

Report for CSC3150 Assignment 3

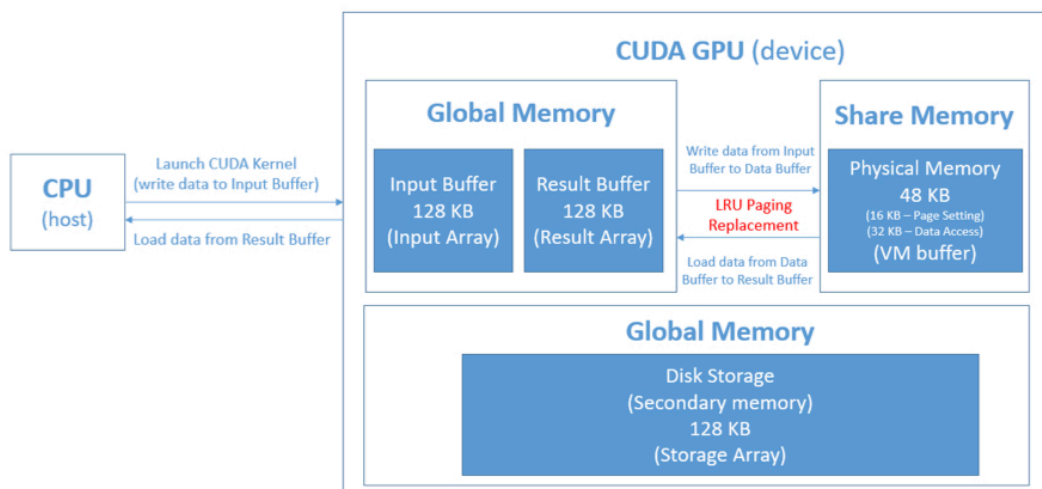
Yang Liang 120090874

1. Program Design Methodology

1.1 Program Task

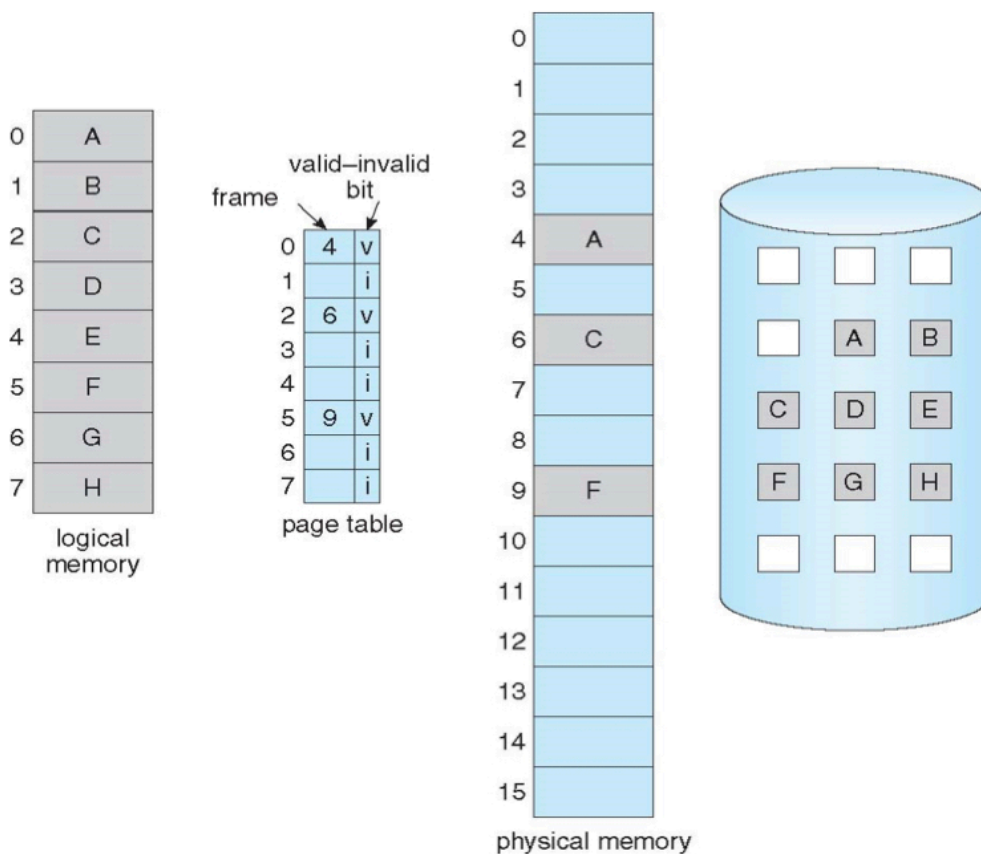
In this assignment, we are required to simulate the mechanism of virtual memory via GPU's memory. In the GPU given, there is a secondary memory (disk) of size 128KB, which is in the global memory area of the CUDA GPU. The physical memory is in the share memory space of the CUDA GPU, which is of size 48KB, where 32KB is used for storing data, and 16KB is used for page table settings. The main workflow of the stimulated process is that: first, data in file *data.bin* is loaded to *input buffer*, which is an array of size 128KB, in the global memory area. Then, those data in the *input buffer* will be loaded to the *data buffer* in physical memory, by the *vm_write()* function we need to implement. Finally, they will be loaded back to the *result buffer*, again an array of size 128KB, with the help of the *vm_read()* function and *vm_snapshot()*. Function *vm_read()* typically gets the data, and *vm_snapshot()* loads them one by one back to the result buffer (an array).

Since the physical memory for data is of size 32KB, much smaller than the result buffer, swap must happen, to swap the data to the disk. To determine which page to be swapped out, an LRU page replacement algorithm is to be implemented.



1.2 Implementation Details

In actual design, we need to tackle 5 objects, logical memory, page table, physical memory, disk storage and an additional swap table. They have some internal relationships. Paging technique means to divide logical memory and physical memory into pages of same size, here, it is 32Bytes. Therefore, it allows for the fact that logical memory can be greater than the physical memory (logical memory can have more pages than physical memory, but the size of each page should be the same), there is a mapping relationship between the frame number (of the physical memory) and the page number (of the logical memory). This mapping information is stored in the page table. Here, we are required to implement an inverted page table, where the row number of the table is the frame number, the value stores in the table is a sequence which contains the page number of the virtual memory. So, the number of the entries (rows) of the page table equals to the total frame number of physical memory.



For the swap table I implement, it records the relationship between the logical memory and the frame number in the disk, where the disk frame number is the row index, and the value stores

in the table is a 32-bit sequence which contains the page number of the virtual memory.

In the inverted page table, each row can contain 32-bit information. Since there only need to be maximum 4096 pages (logical memory) required in this scenario, i.e., the page number occupies at most 13 bits, therefore, we can utilize the rest bits to store other information. (This method of storing information in bits was inspired by my classmate, Liu Qi, but the actual implementation is indeed completely done by me). So, we can utilize this 32-bit sequence to store 4 information.

<i>Valid/invalid bit</i>	<i>process pid</i>	<i>virtual page number</i>	<i>time that it was last referenced</i>
<i>1bit</i>	<i>2bit</i>	<i>13bit</i>	<i>16bit</i>

In the swap table, since it can be declared without occupying the 128KB storage memory, we can declare it in the global memory with `__device__ __managed__`. We allocate a memory size `SWAP_TABLE_SIZE` of 2^{16} bytes, 4 times the size of the page table, since there are totally 4096 rows in the swap table. For simplicity, each row also stores a 32-bit data (it may seem to be sort of waste of storage space to do so...), with the bit-arrangements as below.

<i>Valid/invalid bit</i>	<i>Process pid</i>	<i>Virtual page number</i>
<i>1 bit</i>	<i>2 bit</i>	<i>29 bit (a bit waste of space...)</i>

In `vm_write()` function, we are given the virtual memory address (`addr`), and we need to find its actual location (either in the physical memory or in the disk), then we swap it back to the physical memory (if needed), and writes the value to this specific place (physical memory address). We typically deal with 4 cases, and the workflow is shown below.

1. Search the page table for the frame number, given a virtual page number (virtual page number can be derived in the virtual memory address parameter).
2. **(case 1)**

If it is found, which means that this virtual page block is already stored in the physical memory. We then directly retrieve its corresponding frame number (the index of the page

table), the physical address that is mapped by the virtual memory address given is

$$\text{actual physical address} = \text{frame number} * 32 + \text{offset}.$$

We then write the data to the given place in the data buffer,

$$\text{buffer}[\text{actual physical address}] = \text{value}.$$

(case 2)

If it is not found, which means that this virtual page block is not in the physical memory. Page fault number will be increased by 1. We then need to go to the disk to further search for it, we can search in the swap table.

(case 2.1) if there is still empty space in the page table, which means the physical memory is not full, also, we are able to find that page block in the disk by the swap table.

We can then retrieve its corresponding disk number (the index of the swap table), and then put the block from the disk back to the empty block in the physical memory. We can then write the data to the given place in the data buffer with the same way as case 1.

(case 2.2) if there is no empty space in the page table, which means the physical memory is now full, and we are still able to find that page block in the disk by the swap table.

Now, swapping occurs. We need to choose which frame in the physical memory to be swapped with the target block in the disk by the LRU algorithm. This can be determined by the LRU-timer information stored in the 32-bit sequence in the page table mention above. We will swap out the block with the largest timer, since it is the least referenced one. We then swap it out and swap back the target block in the disk to the physical memory. Then write the data to the given place in the data buffer with the same way as case 1.

(case 2.3) if there is no empty space in the page table, which means the physical memory is now full, and we are not able to find that page block in the disk by the swap table. In this case, we can simply find the empty block in the disk, and swap it to the physical memory. This scenario also requires swapping. Again, we need to choose the frame with largest LRU-timer information in the physical memory to be swapped out, with the target empty block in the disk swapped into the physical memory. Data will then be written to this empty block. Next, we write the data to the given place in the data buffer with the same way as case 1.

Please note that we need to update the information of LRU-timer bit in the 32-bit sequence. In

each `vm_write()`, we need to increment the LRU-timer bit of all the entries in the page table by one, and reset the one referenced just now back to 1. The information in the swap table and page table is also required to be updated, we need to update the contents (pages) stored in the physical memory (recorded in the page table), and the one stores in the disk (recorded in the swap table).

We can implement `vm_write()` the same way as `vm_read()`, since they are the reverse operations, since `vm_write()` writes (modifies) information (*value*) to the corresponding index (physical memory address number) of data buffer, as
buffer[actual physical address]=value,
while `vm_write()` retrieves the information (*info*) out, as
value=buffer[actual physical address].

But worth mentioning, (case2.3) never occurs in `vm_read()`, because it is impossible to read out information of a virtual page number that is stored neither in the disk nor in the physical memory. In actual implementation, a write operation always occurs before a read operation.

1.3 Bonus

In the bonus part, we are required to provide multithread support (here, 4 threads is sufficient) to the virtual memory simulation. According to the *Addition notes on Assignment 3_updated.pdf*, we can choose the below 3 versions to implement the task. **I will choose to implement version 1**, which is to divide the manipulation of this task into 4 threads, for example, thread with `pid = 0` is responsible for reading and writing `addr % 4 == 0` task, thread with `pid = 1` is responsible for reading and writing `addr % 4 == 1` task.

In order to launch a CUDA kernel with 4 threads, we need to modify the second parameter to 4, according to the syntax of CUDA programming, in *main.cu*.

```
/* Launch kernel function in GPU, with single thread
and dynamically allocate INVERT_PAGE_TABLE_SIZE bytes of share memory,
which is used for variables declared as "extern __shared__" */
mykernel<<<1, 4, INVERT_PAGE_TABLE_SIZE>>>(input_size);
```

The below tutorial slides reflect the essence of CUDA multithread support. For example, to

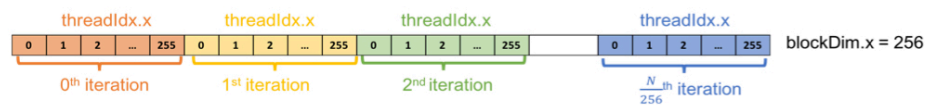
execute the above `vector_add()` function, if there is only 1 thread, we have to iterate n times to finish execution, each iteration calculate the output of 1 element of the output array. However, with more threads, say, 256, we are able in a parallelism way. For example, in the 0-th iteration, the k -th thread computes the addition of the k -th element, in the next iteration, it will compute the addition of $(k+256)$ -th element, if there are 256 threads. Other threads operate in the same manner. By parallelism with multiple threads, the number of iterations needed to calculate the sum of n -element vector will be reduced to $\frac{n}{256}$.

CUDA Example

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    int index = threadIdx.x;
    int stride = blockDim.x;

    for(int i = index; i < n; i += stride){
        out[i] = a[i] + b[i];
    }
}

// Executing kernel
vector_add<<<1,256>>>>(d_out, d_a, d_b, N);
```



Based on the thoughts above, we implement the Version 1, we can distribute our task of reading and writing data, manipulating page faults to 4 threads, one way of distribution is that thread $N(N=1,2,3,4)$ will handle the read and write of data with address whose remainder is N after divide by 4.

After launching the kernel with multithread, we are able to use the 4 threads to execute the same function, with fewer iterations ($input_size/4$). Each CUDA thread has its own thread id, called `threadIdx.x`, we can use this thread id to control them to only execute certain address as the way of distribution we mentioned above. This can be achieved with an *if* statement as shown below. For example, consider a thread with `threadIdx.x = 3`, it will only execute those address that satisfies the *if* condition

If ($addr \% 4 == threadIdx.x$)

Else, this thread won't execute this `vm_read()` or `vm_write()` operations. Therefore, we achieve our target of parallelism that thread with id = 1 will only deal with read or write of address

1,5,9,..., $4k+1$, thread with id = 2 will only deal with read or write of address 2,6,10,..., $4k+2$, so on so forth. We also adopt the use of CUDA thread function `__syncthreads()`, which can be used to synchronize multiple threads in the kernel. It can be viewed as a barrier, where the process won't continue unless all the processes have reached it.

```
__device__ uchar vm_read(VirtualMemory *vm, u32 addr) {
    /* Complete vm_read function to read single element from data buffer */
    __syncthreads();
    if (addr % 4 == threadIdx.x){
        //printf("[thread %d] reading %d\n", threadIdx.x, addr);
        int disk_index;
        uchar value;
```

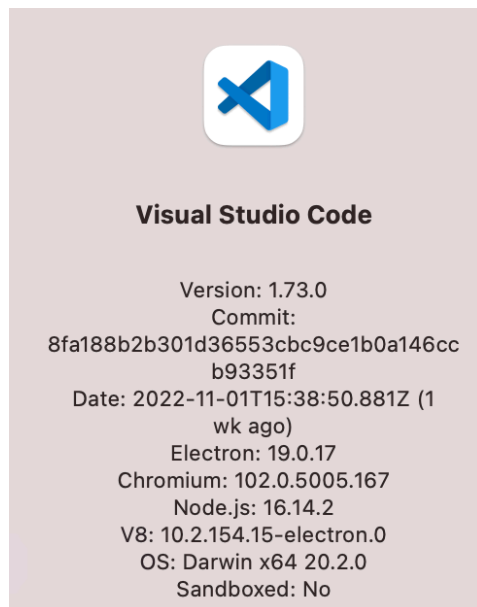
2. Program Execution Environment

2.1 OS



macOS Big Sur
版本 11.1

2.2 VS version



2.3 CUDA Version and GPU information (Cluster Environment)

Cluster Environment

The following information holds for every machine in the cluster.

Item	Configuration / Version
System Type	x86_64
Operating System	CentOS Linux release 7.5.1804
CPU	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz 20 Cores, 40 Threads
Memory	100GB RAM
GPU	Nvidia Quadro RTX 4000 GPU x 1
CUDA	11.7
GCC	Red Hat 7.3.1-5
CMake	3.14.1

3. Program Execution Commands

Please use the commands below in the terminal to execute the program in the cluster, for both the main task and the bonus.

```

● [120090874@node21 ~]$ cd /nfsmnt/120090874/A3-template
○ [120090874@node21 A3-template]$ sbatch ./slurm.sh

```

4. What is Learned from this Assignment

I think this is the hardest assignment by now in this course. But the difficulty is on understanding the concepts and some underlying principles on virtual memory, physical memory, page table, disk, swapping, page replacement. These things cannot be simply covered in one or two tutorial sessions, it requires us to read the textbooks and search for information online, discussing with classmates is also very essential. To finish this assignment, we need to fully understanding the concepts above and we need to think of many different cases in reading and writing data (for example, in my implementation, I divide 3 cases for *vm_read()*, and 4 cases for *vm_write()*). If we managed to make these process clear, the actual implementation may not be that hard. Therefore, I think understanding the concepts is very important to finishing this assignment.

By implementing the project, I indeed get a deeper understanding with the virtual memory management, data swapping and page replacement. The implementation requires a lot of memory address manipulation, and I am now more familiar with manipulating address in C programming. What's more, I learned a bit of CUDA programming and some of its basic syntax.

5. Problems met in this assignment

(1) Understanding the concepts


Search online, read the textbooks, discuss with classmates, listen to the lectures and tutorials

(2) Debugging issues

1. Manipulating memory address using operators such as `&`, `|` (should be very careful with this, since I accidentally write the wrong direction for the shifting operators for several times)
2. Updating the page tables, swap tables when swapping occurs. (since there are too many cases, so we should be very clear about what should be put into the page table and swap table after swapping)

6. Screenshots of the Program Outputs

6.1 Explanation on Page Fault Number of `user_program.cu`

```
A3-template >  user_program.cu
1  #include "virtual_memory.h"
2  #include <cuda.h>
3  #include <cuda_runtime.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  //test case 1
8  __device__ void user_program(VirtualMemory *vm, uchar *input, uchar *results,
9                               int input_size) {
10
11     for (int i = 0; i < input_size; i++){
12         vm_write(vm, i, input[i]);
13     }
14
15     for (int i = input_size - 1; i >= input_size - 32769; i--)
16         int value = vm_read(vm, i);
17
18     vm_snapshot(vm, results, 0, input_size);
19 }
```

The output of the above `user_program.cu` should be 8193. The page fault number of the 3 for loops are 4096, 1 and 4096, respectively. The reason is explained as below.

For the first `vm_write()` section, we write `input_size = 128KB` data ($128KB/32B=4096$ pages) into the physical memory. Since the physical memory and the page table is empty initially, so write operation on each page will lead to a page fault. After we have written 32KB of data, the data buffer will be full (page table will also be full), then swapping may occur (page fault also will

occur). We can see that the request virtual memory address number (in the program, it is the iteration variable i) is increasing, which means it is impossible to find a page hit between i and the page number currently stored in the page table. Therefore, every page write will lead to a page fault. At the end of the first *for* loop, the data in the [96K,128K] (logical address) place of *data.bin* will be stored in the physical memory, and data in the [0K,96K] place of *data.bin* is in disk. Totally, there are $128\text{KB}/32\text{B} = 4096$ page faults.

For the second *vm_read()* loop section, we read out data with logical address from (128KB-1) to (96KB-1). As analyzed above, after the first *vm_write()* loop section, the data stored in the physical memory is the data in the [96K,128K] (logical address) place of *data.bin*, thus, only 1 read page fault will occur, which occurs when attempting to read out data in (96KB-1) logical memory.

The analysis on page fault number for the *vm_snapshot()* is similar to the first *vm_read()* loop section. We want to read out data with logical memory starting from 0 to 128K, but currently the data stored in physical memory is [96K,128K] (logical address) place of *data.bin*. Therefore, reading and loading on every page will lead to a page fault, which is $128\text{KB}/32\text{B} = 4096$ page faults altogether in this section.

To sum up, the total page fault is $8193 = 4096 + 1 + 4096$.

6.2 Output of Main task

(1) When execute the *user_program.cu*

```
A3-template > ≡ result.out
1  main.cu(101): warning #2464-D: conversion from a string literal to "char *" is deprecated
2
3  main.cu(121): warning #2464-D: conversion from a string literal to "char *" is deprecated
4
5  main.cu(101): warning #2464-D: conversion from a string literal to "char *" is deprecated
6
7  main.cu(121): warning #2464-D: conversion from a string literal to "char *" is deprecated
8
9  input size: 131072
10 pagefault number is 8193
11
```

Theoretically, the content in *snapshot.bin* should be identical to *data.bin* after the program. We can use *cmp* command to check, no addition output for this command means is identical.

```

● [120090874@node21 A3-template]$ sbatch ./slurm.sh
Submitted batch job 26864
● [120090874@node21 A3-template]$ cmp data.bin snapshot.bin
○ [120090874@node21 A3-template]$ █

```

(2) When execute the additional test case given by TA.

```

// //test case 2
__device__ void user_program(VirtualMemory *vm, uchar *input, uchar *results, int input_size) {
//   write the data.bin to the VM starting from address 32*1024
  for (int i = 0; i < input_size; i++)
  |   |   vm_write(vm, 32*1024+i, input[i]);
//   write some data (32KB-32B) to the VM starting from 0
  for (int i = 0; i < 32*1023; i++)
  |   |   vm_write(vm, i, input[i+32*1024]);
//   readout VM[32K, 160K] and output to snapshot.bin, which // should be the same with data.bin
  vm_snapshot(vm, results, 32*1024, input_size);
}

```

The output is

```

A3-template > ≡ result.out
1  main.cu(101): warning #2464-D: conversion from a string literal to "char *" is deprecated
2
3  main.cu(121): warning #2464-D: conversion from a string literal to "char *" is deprecated
4
5  main.cu(101): warning #2464-D: conversion from a string literal to "char *" is deprecated
6
7  main.cu(121): warning #2464-D: conversion from a string literal to "char *" is deprecated
8
9  input size: 131072
10 pagefault number is 9215
11

```

Theoretically, the content in *snapshot.bin* should be identical to *data.bin* after this test case. Again, we use *cmp* command to check and compare.

```

● [120090874@node21 A3-template]$ sbatch ./slurm.sh
Submitted batch job 26877
● [120090874@node21 A3-template]$ cmp data.bin snapshot.bin
○ [120090874@node21 A3-template]$ █

```

6.3 Output of Bonus

I choose the **Version 1** (mentioned in *Additional notes on Assignment3_updated.pdf*) way of implementation. When running on the *user_program.cu* (the first test case given). Theoretically, **Version 1** requires us to use 4 threads to collaborately perform the test when implementing version 1, therefore, the result should be same as using a single thread.

```

A3-template-bonus > cat result.out
1  main.cu(101): warning #2464-D: conversion from a string literal to "char *" is deprecated
2
3  main.cu(121): warning #2464-D: conversion from a string literal to "char *" is deprecated
4
5  main.cu(101): warning #2464-D: conversion from a string literal to "char *" is deprecated
6
7  main.cu(121): warning #2464-D: conversion from a string literal to "char *" is deprecated
8
9  virtual_memory.cu(186): warning #940-D: missing return statement at end of non-void function "vm_read"
10
11 input size: 131072
12 pagefault number is 8193
13

```

When we compare the *data.bin* and *snapshot.bin*, the output should still be the same.

```

● [120090874@node21 A3-template-bonus]$ sbatch ./slurm.sh
Submitted batch job 29293
● [120090874@node21 A3-template-bonus]$ cmp data.bin snapshot.bin
○ [120090874@node21 A3-template-bonus]$ █

```

--- End of Report ---