CSC3150 Project 3 Report

1. Program Environment

Operating system: Windows10

IDE: Visual Studio 2017

CUDA Version: CUDA-9.0

GPU Info: GeForce GTX 1060

2. Execution Steps

1. Open Visual Studio 2017, then open a new project with CUDA
2. Replace all .cu files in your project with my files
3. Use Ctrl+F7 to compile all .cu files, and press Ctrl+F5 to run the program

3. Program Design

Operating System Concept	Abstract Data Type Representation
Logical Memory Address (128KB)	input_size (int [131072])
Inverted Page Table (16KB)	vm->invert_page_table (int [2048])
Main Memory / Physical Memory (32KB)	vm->buffer (int [32768])
Disk / Secondary Memory (128KB)	vm->storage (int [131072])
Page Number Stack	vm->LRU_head (struct Page_node *)

Note:

1. that every 32 entries in ADT of main/secondary memory represents a page, and each entry represents data of 1 byte

2. The page table actually only uses the first 1024 entries, with the remaining used for bonus

3. The index of page table represents the page's address in main memory, and the value indicates its corresponding logical address

4. Since the page table is an inverted page table, it only keeps pages that are available in the main memory. It saves space but cost more time to determine whether the demanded page number is in the table

The core functions of my program consist of:

```
void init_invert_page_table(struct VirtualMemory *vm);
```

Initialize the page table, with all entries set to -1

```
void vm_init(struct VirtualMemory *vm, uchar *buffer, uchar *storage...);
```

Initialize the virtual memory, as well as the stack storing the information of least recent used page, implemented by a linked list.

```
int get_LRUidx(struct VirtualMemory *vm);
```

Obtain the least recent used index (main memory address) of the LRU page, extracting this information from the Page Number Stack. The least recent used node is always the first node.

```
int search_pageidx(struct VirtualMemory *vm, int page_num);
```

Iterate through the page table to find the index of a page number. If found, then this page is available, and page fault if not found.

```
void update_stack(struct VirtualMemory *vm, int idx);
```

Each time the program accesses a page, the corresponding node should be moved to the top of the linked list to indicate that it is the most recent used one. Therefore, this function updates the page number stack to keep this information

```
void vm_write(struct VirtualMemory *vm, u32 addr, uchar value);
```

Write the data into the main memory by locating the correct main memory address via page table. If the demanding page number is in the page table, then extract the corresponding main memory address (index of the inverted page table) as well as write data to this location of main memory. If not, then page fault. In this case the program need to swap out the data in main memory to disk, then swap in the page residing in the disk into the main memory, and perform write operation. At the same time, the page table and page number stace both need to be updated.

```
uchar vm_read(struct VirtualMemory *vm, u32 addr);
```

Redd the data from the main memory by locating the correct main memory address via page table. If the demanding page number is in the page table, then extract the corresponding main memory address (index of the inverted page table) as well as read data from this location of main memory. If not, then page fault. In this case the program need to swap out the data in main memory to disk, then swap in the page residing in the disk into the main memory, and perform read operation. At the same time, the page table and page number stace both need to be updated.

```
void vm_snapshot(struct VirtualMemory *vm, uchar *results, int offset, int input_size);
```

Basically extract all the information of the input binary file.

4. Page Fault Number

The page fault number is 4096 for write operation, 1 for read operation, and 4096 for snapshot operation. 8193 page fault in total.

The first 4096 page faults are due to write operation never write the data already in the main memory more than once, therefore each time start write a new page of data, a page fault occurs.

The 1 page fault caused by read operation is because the amount data read is 1 byte more than the capacity of the page table, with the first 32768 byte being the last 32768 byte written into the main memory. Therefore, the excess 1 byte cause 1 page fault.

The last 4096 page faults is because of snapshot operation. In this step, the program reads all data from the lowest address of virtual memory to the highest address of virtual memory. However, the data storing in the main memory at that time is the data in the highest position of the logical memory, so page faults occurs. Snapshot never reads the same data twice, so it causes 4096 page faults – equals the total pages of the actual data.

5. Problems Encountered

CUDA is a piece of shit to be frank, you cannot even use debug function in kernel functions. Indeed you can use Nsight to debug but this is supposed to be taught before we start to get fucked in this assignment. I wrote a program in plain C at first, after I make sure that my algorithm works I modified it into a CUDA program, so I could use debug function of my IDE when I was writing C program.

6. Screenshots

Appended in the end of my report

7. Things Learned

Obviously I learned the working principles of memory management. In addition, I also learned what a pain in my ass it is to write something you have absolute no idea before, i.e. CUDA, as well as how suck Visual Code is. I still prefer Clion, by the way.

8. Bouns

- launch 4 threads in kernel function:

```
mykernel<<<1,4, INVERT_PAGE_TABLE_SIZE>>>(input_size);
```

- Avoid the race condition:

```
int fuck = i + thread_id * (input_size / 4);
// printf("thread id %d, fuck is %d\n", thread_id, fuck);

if (thread_id == 0) {
    vm_write(vm, fuck, input[fuck], thread_id);
    }
    __syncthreads();

    if (thread_id == 1) {
        vm_write(vm, fuck, input[fuck], thread_id);
    }
    __syncthreads();

    if (thread_id == 2) {
        vm_write(vm, fuck, input[fuck], thread_id);
    }
    __syncthreads();

    if (thread_id == 3) {
        vm_write(vm, fuck, input[fuck], thread_id);
    }
    __syncthreads();
```

Therefore, for threads of different id, they will execute write (same for read) operation once, and wait for synchronization triple times. The execution order is 0-1-2-3 since they wait for other threads following this order.

- Modify your paging mechanism

```
__device__ void init_invert_page_table(VirtualMemory *vm) {

    for (int i = 0; i < 1024; i++) {
        vm->invert_page_table[i] = -1; // invalid := MSB is 1
        vm->invert_page_table[i + 1024] = i%4;
    }
}
```

Now the first 1024 entries hold the logical memory address of each page, while the last 1024 entries denote which thread this page belongs to.

```
    if (search_result == -1 || vm->invert_page_table[search_result+1024] != thread_id) { //
```

If the index (page in main memory) does not exist, or it exists but belongs to other threads, a page fault will be invoked.

```
    // update page table
    vm->invert_page_table[LRU_idx] = page_num;
    vm->invert_page_table[LRU_idx + 1024] = thread_id;
```

As the page table is updated, update which thread it belongs to as well.

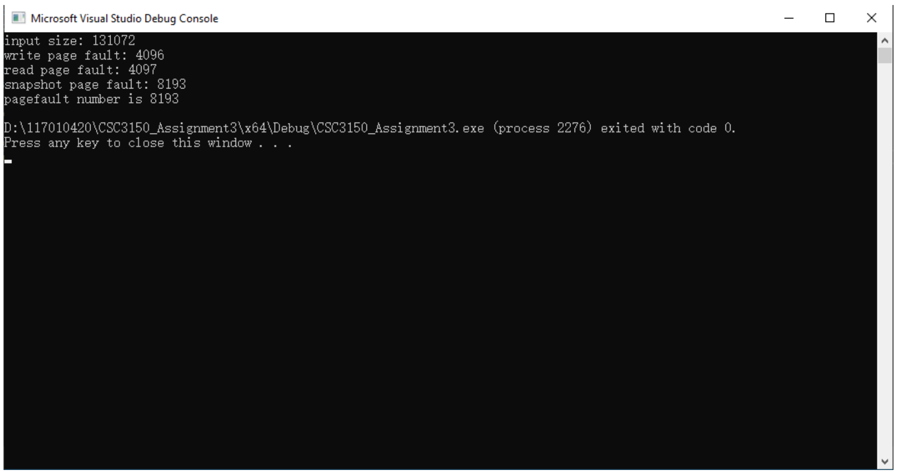
- Print the times of page fault of whole system

Appended in the end

- Correctly dump the contents to "snapshot.bin"

See the zip folder

Normal result screenshot:

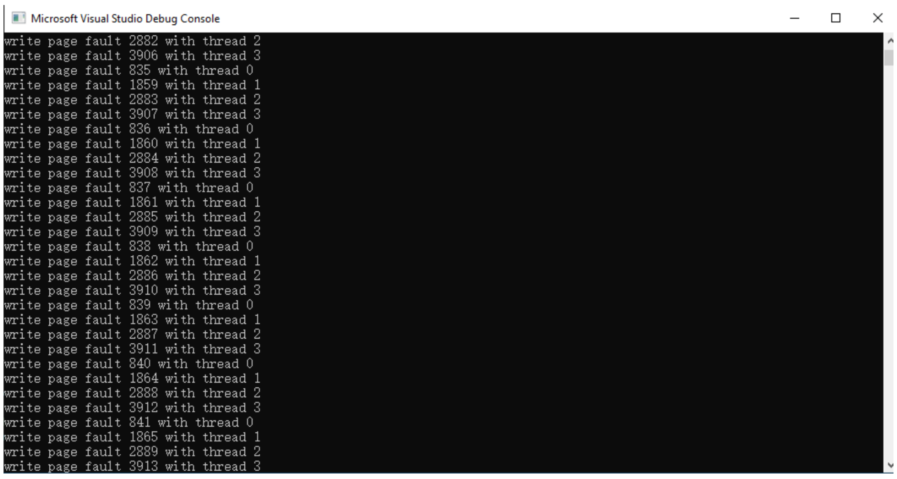


Microsoft Visual Studio Debug Console

```
input size: 131072
write page fault: 4096
read page fault: 4097
snapshot page fault: 8193
pagefault number is 8193

D:\117010420\CSC3150_Assignment3\x64\Debug\CSC3150_Assignment3.exe (process 2276) exited with code 0.
Press any key to close this window . . .
```

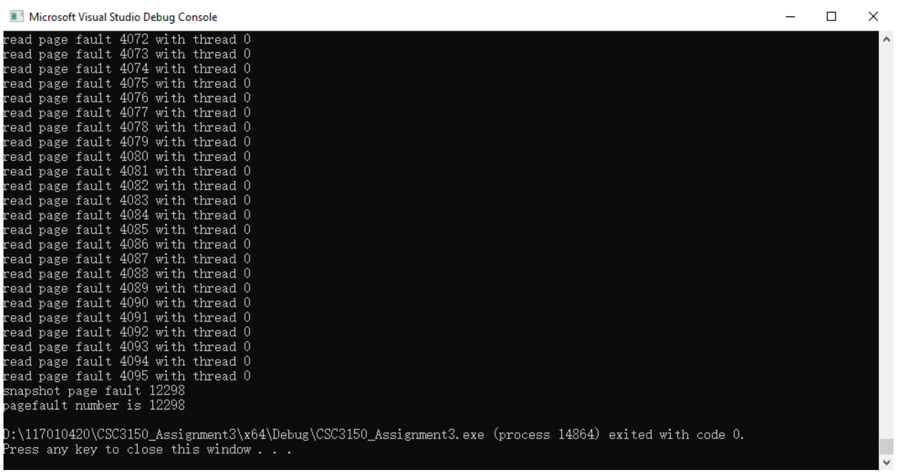
Bonus multi-thread screenshot



Microsoft Visual Studio Debug Console

```
write page fault 2882 with thread 2
write page fault 3906 with thread 3
write page fault 835 with thread 0
write page fault 1859 with thread 1
write page fault 2883 with thread 2
write page fault 3907 with thread 3
write page fault 836 with thread 0
write page fault 1860 with thread 1
write page fault 2884 with thread 2
write page fault 3908 with thread 3
write page fault 837 with thread 0
write page fault 1861 with thread 1
write page fault 2885 with thread 2
write page fault 3909 with thread 3
write page fault 838 with thread 0
write page fault 1862 with thread 1
write page fault 2886 with thread 2
write page fault 3910 with thread 3
write page fault 839 with thread 0
write page fault 1863 with thread 1
write page fault 2887 with thread 2
write page fault 3911 with thread 3
write page fault 840 with thread 0
write page fault 1864 with thread 1
write page fault 2888 with thread 2
write page fault 3912 with thread 3
write page fault 841 with thread 0
write page fault 1865 with thread 1
write page fault 2889 with thread 2
write page fault 3913 with thread 3
```

Bonus result screenshot



Microsoft Visual Studio Debug Console

```
read page fault 4072 with thread 0
read page fault 4073 with thread 0
read page fault 4074 with thread 0
read page fault 4075 with thread 0
read page fault 4076 with thread 0
read page fault 4077 with thread 0
read page fault 4078 with thread 0
read page fault 4079 with thread 0
read page fault 4080 with thread 0
read page fault 4081 with thread 0
read page fault 4082 with thread 0
read page fault 4083 with thread 0
read page fault 4084 with thread 0
read page fault 4085 with thread 0
read page fault 4086 with thread 0
read page fault 4087 with thread 0
read page fault 4088 with thread 0
read page fault 4089 with thread 0
read page fault 4090 with thread 0
read page fault 4091 with thread 0
read page fault 4092 with thread 0
read page fault 4093 with thread 0
read page fault 4094 with thread 0
read page fault 4095 with thread 0
snapshot page fault 12298
pagefault number is 12298

D:\117010420\CSC3150_Assignment3\x64\Debug\CSC3150_Assignment3.exe (process 14864) exited with code 0.
Press any key to close this window . . .
```

See the zip folder