붙임2

2018 학년도 제 2 학기

연구논문/작품 중간보고서

제목	Multiple Producer Single Consumer Double Queue	○ 논문(✓) 작품(　　) ※해당란 체크
GitHub URL	https://github.com/SanghoonARCS/graduation-thesis	
평가등급	지도교수 수정보완 사항	팀원 명단
A, B, F중 택1 (지도교수가 부여)	○ ○ A ○	조 상 훈 (인) [illegible] (학번: 2013314616) 홍 재 완 (인) [illegible] (학번: 2013312784)

2018 년 9 월 21 일

지도교수 : 한 환 수　서명 [illegible]

■ 요약

The memory transfer between host and gpu device is known to be a heavy burden on GPU applications. Memory transfer overheads constitute about 00% 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.

■ 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 pinned된 영역을 버퍼로하는 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) 연구논문의 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 못지 않게 데이터를 이동하는데도 많은 시간이 소요가 된다고 한다. 호스트와 디바이스가 데이터를 주고받을 때 디바이스는 dma를 통해 호스트의 물리 메모리에 직접 접근을 하므로 주고받고자 하는 대상의 메모리 영역은 꼭 pinned되어있어야 한다.

그러므로 쿠다에서는 크게 두 가지의 방법으로 호스트와 디바이스가 데이터를 주고 받는다. 첫번째로는 쿠다 내부적으로 호스트 메모리의 pinned된 버퍼를 생성하여 해당 데이터를 버퍼에 복사하며 디바이스는 버퍼에서 데이터를 읽어오는 방식이다. 두번째로는 전송하고자 하는 데이터의 영역을 전부 pinning해서 디바이스가

직접 접근하는 방법이 있다.

첫번째 방법은 작은 크기의 버퍼만으로 호스트와 디바이스가 데이터를 주고 받을 수 있지만 PCIE 대역폭을 모두 사용하지 못하는 단점이 있다. 두번째 방법은 PCIE 대역폭을 모두 사용 가능하지만 호스트 메모리에 큰 영역의 pinning을 하게 되면 호스트 시스템이 사용할 수 있는 메모리가 부족해져 성능이 낮아 질 수 있다.

이 논문에서는 첫번째 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 2배이상 증가한다.

다) 연구논문 전체 overview

The memory transfer between host and gpu device is known to be a heavy burden on GPU applications. Memory transfer overheads constitute about 00% 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 PCI-Express 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().*

이 논문에서 우리의 contribution은 무엇이다.
CUDA에서 ismatch 되는 부분에서 cuda용 memcpy를 implementation을 한 것, 유용하다는 것 이 논문에서 순서 얘기

레퍼런스 얘기하며 남들과는 다른게 kernel에 집중한 것이 아니라 cpu gpu memcpy에 집중하였다.

■ 관련연구

Choosing Between Pinned and Non-Pinned Memory [1]

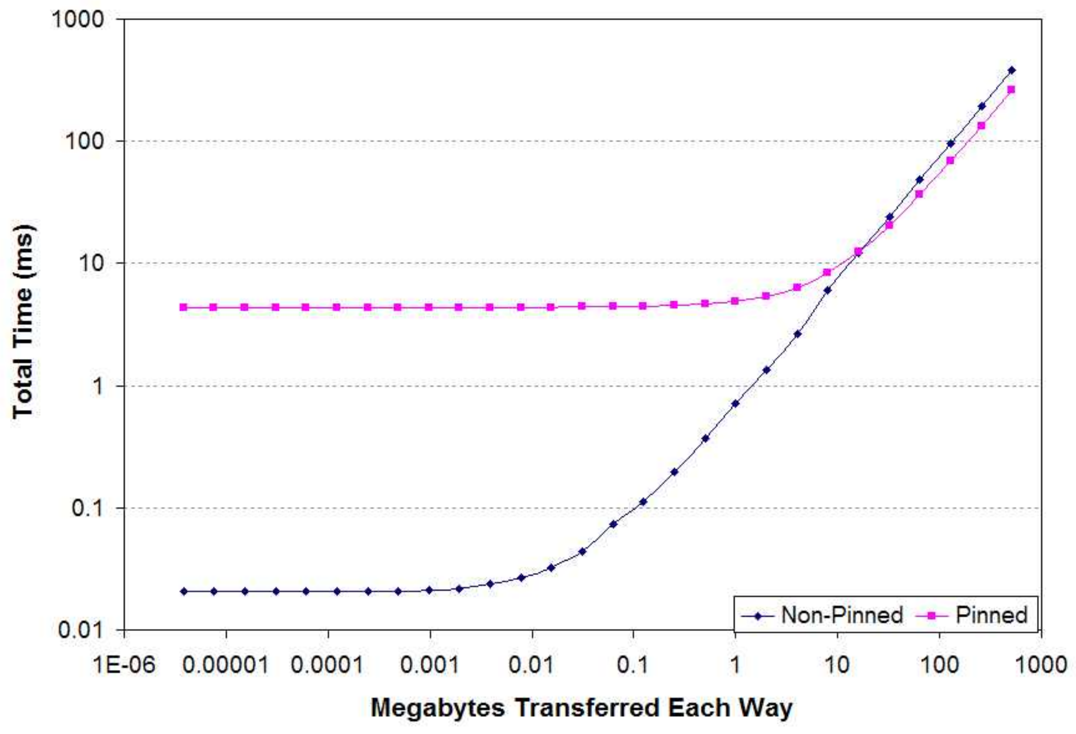


그림 2 Time required to allocate, transfer to the GPU, transfer back to the CPU, and deallocate pinned and non-pinned memory. Note the logarithmic scales.

mlock() [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

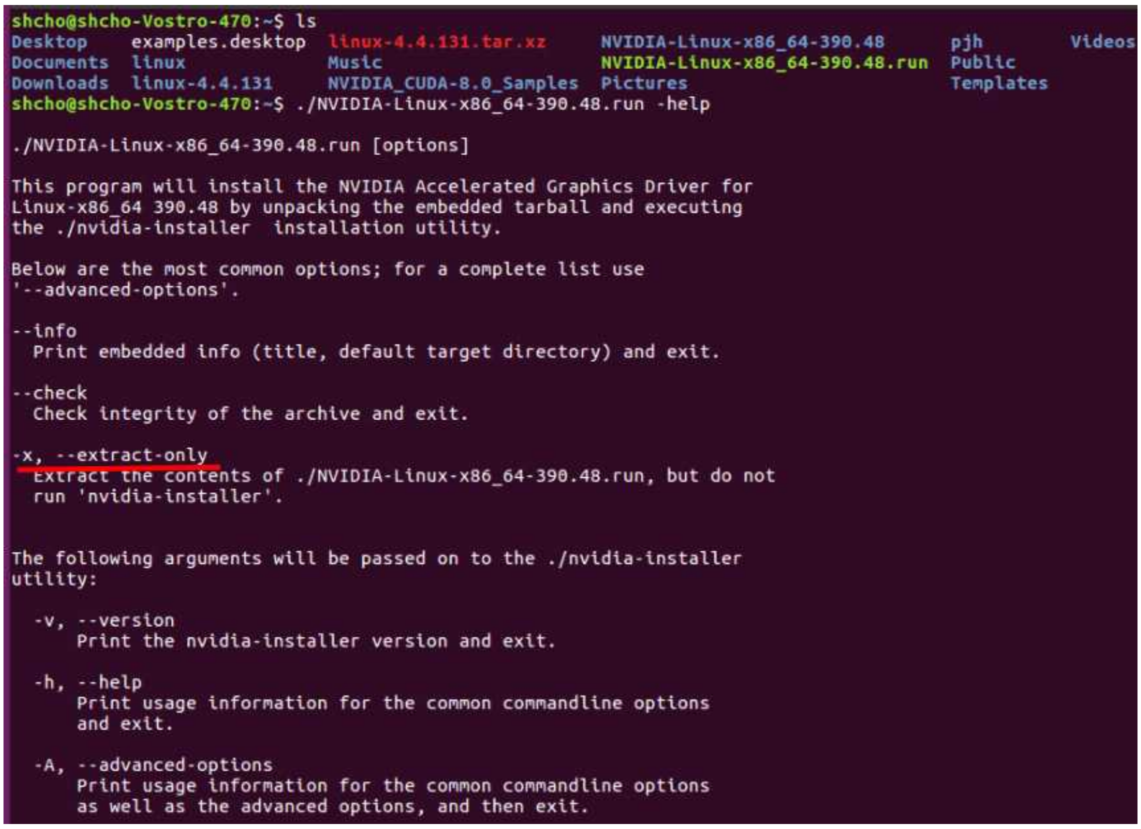
- You load the data exactly once, but you have a lot of computation to perform on it and you want to hide memory transfer latencies through it.
- The host side wants to change/add more data, or read the results, while kernel is still running (e.g. communication)
- The data does not fit into GPU memory

Note that, you can also use multiple streams to copy data and run kernels in parallel.

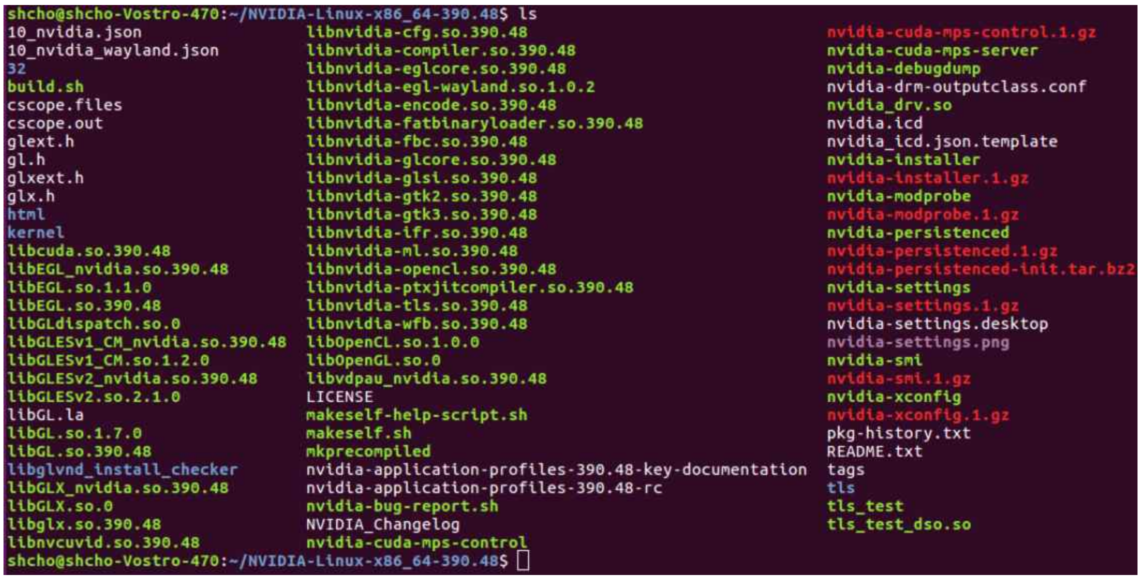
Pinned, but not mapped memory is better:

- 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

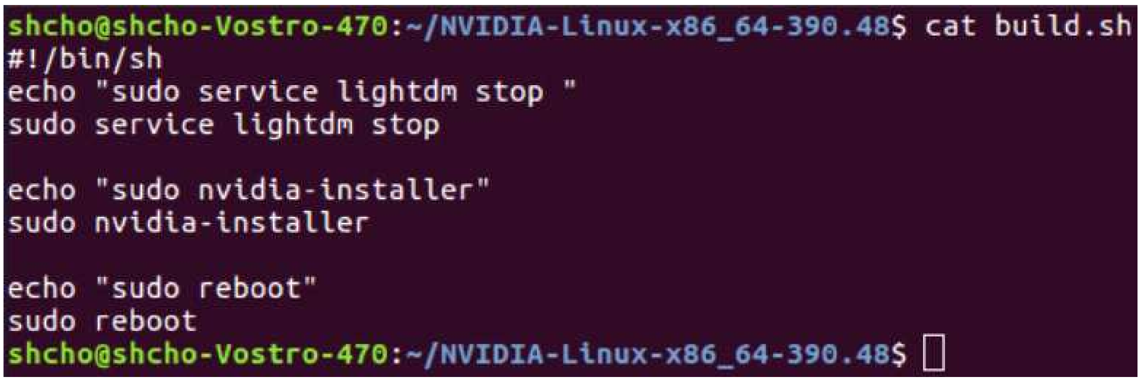
Nvidia Driver License with Linux kernel



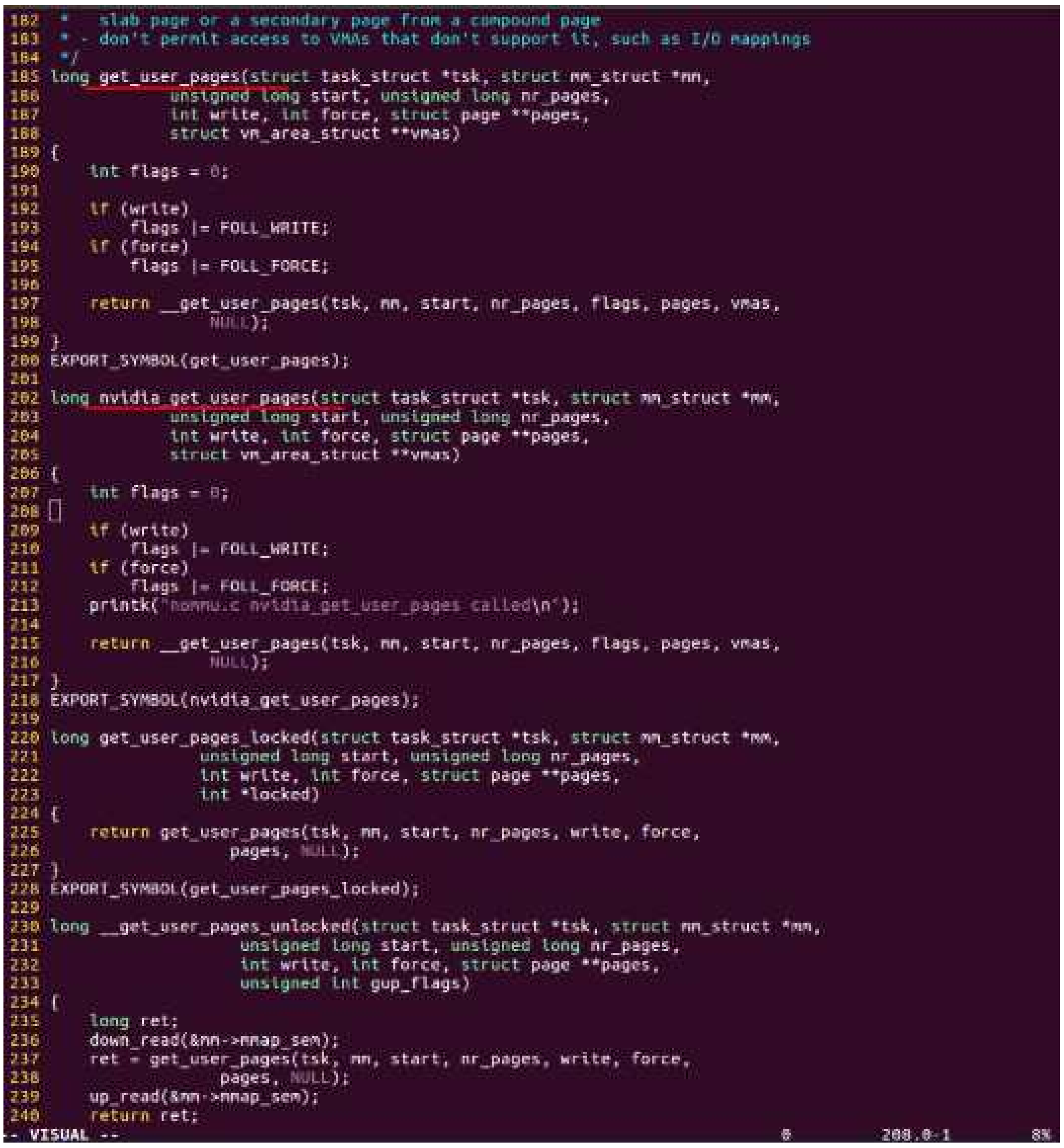
```
shcho@shcho-Vostro-470:~$ ls
Desktop    examples.desktop  linux-4.4.131.tar.xz     NVIDIA-Linux-x86_64-390.48      pjh       Videos
Documents  linux             Music                    NVIDIA-Linux-x86_64-390.48.run  Public
Downloads  linux-4.4.131     NVIDIA_CUDA-8.0_Samples  Pictures         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



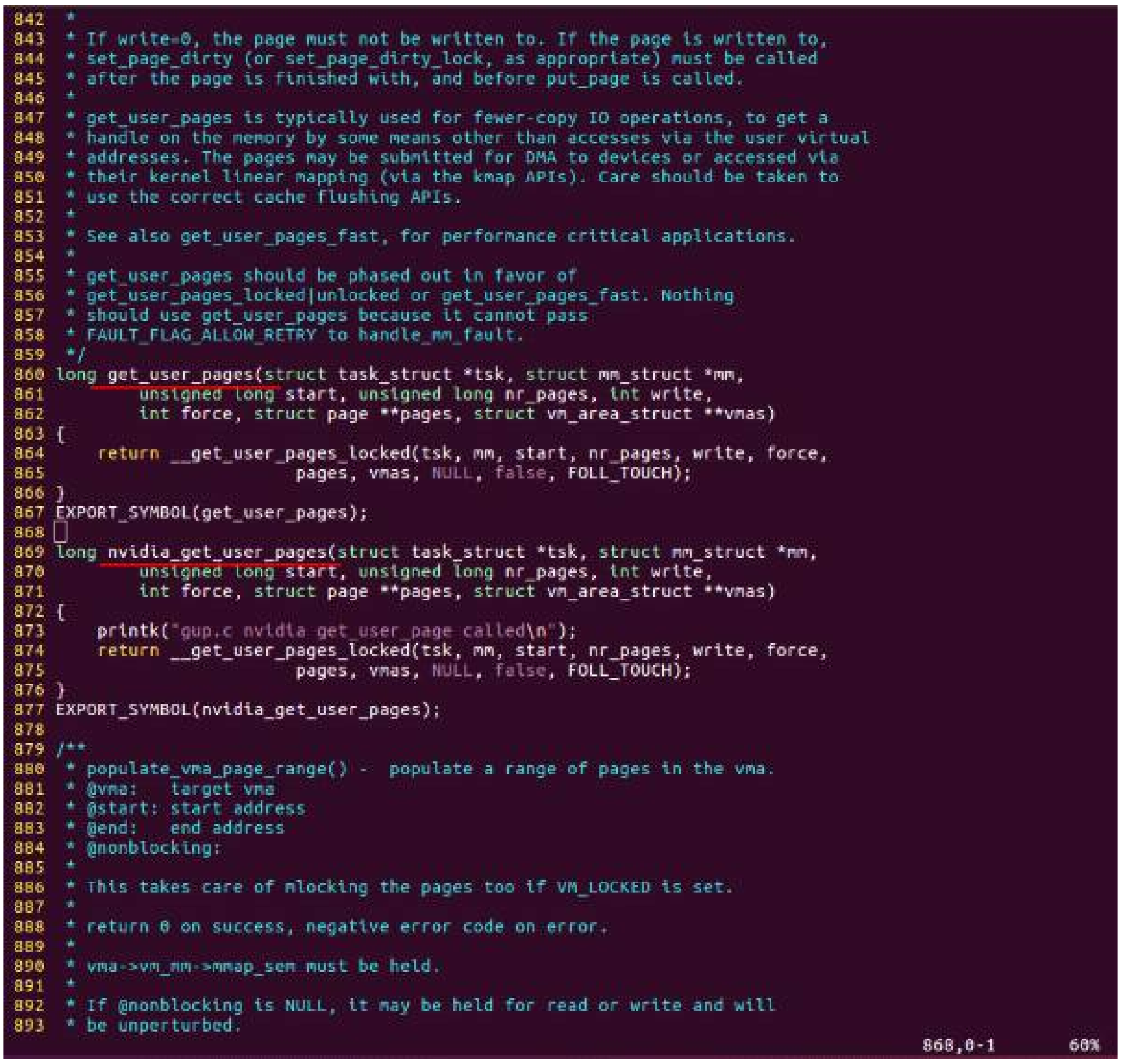
```
shcho@shcho-Vostro-470:~/NVIDIA-Linux-x86_64-390.48$ ls
10_nvidia.json                 libnvidia-cfg.so.390.48                               nvidia-cuda-mps-control.1.gz
10_nvidia_wayland.json         libnvidia-compiler.so.390.48                          nvidia-cuda-mps-server
32                             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$
```



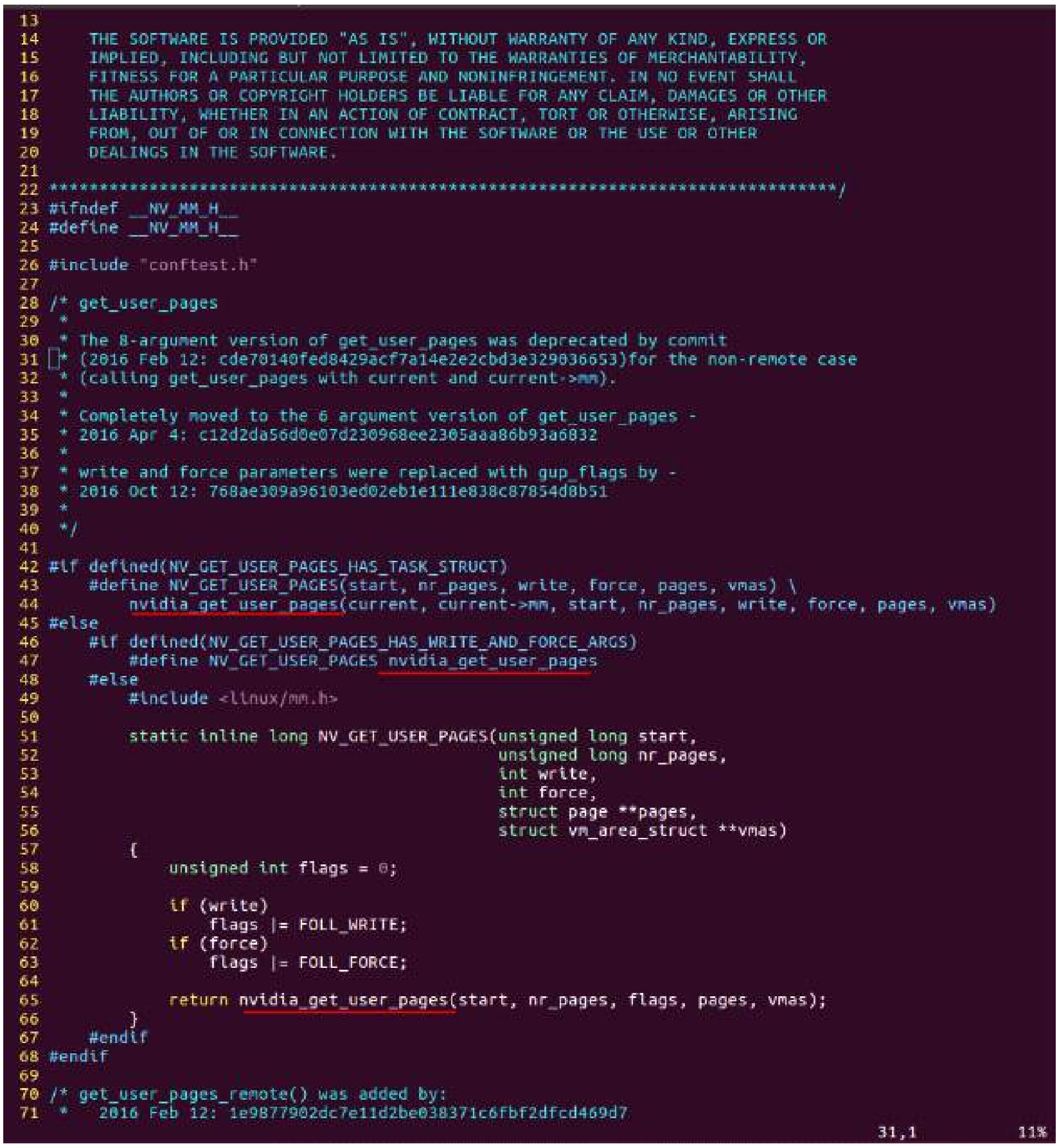
```
shcho@shcho-Vostro-470:~/NVIDIA-Linux-x86_64-390.48$ cat build.sh
#!/bin/sh
echo "sudo service lightdm stop "
sudo service lightdm stop

echo "sudo nvidia-installer"
sudo nvidia-installer

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



```
 *    slab page or a secondary page from a compound page
 * - don't permit access to VMAs that don't support it, such as I/O mappings
 */
long get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
            unsigned long start, unsigned long nr_pages,
            int write, int force, struct page **pages,
            struct vm_area_struct **vmas)
{
    int flags = 0;

    if (write)
        flags |= FOLL_WRITE;
    if (force)
        flags |= FOLL_FORCE;

    return __get_user_pages(tsk, mm, start, nr_pages, flags, pages, vmas,
                NULL);
}
EXPORT_SYMBOL(get_user_pages);

long nvidia_get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
            unsigned long start, unsigned long nr_pages,
            int write, int force, struct page **pages,
            struct vm_area_struct **vmas)
{
    int flags = 0;

    if (write)
        flags |= FOLL_WRITE;
    if (force)
        flags |= FOLL_FORCE;
    printk("nommu.c nvidia_get_user_pages called\n");

    return __get_user_pages(tsk, mm, start, nr_pages, flags, pages, vmas,
                NULL);
}
EXPORT_SYMBOL(nvidia_get_user_pages);

long get_user_pages_locked(struct task_struct *tsk, struct mm_struct *mm,
                unsigned long start, unsigned long nr_pages,
                int write, int force, struct page **pages,
                int *locked)
{
    return get_user_pages(tsk, mm, start, nr_pages, write, force,
                  pages, NULL);
}
EXPORT_SYMBOL(get_user_pages_locked);

long __get_user_pages_unlocked(struct task_struct *tsk, struct mm_struct *mm,
                    unsigned long start, unsigned long nr_pages,
                    int write, int force, struct page **pages,
                    unsigned int gup_flags)
{
    long ret;
    down_read(&mm->mmap_sem);
    ret = get_user_pages(tsk, mm, start, nr_pages, write, force,
                pages, NULL);
    up_read(&mm->mmap_sem);
    return ret;
-- VISUAL --                                                   0          208,0-1        8%
```



```
 *
 * If write=0, the page must not be written to. If the page is written to,
 * set_page_dirty (or set_page_dirty_lock, as appropriate) must be called
 * after the page is finished with, and before put_page is called.
 *
 * get_user_pages is typically used for fewer-copy IO operations, to get a
 * handle on the memory by some means other than accesses via the user virtual
 * addresses. The pages may be submitted for DMA to devices or accessed via
 * their kernel linear mapping (via the kmap APIs). Care should be taken to
 * use the correct cache flushing APIs.
 *
 * See also get_user_pages_fast, for performance critical applications.
 *
 * get_user_pages should be phased out in favor of
 * get_user_pages_locked|unlocked or get_user_pages_fast. Nothing
 * should use get_user_pages because it cannot pass
 * FAULT_FLAG_ALLOW_RETRY to handle_mm_fault.
 */
long get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
        unsigned long start, unsigned long nr_pages, int write,
        int force, struct page **pages, struct vm_area_struct **vmas)
{
    return __get_user_pages_locked(tsk, mm, start, nr_pages, write, force,
                       pages, vmas, NULL, false, FOLL_TOUCH);
}
EXPORT_SYMBOL(get_user_pages);

long nvidia_get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
        unsigned long start, unsigned long nr_pages, int write,
        int force, struct page **pages, struct vm_area_struct **vmas)
{
    printk("gup.c nvidia get_user_page called\n");
    return __get_user_pages_locked(tsk, mm, start, nr_pages, write, force,
                       pages, vmas, NULL, false, FOLL_TOUCH);
}
EXPORT_SYMBOL(nvidia_get_user_pages);

/**
 * populate_vma_page_range() -  populate a range of pages in the vma.
 * @vma:   target vma
 * @start: start address
 * @end:   end address
 * @nonblocking:
 *
 * This takes care of mlocking the pages too if VM_LOCKED is set.
 *
 * return 0 on success, negative error code on error.
 *
 * vma->vm_mm->mmap_sem must be held.
 *
 * If @nonblocking is NULL, it may be held for read or write and will
 * be unperturbed.
                                                        868,0-1       60%
```



```
    THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
    IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
    FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
    LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
    DEALINGS IN THE SOFTWARE.

*******************************************************************************/
#ifndef __NV_MM_H__
#define __NV_MM_H__

#include "conftest.h"

/* get_user_pages
 *
 * The 8-argument version of get_user_pages was deprecated by commit
 * (2016 Feb 12: cde70140fed8429acf7a14e2e2cbd3e329036653)for the non-remote case
 * (calling get_user_pages with current and current->mm).
 *
 * Completely moved to the 6 argument version of get_user_pages -
 * 2016 Apr 4: c12d2da56d0e07d230968ee2305aaa86b93a6832
 *
 * write and force parameters were replaced with gup_flags by -
 * 2016 Oct 12: 768ae309a96103ed02eb1e111e838c87854d8b51
 *
 */

#if defined(NV_GET_USER_PAGES_HAS_TASK_STRUCT)
    #define NV_GET_USER_PAGES(start, nr_pages, write, force, pages, vmas) \
        nvidia_get_user_pages(current, current->mm, start, nr_pages, write, force, pages, vmas)
#else
    #if defined(NV_GET_USER_PAGES_HAS_WRITE_AND_FORCE_ARGS)
        #define NV_GET_USER_PAGES nvidia_get_user_pages
    #else
        #include <linux/mm.h>

        static inline long NV_GET_USER_PAGES(unsigned long start,
                                             unsigned long nr_pages,
                                             int write,
                                             int force,
                                             struct page **pages,
                                             struct vm_area_struct **vmas)
        {
            unsigned int flags = 0;

            if (write)
                flags |= FOLL_WRITE;
            if (force)
                flags |= FOLL_FORCE;

            return nvidia_get_user_pages(start, nr_pages, flags, pages, vmas);
        }
    #endif
#endif

/* get_user_pages_remote() was added by:
 *   2016 Feb 12: 1e9877902dc7e11d2be038371c6fbf2dfcd469d7
                                                                  31,1          11%
```

■ 제안 작품 소개

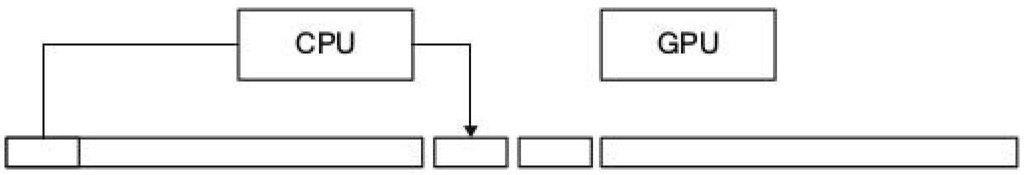
TECHNICAL BACKGROUND

Pinned memory

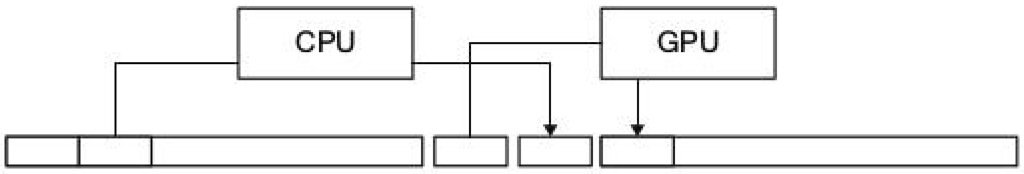
Pinned memory is allocated and freed with specific functions implemented by CUDA. We use cudaHostAlloc() to assign write-combined option to pinned memory. These functions work with the host operating system to allocate page-locked memory and map it for Direct Memory Access(DMA) by devices. The host operating system use kernel function get_user_pages() for page-locked memory. CUDA keeps track of pinned memory of itself and accelerates memory copies that involve host pointer allocated with cudaHostAlloc(). And the asynchronous memcpy functions need pinned memory. Pinned memory can reach maximum bandwidth because of DMA. But allocating pinned memory is expensive because it involves a considerable amount of work for the host and reduces the amount of physical memory available to the operating system and other programs.

Double buffering

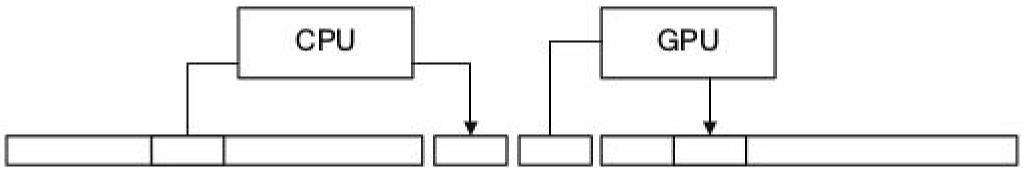
Because the device operates autonomously and can access the host memory without any operating system, only pinned memory is available for asynchronous memcpy between host and devices. The device cannot access pageable memory directly, the driver implements pageable memcpy with a pair of pinned buffers that are allocated with the CUDA context. To perform a host device memcpy, the driver first copies to one pinned buffer, then sends the request of a DMA operation to read that data with the device. While the device starts processing that request, driver copies another data in the other pinned buffer. The host and device keep flip-flopping between pinned buffers, with synchronization, until the device performs the final memcpy. And the host also naturally pages in any cached pages while the data is being copied.



1. “Prime the pump”: CPU copies to first staging buffer

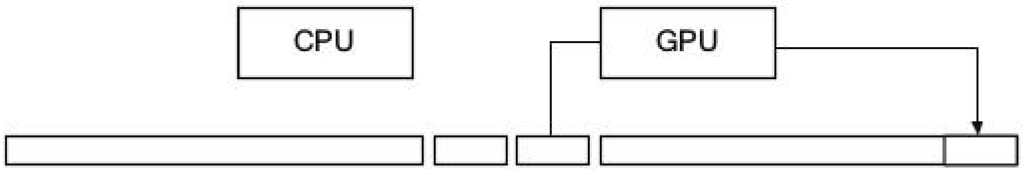


2. GPU pulls from first, while CPU copies to second

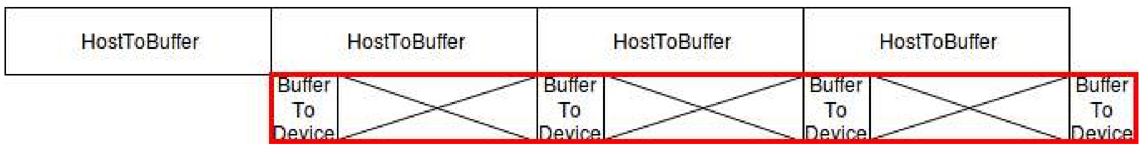


3. GPU pulls from second, while CPU copies to first

. . .



But this method can cause relatively slow data transfer speed if the speed of memcpy from the host to the buffer is slower than the speed of memcpy from the buffer to the device because there are significant delays among memcpys from the buffer to the device.



Write-combined memory

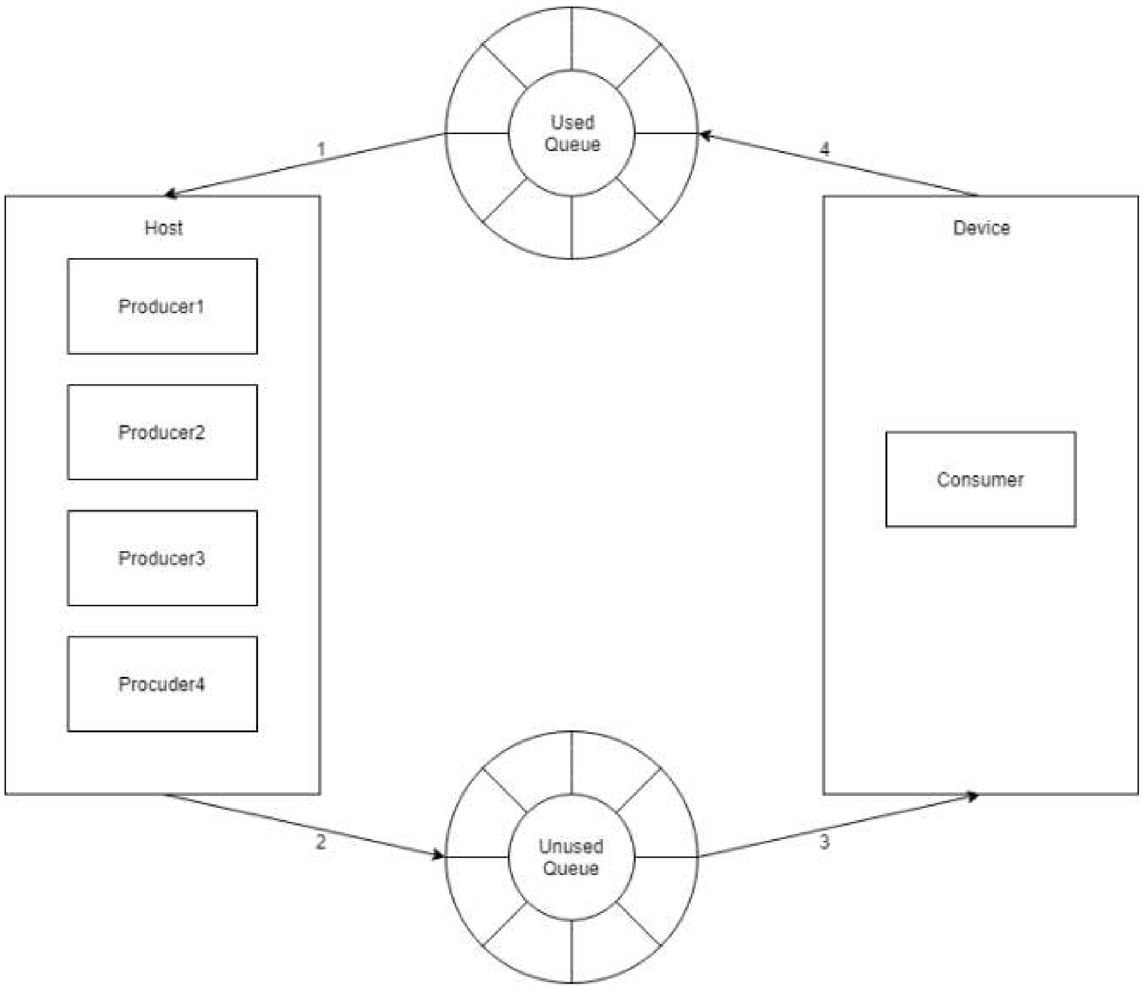
Write-combined memory or uncacheable write-combining memory was created to allow the host to write to device frame buffers fast and no pollution of the host cache. For CUDA, write-combined memory can be allocated by calling cudaHostAlloc() with the cudaHostWriteCombined flag. Besides setting the page table entries to bypass the host caches, write-combined memory also is note snooped during PCI Express bus transfers. Reading WC memory with the host is 6x slower than normal memory but this problem can be circumvented by using MOVNTDQA instruction that is brand new with SSE4.1. NVIDIA implements memcpy function using MOVNIDQA instruction to accelerate memcpy from WC memory to normal memory for device host transfer of gdrcopy(A low-latency GPU memory copy library based on NVIDIA GPUDirect RDMA technology)

EXPERIMENTAL AND COMPUTATIONAL DETAILS

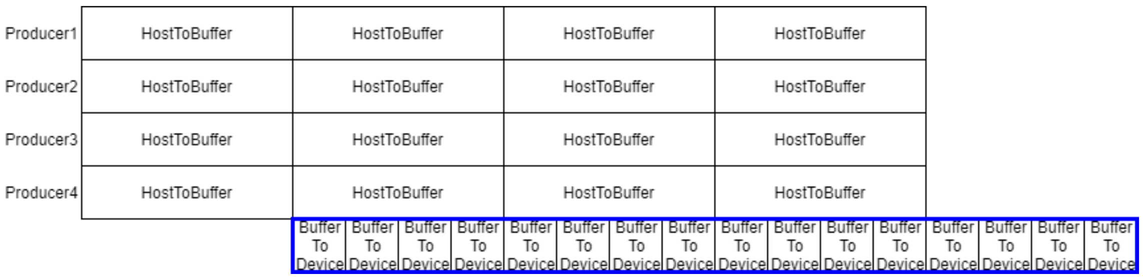
Double queue

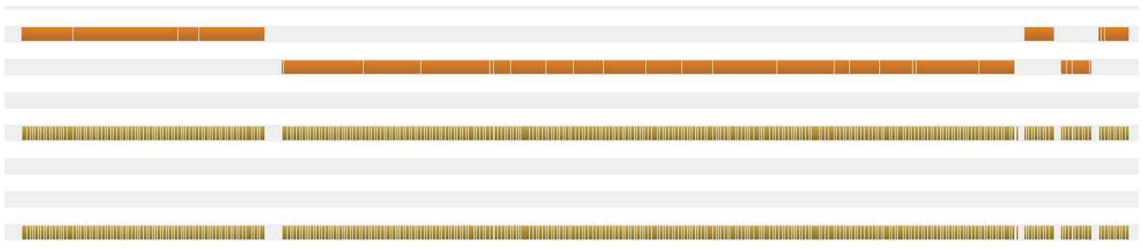
Transferring data between host and device can be transposed to producer-consumer problem. Multiple producers copy data from host memory to the buffer concurrently by multithreading while a consumer copies data of buffer to the device. Numerous producers are required over one consumer to fill the chasm between PCI-Express bandwidth and memcpy bandwidth in cpu memory. The number of buffers is set double of the number of producers, like double buffering. Used queue contains the addresses of the empty buffers and unused queue contains the addresses of filled buffer and index of data. The size of the used and unused queue equates the number of buffers. Operating logic of double queue method follows four steps.

Step one producers get the address of an empty buffer from the used queue and copy data from host to the buffer. Step two producers store the address of filled buffer and index of data in the unused queue. Step three consumer gets the address of filled buffer and index of data from the unused queue and copy data of buffer to a device with the address of buffer and index of data. Step four consumer puts the address of the empty buffer in the used queue. Steps from one to four must be repeated until all data is transfer from the host to the device. Step one and two can be done by multiple producers concurrently but step 3 and 4 are done by one consumer.



Because producers copy data of host to buffer concurrently and used and unused queue guarantee first in first out, the consumer can copy data of buffer to the device without any delay.





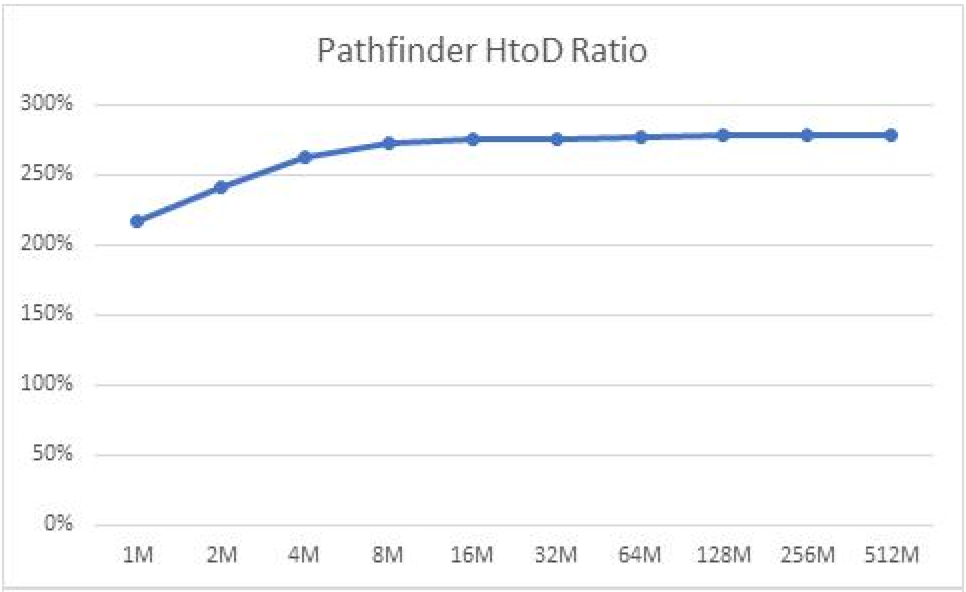
But there are two conditions to reach maximum utilization level of PCI-Express bandwidth. These two conditions are also applied to double buffering.

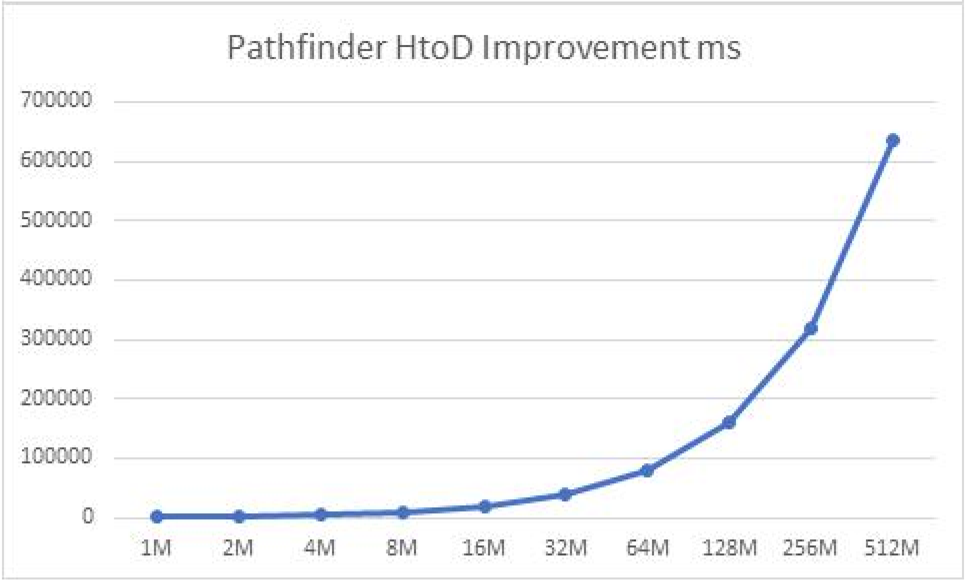
1. The speed of copying data of buffer to device <= The number of producers * The speed of copying data of host to buffer.
2. The speed of copying data of buffer to device + 2 * The number of producers * the speed of copying data of host to buffer <= The maximum bandwidth of host's main memory.

■ 구현 및 결과분석

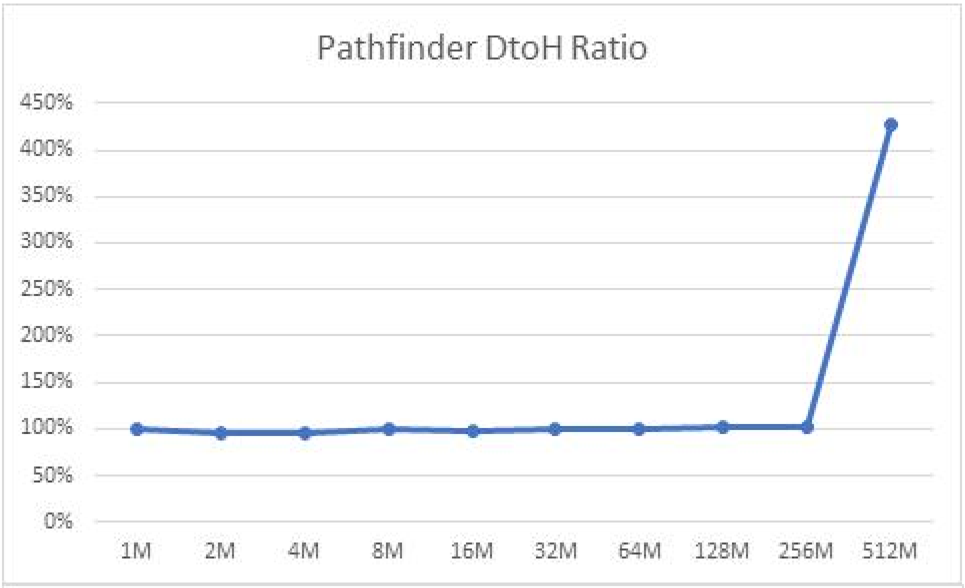
<u>영여 논문 결과 부분 복붙</u>

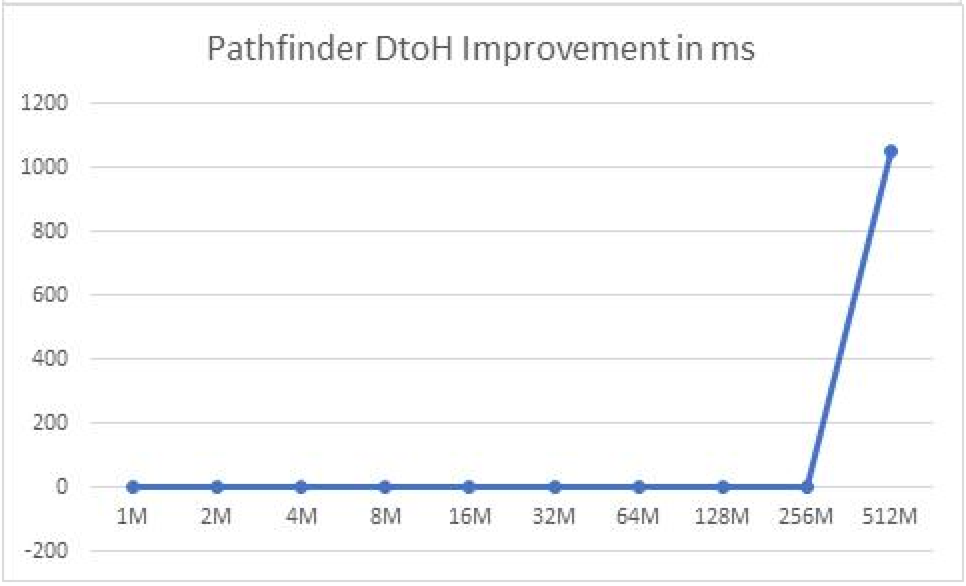
	1M	2M	4M	8M	16M	32M	64M	128M	256M	512M
HtoD Ratio	2.173411	2.410626	2.621692	2.723439	2.745902	2.757213	2.769511	2.78624	2.787276	2.782014
	1M	2M	4M	8M	16M	32M	64M	128M	256M	512M
Difference	1016.35	2160.98	4652.91	9649.64	19449	39119.6	78592.3	158732.8	317742	635436



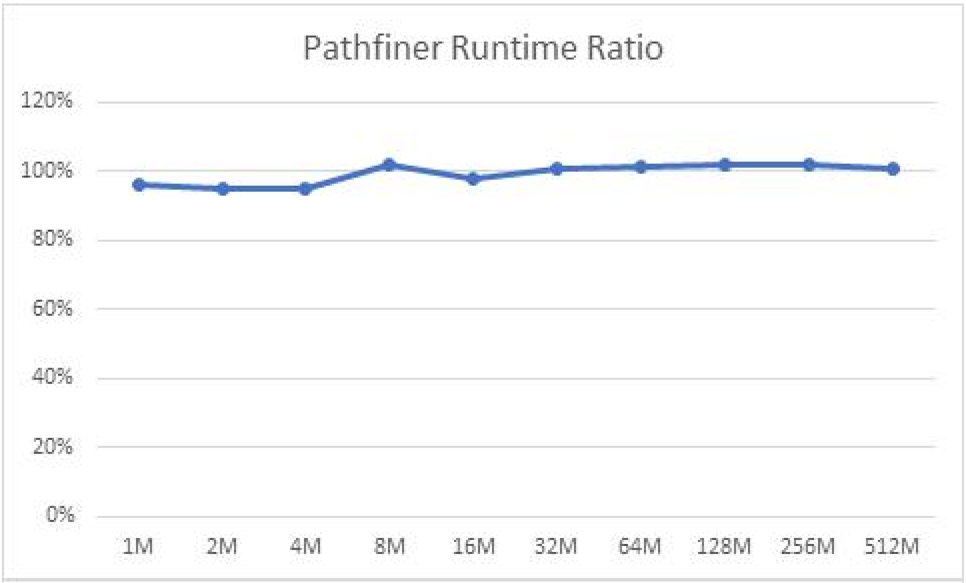


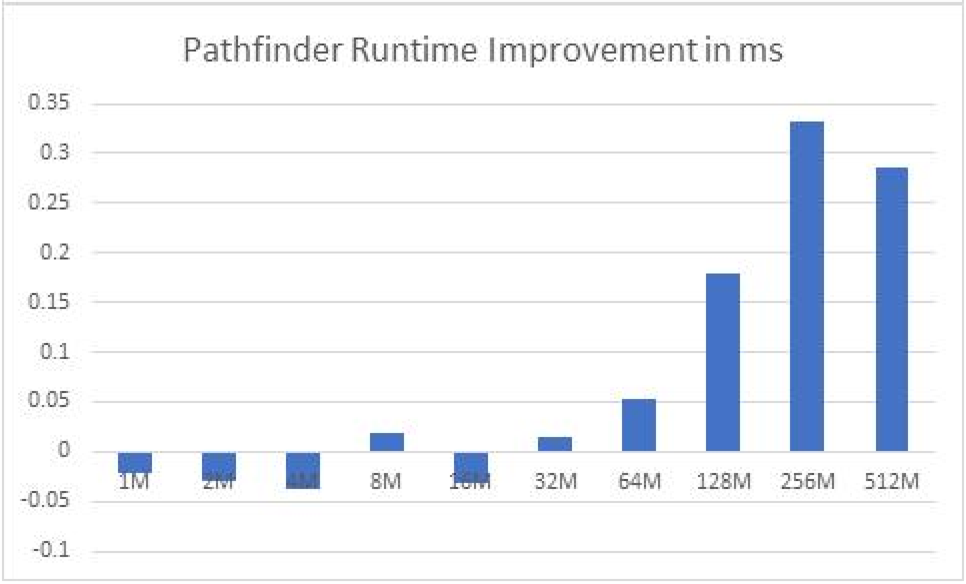
	1M	2M	4M	8M	16M	32M	64M	128M	256M	512M
DtoH Ratio	0.994227	0.958177	0.954209	0.990829	0.983162	0.99857	1.00007	1.012714	1.015127	4.268378
	1M	2M	4M	8M	16M	32M	64M	128M	256M	512M
Difference	-0.0353	-0.4802	-0.9825	-0.3778	-0.6976	-0.1157	0.0058	2.056	2.444	1049.035

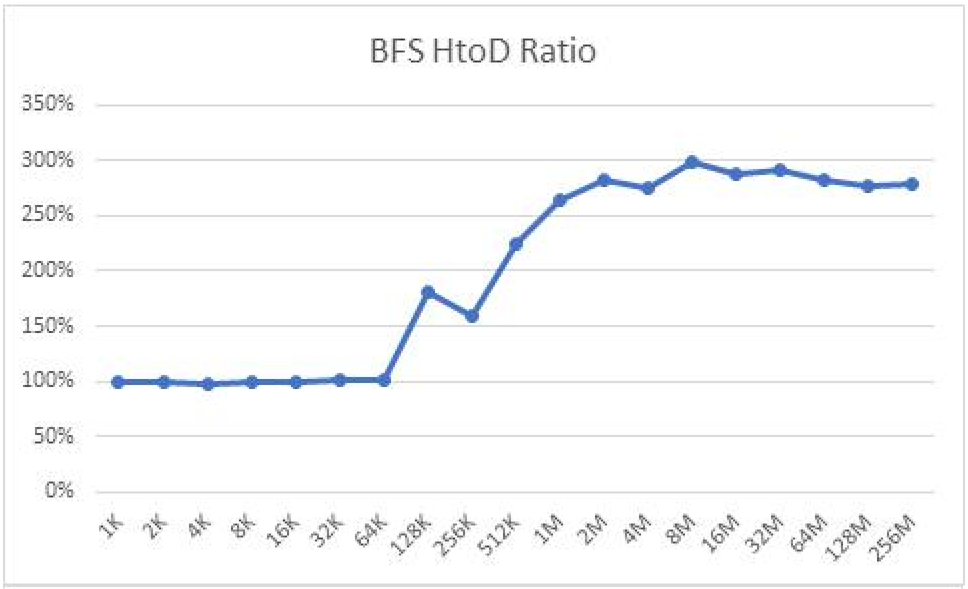


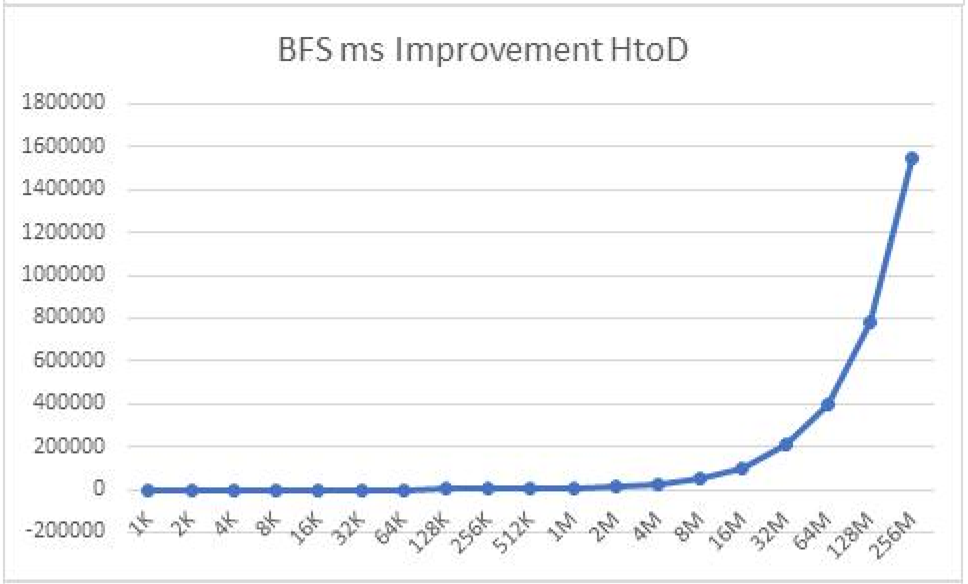


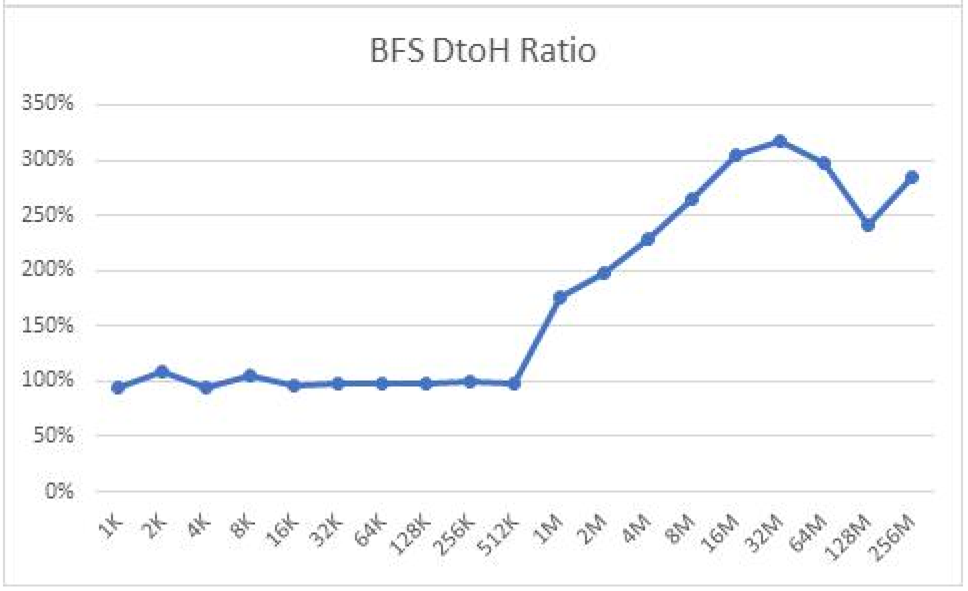
	1M	2M	4M	8M	16M	32M	64M	128M	256M	512M
Runtime Ratio	0.960795	0.951946	0.947811	1.019809	0.978423	1.006378	1.012485	1.022529	1.021518	1.009309
	1M	2M	4M	8M	16M	32M	64M	128M	256M	512M
Difference	-0.02111	-0.02884	-0.03775	0.018421	-0.03081	0.014963	0.052537	0.178996	0.332351	0.286782

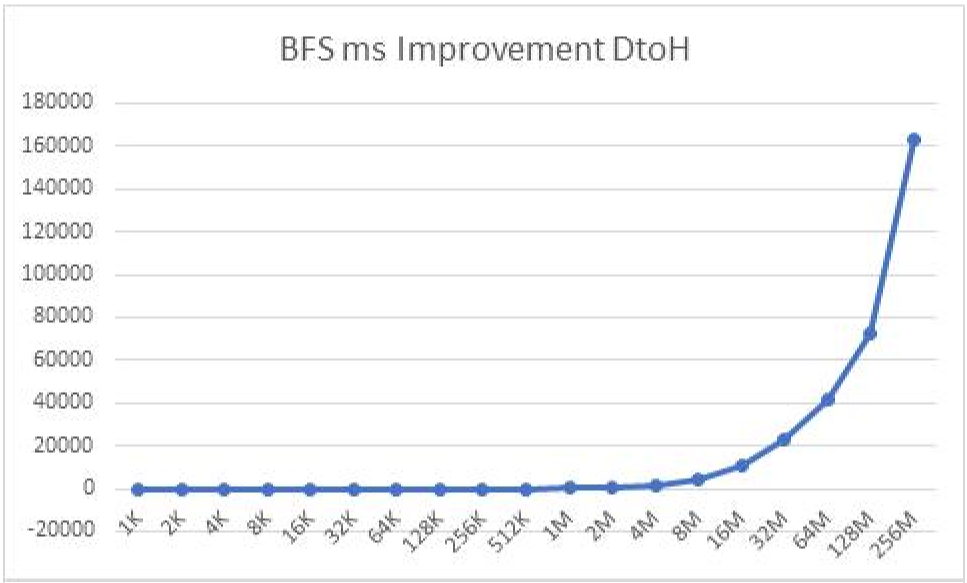


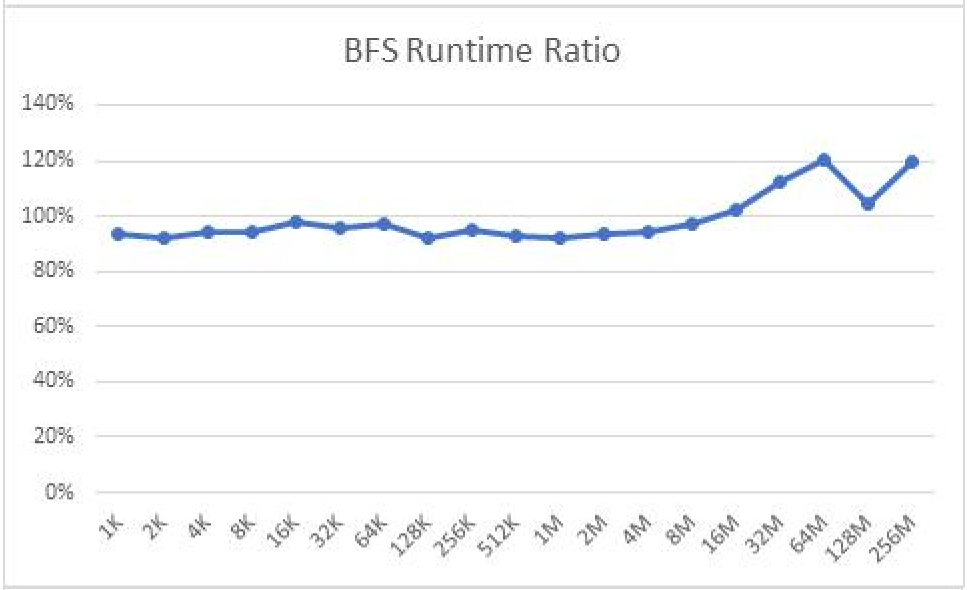


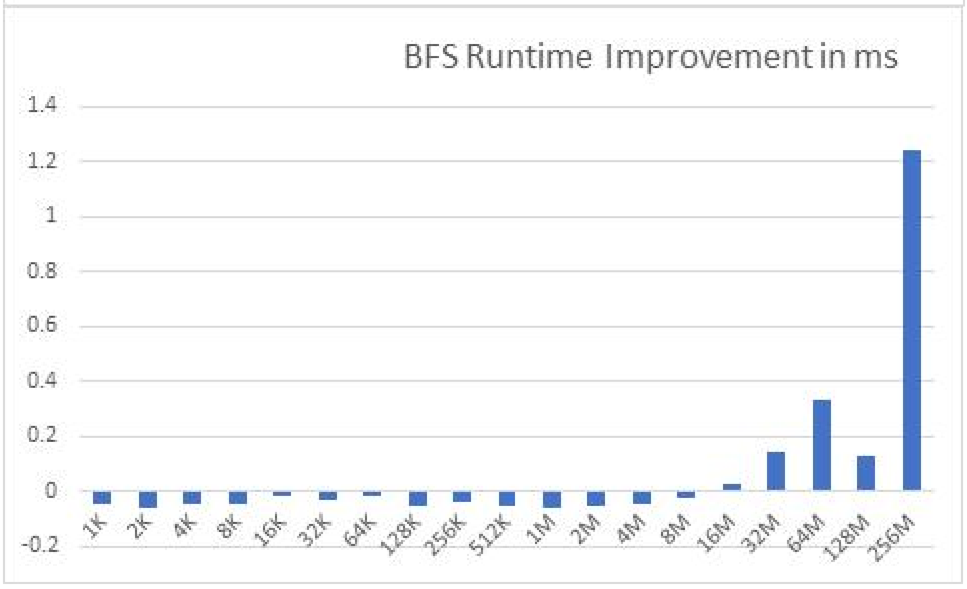

BFS HtoD Ratio
350%
300%
250%
200%
150%
100%
50%
0%
1K
2K
4K
8K
16K
32K
64K
128K
256K
512K
1M
2M
4M
8M
16M
32M
64M
128M
256M


BFS ms Improvement HtoD
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
-200000
1K
2K
4K
8K
16K
32K
64K
128K
256K
512K
1M
2M
4M
8M
16M
32M
64M
128M
256M


BFS DtoH Ratio
350%
300%
250%
200%
150%
100%
50%
0%
1K
2K
4K
8K
16K
32K
64K
128K
256K
512K
1M
2M
4M
8M
16M
32M
64M
128M
256M


BFS ms Improvement DtoH
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
-20000
1K
2K
4K
8K
16K
32K
64K
128K
256K
512K
1M
2M
4M
8M
16M
32M
64M
128M
256M


BFS Runtime Ratio
140%
120%
100%
80%
60%
40%
20%
0%
1K
2K
4K
8K
16K
32K
64K
128K
256K
512K
1M
2M
4M
8M
16M
32M
64M
128M
256M


BFS Runtime Improvement in ms
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
1K
2K
4K
8K
16K
32K
64K
128K
256K
512K
1M
2M
4M
8M
16M
32M
64M
128M
256M

■ 결론 및 소감

기존 데이터 전송 로직은 pcie의 최대 대역폭을 모두 사용하지 못했다. 그 이유는 호스트와 버퍼간의 속도가 버퍼와 디바이스의 속도보다 느리기 때문이었다. 이 문제는 호스트에서 버퍼로 복사하는 것을 생산자 버퍼에서 디바이스로 복사하는 것을 소비자라 간주하면 다중 생산자와 단일 소비자 문제로 치환이 가능하고 이는 더블 큐 기법을 사용하여 구현이 가능하며 데이터 전송 속도 측면에서 많은 개선을 보여주었다. 개선사항으로는 cudaMemcpyFixed가 호출될 때 마다 생산자 쓰레드와 소비자ㅣ 쓰레드가 생성되기에 그 생성시간이 오버헤드로 존재했는데 이는 데이터의 크기가 작은 경우 기존의 기법으로 우회하거나 아님 생산자 쓰레드와 소비자 쓰레드를 항상 백그라운드에 상주하게 해서 cudaMemcpyFixed가 호출될때마다 생성할 필요가 없게 해 오버헤드를 줄일 수 있을 것이다.

■ 참고문헌

[1] Choosing Between Pinned and Non-Pinned Memory, https://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.htm

[2] Locking and pinning, https://lwn.net/Articles/600502/

[3] Difference between cudaMallocManaged and zero copy memory function, https://devtalk.nvidia.com/default/topic/1030507/cuda-programming-and-performance/difference-between-cudamallocmanaged-and-zero-copy-memory-function/

[4] David B. Kirk, Wen-mei W. Hwu, "Programming Massively Parallel Processors: A Hands-on Approach", Morgan Kaufmann, 7th December 2016

[5] Default Pinned Memory Vs Zero-Copy Memory, https://stackoverflow.com/questions/5209214/default-pinned-memory-vs-zero-copy-memory