Report

Li Huayue 118010138

1 Environment of running the program:

The environment of running the program is the same as the computers in TC301 which is shown as follows:

OS: Windows 10

VS version: VS2017

CUDA version: CUDA 9.2

GPU information: NVIDIA Geforce GTX 1060

Compute capacity: 6.1

2 The execution steps of running the program:

The execution steps of running the program in corresponding environment are shown as follows:

(1) Launch VS2017 and set the environment for CUDA programming following the configuration steps in "Assign3_setup.pdf" on blackboard

(2) Click "File→Open→Project/Solution…" in order and then find the "CSC3150_A3.sln" and click it

(3) Make sure that the five files ("main.cu", "virtual_memory.cu" ,"virtual_memory.h", "user_program.cu" and "data.bin") are in the same folder "CSC3150_A3"

(4) Use "Ctrl+F7" to compile the program

(5) Use “Ctrl+F5” to run the program

(6) Wait until the program finished

(7) Then you will see the output in the terminal and there will form a “snapshot.bin” in the folder.

3 The thought of design:

After reading the requirements of the assignment, my whole thought is to finish three main functions which are shown as follows:

```
__device__ uchar vm_read(VirtualMemory *vm, u32 addr);
__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value);
__device__ void vm_snapshot(VirtualMemory *vm, uchar *results, int offset,
                            int input_size);
```

And to implement with LRU paging algorithm when swapping memory, I create a function of my own to swap memory when reading and writing which are shown as follows:

```
__device__ u32 LRU_swap(VirtualMemory *vm,u32 pageNum);
```

First, we will talk about the “vm_write” part. When the user wants to write data to the virtual memory. It will give the data (which is shown as value in the function), the corresponding virtual memory (use the “vm” pointer in the function to represent it) and the relative address (use the “addr” in the function to represent it). Because we use the page table to get the corresponding address in the virtual memory, we can get the page number and page offset for every address input by using “%” and “/” (because the page table contains many pages and every page is of the same size and one page corresponds

to contain page size number of data):

```
u32 page_number = addr / vm->PAGESIZE;
u32 page_offset = addr % vm->PAGESIZE;
```

However, because we use the invert page table instead of page table which means that the entries of the page table array are page number and the indexes of the page table array are frame number. We need to get the frame number and then use frame number and page offset to get the corresponding virtual memory address of corresponding value.

What we should pay attention to is that the first 0-1023 indexes of the invert page table are the status of page numbers of 1024-2048 indexes of the invert page table. The status is invalid or valid and to simplify the complexity of the program. I will use them to help me implement the LRU algorithm as timestamps instead of using a double linked list. Because the initial of the invert page table in the program is almost meaningless (all the status are invalid and the index is equal to the page number), we consider the initial case of the page table is that the page table is empty which is the same as describing in the textbook. Therefore, when the page table is not full and there is a new data written in, the program will first judge if its page number corresponds to the one of the page numbers in the page table, if yes, we can get the index which is the frame number of that page number but if no, we will search the page table in order to find a index that the status of that index is invalid (this index of page is NULL) and then put the new page number into that index and get the index which is the frame number of that page number:

```
for (u32 i = 0; i < vm->PAGE_ENTRIES; i++) {
        if (vm->invert_page_table[i] == 0x80000000)
        {
                condition = true;
                *(vm->pagefault_num_ptr) += 1;
                vm->invert_page_table[i] = 0;
                vm->invert_page_table[i + vm->PAGE_ENTRIES] == page_number;
                frame_number = i;
                break;
        }
        else if ((vm->invert_page_table[i] != 0x80000000)&&(vm->invert_page_table[i + vm->PAGE_ENTRIES] == page_number))
        {
                condition = true;
                vm->invert_page_table[i] = 0;
                frame_number = i;
                break;
        }
}
```

What we should pay attention is that when we change a NULL index to an index with page number, we should change the status of that index from invalid to valid. If the page table is full and we can now match any of page numbers in the page tables to the page number of the new data. It means that we need to use LRU algorithm to swap memory to make there exists the page number of the new data in the page table. Therefore, I create a variable called “condition” which means that if we need to use LRU, the “condition” variable will contain false to mention the program to use LRU. Then I will talk about my thought of design of LRU based on the rules of LRU algorithm.

The rules of LRU (Least Recently Used algorithm) is a Greedy algorithm where the page to be replaced is least recently used. I use the status bit of the invert page table as the timestamps. When every operation (one time of calling the function) almost ends, all the timestamps of valid bit will increase 1:

```
for (u32 i = 0; i < vm->PAGE_ENTRIES; i++) {
        if (vm->invert_page_table[i] != 0x80000000)
        {
                vm->invert_page_table[i] += 1;
        }
}
```

And from the foregoing code shown, we can find that whenever a page number is used in the invert page table (no matter read or write), the timestamp of the corresponding page number will change to 0 (it means the page number is most recently used). Therefore, it is obvious that when we call the function of LRU, we just need to replace the page number with the largest timestamp. I will use the LRU for vm_write as an example to describe how to implement the LRU and swapping memory which is shown as follows:

```
__device__ u32 LRU_swap(VirtualMemory *vm,u32 pageNum) {
        *(vm->pagefault_num_ptr) += 1;
        u32 index = 0;
        for (u32 i = 0; i < vm->PAGE_ENTRIES; i++) {
                if (vm->invert_page_table[i] > vm->invert_page_table[index]) {
                        index = i;
                }
        }
        for (u32 i = 0; i < vm->PAGESIZE; i++) {
                vm->storage[vm->invert_page_table[index + vm->PAGE_ENTRIES] * vm->PAGESIZE + i] = vm->buffer[index*vm->PAGESIZE + i];
                vm->buffer[index*vm->PAGESIZE + i] = vm->storage[pageNum*vm->PAGESIZE + i];
        }
        vm->invert_page_table[index + vm->PAGE_ENTRIES] = pageNum;
        u32 frameNum = index;
        vm->invert_page_table[index] = 0;
        return frameNum;
}
```

We first get the index variable which is the index of old page number in the page table with the largest timestamp. Then we will swap data between the buffer and the storage. We will put the data of the old page number which will be replaced in the buffer to the corresponding address of that page number in the storage and then put the data of the new page number will replace the old page number in the storage to the corresponding address of that page number in the buffer. Then we will update the invert page number with replacing the old page number to the new page number and change the timestamp of that index to 0 and then return that index as the frame number. That is the whole process of the LRU. Finally, we will store the value into the buffer using the frame number and page offset.

```
vm->buffer[frame_number*vm->PAGESIZE+page_offset] = value;
```

Then we will talk about the “vm_read” part. The “vm_read” part is similar to the “vm_write” part but it is used to read data from virtual memory so that the return value of this function is the value of the data in corresponding address. Most parts of “vm_read” function are similar to “vm_write” function. The different part is that in the “vm_write” function, we need to handle the case that the page table is empty or not full but in the “vm_read” function, we do not need to consider this case which means that we will not read data when the page table is empty or when the data which we want to get do not exist in the buffer or the storage.

```
for (u32 i = 0; i < vm->PAGE_ENTRIES; i++) {
        if (vm->invert_page_table[i + vm->PAGE_ENTRIES] == page_number) {
                condition = true;
                vm->invert_page_table[i] = 0;
                frame_number = i;
                break;
        }
}
```

Finally, we will talk about the “vm_snapshot” part. In this part, we will use “vm_read” in this function to put the data in the virtual memory “buffer” to the result. The data we want to read is from offset to the input_size + offset, we will read them one by one and use the pointer of result to get them into the result.

```
__device__ void vm_snapshot(VirtualMemory *vm, uchar *results, int offset,
                int input_size) {
  /* Complete snapshot function togther with vm_read to load elements from data
   * to result buffer */
        for (u32 i = offset; i < input_size + offset; i++) {
                uchar result = vm_read(vm, i);
                *(results + i) = result;
        }
}
```

That is the whole process of my thought of design to implement this program.

4 The page fault number of the output and how does it come out:

The page fault number of the output when using the “user_program.cu” to test is 8193. Now I will try to analysis the hot does it come out. First, according to the definition of page fault number, there will be two cases that we will increase the page fault number by 1.

(1) When the page table is not full and the new page number input does not exist in the page table, the initial case of the page table shows that the status bit of the new page number is invalid, therefore, in this case, the page fault number will increase 1.

(2) When the page table is full but the new page number input does not exist in the page table, which means that we need to use LRU to swap memory. In this case, the status of that new page number is invalid, so the page fault number will increase 1.

After learning about when the page fault number will increase 1, let us consider the logic of the code of “user_program.cu” which is shown as follows:

```
__device__ void user_program(VirtualMemory *vm, uchar *input, uchar *results,
                             int input_size) {
  for (int i = 0; i < input_size; i++)
    vm_write(vm, i, input[i]);

  for (int i = input_size - 1; i >= input_size - 32769; i--)
    int value = vm_read(vm, i);

  vm_snapshot(vm, results, 0, input_size);
}
```

The code first writes all the data in the “data.bin” into the virtual memory char by char. By verifying, we can find that the input_size is 32×4096. Because there are 1024 pages in the page table and at the beginning, the page table is empty and all the status are invalid and the size of every page is 32. Therefore, when the first 32×1024

data input into the virtual memory, we just fill the page table using them and every 32 data will fill a page (every 32 data have the same page number) and when an index of the page table has the page number, the status of that index will change from invalid to valid. Therefore, when the first 32×1024 data input, the page fault number will increase to 1024. When the rest 32×3072 data input, because the front data has already filled the virtual memory buffer and the corresponding page number has already filled the page table and all the page numbers for every 32 data in the write process of the "user_program.cu" are different, the program needs to use LRU swap memory to swap memory 3072 times. Therefore, after the rest of the write process, the page fault number will increase 3072. Therefore, we can find that after the write process of the "user_program.cu", the page fault number becomes 4096.

```
for (int i = 0; i < input_size; i++)
  vm_write(vm, i, input[i]);
```

Then the code use "vm_read" to read data from virtual memory. From the structure of this loop we can find that the program reads data in the reverse order. From the last data to the data in $input\_size - 32769 - 1$ index. The total number of data the program read is that 32769 which is $32 \times 1024 + 1$. Because when the last step "write process" ends, the page numbers in the page tables are the largest 1024 (3072-4095). Therefore, when we read the first 32×1024 data, because their corresponding page numbers are in the page table, so they do not need to swap memory using LRU. Therefore, the page fault number will not increase when we read the first 32×1024 data. However, when we read the $32 \times 1024 + 1$ data, because the page number of it does not exist in the page table, we need to swap memory using LRU. Therefore, the

page fault number will increase by 1. Therefore, after the process of “read”, the page fault becomes 4097.

```
for (int i = input_size - 1; i >= input_size - 32769; i--)
  int value = vm_read(vm, i);
```

The last step of the code is to use the “vm_snapshot” to read data from the virtual memory to the result. In the “user_program.cu”, the offset is 0. Therefore, the logic of this part of code is to read from the beginning to the end of the original data (positive sequence). After the end of the front “vm_read”, the page numbers of the page table are all the last part of the 4096. Therefore, in the “vm_snapshot” part, every time we read 32 data, we need to swap memory using LRU. Therefore, the page fault number will increase by 4096. Therefore, the finally page fault number will become $4097 + 4096 = 8193$.

That is the whole process how does 8193 come out.

5 The problems I met in this assignment and the solutions:

I have met a lot of problems in this assignment which are shown as follows:

(1) When I try to implement the LRU. I need a structure to help me record and find the page to be replaced is least recently used. At first, I try to use the double linked list to implement the LRU, but the design of the double linked list and logic of it are a little complex. Therefore, I finally decide not to use double linked list to implement the process. After observation, I find the first 1024 index of the invert page table when all the status become valid will become useless. I try to take full advantage of them. Therefore, I try to use the value in them as the timestamps of the

corresponding index page number and change the timestamps after every operation. This makes my program clear compared with using double linked list. That is the solution of this problem I met.

(2) When I first write this program, the initial invert page table function makes me confused and I have known that the "user_program.cu" is just one of test program and my code should get right result for all kinds of input order test program. Therefore, at first I do not known how to consider all kinds of special input order. After observation, I think the initial invert page table function is almost meaningless because all the status at the beginning are invalid. Therefore, I assume that the page table to be empty at the beginning. Therefore, the whole process no matter what the input order is becomes clear. I can consider into all kinds of special input order with a little code.

6 The screenshot of the program output:

(1) When the program finished, you will see the input size and page fault number in the terminal:

```
input size: 131072
pagefault number is 8193

D:\118010138\CSC3150_A3\x64\Debug\CSC3150_A3.exe (process 10656) exited with code 0.
Press any key to close this window . . .
```

(2) You can also use "fc" command to make sure that the input file "data.bin" is the same as the output file "snapshot.bin":

```
D:\118010138\CSC3150_A3\CSC3150_A3>fc data.bin snapshot.bin
Comparing files data.bin and SNAPSHOT.BIN
FC: no differences encountered
```

7 Bonus

I have used the invert page table to solve the problem which is the content of bonus. In this bonus part, I launch 4 threads in kernel function, all threads concurrently execute it:

```
dim3 grid, block;
grid.x = 2;
grid.y = 2;
block.x = 2;
block.y = 2;
mykernel<<<grid, block, INVERT_PAGE_TABLE_SIZE>>>(input_size);
```

And to avoid the race condition, I make the thread runs in order:

```
if (gridDim.x == 0 && gridDim.y == 0) {
        user_program(&vm, input, results, input_size);
}
if (gridDim.x == 1 && gridDim.y == 0) {
        user_program(&vm, input, results, input_size);
}
if (gridDim.x == 0 && gridDim.y == 1) {
        user_program(&vm, input, results, input_size);
}
if (gridDim.x == 1 && gridDim.y == 1) {
        user_program(&vm, input, results, input_size);
}
```

Therefore, because I use the invert page table, all the threads use the same page table, so I can manage multiple threads. And because the page fault number is added together by the four threads. The total page fault number is $8193 \times 4 = 32772$. The program can correctly dump the contents to “snapshot.bin”.

8 What did I learn from this assignment:

In this assignment, I have learned how to use page table in storing and reading data

from virtual memory and the interaction of different memory (buffer and storage). I also learn how to get data from virtual memory using pointers. What's more, I also learn how to implement LRU when swapping memory. And I also learn to take patience and take careful of programming because I almost waste 5 hours to debug only because a "<" is written to ">" by mistake. All in all, I have learned a lot from this assignment.