CSC3150 Project 3 Report

119010341 WU Yifan

Running Environment

- Windows 10
- NVIDIA CUDA 11.2.162 driver
- NVCC version:

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Mon_Nov_30_19:15:10_Pacific_Standard_Time_2020
Cuda compilation tools, release 11.2, V11.2.67
Build cuda_11.2.r11.2/compiler.29373293_0
```

- NVIDIA Geforce GTX 1650
- Visual Studio 2019 (developing toolkit for c++/c)

Execution Guide

Go to the assignment directory

```
|  README.txt
|
├─bonus
|      data.bin
|      main.cu
|      user_program.cu
|      virtual_memory.cu
|      virtual_memory.h
|
└─project_3
        data.bin
        main.cu
        user_program.cu
        virtual_memory.cu
        virtual_memory.h
```

Go to the basic task directory or the bonus directory

```
data.bin
main.cu
user_program.cu
virtual_memory.cu
virtual_memory.h
```

Compilation

```
nvcc --relocatable-device-code=true main.cu virtual_memory.cu user_program.cu -o
main.out
```

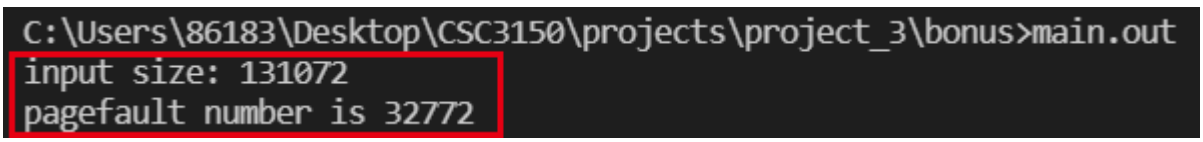
3 new files should appear, and then choose `main.out` to execute

```
main.exp
main.lib
main.out
```

Sample Output

Sample output for the basic task

Sample output for the bonus task



Program Design--Basic Task

1. Task

In this task, we are going to implement the **paging system** using CUDA (Compute Unified Device Architecture) in the GPU due to GPU's small size and low latency access.

Below are the requirements for the implementation

Paging System Parameters	Values
Secondary Memory Size	128KB
Physical Memory Size = Page Table Size + Data Size	48KB = 16KB + 32KB
Page Size	32B

Also, when there is no available entries in the page table for reading or writing, we need to use LRU algorithm for swapping

2. Memory Allocation Design

First, we totally have 4 collections of data

- Input Buffer ---- Virtual Memory
- Share Memory ---- Physical Memory
- Disk Storage ---- Secondary Memory
- Result Buffer contains all the data read from the physical memory

Physical memory is divided into 2 parts: page table(16KB) and data access(32KB)

The most important part in the paging system is the design of the page table

> **Page table at least contains**
>
> - physical frame number or virtual page number(inverted page table)
> - a valid flag for each page number

Since we have 16KB for the page table settings, we can at most allocate 8KB for the page number, which is 8KB / 4B = 2048 page entries

The page size is set to be 32B, there are totally 32KB / 32B = 1024 entries in the data access part of the physical memory

The size of the virtual memory, however, can be much larger than 32KB because the input size is determined by the user. So it may be impossible to insert all the virtual page number into the page table

Thus we choose the inverted page table:

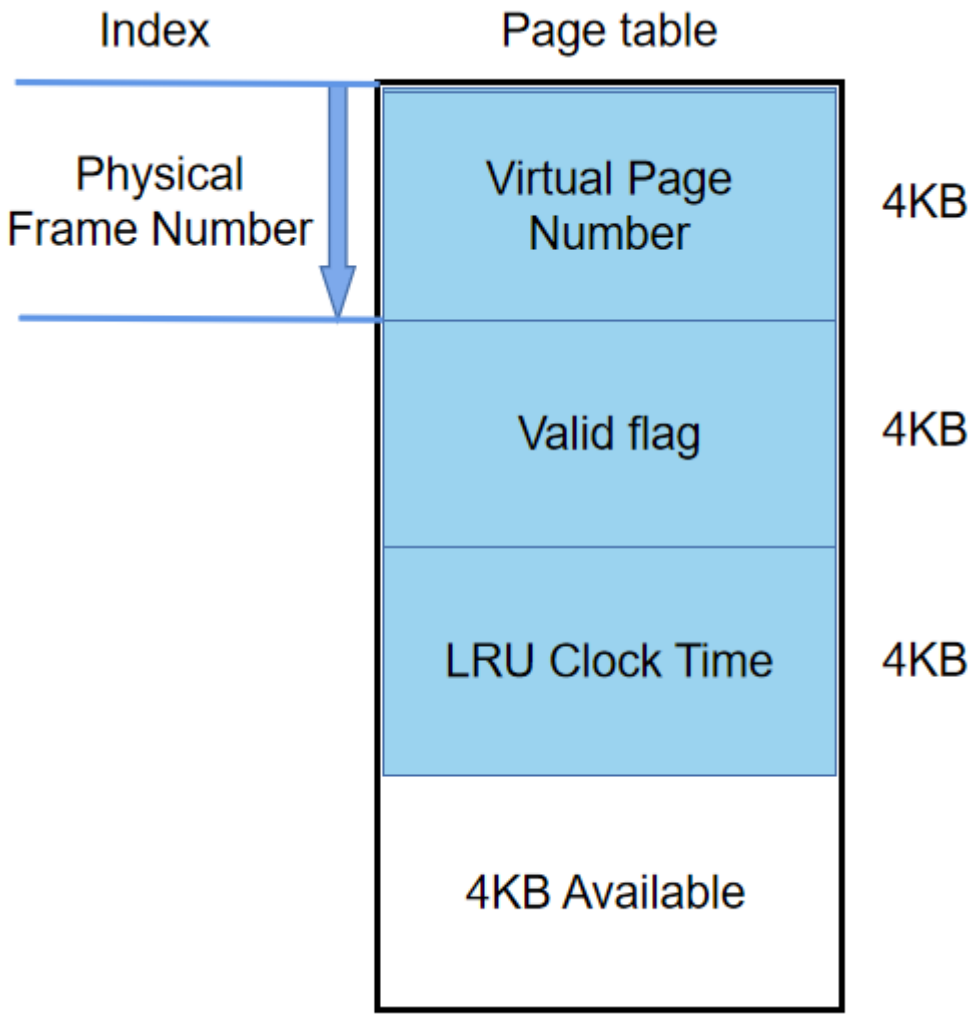


Figure 1

For each physical frame number, we will have assign 1 mapped virtual page number, 1 valid flag, and 1 LRU clock time. In this way, in total 1024 physical frame entries will lead to 1024 * 3 * 4 = 12KB memory, which is less than 16KB

3. Logic for vm_read & vm_write

The main arguments for the function `vm_read()` and `vm_write()` is a virtual memory address, so we first need to convert the virtual memory address to the corresponding virtual page number

```
virtual_page_number = addr / vm->PAGESIZE;
page_offset = addr % vm->PAGESIZE;
```

Then we will do operations based on this `virtual_page_number`

3.1 Search for virtual_page_number in the page table

First traverse the "Virtual Page Number" part (the first 1024 entries) of the page table, if we can find the same `virtual_page_number`, the index of this non-empty entry will be the corresponding physical frame number

So we just use this physical frame number together with the `page_offset` which we have obtained earlier to conduct reading or writing from the physical memory

3.2 Add virtual_page_number into the page table

In 3.1, suppose we fail to find exactly the same `virtual_page_number` in the page table, we may need to write this `virtual_page_number` into the page table

- find an empty entry in the page table with its index as physical frame number
- change the valid flag of this physical frame number to "valid"
- change the mapped virtual page number to the value of `virtual_page_number`

3.3 The LRU algorithm

If the page table is already full and there is no empty entry for us to write in virtual page number, we may need to pick a victim entry and put all its corresponding data into the secondary storage to give us an empty entry

In **Figure 1**, we use 4KB of memory in the page table to store the "clock time" attribute, the mechanism for "clock time" is:

- Whenever we conduct `vm_read()` or `vm_write()` at a certain physical frame number, we will reset its clock time to 0, and increase the clock time at all the other non-empty frame numbers by 1

```
for(int i=0; i<vm->PAGE_ENTRIES; i++){
    if(vm->invert_page_table[i]==0x00000000){ // non-empty frame
        vm->invert_page_table[i + vm->PAGE_ENTRIES * 2] += 1;
    }
}
```

- When we initialize the memory, all of the clock time integers are set to be 0

```
// reset
vm->invert_page_table[physical_frame_number + vm->PAGE_ENTRIES * 2] = 0;
```

In this way, the most recent frame that we use will have the maximum clock time, which is 0. The least recent frame that we use, on the other hand, will have the minimum clock time.

Thus we can just traverse the clock time of each physical frame and find frame number with the smallest clock time. This frame will be the victim.

The "swapping" operations are different for `vm_read()` and `vm_write()`

For `vm_write()`, we only need to put the victim into the disk

- Use the frame number of the victim to locate the starting address of the its physical memory
- move 1 page of data after the starting address from the physical memory to the secondary memory, the starting address in the secondary memory will be determined by the mapped virtual page number of the frame number

```
for(int i=0; i<vm->PAGESIZE; i++){ // transferring data
    vm->storage[i + frame_number] = vm->buffer[i + mapped_vpn];
}
```

4. vm_snapshot

In this part there is no complicated logic. The program just invokes `vm_read()` to read all data in the physical memory and the secondary memory in the virtual memory sequence `0->inputsize`

Program Design--Bonus

1. Task

To launch multiple threads (4 threads) and each thread use the mechanism of paging, we should design a new page table for managing multiple threads.

2. Lauching Multiple Threads

We have defined the page table, the page fault number, the physical memory, and the secondary memory as **macro variables**. which means that all the 4 threads will share the same memory as well as the page fault number

Shared memory will be fine, however the share upon **page fault number** may possibly result in wrong output when multiple threads are accessing the same variable

Thus we need to particularly keep different threads independent from each other in this program. We can distinguish different threads by its **thread id** and let only one thread run each time

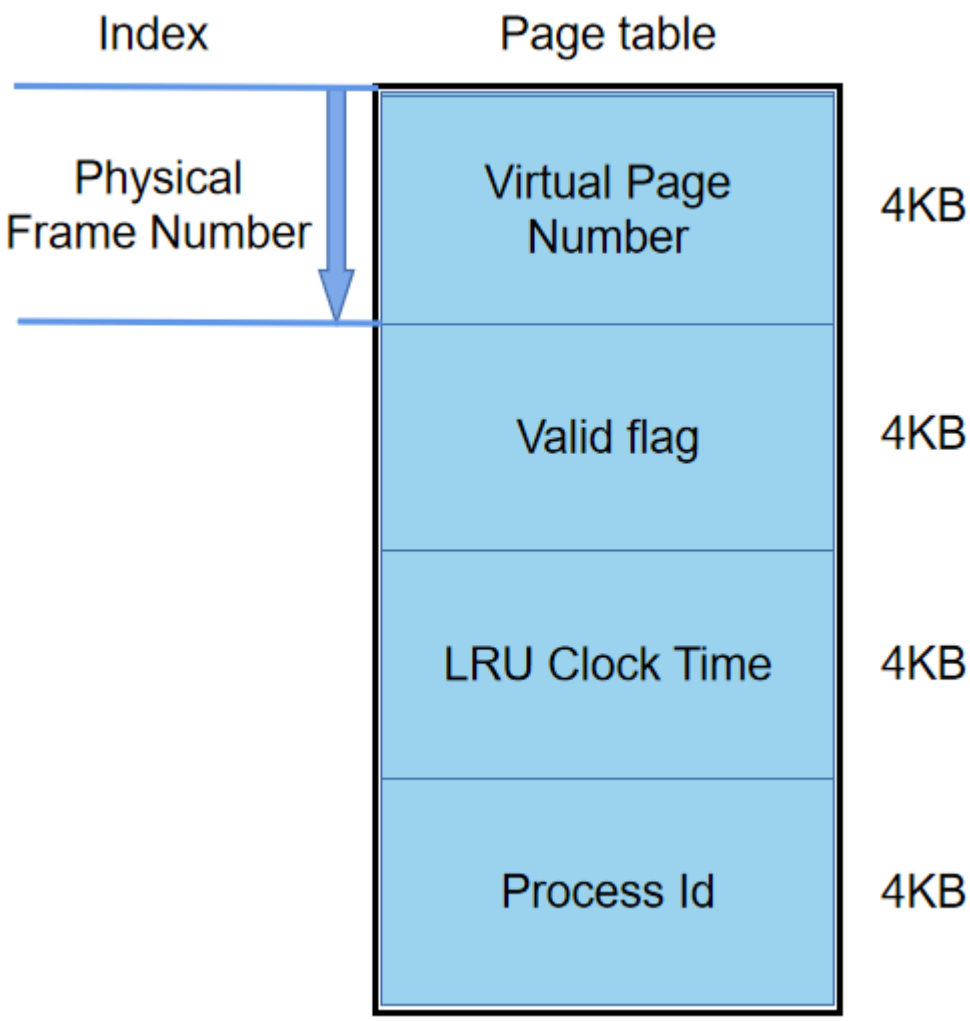
```
if(pid==0){
    user_program(&vm, input, results, input_size);
}
__syncthreads(); // the other threads wait for thread-0

...
```

3. Modify Page Table

In order to let the page table differentiate virtual page numbers of different threads, we need to add a "pid" attribute to each frame

In this way, the allocation of page table will become



So when we conduct `vm_read()` and `vm_write()` (in **3.1**, **3.2**, and **3.3**)

- we need to enclose the corresponding pid of that frame into the page table

```
vm->invert_page_table[physical_frame_number + vm->PAGE_ENTRIES * 3] = pid;
```

- we need to compare both the `virtual_page_number` and the `pid` when we traverse the page table

```
if (
vm->invert_page_table[physical_frame_number + vm->PAGE_ENTRIES] ==
virtual_page_number
&&
vm->invert_page_table[physical_frame_number + vm->PAGE_ENTRIES * 3] == pid
)
```

Sample Output Analysis

In the input file `data.bin`, there are totally 128KB (131072KB) data, which is 131072KB / 32B = 4096 pages

1. vm_write

In the sample user program, 4096 pages of data are written into the physical memory together. However, the page table can only hold 1024 pages.

So for the first 1024 pages, the page table is completely empty and every page's first data that is written into the memory will cause a page fault. Thus 1024 page faults are raised.

- page fault number = 1024

For the remaining 3072 pages, there is no space in the page table so the paging system will carry out LRU algorithm. Frames are swapped out from 0 index to 1023 index for 3 cycles, where page fault number will increase to 4096. The current pages left in the page table will be the last 1024 pages in the virtual memory.

- page fault number = 1024+3072 = 4096

2. vm_read

In the sample user program, the last 32,800B data, which is 1025 pages, are read from the physical memory. Nevertheless, the page table only contains the last 1024 pages of data. So at the last page that the user intends to read will cause a page fault

- page fault number = 4096+1 = 4097

3. vm_snapshot

The snapshot function will read data in a sequence from virtual memory address 0 to the end. However, the current page table only contains 1024 pages in the last 1025 pages. So every page that is read will cause a page fault

- page fault number = 4097+4096 = 8193

Thus the sample output for the basic task is

```
input size: 131072
page fault number is 8193
```

For the bonus part, essentially the program just run the user program for 4 times, so the output should be

```
input size: 131072
page fault number is 32772
```

What Have I Learned & What Problems Have I Met

1. Understand the Paging Mechanism

This assignment has greatly enforced my understanding about the paging system in computers.

It is not just about making memory allocation possible, but we also should pay attention to the efficiency, memory usage, and the multi-thread managing of the paging system

To use less memory, we realize that it is unnecessary to put all virtual page numbers into the page table. The advantage of inverted page table is truly shown in this assignment.

However, if we try to save memory, the increase of time complexity is inevitable. In this program, I just use the "clock time" attribute to implement the LRU, but it is quite time consuming to traverse 1024 integers in the page table to find the minimum clock time. It may be better to implement LRU with a double linked list or a stack.

I also gain better understanding that the memory for a computer is multi-leveled. The main memory (or the physical memory in this assignment) is the fastest but the smallest. Together with the secondary memory (disks) can the computer handle the huge amount of data.

2. Difficulties Designing the LRU Algorithm

For one period my output of the page fault number was 9,216

I printed out the page fault number right after `vm_write()` and it turned out to be correctly 4096. However, after `vw_read()` it became 4096+1024=5120. So I realized that there must be something wrong with the LRU algorithm.

I printed out the victim entry index of the page table and I found that it kept being 1023 (the last entry) while the program conducted `vm_write()`

After some time I found out where the bug is, at that time I designed the LRU algorithm to be

- every time that we read or write, we increase the clock time of that frame by 1, while the others remain the same
- we pick the last frame with the smallest clock time to be the victim

In this case, after the `vm_write()` write the first 1024 pages in to the page table, all of the clock time will be 1, and there is no way to distinguish the least_recent_entry from others. Particularly, the program will pick the last frame since it is the "last frame with the smallest clock time".

So I change the algorithm to the current design (see it above).