분임3

연구작품/논문 최종보고서

2018 학년도 제 2 학기

제목 : Multithreaded Double Queuing for GPGPU

조상훈(2013314616), 홍재완(2013312784)
(※팀원 이름 모두 기재)

2018 년 11 월 07 일

지도교수: 한 환 수 서명

계획(10)	주제(20)	개념(20)	상세(30)	보고서(20)	총점(100)
10	20	20	28	20	98

* 지도교수가 평가점수 기재

■ 요약

CPU-GPU간 데이터 통신은 GPU를 활용하는 프로그램에서 상당한 비중을 차지한다. 특히 빅 데이터와 인공지능 등 많은 데이터를 소모하는 분야에서는 그 비중이 더욱 증가한다. 하지만 이러한 비중에도 불구하고 GPGPU 분야에서는 데이터 통신 개선은 하드웨어 분야의 몫이라 생각하고 있는 현실이다. 하지만 이미 하드웨어는 충분히 발전했고 현재는 개선되지 않고 사용한 로직에 의해 오히려 소프트웨어적인 문제가 병목으로 작용하고 있다. 이 논문에서는 이러한 병목을 해결하기 위해 기존의 로직을 개선할 것이다.

■ 서론

가) 제안배경 및 필요성

과거에는 시스템에서 하드웨어가 병목인 경우가 대다수였지만 현재는 하드웨어의 발전으로 오히려 소프트웨어가 병목인 경우가 증가했다. 호스트와 디바이스 간의 PCIe 통신에서 호스트의 메모리의 pinning한 영역을 버퍼로 하는 더블 버퍼링 기법을 사용하고 있다. 하지만 부적절한 알고리즘으로 인해 호스트와 버퍼간의 속도가 디바이스와 버퍼간의 속도보다 많이 느려서 PCIe 대역폭을 모두 사용하지 못해 소프트웨어가 병목인 상황이다. 그래서 호스트와 버퍼간의 통신을 생산자 디바이스와 버퍼간의 통신을 소비자로 간주해 다중 생산자 단일 소비자 문제로 치환을 하고 이에 필요한 더블 큐로 PCIe 대역폭을 최대한 사용해 데이터 통신의 속도를 향상시키고자 한다.

나) 연구논문/작품의 목표

최근 인공지능과 암호화폐 등과 같이 GPGPU를 이용하는 사례가 급격히 증가하고 있다. 이러한 추세에 편승해 가용한 자원들을 최대한 사용해 연산 효율을 높이고자 하는 노력도 활발하게 이루어지고 있다.

GPU를 이용해 연산은 먼저 호스트 메모리에서 디바이스 메모리로 처리할 데이터를 복사하고 이것을 이용해 디바이스에서 연산을 한 다음 결과를 디바이스 메모리에서 호스트 메모리로 다시 복사하는 일련의 과정을 거친다. 디바이스에서 연산하는 것 못지않게 데이터를 이동하는데도 많은 시간이 소요가 된다고 한다. 호스트와 디바이스가 데이터를 주고받을 때 디바이스는 DMA(Direct Memory Access)를 통해 호스트의 물리 메모리에 직접 접근을 하므로 주고받고자 하는 대상의 메모리 영역은 꼭 pinning해야 한다.

그러므로 CUDA에서는 크게 두 가지의 방법으로 호스트와 디바이스가 데이터를

주고받는다. 첫 번째로는 CUDA 내부적으로 호스트 메모리의 pinning한 버퍼를 생성하여 해당 데이터를 버퍼에 복사하며 디바이스는 버퍼에서 데이터를 읽어오는 방식이다. 두 번째로는 전송하고자 하는 데이터의 영역을 전부 pinning해서 디바이스가 직접 접근하는 방법이 있다.

첫 번째 방법은 작은 크기의 버퍼만으로 호스트와 디바이스가 데이터를 주고받을 수 있지만 PCIe 대역폭을 모두 사용하지 못하는 단점이 있다. 두 번째 방법은 PCIe 대역폭을 모두 사용 가능하지만 호스트 메모리에 큰 영역에서 pinning을 하면 호스트 시스템이 사용할 수 있는 메모리가 부족해져 성능이 낮아 질 수 있다.

이 논문에서는 첫 번째 방법의 문제점 해결에 집중 할 것이다. 대역폭을 최대로 활용하기 위해 문제를 생산자 소비자 문제의 관점에서 다수의 생산자와 하나의 소비자로 설정해 더블 큐를 이용할 것이다. 이 방법을 사용하면 비교적 큰 데이터를 전송할 때 속도가 대략 2배 이상 증가한다.

다) 연구논문/작품 전체 overview

The memory transfer between CPU host and GPU device is known to be a heavy burden on GPU applications. Memory transfer overheads constitute about a lot of total running time in rodinia benchmark. While such 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. Unlike conventional hardware bottleneck, software bottleneck is throttling back the performance in inter CPU-GPU memory transfer. In this paper, we propose a software optimization technique for memory transfer between device and host.

CUDA adopted double buffering system for memory transfer in NVIDIA GPUs. This double buffering technique involves memory pinning for stable remote DMA. However there is a mismatch between PCIe bandwidth and CPU memory bandwidth. With 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 like industry standard GDDR5. Furthermore, hardware vendors have been introducing a new standard to market nearly triennial interval. This caused fragmented memory bandwidth within a system. Despite this fragmented memory bandwidth, CUDA insists on static memory transfer technique in its cudaMemcpy().

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().

■ 관련연구

Choosing Between Pinned and Non-Pinned Memory[1]

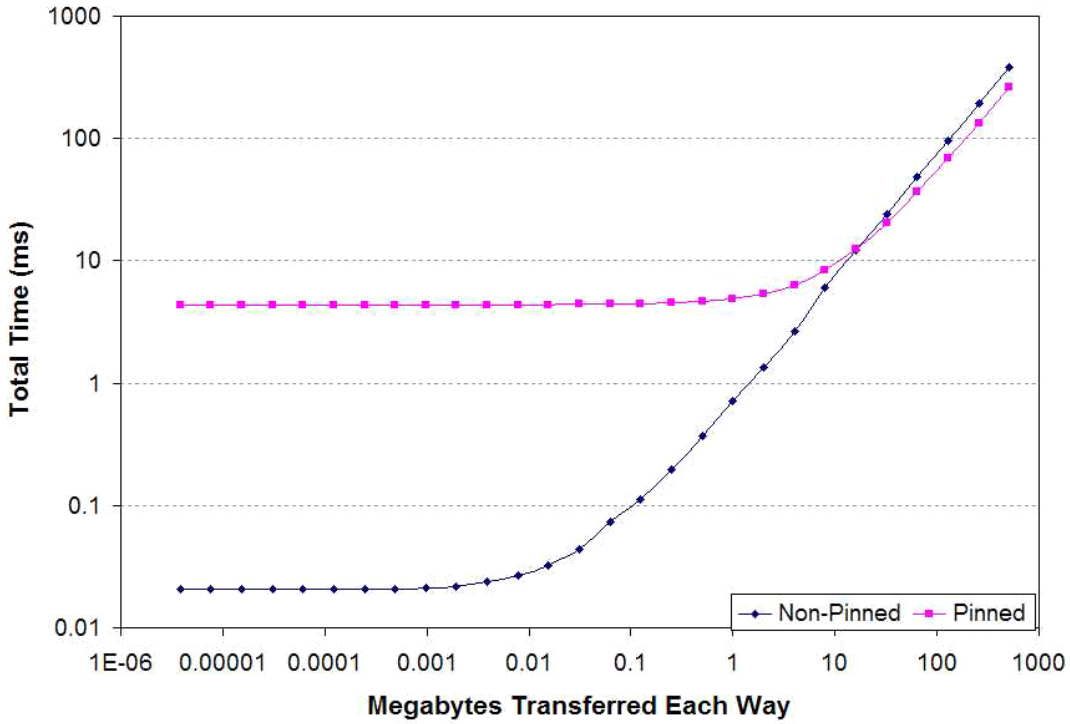


그림 2 Time required to allocate, transfer to the GPU, transfer back to the CPU, and deallocate pinned and non-pinned memory. Note the logarithmic scales.

Page-Locked Memory[2]

- mlock() ensures a page to be on physical memory but not fixed.
- A page that is being migrated has been isolated from the LRU lists and is held locked across unmapping of the page, updating the page's address space entry and copying the contents and state, until the page table entry has been replaced with an entry that refers to the new page.
- Linux supports migration of mlocked pages and other unevictable pages. This involves simply moving the PG_mlocked and PG_unevictable states from the

old page to the new page.
- May cause soft page fault.

Unified Memory[3]

- Unified Memory offers a "single-pointer-to-data" model that is conceptually similar to CUDA's zero-copy memory.
- Unified Memory eliminates the need for explicit data movement via the cudaMemcpy*() routines without the performance penalty incurred by placing all data into zero-copy memory.
- Data movement still takes place so a program's run time typically does not decrease.
- Unified Memory instead enables the writing of simpler and more maintainable code.

Zero copy(Mapped Pinned Memory)[4]

- Zero-copy system memory has identical coherence and consistency guarantees to global memory
- A kernel may not allocate or free zero-copy memory, but may use pointers to zero-copy passed in from the host program.
- With zero-copy allocations the physical location of memory is pinned in CPU system memory such that a program may have fast or slow access to it depending on where it is being accessed from
- Unified Memory, on the other hand, decouples memory and execution spaces so that all data accesses are fast.

Pinned Memory[2]

- Lock pages in memory via actions like the creation of remote DMA (RDMA) buffers. Those pages are not currently counted against the limit on locked pages.
- These "back door" locked pages also create another sort of problem.
- Normally, the memory management subsystem goes out of its way to separate pages that can be moved from those that are fixed in place.
- The pages are often allocated as normal anonymous memory — movable pages, in other words.

- Memory management code tries to create contiguous ranges of memory by shifting pages around.
- They are in a place reserved for movable pages, but, being unmovable, they cannot be moved out of the way to make the creation of larger blocks possible.

Mapped pinned memory vs Pinned Memory[5]

Mapped, pinned memory (zero-copy) is useful when either:

- The GPU has no memory on its own and uses RAM anyway
- You load the data exactly once, but you have a lot of computation to perform on it and you want to hide memory transfer latencies through it.
- The host side wants to change/add more data, or read the results, while kernel is still running (e.g. communication)
- The data does not fit into GPU memory

Note that, you can also use multiple streams to copy data and run kernels in parallel. Pinned, but not mapped memory is better:

- When you load or store the data multiple times. For example: you have multiple subsequent kernels, performing the work in steps - there is no need to load the data from host every time.
- There is not that much computation to perform and loading latencies are not going to be hidden well

NVIDIA Driver License with Linux kernel

```
shcho@shcho-Vostro-470:~/NVIDIA-Linux-x86_64-390.48$ cat build.sh
#!/bin/sh
echo "sudo service lightdm stop "
sudo service lightdm stop

echo "sudo nvidia-installer"
sudo nvidia-installer

echo "sudo reboot"
sudo reboot
shcho@shcho-Vostro-470:~/NVIDIA-Linux-x86_64-390.48$
```

그림 3 NVIDIA Driver License with Linux kernel

```
shcho@shcho-Vostro-470:~$ ls
Desktop    examples.desktop  linux-4.4.131.tar.xz     NVIDIA-Linux-x86_64-390.48      pjh        Videos
Documents  linux             Music                    NVIDIA-Linux-x86_64-390.48.run  Public
Downloads  linux-4.4.131     NVIDIA_CUDA-8.0_Samples  Pictures                        Templates
shcho@shcho-Vostro-470:~$ ./NVIDIA-Linux-x86_64-390.48.run -help

./NVIDIA-Linux-x86_64-390.48.run [options]

This program will install the NVIDIA Accelerated Graphics Driver for
Linux-x86_64 390.48 by unpacking the embedded tarball and executing
the ./nvidia-installer  installation utility.

Below are the most common options; for a complete list use
'--advanced-options'.

--info
  Print embedded info (title, default target directory) and exit.

--check
  Check integrity of the archive and exit.

-x, --extract-only
  Extract the contents of ./NVIDIA-Linux-x86_64-390.48.run, but do not
  run 'nvidia-installer'.


The following arguments will be passed on to the ./nvidia-installer
utility:

  -v, --version
      Print the nvidia-installer version and exit.

  -h, --help
      Print usage information for the common commandline options
      and exit.

  -A, --advanced-options
      Print usage information for the common commandline options
      as well as the advanced options, and then exit.
```

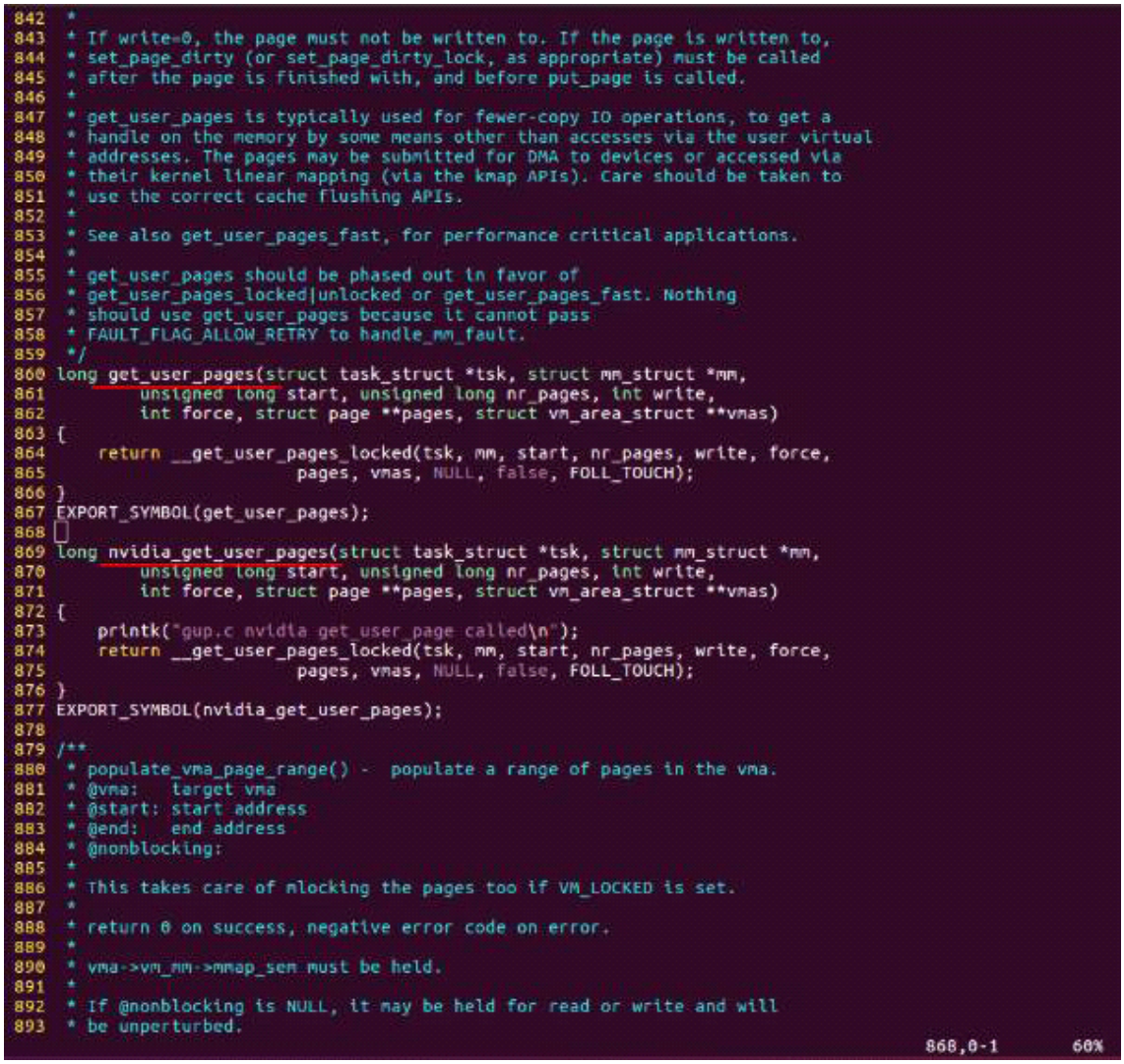
그림 4 NVIDIA Driver License with Linux kernel

```
shcho@shcho-Vostro-470:~/NVIDIA-Linux-x86_64-390.48$ ls
10_nvidia.json                libnvidia-cfg.so.390.48                                 nvidia-cuda-mps-control.1.gz
10_nvidia_wayland.json        libnvidia-compiler.so.390.48                            nvidia-cuda-mps-server
32                            libnvidia-eglcore.so.390.48                             nvidia-debugdump
build.sh                      libnvidia-egl-wayland.so.1.0.2                          nvidia-drm-outputclass.conf
cscope.files                  libnvidia-encode.so.390.48                              nvidia_drv.so
cscope.out                    libnvidia-fatbinaryloader.so.390.48                     nvidia.icd
glext.h                       libnvidia-fbc.so.390.48                                 nvidia_icd.json.template
gl.h                          libnvidia-glcore.so.390.48                              nvidia-installer
glxext.h                      libnvidia-glsi.so.390.48                                nvidia-installer.1.gz
glx.h                         libnvidia-gtk2.so.390.48                                nvidia-modprobe
html                          libnvidia-gtk3.so.390.48                                nvidia-modprobe.1.gz
kernel                        libnvidia-ifr.so.390.48                                 nvidia-persistenced
libcuda.so.390.48             libnvidia-ml.so.390.48                                  nvidia-persistenced.1.gz
libEGL_nvidia.so.390.48       libnvidia-opencl.so.390.48                              nvidia-persistenced-init.tar.bz2
libEGL.so.1.1.0               libnvidia-ptxjitcompiler.so.390.48                      nvidia-settings
libEGL.so.390.48              libnvidia-tls.so.390.48                                 nvidia-settings.1.gz
libGLdispatch.so.0            libnvidia-wfb.so.390.48                                 nvidia-settings.desktop
libGLESv1_CM_nvidia.so.390.48 libOpenCL.so.1.0.0                                      nvidia-settings.png
libGLESv1_CM.so.1.2.0         libOpenGL.so.0                                          nvidia-smi
libGLESv2_nvidia.so.390.48    libvdpau_nvidia.so.390.48                               nvidia-smi.1.gz
libGLESv2.so.2.1.0            LICENSE                                                 nvidia-xconfig
libGL.la                      makeself-help-script.sh                                 nvidia-xconfig.1.gz
libGL.so.1.7.0                makeself.sh                                             pkg-history.txt
libGL.so.390.48               mkprecompiled                                           README.txt
libglvnd_install_checker      nvidia-application-profiles-390.48-key-documentation    tags
libGLX_nvidia.so.390.48       nvidia-application-profiles-390.48-rc                   tls
libGLX.so.0                   nvidia-bug-report.sh                                    tls_test
libglx.so.390.48              NVIDIA_Changelog                                        tls_test_dso.so
libnvcuvid.so.390.48          nvidia-cuda-mps-control
shcho@shcho-Vostro-470:~/NVIDIA-Linux-x86_64-390.48$
```

그림 5 NVIDIA Driver License with Linux kernel

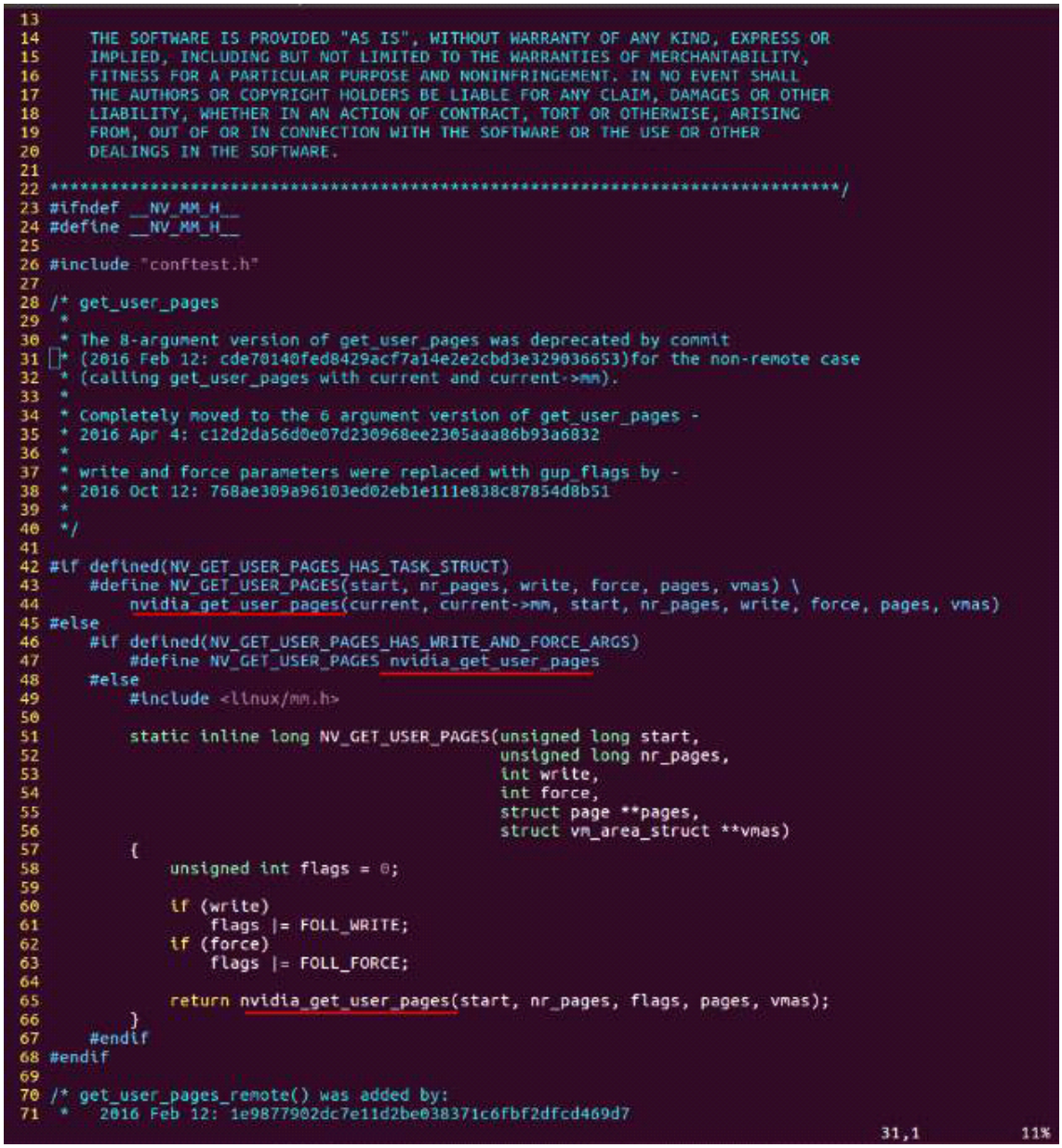
```
182  *   slab page or a secondary page from a compound page
183  * - don't permit access to VMAs that don't support it, such as I/O mappings
184  */
185 long get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
186             unsigned long start, unsigned long nr_pages,
187             int write, int force, struct page **pages,
188             struct vm_area_struct **vmas)
189 {
190     int flags = 0;
191
192     if (write)
193         flags |= FOLL_WRITE;
194     if (force)
195         flags |= FOLL_FORCE;
196
197     return __get_user_pages(tsk, mm, start, nr_pages, flags, pages, vmas,
198                 NULL);
199 }
200 EXPORT_SYMBOL(get_user_pages);
201
202 long nvidia_get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
203             unsigned long start, unsigned long nr_pages,
204             int write, int force, struct page **pages,
205             struct vm_area_struct **vmas)
206 {
207     int flags = 0;
208
209     if (write)
210         flags |= FOLL_WRITE;
211     if (force)
212         flags |= FOLL_FORCE;
213     printk("nonmu.c nvidia_get_user_pages called\n");
214
215     return __get_user_pages(tsk, mm, start, nr_pages, flags, pages, vmas,
216                 NULL);
217 }
218 EXPORT_SYMBOL(nvidia_get_user_pages);
219
220 long get_user_pages_locked(struct task_struct *tsk, struct mm_struct *mm,
221                unsigned long start, unsigned long nr_pages,
222                int write, int force, struct page **pages,
223                int *locked)
224 {
225     return get_user_pages(tsk, mm, start, nr_pages, write, force,
226                   pages, NULL);
227 }
228 EXPORT_SYMBOL(get_user_pages_locked);
229
230 long __get_user_pages_unlocked(struct task_struct *tsk, struct mm_struct *mm,
231                    unsigned long start, unsigned long nr_pages,
232                    int write, int force, struct page **pages,
233                    unsigned int gup_flags)
234 {
235     long ret;
236     down_read(&mm->mmap_sem);
237     ret = get_user_pages(tsk, mm, start, nr_pages, write, force,
238                  pages, NULL);
239     up_read(&mm->mmap_sem);
240     return ret;
-- VISUAL --                                                    8          208,0-1        8%
```

그림 6 NVIDIA Driver License with Linux kernel



```
842  *
843  * If write=0, the page must not be written to. If the page is written to,
844  * set_page_dirty (or set_page_dirty_lock, as appropriate) must be called
845  * after the page is finished with, and before put_page is called.
846  *
847  * get_user_pages is typically used for fewer-copy IO operations, to get a
848  * handle on the memory by some means other than accesses via the user virtual
849  * addresses. The pages may be submitted for DMA to devices or accessed via
850  * their kernel linear mapping (via the kmap APIs). Care should be taken to
851  * use the correct cache flushing APIs.
852  *
853  * See also get_user_pages_fast, for performance critical applications.
854  *
855  * get_user_pages should be phased out in favor of
856  * get_user_pages_locked|unlocked or get_user_pages_fast. Nothing
857  * should use get_user_pages because it cannot pass
858  * FAULT_FLAG_ALLOW_RETRY to handle_mm_fault.
859  */
860 long get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
861         unsigned long start, unsigned long nr_pages, int write,
862         int force, struct page **pages, struct vm_area_struct **vmas)
863 {
864     return __get_user_pages_locked(tsk, mm, start, nr_pages, write, force,
865                        pages, vmas, NULL, false, FOLL_TOUCH);
866 }
867 EXPORT_SYMBOL(get_user_pages);
868
869 long nvidia_get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
870         unsigned long start, unsigned long nr_pages, int write,
871         int force, struct page **pages, struct vm_area_struct **vmas)
872 {
873     printk("gup.c nvidia get_user_page called\n");
874     return __get_user_pages_locked(tsk, mm, start, nr_pages, write, force,
875                        pages, vmas, NULL, false, FOLL_TOUCH);
876 }
877 EXPORT_SYMBOL(nvidia_get_user_pages);
878
879 /**
880  * populate_vma_page_range() -  populate a range of pages in the vma.
881  * @vma:   target vma
882  * @start: start address
883  * @end:   end address
884  * @nonblocking:
885  *
886  * This takes care of mlocking the pages too if VM_LOCKED is set.
887  *
888  * return 0 on success, negative error code on error.
889  *
890  * vma->vm_mm->mmap_sem must be held.
891  *
892  * If @nonblocking is NULL, it may be held for read or write and will
893  * be unperturbed.
                                                        868,0-1       60%
```

그림 7 NVIDIA Driver License with Linux kernel



```
13
14      THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
15      IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
16      FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
17      THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
18      LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
19      FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
20      DEALINGS IN THE SOFTWARE.
21
22  *******************************************************************************/
23  #ifndef __NV_MM_H__
24  #define __NV_MM_H__
25
26  #include "conftest.h"
27
28  /* get_user_pages
29   *
30   * The 8-argument version of get_user_pages was deprecated by commit
31   * (2016 Feb 12: cde70140fed8429acf7a14e2e2cbd3e329036653)for the non-remote case
32   * (calling get_user_pages with current and current->mm).
33   *
34   * Completely moved to the 6 argument version of get_user_pages -
35   * 2016 Apr 4: c12d2da56d0e07d230968ee2305aaa86b93a6832
36   *
37   * write and force parameters were replaced with gup_flags by -
38   * 2016 Oct 12: 768ae309a96103ed02eb1e111e838c87854d8b51
39   *
40   */
41
42  #if defined(NV_GET_USER_PAGES_HAS_TASK_STRUCT)
43      #define NV_GET_USER_PAGES(start, nr_pages, write, force, pages, vmas) \
44          nvidia_get_user_pages(current, current->mm, start, nr_pages, write, force, pages, vmas)
45  #else
46      #if defined(NV_GET_USER_PAGES_HAS_WRITE_AND_FORCE_ARGS)
47          #define NV_GET_USER_PAGES nvidia_get_user_pages
48      #else
49          #include <linux/mm.h>
50
51          static inline long NV_GET_USER_PAGES(unsigned long start,
52                                               unsigned long nr_pages,
53                                               int write,
54                                               int force,
55                                               struct page **pages,
56                                               struct vm_area_struct **vmas)
57          {
58              unsigned int flags = 0;
59
60              if (write)
61                  flags |= FOLL_WRITE;
62              if (force)
63                  flags |= FOLL_FORCE;
64
65              return nvidia_get_user_pages(start, nr_pages, flags, pages, vmas);
66          }
67      #endif
68  #endif
69
70  /* get_user_pages_remote() was added by:
71   *   2016 Feb 12: 1e9877902dc7e11d2be038371c6fbf2dfcd469d7
                                                                    31,1          11%
```

그림 8 NVIDIA Driver License with Linux kernel

■ 제안 작품 소개

BACKGROUND

Pinned Memory

Pinned memory is the page-locked memory that does not cause a soft page fault. 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. The host operating system uses the kernel function get_user_pages() for pinned memory in Linux. CUDA keeps track of pinned memory of itself and accelerates memory copies that involve host addresses pointing at pinned memory. 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.

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

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 one pinned buffer(Step 1 in 그림 9 Logic of Double Buffering), 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 그림 9 Logic of Double Buffering). The host keeps flip-flopping with the device between pinned buffers, with synchronization, until the device performs the last memory copy(Step 3 in 그림 9 Logic of Double Buffering). 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 그림 10 Timeline of Memory Transfers in Double Buffering.

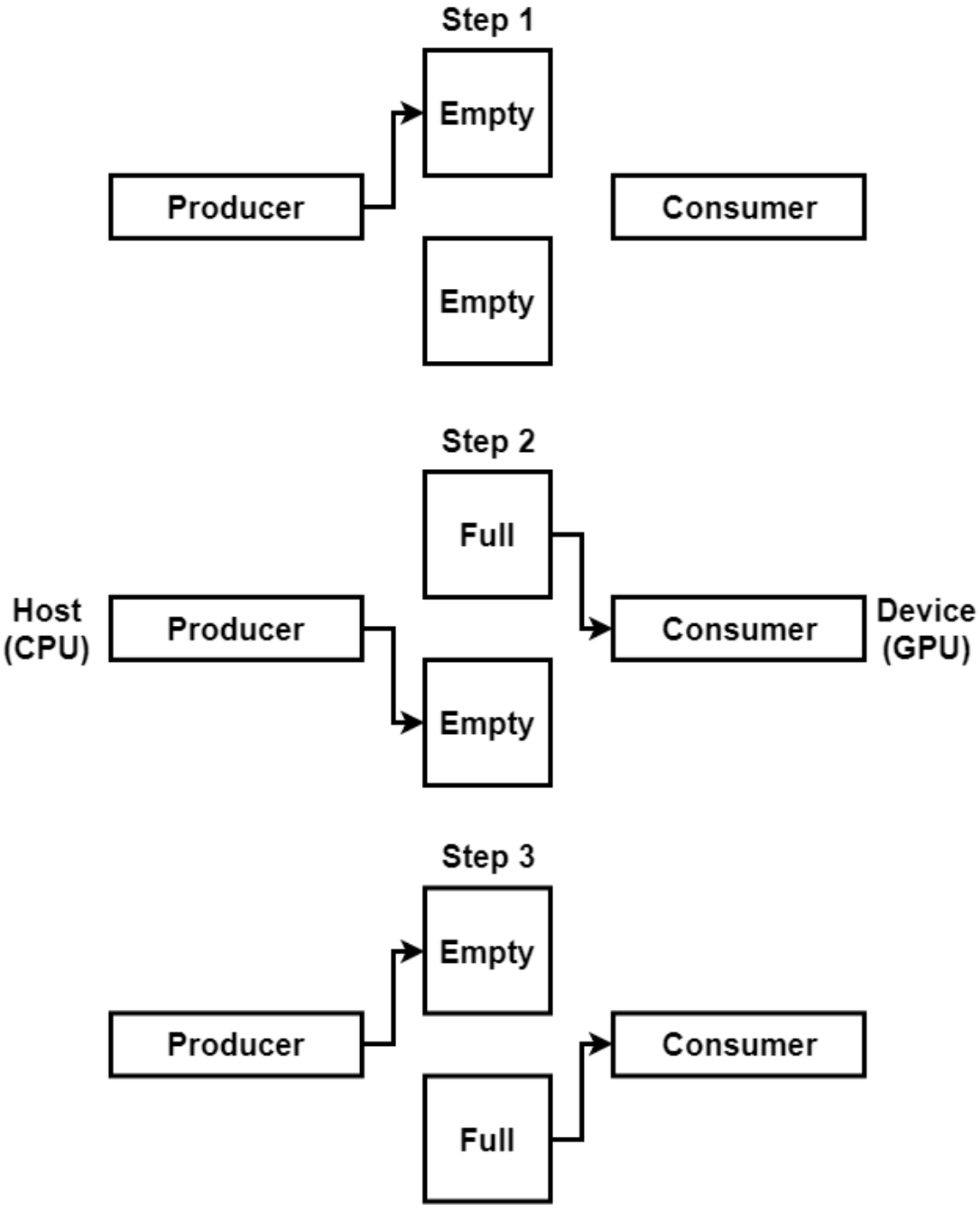


그림 9 Logic of Double Buffering

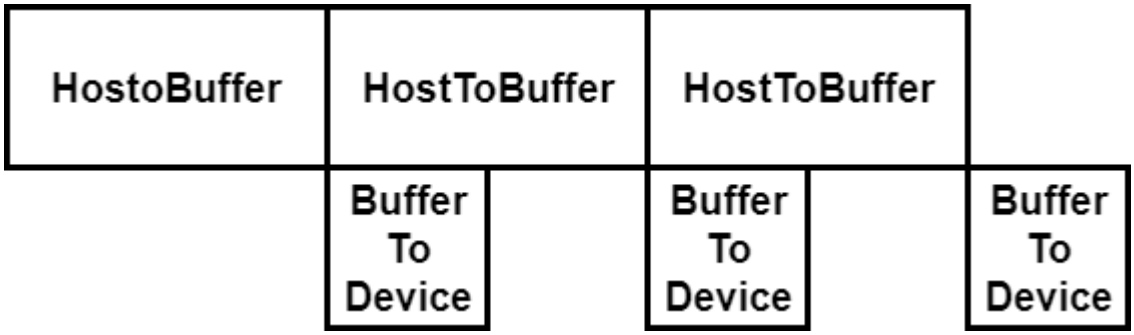


그림 10 Timeline of Memory Transfers in Double Buffering

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. For CUDA, write combined memory can be allocated by calling cudaHostAlloc() with cudaHostWriteCombined as a parameter. Besides setting the page table entries to bypass the host caches, write-combined memory is ensured not to be snooped during PCIe bus transfers.

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. 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). 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.

Multithreaded Double Queuing

Transferring data from the host to devices can be transposed to a producer-consumer problem. 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 그림 11 Algorithm of Producer and 그림 12 Algorithm of Consumer.

Multiple producers execute the steps of producers and one consumer executes the steps of a consumer as shown in 그림 13 Multithread Double Queuing: N producers and 1 consumer. Because producers copy the data of the host to the buffer concurrently and queues guarantee first in first out, the consumer can copy the data of the buffer to the device without any delay expressed in 그림 14 Timeline of Memory Transfers in 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.

```
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)

}
```

그림 11 Algorithm of Producer

```
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)

}
```

그림 12 Algorithm of Consumer

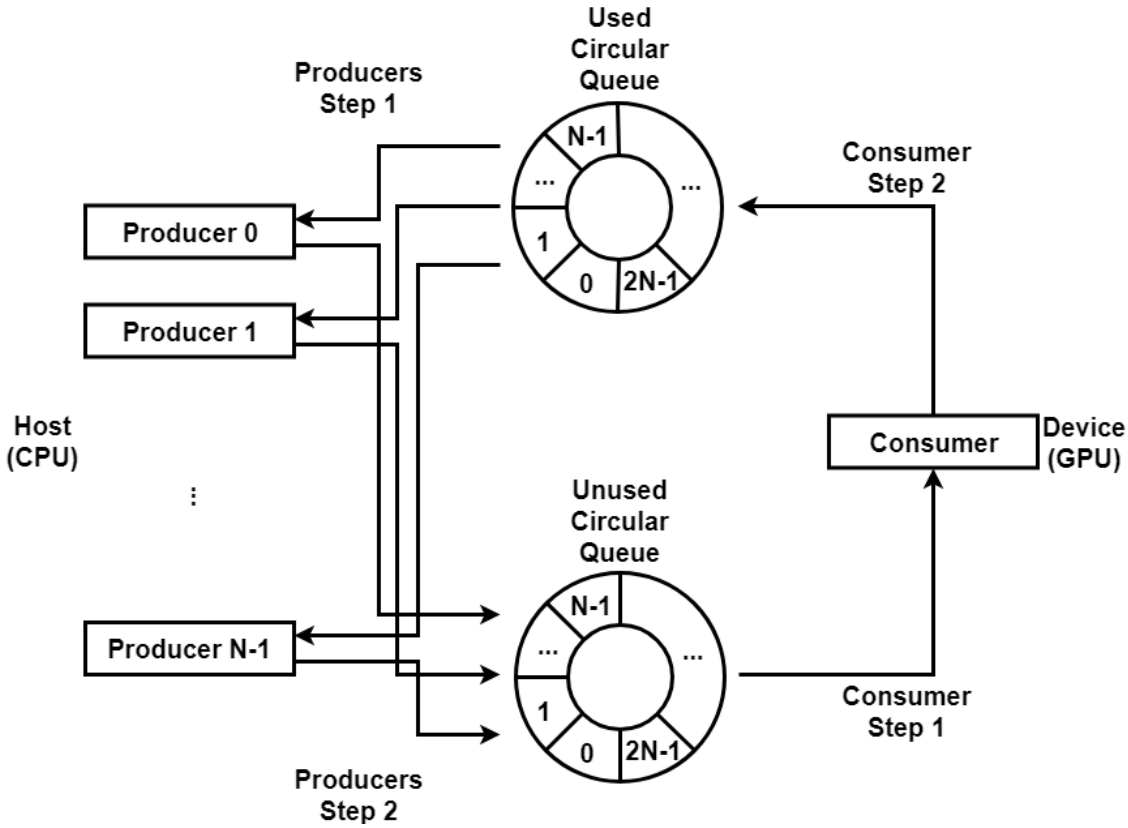


그림 13 Multithread Double Queuing: N producers and 1 consumer

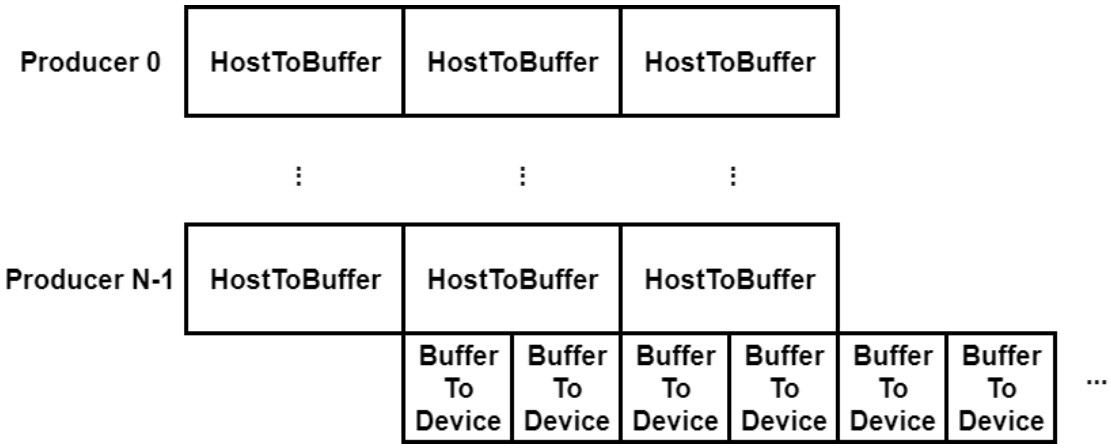


그림 14 Timeline of Memory Transfers in Double Buffering

■ 구현 및 결과분석

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

그림 15 Specification of Experiment Environment

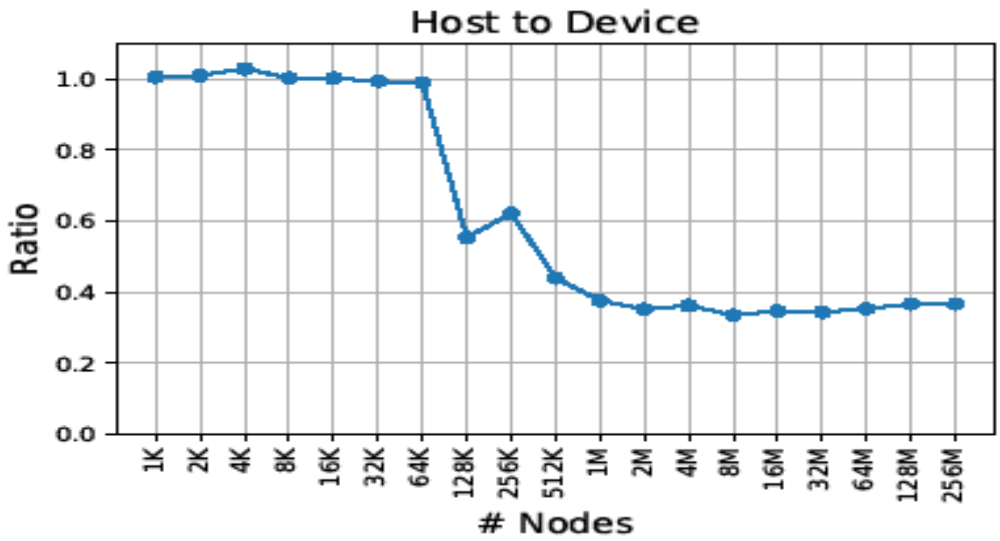


그림 16 BFS Normalized Memory Transfer Host to Device

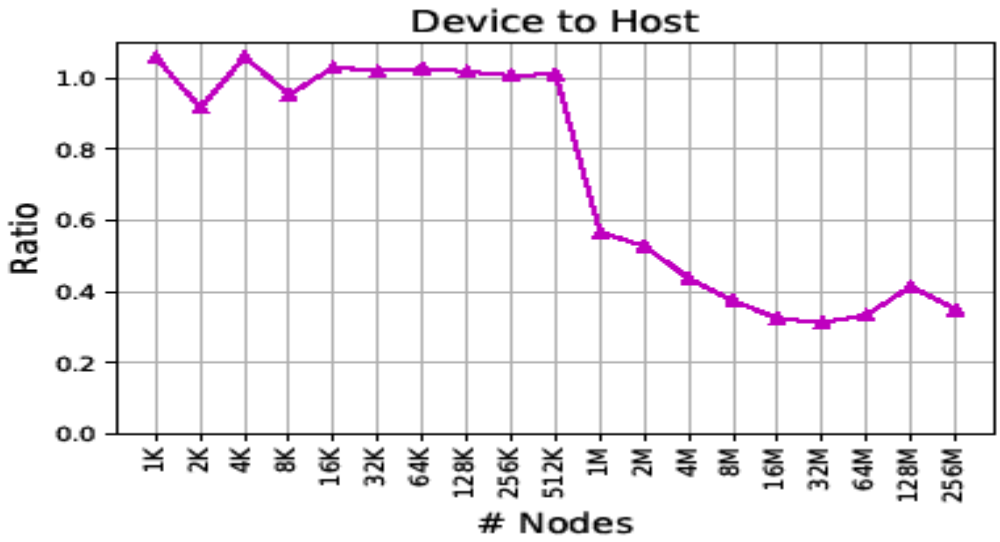


그림 17 BFS Normalized Memory Transfer Device to Host

실험환경은 그림 15 Specification of Experiment Environment 와 같다. 그림 16 BFS Normalized Memory Transfer Host to Device 와 그림 17 BFS Normalized Memory Transfer Device to Host 은 BFS 벤치마크에서 Host to Device와 그 반대에서 기존에 비해 구현 사항의 속도가 얼마나 증가했는지 비율로 보여주고 있다. Host to Device와 그 반대의 경우 각각 노드 수가 128K와 1M을 기준으로 급격하게 성능향상을 보여주고 있는데 기준점 이전에서는 데이터의 크기가 1MB 미만이기 때문에 기존 것의 성능 하락이 나타나지 않는다. 1MB인 이유는 더블 버퍼링에서 두 개의 버퍼들을 모두 사용하는 경우는 각각 버퍼의 크기인 1MB을 초과할 때 발

생하고 이 크기를 초과하지 않으면 버퍼는 한 개만 사용되기 때문에 PCIe 대역폭에서 대기 상태가 발생하지 않기 때문이다. 데이터 크기가 일정 이상이면 대략 270%의 성능 향상을 보여준다.

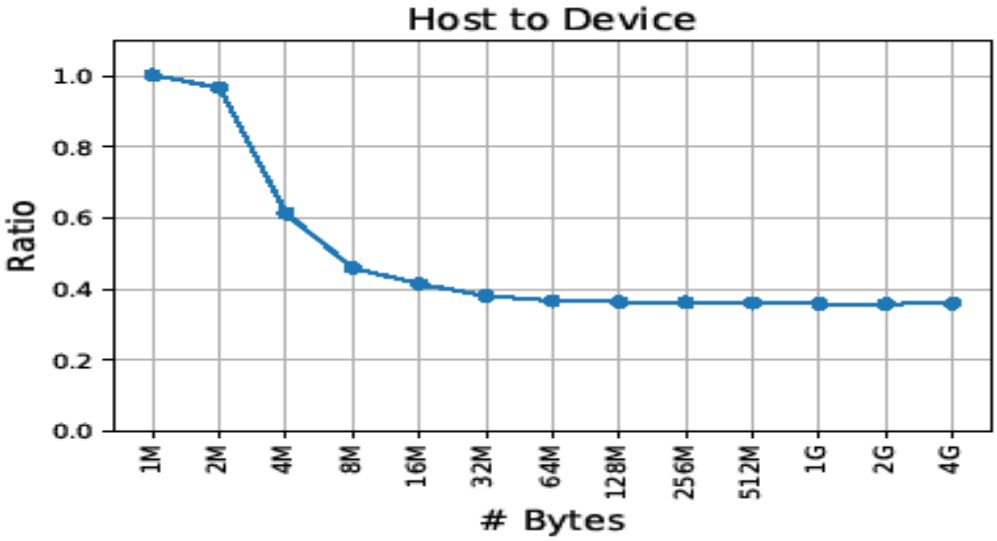


그림 18 Pathfinder Normalized Memory Transfer Host to Device

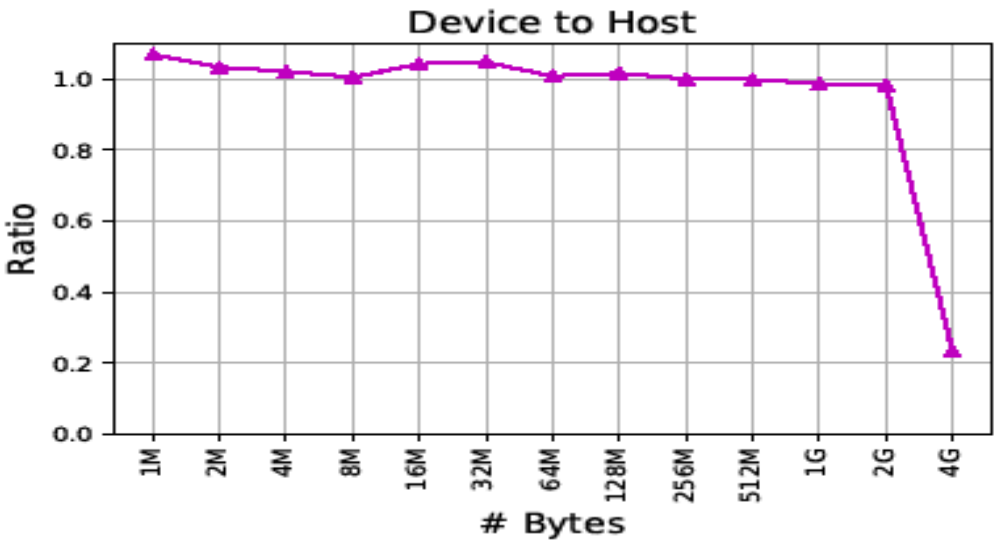


그림 19 Pathfinder Normalized Memory Transfer Device to Host

그림 18 Pathfinder Normalized Memory Transfer Host to Device 와 그림 19 Pathfinder Normalized Memory Transfer Device to Host 은 Pathfinder 벤치마크에서 Host to Device와 그 반대에서 기존에 비해 구현 사항의 속도가 얼마나 증가했는지 비율로 보여주고 있다. BFS의 경우와 마찬가지로 Host to Device에서는 1MB

를 초과하자 성능 향상이 발견되고 있다. 그 반대의 경우에는 2GB 이후에 성능 향상을 보여주고 있는데 그 이유는 Pathfinder에서 결과 값의 데이터 크기를 입력 값에 비해 매우 작기 때문에 해당 입력 값 데이터 크기 이후에 결과 값의 데이터 크기가 1MB을 초과하기 때문이다. 성능향상은 270% 정도로 BFS와 거의 같다.

한편 BFS와 Pathfinder 모두 Device to Host에서 결과 값들이 다소 불안정한 것을 볼 수 있는데 이는 기존 것의 측정값들이 불안정하여 발생한 것이다.

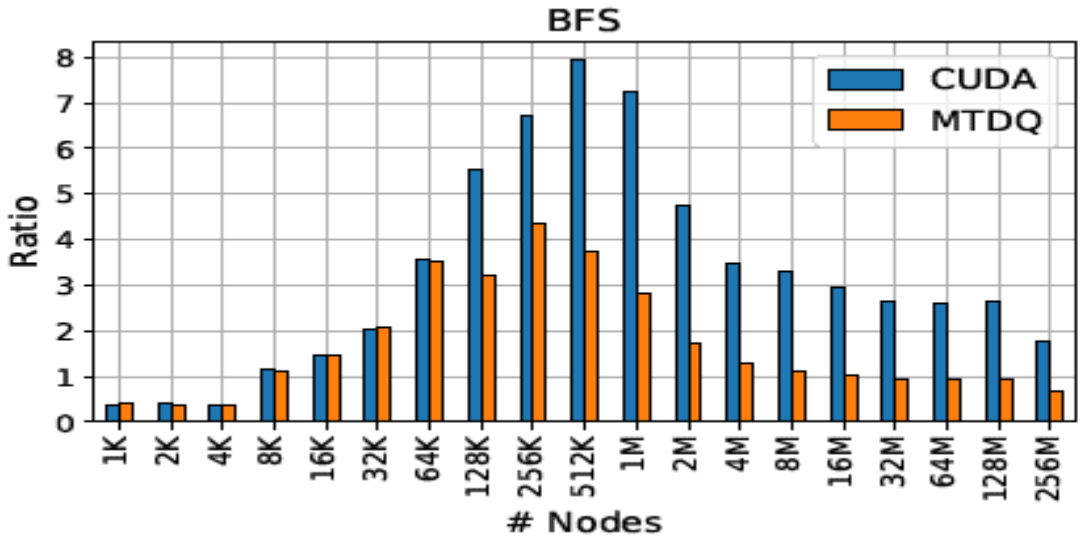


그림 20 Memory Transfer Time/Kernel Execution Time of Plain CUDA and Multithreaded Double Queuing BFS

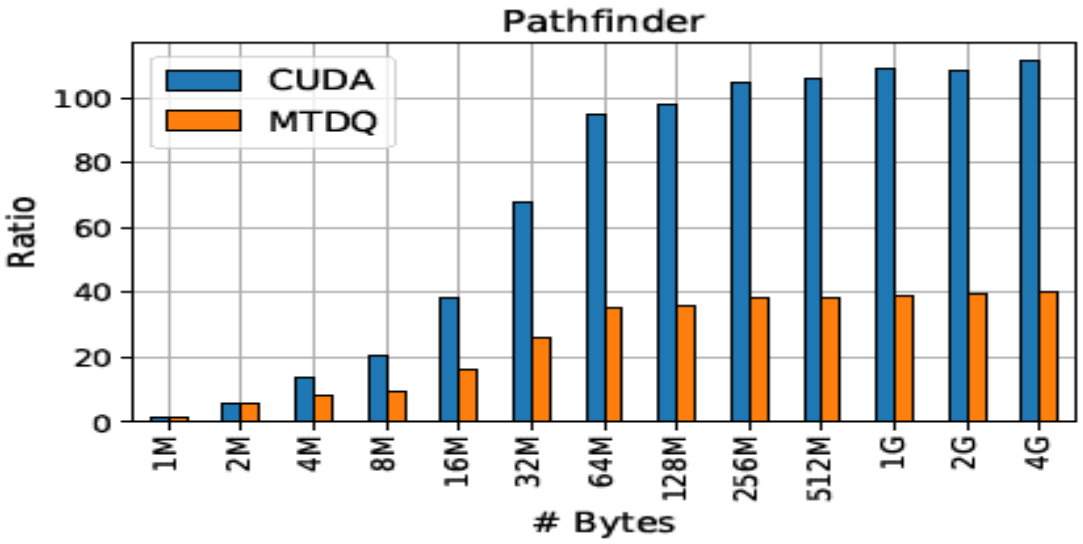


그림 21 Memory Transfer Time/Kernel Execution Time of Plain CUDA and Multithreaded Double Queuing Pathfinder

그림 20 Memory Transfer Time/Kernel Execution Time of Plain CUDA and Multithreaded Double Queuing BFS 와 그림 21 Memory Transfer Time/Kernel Execution Time of Plain CUDA and Multithreaded Double Queuing Pathfinder 은 순수 연산 시간 대비 데이터 전송 시간의 비율을 기존의 것과 구현한 것으로 구별하여 보여주고 있다. BFS와 Pathfinder 모두 기존 것에 비해 그 비율이 현저하게 낮아지는 것을 볼 수 있다. 입력 값의 크기에 따라 다소 차이는 있지만 일정 크기를 초과하면 비율이 50%이상 감소하는 것을 볼 수 있다. 즉 구현 한 것을 사용한다면 총 GPU 연산 시간을 대폭 줄일 수 있다.

■ 결론

기존 CUDA에서 구현되어 있는 CPU-GPU간 메모리 복사 로직은 PCIe 대역폭을 모두 사용하지 못하고 있다. 원인은 호스트에서 버퍼로의 메모리 복사와 버퍼에서 디바이스로의 메모리 복사간의 속도 불균형 때문이다.

이 문제는 전자를 producer로 후자를 consumer로 치환해서 multiple producers single consumer 상황으로 간주가 가능하다. 그리고 이 상황을 해결하기 위해서는 여기서 제안한 새로운 로직인 double queue 방식을 사용하는 것이 적절하다.

다만 개선사항으로 로직을 실제 구현 시 메모리 복사 함수가 호출될 때 마다 생산자와 소비자를 매번 생성해야하는 비효율성이 존재한다. 이는 생산자와 소비자를 한번 생성 후 계속 재사용하는 방향으로 가면 해결이 가능할 것이다.

■ 후기

최근 딥 러닝이나 인공지능이 컴퓨터 공학 계에서 뜨거운 감자인 것이 사실이다. 그래서 기존의 알고리즘을 활용하고 효율적으로 구현하는데 관심이 집중되고 있다. 하지만 알고리즘이 실행되는 하드웨어적인 시스템 환경에는 접근성이 떨어진다는 이유로 상대적으로 관심이 덜하다. 사실 남들이 하는 것은 흥미가 없었기 때문에 후자에 관심이 쏠렸다. 그래서 최근 이러한 알고리즘을 가속하기위해 GPU를 적극적으로 사용한다는 사실을 알고 기존 환경의 문제점을 탐색해 본 결과 GPU가 연산해야 할 데이터를 전송하는 행위 자체가 비효율적으로 동작하고 있음을 알 수 있었다. 이 분야는 처음이었기 때문에 연구를 하면서 많은 어려움이 있었지만 이렇게 무사히 개선점을 찾을 수 있어서 다행이라 생각한다.

■ 참고문헌

[1] Pinned vs. Non-Pinned Memory, www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html.

[2] “Locking and Pinning.” [LWN.net], lwn.net/Articles/600502/.

[3] “CUDA C Programming Guide.” NVIDIA Developer Documentation, docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#ixzz58QrvcAsN.

[4] Kirk, David B., and W. Hwu Wen-Mei. Programming massively parallel processors: a hands-on approach. Morgan kaufmann, 2016.

[5] “Default Pinned Memory Vs Zero-Copy Memory.” Stack Overflow, stackoverflow.com/questions/5209214/default-pinned-memory-vs-zero-copy memory.