficiently large input. In BFS inputs below 32K show slower transfer time than the current CUDA transfer time Figure 5. 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

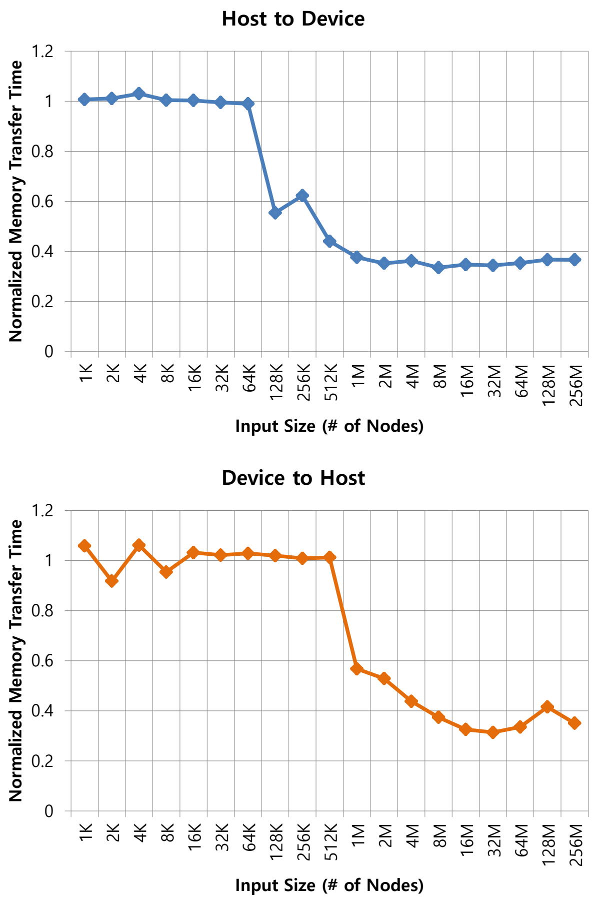

Figure 5: BFS Normalized Memory Transfer Time

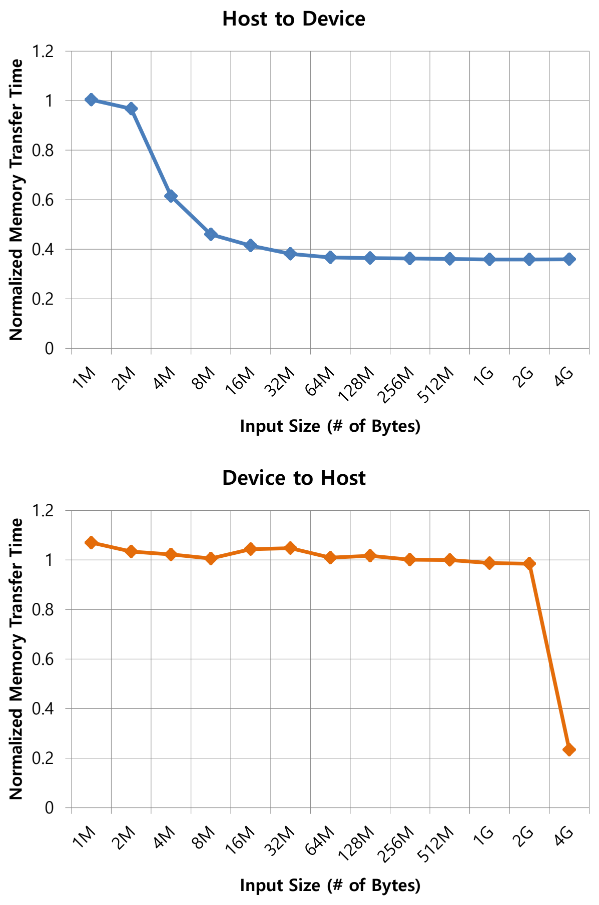

Figure 6: Pathfinder Normalized Memory Transfer Time

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 input of 32K nodes and larger show performance increase in data transfer rates and converge to 40% of CUDA . Figure 5 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 6. The input that larger than 1MB involves 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 input as this is the point where data is sufficiently large enough to utilize Multithreaded Double Queuing effectively. The 4GB input 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 input 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.

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 7. This portion is recalculated with Multithreaded Double Queuing and compared with plain CUDA version in Figure 8. The portion memory transfer takes dramatically reduces with Multithreaded Double Queuing. With a large input, the portion is halved compared to the plain benchmark portion. As more data are transferred over PCIe, more time is saved.

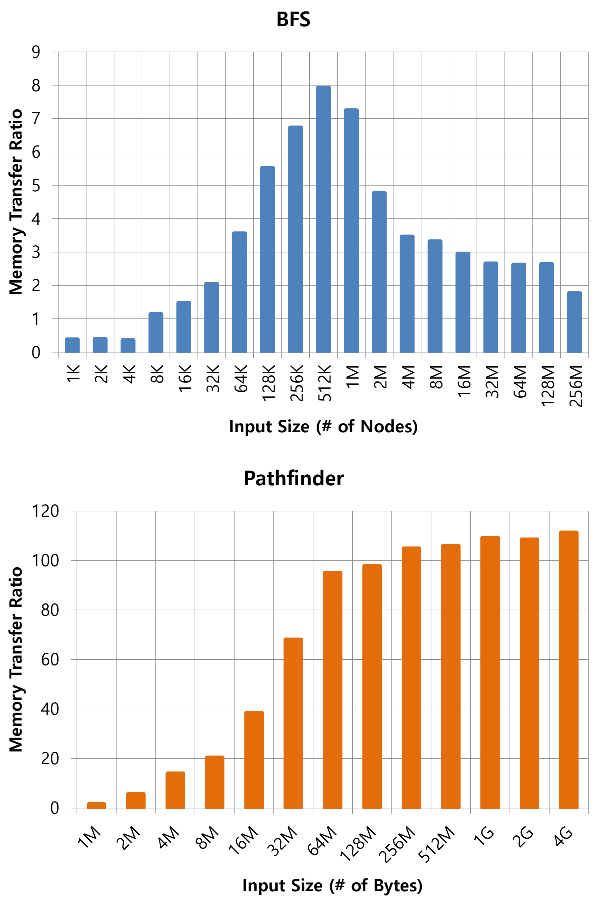

Figure 7: The Ratio of Memory Transfer Time to Kernel Execution Time

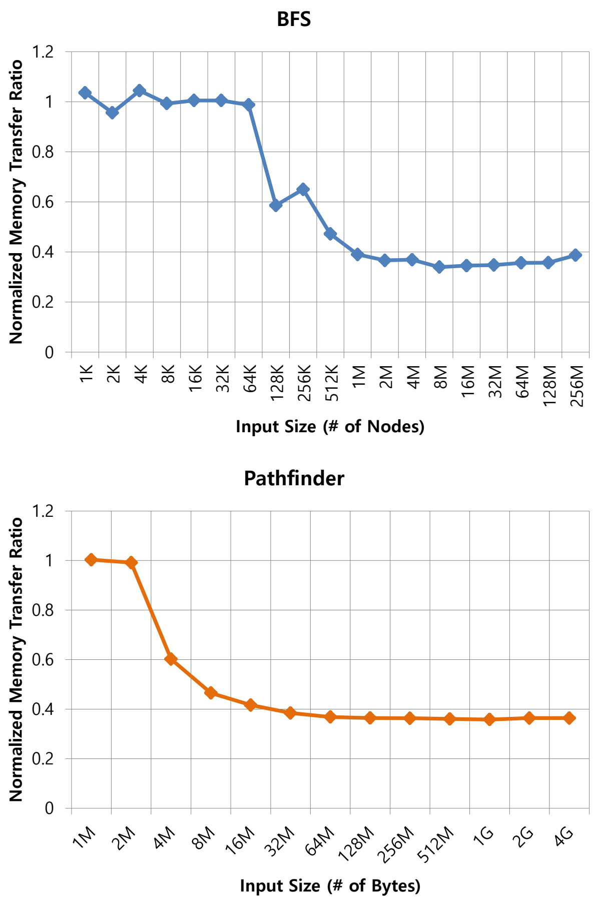

Figure 8: Normalized Memory Transfer Ratio

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 Queuing can fill up the hole caused by the tiny input 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 Queuing 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 lamenting. Multithreaded Double Queuing ensures to make the most of PCIe bandwidth. It will make a GPGPU program to reach the full potential of a GPU device on any system configuration.

ACKNOWLEDGMENTS

The work is supported by the Korea Ministry of Science and ICT under Grant NRF-2016M3C4A7952587 and IITP-2015-0-00914.

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] André R Brodtkorb, Trond R Hagen, and Martin L Sætra. 2013. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *J. Parallel and Distrib. Comput.* 73, 1 (2013), 4–13.
[6] Jonathan Corbet. 2014. Locking and pinning. https://lwn.net/Articles/600502/
[7] 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

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

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

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

[11] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.