THE CNINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC3150

Operating Systems

Report for Assignment 3

Author:
Li Jingyu 李璟瑜

Student Number:
118010141

Nov 8, 2020

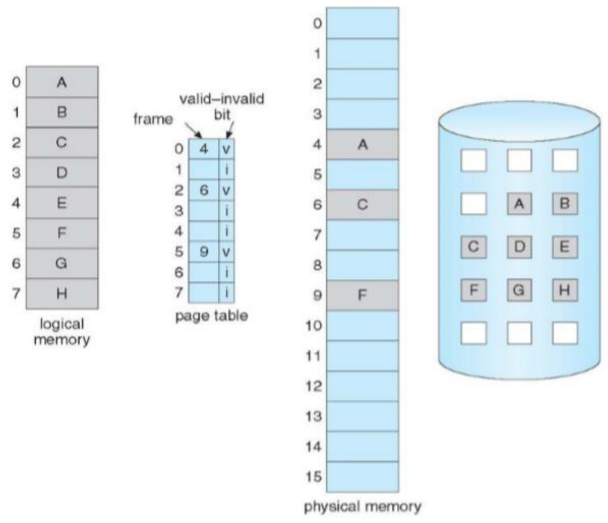
Contents

1 Design

Assignment 3 requires us to write a program which simulates the mechanism of virtual memory. Virtual memory is the logical abstraction of physical memory, which is limited in size. When user launches a program, the program will be loaded into the main memory. Virtual memory makes it seem like it has infinite memory size to use.

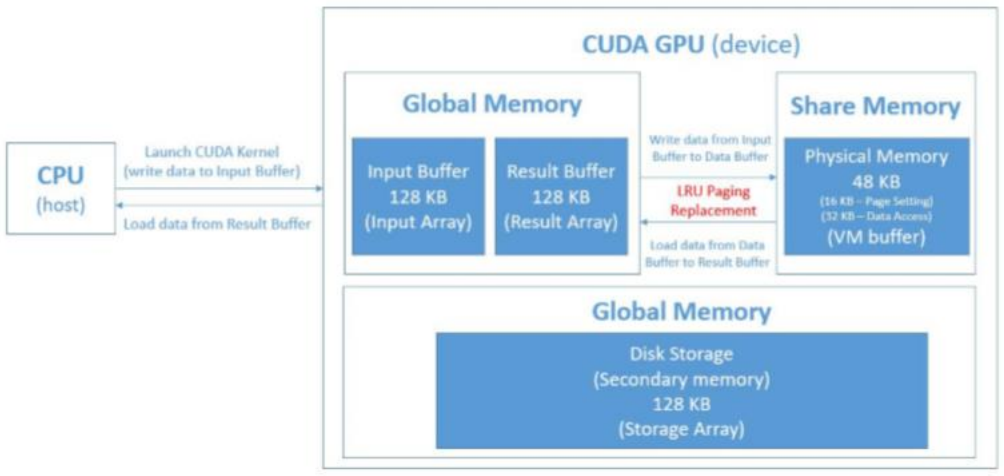
This project implements the paging algorithm. A page is a basic unit of virtual memory program block, which is 32 bytes in the program. When we access the memory, we are actually performing the operations towards pages. Each virtual memory position corresponds to a page, while each physical address corresponds to a frame, which is as large as a page. A page table is used to record the relationship between the page position in virtual address and the real physical address. Since the space to set page table is limited in this project (16KB for 1024 entries, 16bytes/entry), I implement the inverted-page table. The difference between inverted one and the traditional one is that it takes the page number as the value of each row and the key(index) is the frame number (in traditional table page number should be the key and frame number is the value). We need to go through the whole page table to access the physical position in the worst case, which takes O(N).

Another important issue is the swap algorithm. There are 1024 page entries in the page table recording 32KB main memory information, but the total data size is 4 times as large as the main memory (128KB). As a result, the page table is not sufficient to store the relationships. When there is a new page coming, an old page should be swapped out. Here the project requires us to implement LRU (least recently used algorithm). The operating system should swap the page which has been used furthest from now. The point is to replace the new memory part with the oldest one that has the least probability to be used for the program. In order to track the time/order for the page operations, extra data structures should be used. I make use of an array to achieve this. The array is attached just after the page table (1024 entries for valid/invalid bit, 1024 entries for recording virtual page number, and another 1024 entries for the LRU array). The index of the array just represents the time from current operations (0-newest, 1-1 unit before, 2-2 unit time before…, 1023-the least used one). When performing read or write operations toward a page, the page will be moved to the first entry, while the last element in the LRU array will be moved (if page table is full) and all the elements will be moved back by one step, which takes O(N). In this way we can quickly get to know the least used page number and find the corresponding frame in the main memory for swapping.



The general process to access a page is like, we first obtain a logical address (e.g.2, the data stored is C), and we check the page table, if it is in the table, then we takes the index as the frame number and access it in the main memory. If it is not in the page table, then we find the corresponding page in the disk and do the swapping (depends on read/write operations).

About the memory architecture:

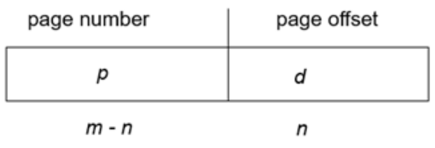


We enter the program from CPU, do the programming in CUDA GPU. Input buffer stores the data for input and result buffer stores to output. Share memory serves as the main memory (fast) which includes the space for 1024 pages and space for the page table. Global memory is for the secondary storage (disk) which is slower than share memory.

Next I will talk about my design for the three specific functions we need to change in the project, following the program logic flow.

The detailed process for vm_read():

1. From the logical address, calculate the page offset (the relative position of the uchar in a page) and the page number (addr >> page size).



2. Find the page number in the inverted page table (iterating).

   (a) If the page number is found and the valid bit is 0x00000000 (on), set the frame number as the index of that entry, and then update the LRU array, find the original position of the page number and move all the entries before back by one step. Set the first element as the current page number.

   (b) If the page number is not in the table, a page fault occurs. Page fault counter increases by 1.

   (b-1) Check whether the page table is full, if it is, take the least used page number from the LRU array, finding its corresponding frame in the page table, record the least frame. Update the page table by replacing the current page number with the least used one.

   (b-2) Update the LRU array, move the existing element back by 1 step and set the first element as the current page number.

   (b-3) Check whether there is an empty frame, if the page table is full, swap out the least used element in the physical memory to its position in disk storage (page number address). Swap in the targeting page (access by current page number) in the disk to the main memory. If the page table is not full, we just find the next entry which is invalid, setting it to valid, and swap the page in the memory. Then we update the page table, adding one row.
3. Finally, the targeted data are already put in the certain position (either it is there originally or is swapped to there) and we have known the frame number

from the previous steps. We access the vm buffer directly to read the data. The returned value is an uchar.

The detailed process for vm_write(): write and read is pretty similar, here I will simplify the procedure repeated used in vm_read().

1. From the logical address, calculate page offset and page number.
2. Find the page number in the inverted-page table.
   (a) If the page is inside the table and is valid, record the frame number (index) and update the LRU array (move back, set the first element).
   (b) If page fault occurs, increase the page fault counter.
   (b-1) Check whether the page table is full, if it is, find the least used page number and its frame number in LRU array and the page table.
   (b-2) Update the LRU array.
   (b-3) Check if there is an empty frame (invalid page table entry), if there is no empty space, swap the targeted page from disk storage to the main memory, and swap the least used page back to the disk. If there is an empty frame, just set it into valid and update the page table.
3. Finally, write the value into the position in vm buffer. The information of frame number has been obtained from the previous steps.

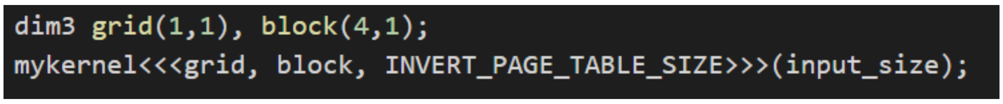
The detailed process for vm_snapshot():

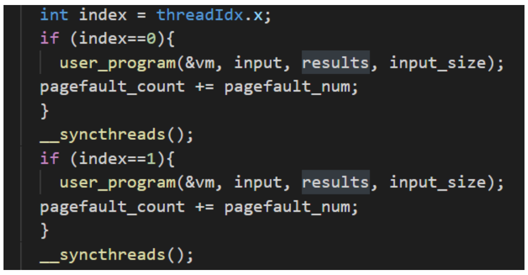
This function loads all the data from the virtual memory and disk to the result buffer. It starts from the position "offset", using vm_read() to read all the data from logical address offset to the last data. The data will be stored in result buffer.

At the end of the program execution, the total number of page fault will be printed, whose result is 8193.

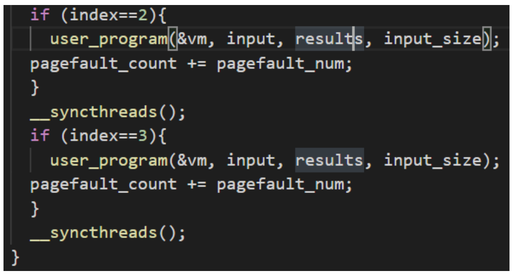
The strategy for Bonus: Bonus problem requires us to launch 4 threads to execute the program concurrently. They have strict execution order to prevent preemptive behaviors (0>1>2>3). The key step to schedule the threads is to use __syncthreads() functions that act as a barrier in the block to synchronize all the threads. It is just like a plank to stop the water falling down, and when all the water flow reaches the plank, and then it will relieve itself.

I modify the code in the main function. Define the argument in the mykernel function as 1,4 (4*1 block, 4 threads). They will concurrently execute mykernel function. Obtain the thread number using threadIdx.x, set 4 if conditions to execute the user program. There is the __synthreads function between each if condition to make sure all the threads reach that point before continue the execution, so that the priority order can be guaranteed. A variable is used to sum up all the page faults.


```
dim3 grid(1,1), block(4,1);
mykernel<<<grid, block, INVERT_PAGE_TABLE_SIZE>>>(input_size);
```



```
int index = threadIdx.x;
if (index==0){
  user_program(&vm, input, results, input_size);
pagefault_count += pagefault_num;
}
__syncthreads();
if (index==1){
  user_program(&vm, input, results, input_size);
pagefault_count += pagefault_num;
}
__syncthreads();
```



```
if (index==2){
  user_program(&vm, input, results, input_size);
pagefault_count += pagefault_num;
}
__syncthreads();
if (index==3){
  user_program(&vm, input, results, input_size);
pagefault_count += pagefault_num;
}
__syncthreads();
}
```

For more details, please refer to the codes.

2 Problems and Solutions

In this part I will discuss some problems I met and how I solved them when writing the assignment.

(a) **Problem**: Understanding of memory management in operating system. Hard to make the mechanism clear.

Solution: Go to the lecture and the tutorial. Discuss with peers to order the thoughts. Use Google, Baidu, CSDN to learn some knowledge about virtual memory, paging, swap, and so on.

(b) **Problem**: Swap strategy. At first I make it wrong. I just think that swap is simply the exchange between two page/frame positions.

Solution: Swap should be related to 3 positions, the frame in the main memory, the targeted frame in the disk to be swapped in and the frame position of the page to be swapped out.

(c) **Problem**: LRU algorithm. At first I am confused about how to implement the double linked list to finish LRU (taught in leetcode).

Solution: Use an array instead, the index just represents the accessing time of the page from now. Each time we need to move all the elements to keep the order. (Insert or move an element to the head costs O(N)).

(d) **Problem**: Fail to understand the usage of the disk. Equalize disk storage with the input buffer.

Solution: The disk is empty at the beginning. It is different from the input buffer. The variable storage[STORAGE_SIZE] is used to represent it.

(e) **Problem**: Program Bugs.

Solution: Debug. For example, I reverse the order of two arguments. I forget to add vm-> to access some variables defined in the virtual memory.

3 Environment and Execution

OS : **Win 10**

VS version: **Visual Studio 2015 (v140)**

CUDA version: **9.2**

GPU information: **NVIDIA Geforce GTX 1060**

Open Visual Studio, load the project (.sln) file, use Ctrl+F7 to compile all the .cu files, and then use ctrl+F5 to run the program.

I have printed the value of **i** in the for loop in order to view the current schedule.

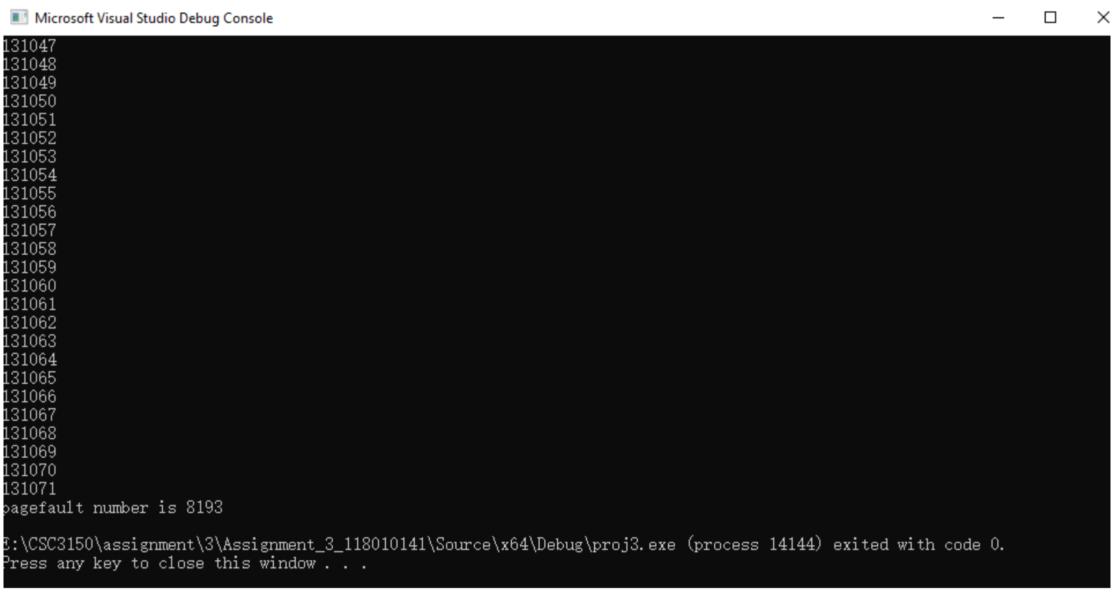
4 Output and Explanation

In this part the relevant program outputs will be shown.

Input size: 130172

```
input size: 131072
```

Final Result: The page fault number is 8193



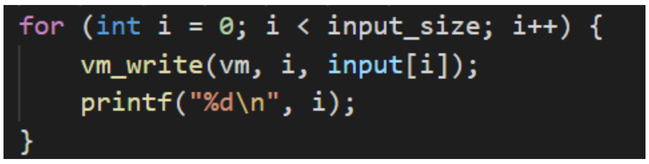
Microsoft Visual Studio Debug Console

```
131047
131048
131049
131050
131051
131052
131053
131054
131055
131056
131057
131058
131059
131060
131061
131062
131063
131064
131065
131066
131067
131068
131069
131070
131071
pagefault number is 8193

E:\CSC3150\assignment\3\Assignment_3_118010141\Source\x64\Debug\proj3.exe (process 14144) exited with code 0.
Press any key to close this window . . .
```

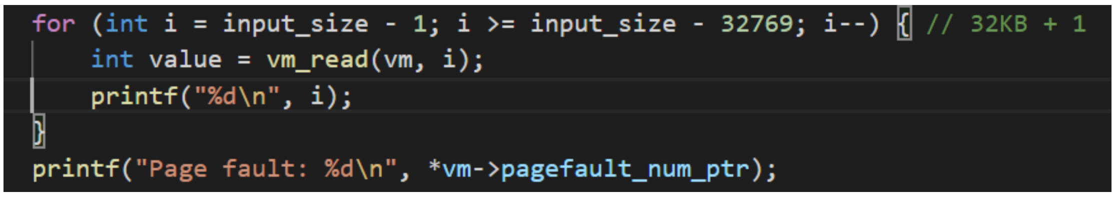
Specifically, the total pagefault number for vm_write() is 4096, and 1 for vm_read(), and 4096 for vm_snapshot().

Explanation: Page fault occurs when the targeted page is not found in the page table. When the logical address is transmitted to the function, the page number is obtained by certain calculation. The program will search the page number in the page table. As long as it does not find the page number, a page fault will occur.



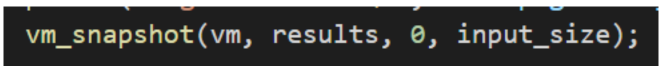
```
for (int i = 0; i < input_size; i++) {
    vm_write(vm, i, input[i]);
    printf("%d\n", i);
}
```

For the vm_write() function, at first, the page table is totally empty, logical address increases from 1 to input size (all the data bin file). Every 32 bytes form a page, so the page fault occurs every 32 iterations. Since the address just increases strictly, all the pages will not repeat. So there will be 131012/32 = **4096** page fault occurrence. (Each page will cause a page fault since they are not in the page table before: page 0-4095)



```
for (int i = input_size - 1; i >= input_size - 32769; i--) { // 32KB + 1
    int value = vm_read(vm, i);
    printf("%d\n", i);
}
printf("Page fault: %d\n", *vm->pagefault_num_ptr);
```

For the vm_read() function, the logical address starts from the last index (input_size - 1) to input_size-32769, with 32769 iterations in total. When vm_write() finishes the execution, the page number in the page table is all the last 1024 virtual addresses (page number: 3072-4095), so there is no page fault in the first 1024*32 = 32768 iterations. The only page fault occurs in the last iteration (i=input_size-32769, page numer: 3071). So there is only **1** page fault.

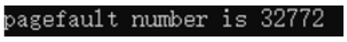


```
vm_snapshot(vm, results, 0, input_size);
```

For the vm_snapshot() function, the effect is the same as executing vm_read() for input_size times, accessing 4096 pages. The logical address also starts from 0 (offset=0). At that state, the page number in the page table is 3071-4094, but the page starts to be loaded to memory from 0. There will be no overlap in range of 1024 entries. (0-1023 will replace all the pages of 3071-4094), and the following execution is the same as the first vm_read() description. The total page_fault number is also 4096.

Finally, the total page fault number is 4096 + 1 + 4096 = 8193

Bonus output:

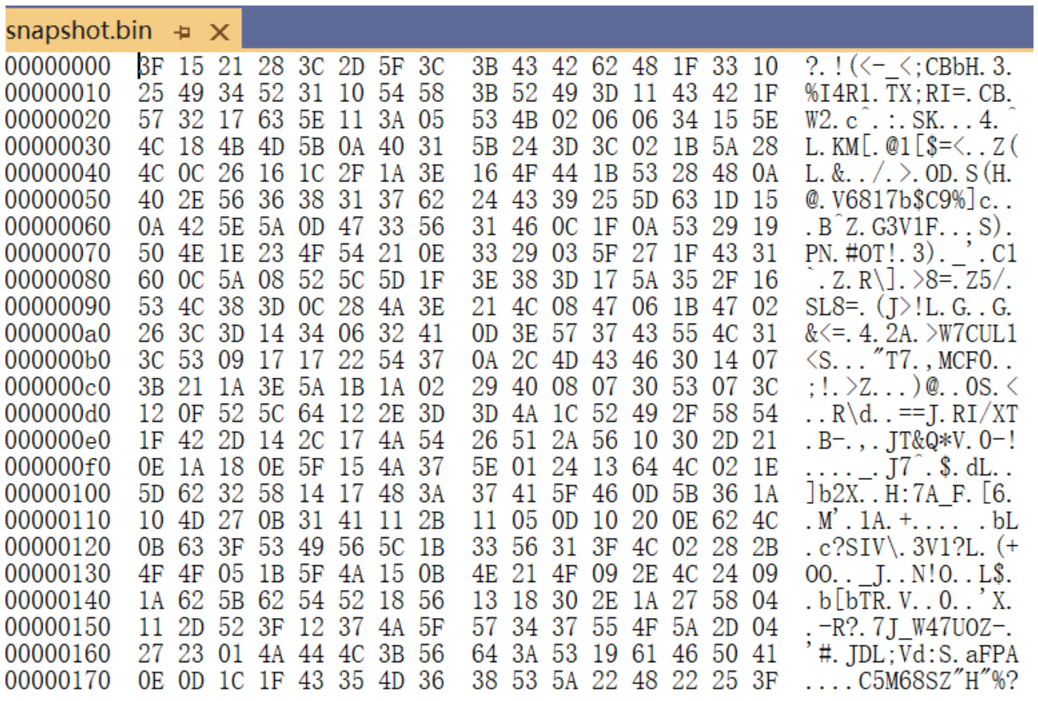


```
pagefault number is 32772
```

Page fault number is 32772. (4*8193)

Explanation: In bonus problem we create 4 threads executing the same user programs. The effect is just like using a for loop to run the user program 4 times. So the total page fault number equals to 4 times the original page fault count. (Two threads' page number will not overlap)

Snapshot.bin:


snapshot.bin

```
00000000  3F 15 21 28 3C 2D 5F 3C  3B 43 42 62 48 1F 33 10  ?.!(<-_<;CBbH.3.
00000010  25 49 34 52 31 10 54 58  3B 52 49 3D 11 43 42 1F  %I4R1.TX;RI=.CB.
00000020  57 32 17 63 5E 11 3A 05  53 4B 02 06 06 34 15 5E  W2.c^.:.SK...4.^
00000030  4C 18 4B 4D 5B 0A 40 31  5B 24 3D 3C 02 1B 5A 28  L.KM[.@1[$=<..Z(
00000040  4C 0C 26 16 1C 2F 1A 3E  16 4F 44 1B 53 28 48 0A  L.&../.>.OD.S(H.
00000050  40 2E 56 36 38 31 37 62  24 43 39 25 5D 63 1D 15  @.V6817b$C9%]c..
00000060  0A 42 5E 5A 0D 47 33 56  31 46 0C 1F 0A 53 29 19  .B^Z.G3V1F...S).
00000070  50 4E 1E 23 4F 54 21 0E  33 29 03 5F 27 1F 43 31  PN.#OT!.3)._'.C1
00000080  60 0C 5A 08 52 5C 5D 1F  3E 38 3D 17 5A 35 2F 16  `.Z.R\].>8=.Z5/.
00000090  53 4C 38 3D 0C 28 4A 3E  21 4C 08 47 06 1B 47 02  SL8=.(J>!L.G..G.
000000a0  26 3C 3D 14 34 06 32 41  0D 3E 57 37 43 55 4C 31  &<=.4.2A.>W7CUL1
000000b0  3C 53 09 17 17 22 54 37  0A 2C 4D 43 46 30 14 07  <S..."T7.,MCF0..
000000c0  3B 21 1A 3E 5A 1B 1A 02  29 40 08 07 30 53 07 3C  ;!.>Z...)@..0S.<
000000d0  12 0F 52 5C 64 12 2E 3D  3D 4A 1C 52 49 2F 58 54  ..R\d..==J.RI/XT
000000e0  1F 42 2D 14 2C 17 4A 54  26 51 2A 56 10 30 2D 21  .B-.,.JT&Q*V.0-!
000000f0  0E 1A 18 0E 5F 15 4A 37  5E 01 24 13 64 4C 02 1E  ...._.J7^.$.dL..
00000100  5D 62 32 58 14 17 48 3A  37 41 5F 46 0D 5B 36 1A  ]b2X..H:7A_F.[6.
00000110  10 4D 27 0B 31 41 11 2B  11 05 0D 10 20 0E 62 4C  .M'.1A.+.... .bL
00000120  0B 63 3F 53 49 56 5C 1B  33 56 31 3F 4C 02 28 2B  .c?SIV\.3V1?L.(+
00000130  4F 4F 05 1B 5F 4A 15 0B  4E 21 4F 09 2E 4C 24 09  OO.._J..N!O..L$.
00000140  1A 62 5B 62 54 52 18 56  13 18 30 2E 1A 27 58 04  .b[bTR.V..0..'X.
00000150  11 2D 52 3F 12 37 4A 5F  57 34 37 55 4F 5A 2D 04  .-R?.7J_W47UOZ-.
00000160  27 23 01 4A 44 4C 3B 56  64 3A 53 19 61 46 50 41  '#.JDL;Vd:S.aFPA
00000170  0E 0D 1C 1F 43 35 4D 36  38 53 5A 22 48 22 25 3F  ....C5M68SZ"H"%?
```

5 Feeling

Here I will list several feelings I have when writing the assignment.

a. Assignment 3 is different from the previous two assignments. The knowledge part is more abstract and difficult to understand. I spend more time on discussion and learning the theory before writing the code. The code just costs me several hours to finish (less than the previous two). I feel like I am writing a project of CSC1002.

b. The project introduces GPU, a powerful computing unit I did not even know before. GPU is much faster than CPU, supporting the graph computing functions. TA said we should use N card instead of A card, but I finally found that my computer has an I card (rubbish).

c. TA said the project is straightforward, and I agree with him. The main difficulty is to understand the memory management mechanism. As long as we make the logic clear, the code is easy to write.

d. Writing in TC301: This time we need to go to the computer room to use the GPU to run the program. There are 40 seats in total, but there are 150 students in this course. I went to TC301 this Saturday and Sunday afternoon, and the room is full. It makes me feel like I am back to the senior high school. In memory of the school life!
e. I also improve my programming ability a little bit. Known CUDA programming, debug, implement the logic……
f. This time lecture knowledge is useful, and tutorial notes are also practical. The materials are sufficient. However, I think the instruction is a little bit confused.
g. By the way, TA speaks clear English. Good.

That’s all.