CSC3150 Project3 Report

118010045 Cui Yuncong

November 2020

Contents

1 Introduction

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. This project implements simple virtual memory in a kernel function of GPU that have single thread, limit shared memory and global memory by using CUDA API to access GPU. CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model. The program is implemented on CUDA and tested on the windows OS with CUDA 11.1, VS 2019, and NVIDIA GeForce GTX 1060.

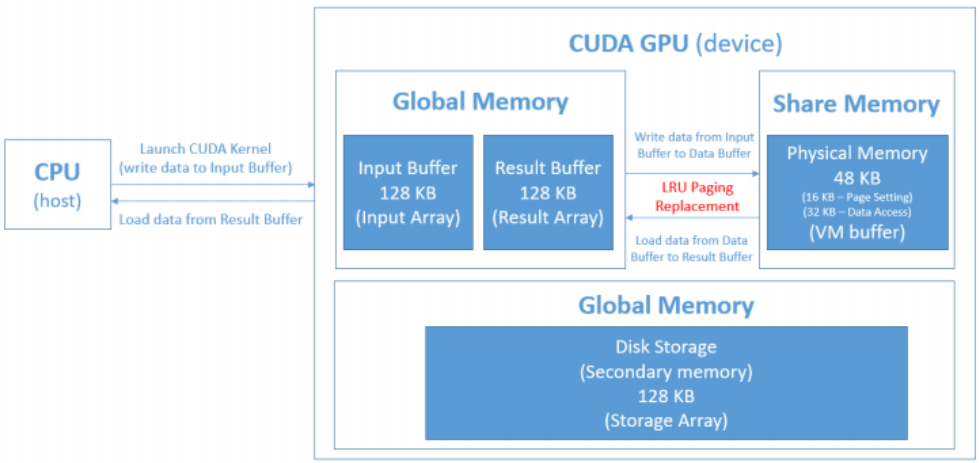

Figure 1: CUDA Memory Structure

2 Design

2.1 Program Flow

The program consists of 4 parts:

1. Load the data from the binary data file “data.bin” to the input buffer.

2. Write the data from the input buffer to the physical memory. Implement the LRU algorithm to decide which page should be removed from the physical memory to the disk storage and put the current page into that frame.

3. Find the required pages first in the physical memory. If the target page is not in the physical memory, use the LRU algorithm to swap the page in the storage with the page in the physical memory.

4. Output the data to binary data file "snapshot.bin".

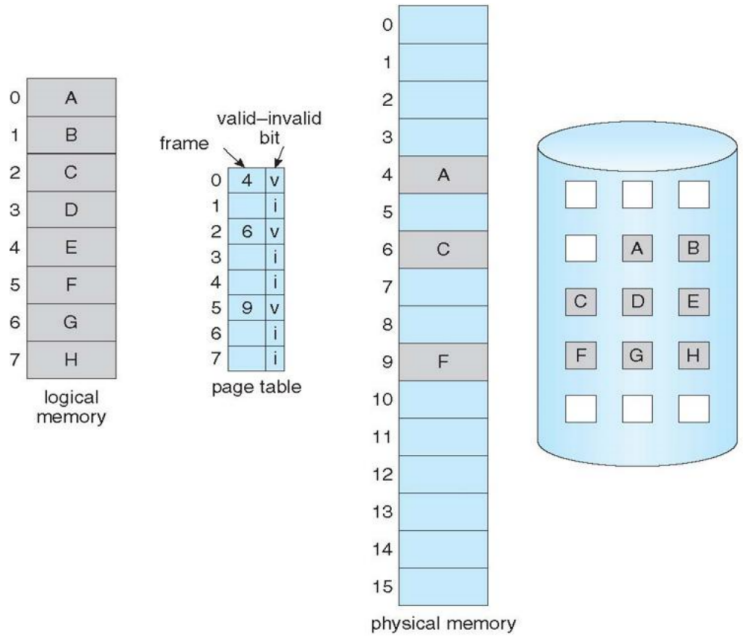

Figure 2: Memory Structure of Page and Frame

2.2 LRU Algorithm

In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operat-

ing System replaces one of the existing pages with newly needed page. Least Recently Used (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used.

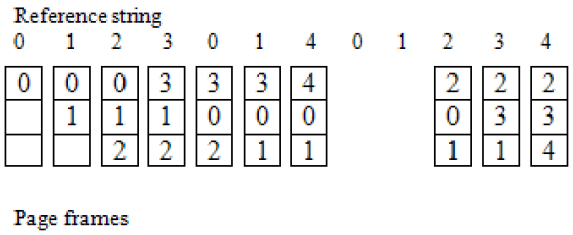

Figure 3: The Demo for LRU Algorithm

2.3 Functions

This section describes the functions in the virtual_memory.cu.

1. copy_data()

This copy_data() function shown in Figure 4 is used to move the data from the storage to the physical memory according to the given page number and frame number.

```
__device__ void copy_data(VirtualMemory* vm, u32 page_number, u32 frame_number)
{
    for (int i = 0; i < 32; i++)
    {
        vm->storage[vm->invert_page_table[frame_number] * 32 + i] = vm->buffer[frame_number * 32 + i];
        vm->buffer[frame_number * 32 + i] = vm->storage[page_number * 32 + i];
    }
    vm->invert_page_table[frame_number] = page_number;
}
```

Figure 4: copy_data() Function

2. page_fault()

This page_fault() function shown in Figure 5 is used to find if the given page is in the frame. The function returns false if the page fault occurred; otherwise, returns true.

```
__device__ bool page_fault(VirtualMemory* vm, u32 page_number)
{
    for (int i = 0; i < vm->PAGE_ENTRIES; i++)
        if (vm->invert_page_table[i] == page_number)
            return false;
    return true;
}
```

Figure 5: page_fault() Function

3. invalid()

This invalid() function shown in Figure 6 is used to update the frame list after each page is loaded in the physical memory. It firstly finds the corresponding block in the frame list. The frame list is the later part in the invert page table. Then, it puts the top frame number at the bottom of the list since this frame is just be used and put other frame numbers forward by one block.

```
__device__ void invalid(VirtualMemory* vm, u32 frame_number)
{
    int head, tmp;
    bool tag = true;
    for (int i = 0; i < vm->PAGE_ENTRIES; i++)
        if (vm->invert_page_table[i + vm->PAGE_ENTRIES] == frame_number)
        {
            tag = false;
            head = i;
        }
    if (tag)
        return;
    tmp = vm->invert_page_table[vm->PAGE_ENTRIES + head];
    for (int i = vm->PAGE_ENTRIES + head; i < 2 * vm->PAGE_ENTRIES - 1; i++)
        vm->invert_page_table[i] = vm->invert_page_table[i + 1];
    vm->invert_page_table[2 * vm->PAGE_ENTRIES - 1] = tmp;
}
```

Figure 6: invalid() Function

4. search()

This search() function shown in the Figure 7 is used to find the corresponding frame number according to the given page number. It searches the page table to check if the page number storage in each entry equals to the given page number and returns the frame number.

```
__device__ int search(VirtualMemory* vm, u32 page_number)
{
    u32 frame_number = -1;
    for (int i = 0; i < vm->PAGE_ENTRIES; i++)
        if (vm->invert_page_table[i] == page_number)
        {
            frame_number = i;
            break;
        }
    return frame_number;
}
```

Figure 7: search() Function

5. vm_read()

The vm_read() function shown in Figure 8 is used to read the target pages from the buffer. It firstly invke the page_fault() function to check if the target pages are in the physical memory or not. If the target pages are not in the physical memory it uses the copy_data() function to move the pages from storage to physical memory. For each byte, the frame list will be changed since the recent used frame may be changed.

```
__device__ uchar vm_read(VirtualMemory* vm, u32 addr)
{
    u32 page_number = addr / 32, frame_number;
    if (page_fault(vm, page_number))
        frame_number = search(vm, page_number);
    else
    {
        *vm->pagefault_num_ptr = *vm->pagefault_num_ptr + 1;
        frame_number = vm->invert_page_table[vm->PAGE_ENTRIES];
        copy_data(vm, page_number, frame_number);
    }
    convert_to_invalid(vm, frame_number);
}
```

Figure 8: vm_read() Function

6. vm_write()

The vm_write() function shown in Figure 9 is used to write the data from input buffer to the physical memory and storage. Each time when we are loading the page into the memory we should check if the frame is full or not. If the frame is already occupied by some page,

the page will be moved to the storage, and then the new page will be put into the frame.

```
__device__ void vm_write(VirtualMemory* vm, u32 addr, uchar value)
{
    u32 page_number = addr / 32, frame_number, page_number_storage;
    if (!page_fault(vm, page_number))
        frame_number = search(vm, page_number);
    else
    {
        *vm->pagefault_num_ptr = *vm->pagefault_num_ptr + 1;
        frame_number = vm->invert_page_table[vm->PAGE_ENTRIES];
        if (vm->invert_page_table[frame_number] != 0x80000000)
        {
            page_number_storage = vm->invert_page_table[frame_number];
            for (int i = 0; i < 32; i++)
                vm->storage[page_number_storage * 32 + i] = vm->buffer[frame_number * 32 + i];
        }
        vm->invert_page_table[frame_number] = page_number;
    }
    vm->buffer[frame_number * 32 + addr % 32] = value;
    invalid(vm, frame_number);
}
```

Figure 9: vm_write() Function

7. vm_snapshot()

The vm_snapshot() function shown in Figure 10 is used to put the data into result buffer. It finds the page of input buffer in the physical memory. The page will be put the page from physical memory into the result buffer if it is in the physical memory. Otherwise, the page in storage will be swapped with the page in the target frame and moved into the result buffer.

```
__device__ void vm_snapshot(VirtualMemory* vm, uchar* results, int offset, int input_size)
{
    for (int i = 0; i < input_size; i++)
    {
        u32 page_number = i / 32, frame_number;
        if (!page_fault(vm, page_number))
            frame_number = search(vm, page_number);
        else
        {
            *vm->pagefault_num_ptr = *vm->pagefault_num_ptr + 1;
            frame_number = vm->invert_page_table[vm->PAGE_ENTRIES];
            copy_data(vm, page_number, frame_number);
        }
        for (int j = 0; j < 32; j++)
            results[page_number * 32 + j] = vm->buffer[frame_number * 32 + j];
        invalid(vm, frame_number);
    }
}
```

Figure 10: vm_snapshot() Function

3 Problem and Solution

3.1 Construct the Frame List with LRU Algorithm

The program implements the LRU algorithm by using an array to store the sequence. The top entry will store the frame number of the least recently used and the bottom one will store the frame number of the most recently used. Whenever a frame has been used, the frame will be swapped to the bottom of the list and every frame number stored below it will be moved forwards by one entry.

3.2 Get the Frame Number

The program puts the first frame in the frame list if the page fault occurred. Otherwise, it goes through the invert page table to find the corresponding frame for each data according to its page number.

3.3 Get the Page Number

The program stores the data in the storage according to their page number. Tracking the page from the storage can get its page number according to its position.

4 Running Environment

Version of OS: Window10

IDE: Visual Studio 2019

Video Card: NVIDIA Geforce GTX 1060

CUDA Cersion: 11.1

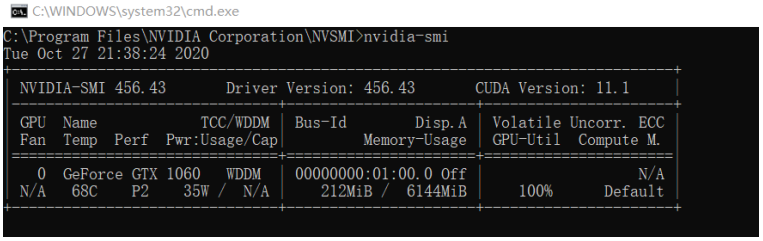

Figure 11: GPU Information and CUBA Version

5 Execution and Program Output

Use Visual Stdio to open the project. Press 'Ctrl' + 'F7' to compile each CUDA file and use the 'Ctrl' + 'F5' to execute the program.

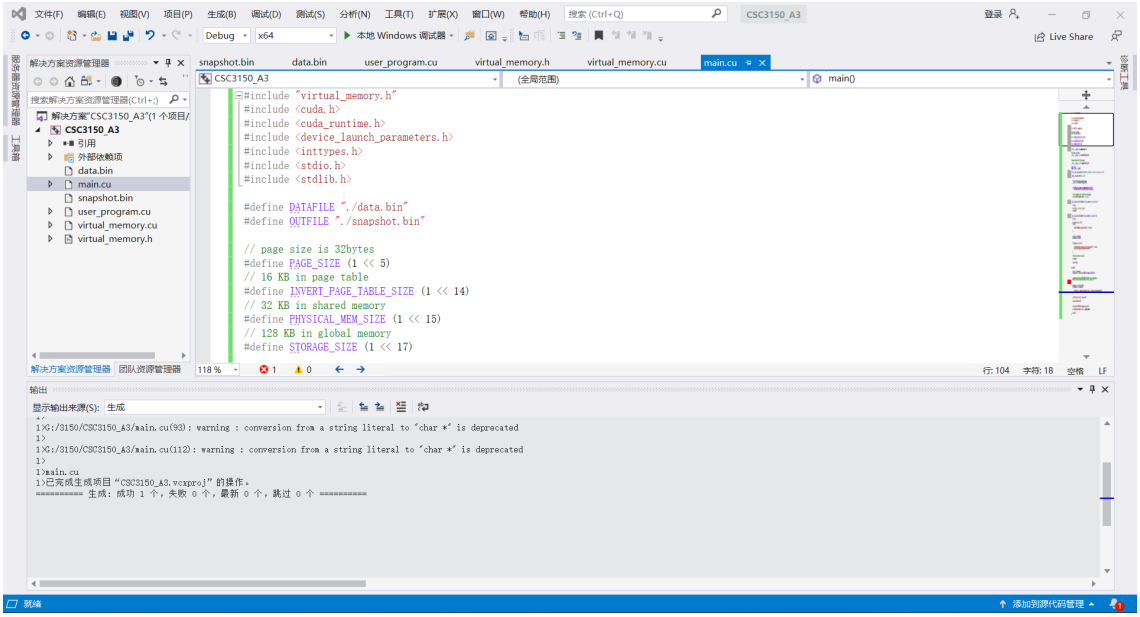

Figure 12: Compile Successfully

The page fault number is shown as the figure below:

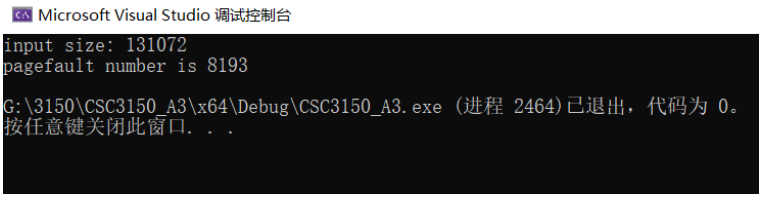

Figure 13: Page Fault Number

The output file is shown as the figure below:

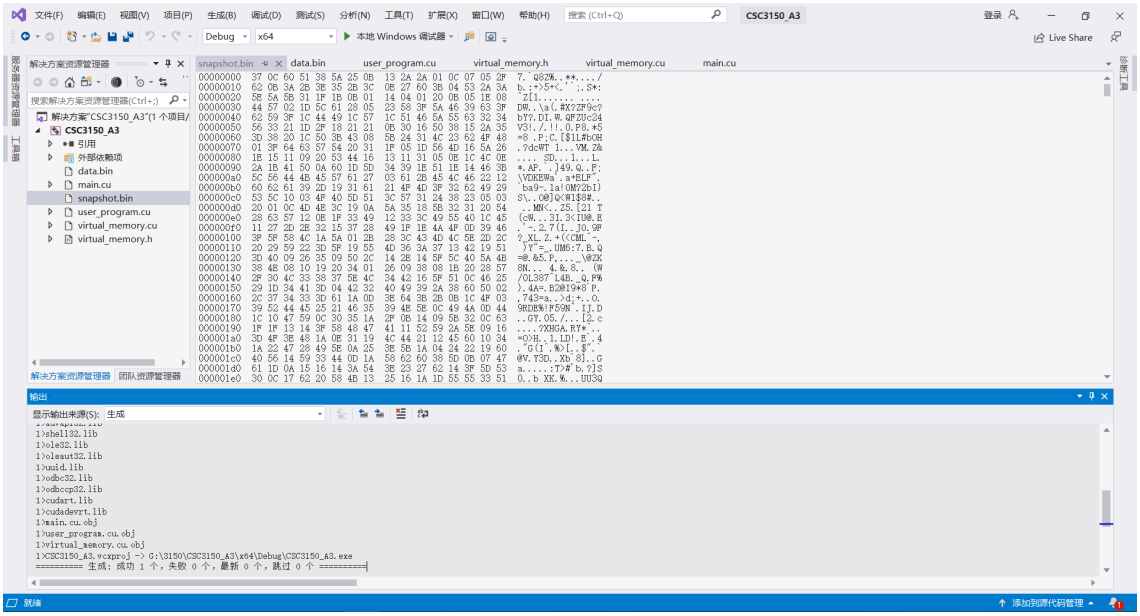

Figure 14: Snapshot

6 Analysis

The page fault is 8193 as shown in Figure 13, which contains 4096 page faults from the vm_write() function, 1 page fault from the vm_read() function and 4096 page faults from the vm_snapshot() funciton.

There are totally $2^{15} + 1$ bytes required to be read in the physical memory according the

Function	Page Fault
vm_read	1
vm_write	4096
vm_snapshot	4096
Total	8193

Table 1: The Number of Page Fault

user program. The size for those data is equal to 1024 pages and 1 byte with each page contains 32 bytes. The user program reads the data from the bottom of the data binary file, and the data are written from the top to the bottom in the vm_write() section, the first 1024 pages will be found in the physical memory. Therefore, only the last byte will generate page fault. The page faults in the vm_read() function is 1.

Every page loaded from the input buffer is a new page for the physical memory. There are totally 128KB (2^{17} bytes) data with 2^5 bytes in each page, we can have 4096 (2^{12}) pages. Therefore, vm_read() function has 4096 page faults.

The vm_snapshot() funciton is required to read the data again from the top to the bottom. The number of page faults in this section is as the same as that in the vm_write() function. Each page will generate the page fault since only the bottom pages are stored in the physical memory at the beginning. Therefore, vm_snapshot() function has 4096 page faults.

Therefore, there are totally 8193 ($1 + 2^{12} + 2^{12}$) page faults generated.

7 Learning Remarks

7.1 Memory Structure

This program aims to implement the a strategy to manage the memory. In the Figure 1, the data that needs to be used can be stored in the both shared memory and global memory. When the CPU tries to read the data, the memory firstly searches the page table to check if this page is in the physical memory. The page in the storage will be swapped with the page in the memory if the page is not in the physical memory

7.2 LRU Algorithm

Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is LRU (Least Recently Used) paging.

In the LRU, the program needs to decide which is the page that has been unused for the longest time. Thus, the program designs a frame list. It is an array which contains the number of rounds that the last time when this frame has been used. Once a frame is used, its rank in the frame list will be updated while other counters remains the same. The index with the largest counter will be the frame number of the page that has been unused for the longest time.

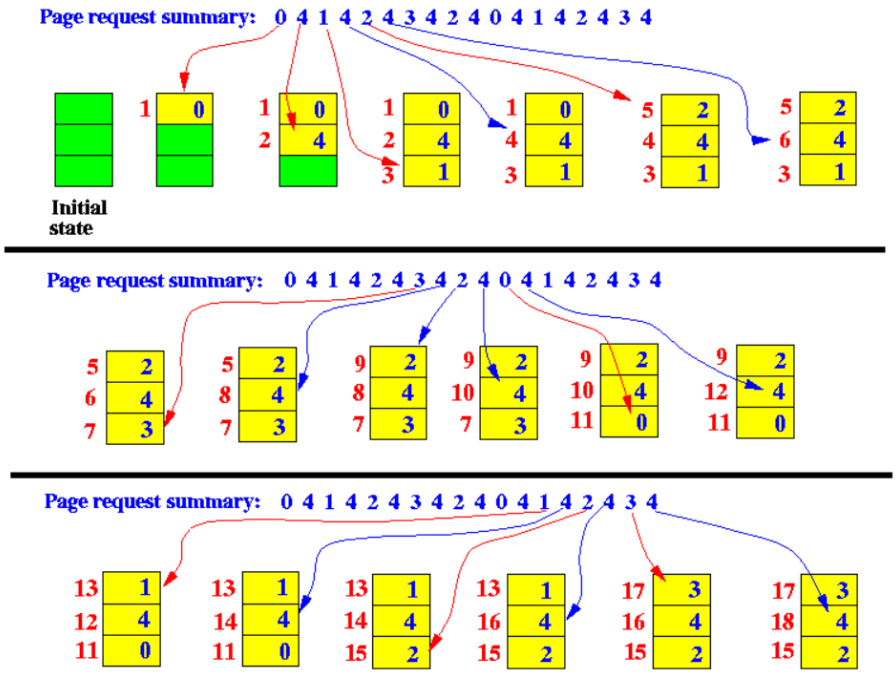

Figure 15: Snapshot

7.3 Invert Page Table

The invert page table is used to store the corresponding page number in the table with the index of the frame number. The logical memory can be linked to the physical memory by this table since it can check the frame number of the page in the physical memory.