Assignment4 Report

Name: Huang Yingyi

Student ID: 119010114

File Tree

```
.
├── bonus
|   ├── data.bin
|   ├── main.cu
|   ├── Makefile
|   ├── snapshot0.bin
|   ├── snapshot1.bin
|   ├── snapshot2.bin
|   ├── snapshot3.bin
|   ├── user_program.cu
|   ├── virtual_memory.cu
|   └── virtual_memory.h
├── report.pdf
└── source
    ├── data.bin
    ├── main.cu
    ├── Makefile
    ├── snapshot.bin
    ├── user_program.cu
    ├── virtual_memory.cu
    └── virtual_memory.h
```

Under **source** directory:

> data.bin: input data to the virtual memory
>
> main.cu: main file for the program
>
> Makefile: help compile the program
>
> snapshot.bin: data output from the virtual memory
>
> user_program.cu: you can change it for another program reading or writing to the virtual memory
>
> virtual_memory.h: header file for the virtual_memory.cu
>
> virtual_memory.cu: our implementation of read / write operations on the memory

Under **bonus** directory:

> data.bin: input data to the virtual memory
>
> main.cu: main file for the program
>
> Makefile: help compile the program
>
> snapshot0.bin: data output from the virtual memory from after the thread0 user program terminates
>
> snapshot1.bin: data output from the virtual memory from after the thread0 user program terminates
>
> snapshot2.bin: data output from the virtual memory from after the thread0 user program terminates
>
> snapshot3.bin: data output from the virtual memory from after the thread0 user program terminates
>
> user_program.cu: you can change it for another program reading or writing to the virtual memory
>
> virtual_memory.h: header file for the virtual_memory.cu
>
> virtual_memory.cu: our implementation of read / write operations on the memory

1. Running Environment

For the **homework & bonus** program:

Operating System

```
cat /etc/redhat-release
```

CUDA Version

```
nvcc --version
```

```
cat /usr/local/cuda/version.txt
```

GPU Information

```
lspci | grep -i vga
```

NOTICE: the machine is located at TC301 room, the $12th$ machine.

2. Execution Steps

Homework

```
cd <DIRCT_TO_PROJ>                          # come to the project
directory
cd source                                   # go to the source file
make all                                    # compile all the files
./test                                      # execute the program
make clean                                  # clean all the compiled files
diff data.bin snapshot.bin                  # compare the input and output
data bin
diff <(xxd data.bin) <(xxd snapshot.bin)    # compare to view dirty data
```

Bonus Task

```
cd <DIRCT_TO_PROJ>                          # come to the project
directory
cd bonus                                    # go to the bonus file
make all                                    # compile all the files
./test                                      # execute the program
make clean                                  # clean all the compiled files
diff data.bin snapshot0.bin                 # compare the input and output
data bin
diff data.bin snapshot1.bin
diff data.bin snapshot2.bin
diff data.bin snapshot3.bin
diff <(xxd data.bin) <(xxd snapshot0.bin)   # compare to view dirty data
diff <(xxd data.bin) <(xxd snapshot1.bin)
diff <(xxd data.bin) <(xxd snapshot2.bin)
diff <(xxd data.bin) <(xxd snapshot3.bin)
```

3.Progame Design

Homework Program

Overall Ideas

In this program we are going to implement the **write, read, and snapshot** functions to the virtual memory.

The memory system in this program is consist of 3 parts, which are disk, physical memory for data, physical memory for page table. We can use virtual page number as the index of disk and physical page number as the index of physical memory. The page table is inverted, in which the index is physical page number and the value is {valid, dirty, pid, count, virtual page number}.

In the three kinds of operation, we first get to the page table, check the valid entry that whether the target virtual page number appears in the value. If yes, we get a **page hit**. Otherwise, we get into a **page fault**. To solve page fault, we check whether the page table is full. If there are positions in the page table, we insert the target page from disk into the physical memory and update the page table entry. If the table is full, we swap the least recently used page, write back to disk based on write back policy, and insert the new page into physical memory, update the page table.

How to Make a Good Write Back Policy

Instead of write back the page in physical memory to disk in every page swap, we should figure out a better solution. We maintain a dirty bit for the page table entry. Once there is a **write** to the page, we update this bit as 1. When there is a page swap, we check the dirty bit of the page table entry. If the dirty bit is 1, we write it back into the disk. Otherwise, we replace the page directly.

How to count the page fault

If the memory operation (read / write) cannot find a corresponding entry in the page table, we can conclude that the target page is not in the physical memory. Before doing the next step, we add 1 to the **global** number count of page fault.

Bonus Program

Overall Ideas

The overall ideas are the same as the one of required task. The major difference is that we simulate the memory access of 4 processes by CUDA threads. We let this 4 threads to invoke the **user_program()** function one by one. That is, only when the ith thread's **user_program()** returns, the $i+1th$ thread invoke its **user_program()**.

How to Isolate Memory Address Space of Different Processes

Since the memory address space of different processes should be different, we maintain the pid value in page table using **2-bit** to verify the page owners. Only when the process id (thread id indeed) and the virtual page number are checked consistent with the target page in a valid page table entry, we conclude it as a page hit.

Thread Scheduling

To make the 4 threads execute in sequence, we make use of the CUDA barrier function **__syncthreads()** for thread synchronization.

4. Test Results

Required Homework - Provided User Program

Compilation

```
cd <DIREC_TO_PROJ>
cd source
make all
```

Execution

```
./test
```

Clean the Compiled Files

```
make clean
```

Result Validation

The Number of Page Faults

In the test user program, we firstly invoke **vm_write()** to write $4K$ pages into the virtual memory, which will introduce $4K$ page faults. The writing order is from page 0 to page $4K$.

Then we invoke **vm_read()** to read $1k+1$ pages in the virtual memory, from page $4K$ to $3k-1$, which introduces 1 page faults.

After all, we invoke **vm_snapshot()** to get a snapshot of the virtual memory, which invokes **vm_read()** from 0 to $4K$, introducing $4K$ page faults.

In sum, there are $4K + 4K + 1 = 8193$ pages faults, **consistent** with our output results.

The snapshot of virtual memory

We compare the input data bin and the snapshot data bin.

```
diff data.bin snapshot.bin
```

They are **identical**! Our results for the provided user program is **correct**!

Required Homework - My User Program

Here are some of my user programs for test.

In the first two examples we want to figure out **whether the page fault counter works well**. Therefore, we implement two user programs, in which we change the number of page fault introduced.

```
// program1: expected 4096+4096 = 8192 page faults

__device__ void user_program(VirtualMemory *vm, uchar *input, uchar
*results,
                             int input_size) {
  for (int i = 0; i < input_size; i++)
    vm_write(vm, i, input[i]);

  for (int i = input_size - 1; i >= input_size - 32768; i--)
    int value = vm_read(vm, i);

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

The result is correct:

```
// program2: expected 4096+4096+1+1024 = 9217 page faults

__device__ void user_program(VirtualMemory *vm, uchar *input, uchar
*results,
  int input_size) {
for (int i = 0; i < input_size; i++)
vm_write(vm, i, input[i]);

for (int i = input_size - 1-32768; i >= input_size - 32769-32768; i--)
int value = vm_read(vm, i);

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

The result is correct:

In the last user program we want to test **whether the write back policy works well.**

```
// user program3: 4096+1+2+4096 = 8195 page faults

__device__ void user_program(VirtualMemory *vm, uchar *input, uchar
*results,
                             int input_size) {
  for (int i = 0; i < input_size; i++)
    vm_write(vm, i, input[i]);

  for (int i = 0; i < 16; i++)
    vm_write(vm, i, (uchar)i);

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

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

In this version of user program, we try to write $0 - 15$ to virtual address $0 - 15$ and let these 16 pages written back into the memory, introducing 8195 page faults. The following is the test result:

```
diff <(xxd data.bin) <(xxd snapshot.bin)  # compare to view dirty data
```

In the result picture we can see that the first 16 address positions the correct value $0 - 15$. And the page fault number is correct. The write back policy functions well.

Bonus Task - Provided User Program

Compilation

```
cd <DIREC_TO_PROJ>
cd bonus
make all
```

Execution

```
./test
```

Clean the Compiled Files

```
make clean
```

Result Validation

The Number of Page Faults

We let 4 threads to execute the **user_program()** function one by one. Therefore, we firstly get the page fault number after one thread finishes its function.

In the test user program, we firstly invoke **vm_write()** to write $4K$ pages into the virtual memory, which will introduce $4K$ page faults. The writing order is from page 0 to page $4K$.

Then we invoke **vm_read()** to read $1k+1$ pages in the virtual memory, from page $4K$ to $3k-1$, which introduces 1 page faults.

After all, we invoke **vm_snapshot()** to get a snapshot of the virtual memory, which invokes **vm_read()** from 0 to $4K$, introducing $4K$ page faults.

In sum, there are $4K+4K+1=8193$ pages faults, **consistent** with our output results. For 4 threads, there are $8193*4=32772$.

The snapshot of virtual memory

We compare the input data bin and the snapshot data bin.

```
diff data.bin snapshot0.bin
diff data.bin snapshot1.bin
diff data.bin snapshot2.bin
diff data.bin snapshot3.bin
```

They are **identical**! Our results for the provided user program is correct!

Bonus Task - Provided User Program

Here are some of my user programs for test.

In the first two examples we want to figure out **whether the page fault counter works well**. Therefore, we implement two user programs, in which we change the number of page fault introduced.

```
// program1: expected 4096+4096 = 8192 page faults for each thread: 32768

__device__ void user_program(VirtualMemory *vm, uchar *input, uchar
*results,
                             int input_size) {
  for (int i = 0; i < input_size; i++)
    vm_write(vm, i, input[i]);

  for (int i = input_size - 1; i >= input_size - 32768; i--)
    int value = vm_read(vm, i);

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

The result is correct:

```
// program2: expected 4096+4096+1+1024 = 9217 page faults for each
thread: 36,868

__device__ void user_program(VirtualMemory *vm, uchar *input, uchar
*results,
  int input_size) {
for (int i = 0; i < input_size; i++)
vm_write(vm, i, input[i]);

for (int i = input_size - 1-32768; i >= input_size - 32769-32768; i--)
int value = vm_read(vm, i);

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

The result is correct:

In the last user program we want to test **whether the write back policy works well.**

```
// user program3: 4096+1+2+4096 = 8195 page faults for each: 32,780

__device__ void user_program(VirtualMemory *vm, uchar *input, uchar
*results,
                             int input_size) {
```

```
  for (int i = 0; i < input_size; i++)
    vm_write(vm, i, input[i]);

  for (int i = 0; i < 16; i++)
    vm_write(vm, i, (uchar)i);

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

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

In this version of user program, we try to write $0-15$ to virtual address $0-15$ and let these 16 pages written back into the memory, introducing 8195 page faults. The following is the test result:

```
diff <(xxd data.bin) <(xxd snapshot0.bin)  # compare to view dirty data
diff <(xxd data.bin) <(xxd snapshot1.bin)
diff <(xxd data.bin) <(xxd snapshot2.bin)
diff <(xxd data.bin) <(xxd snapshot3.bin)
```

In the result picture we can see that the first 16 address positions the correct value $0-15$. And the page fault number is correct. The write back policy functions well.

5. Conclusion

According to the tests results, we can conclude that the design and code implementation of these two programs are successful.

In these tasks, I learnt:

- We should make good use of the space of page table to include more useful information in small space if possible.
- Page swap is really complicated and requires design to reduce it or make it more efficient. LRU can reduce the number of page swap based on the locality. A good write back policy can reduce the overhead of page swap.
- It is important to make the memory address space isolated among processes to avoid memory access error.