Report for CSC3150 Assignment3

Zhixuan LIU

118010202

1 Introduction

In this Project, I use GPU to implement a virtual memory. The main component contains the input buffer and the output buffer, which are used in the Global Memory of CUDA GPU; in CADA share memory, i have one page table and one vm buffer; in CUDA global memory, i implement the secondary storage.

The highlight of my design is the page table, which i use doubly linked list in 16KB Page Setting to implement. Due to this structure, the all program will only take 1min 40s.

The following shows the details of my design.

2 Environment of Running my Program

I use the computer in TC301, with the following environment:

Win 10
VS2017
CUDA9.2
NVIDIA Geforce GTX 1060
Compute capacity: 6.1

3 Execution Step to Run my Program

My assignment program is in "3150_A3" folder, my bonus program is in the "3150_A3_bonus" folder. In VS2017, press "Ctrl + F7" to compile my programs, and press "Ctrl + F5" to run my program.

4 How did I Design my Program

I only change the "virtual_memory.cu" program. There are four parts: init_invert_page_table, vm_read, vm_write and snapshot.

4.1 init_invert_page_table

In this init_invert_page_table function, a 16KB memory size is given. The page entries will only use 4KB. For the other space, i design a doubly linked list to implement the LRU. My structure is shown in the figure:

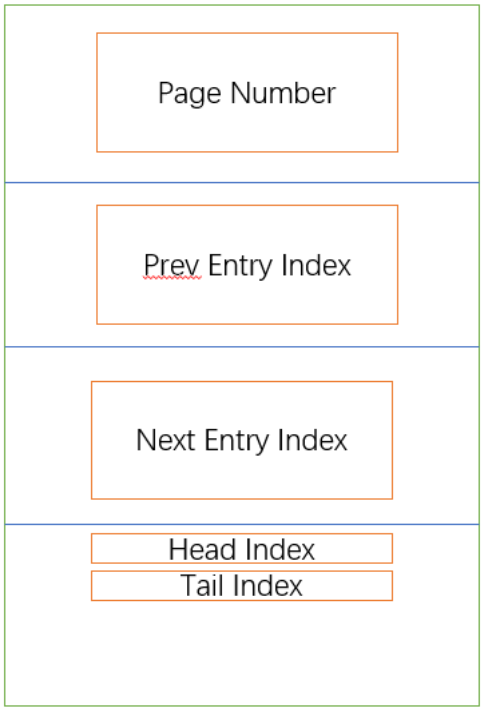

Figure 1: The structure of Page table to implement linked list.

4.2 vm_write

This function will write the value in the input buffer into the vm buffer. The logic is shown as follows:

First, we will get a virtual address i generated by CPU. Then, we need to get its page number and offset:

```
1 uint32_t page_num = addr / 32;
2 uint32_t offset = addr % 32;
```

Then we need to check whether this page number has already existed in the page table by checking through the page table.

If so, this means that the frame is in the vm buffer. Therefore, we can directly write the value into the corresponding physical address.

```
physical_addr = (flag_in) * 32 + offset;
value = vm->buffer[physical_addr];
```

Because in this system i use the LRU structure, so we need to put the page number we attain to the tail of my doubly linked list.

However, if we do not find the page number in the page table, this means that this page (or frame) is not in the vm buffer. Then, we encounter a page fault. Then we need to judge whether this page table is full. If the page table is not full, then we can put the page number into the empty page table entry, then put the value in to the vm buffer. Otherwise, we should use the LRU structure, which is the head of my doubly linked list, to put the least recent used frame in to the secondary memory. Then, replace it by the new page number.

4.3 vm_read

This function is similar to the vm_write function. The difference is that it read things from the memory. First, we get a virtual memory address i generate by CPU. Then, I get the page number and offset similar to wm_write. Next, i check whether this page number is in my page table.

If it is in my page table, this means that, the value I need is in the vm buffer. Then i just go directly into the um buffer to get the value.

If it is not in my page table ,this means that the value I need is in the secondary memory. Then i need to replace the least recent used frame in the vm buffer by the frame I need in the secondary memory, as well as change it in the page table. Then we access the data in the vm buffer.

4.4 snapshot

This function is to read the value according to the address to the result buffer.

```
__device__ void vm_snapshot
    (VirtualMemory *vm, uchar *results, int offset,
        int input_size) {
        for (int i = 0; i < input_size; i++) {
                results[i] = vm_read(vm, i);
        }
}
```

4.5 Declaration of my LRU

I use doubly linked list to implemt the LRU as shown in Fig.1. The least recent used page is the head of my linked list, while the page that is the recent access will be put to the tail of my linked list.

When there is a snap, the head of my linked list will be replaced; when one page is accessed, its index will be marked as the tail of the linked list.

4.6 Advantages

By using this LRU structure, the running time of my whole program will only takes 1 min 40 s.

5 Page Fault Number

My page fault number is 8193. In vm_write part, the page fault is 4096. Because There 32*4096 virtual addresses, and each page size is 32. Which means in vm_write, it will encounter 4096 page faults.

In vm_read part, the page fault number is 1. This is because the reading is from down to top:

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

For the previous part, because those page is the recent write in the page table and vm buffer, so the page number will be found in page table. Then the previous part will not cause page fault. Only the end read part will cause one page fault.

In snapshot part, the page fault number is 4096. It read the things from begin to the end, just like vm_write. So it will cause 4096 page faults.

6 The Problem I met

In the first version of my program, i did not use doubly linked list to implement my LRU structure. Instead, i use "count". More specifically, each page table entry has an attribute denotes how long it has exits in the whole program running time. For example, the newly accessed page number has an "count = 0"; every time a page number is accessed, the other page number "count += 1". The least recent used page is the page with the largest "count". However, it takes 10 minutes to run this program by using this structure, which is drastically long.

I solve this problem by using doubly linked list, and it only takes 1 min 4 s.

7 Screen Shot

The only screen shot is the page fault number:

I also checked the difference of data.bin and snapshot.bin using linux (for both A3 part and bonus part):

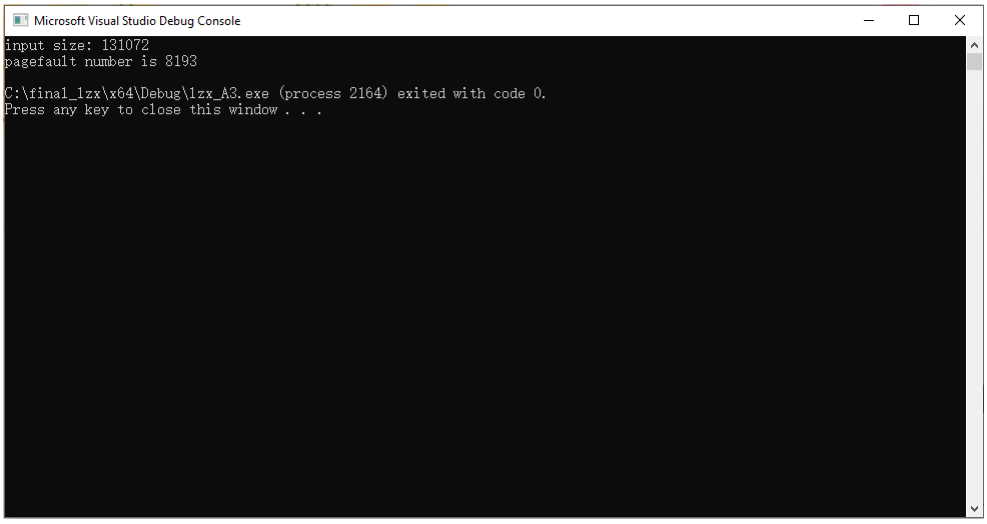

Figure 2: The screen shot of my program.

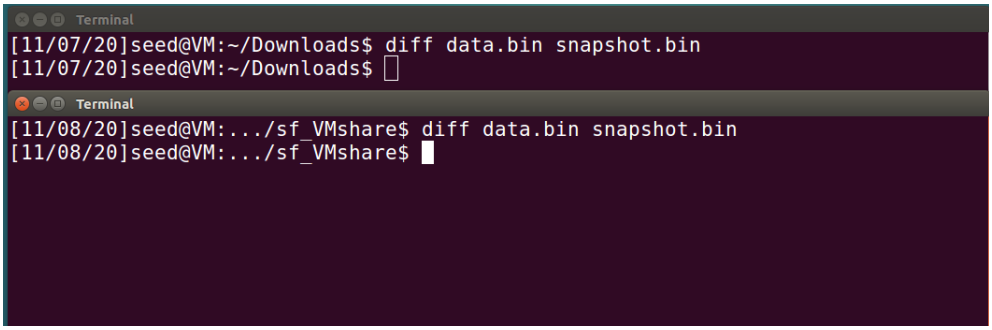

Figure 3: The screen shot of the difference between to bin file.

8 What did I Learned

In this program, I learned how virtual memory works. How to use the LRU structure to replace the frame and how to design the LRU structure. I learned how to improve the structure and make it more efficiently.

9 Bonus

In my bonus part, i create 4 thread to implement my program concurrently.

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

In the user program, i use *threadIdx.x* to identify each thread. I also use *__syncthreads*() to synchronize them. In my virtual memory program, i import the structure called "CUDA Atomic Lock" from the internet, to mutex my functions.

By implementing this method, the running time of my program reduced to 30s. Which means that the job is divided into four parts and each thread implement those four parts concurrently.