CSC3150 Assignment 3

Homework Requirements

Environment

- We recommend you to do the next two assignments using 1) the cluster, and 2) computers in TC301 classroom (40 available PCs). Please compile and test your code on the cluster before submission. (The cluster and slurm manual is on CSC4005_Slurm User Guide · GitHub, which will be introduced in tutorials)

Submission

- **Due on: 23:59, Nov 9. 2022**
- Please note that, TAs may ask you to explain the meaning of your program, to ensure that the codes are indeed written by yourself. Please also note that we will use a plagiarism detector to check if your program is too similar to the code of previous years' students.
- Violation against the format requirements will lead to grade deduction.

Here is the format guide. The project structure is illustrated as below. You can also use `tree` command to check if your structure is fine. Structure mismatch would cause grade deduction.

```
main@ubuntu:~/Desktop/Assignment_3_<student_id>$ tree
.
├── bonus
|   ├── data.bin
|   ├── main.cu
|   ├── slurm.sh
|   ├── user_program.cu
|   ├── virtual_memory.cu
|   └── virtual_memory.h
├── report.pdf
└── source
    ├── data.bin
    ├── main.cu
    ├── slurm.sh
    ├── user_program.cu
    ├── virtual_memory.cu
    └── virtual_memory.h
2 directories, 13 files
```

Visual Studio project files WILL NOT BE ACCEPTED! Please compress all files in the file structure root folder into a single zip file and **name it using your student id** as the code showing below and above, for example, Assignment_3_120010001.zip. The report should be submitted in the format of **pdf**, together with your source code. Format mismatch would cause grade deduction. Here is the sample step for compress your code.

```
main@ubuntu:~/Desktop$ zip -q -r Assignment_3_<student_id>.zip
Assignment_3_<student_id>
main@ubuntu:~/Desktop$ ls
Assignment_3_<student_id>                    Assignment_3_<student_id>.zip
```

Task Description

In Assignment 3, you are required to simulate a mechanism of virtual memory via GPU's memory.

Background:

- Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.
- In this project, you should implement simple virtual memory in a kernel function of GPU that have single thread, limit shared memory and global memory.
- We use CUDA API to access GPU. CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model.
- We don't consider any parallel computing technique in this project, only use single thread to serial access that let us focus our virtual memory implementation.
- There are many kinds of memory in CUDA GPU, we only introduce two memories (global memory and shared memory) which relate to our project.
- Global memory
-
    - Typically implemented in DRAM
    - High access latency: 400-800 cycles
- Shared memory

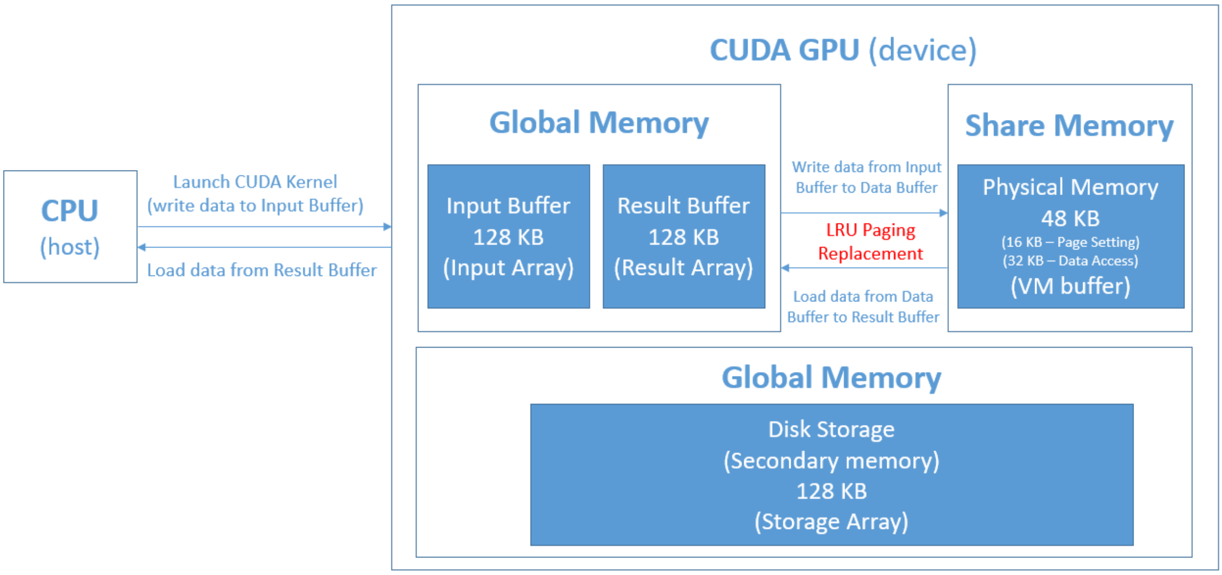
- - Extremely fast
  - Configurable cache
  - Memory size is small (16 or 48 KB)

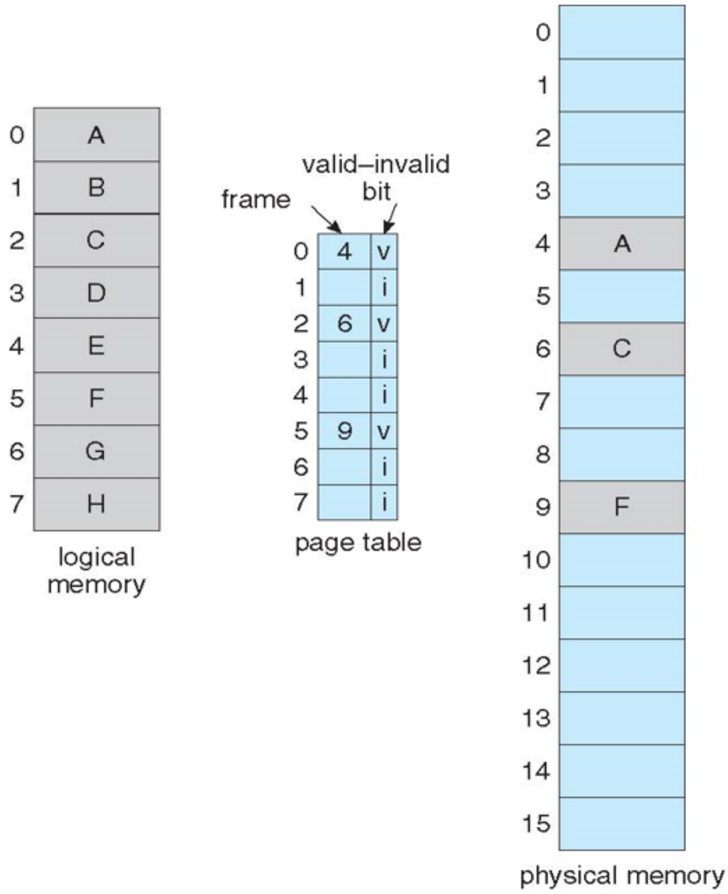
The GPU Virtual Memory we need to design:

- Because the shared memory in GPU with small size and low latency access, we take the shared memory as the traditional CPU physical memory.
- Take the global memory as the disk storage (secondary memory).
- In CUDA, the function executed on GPU that defined by programmer, is called kernel function.
- A kernel function would no longer be constrained by the amount of shared memory that is available. Users would be able to write kernel functions for an extremely large virtual address space, simplifying the programming task.
- Implement a paging system with swapping where the thread access data in shared memory and retrieves data from global memory (secondary memory).
- We only implement the data swap when page fault occurs (not in the instruction level).

Specification of the GPU Virtual Memory we designed:

- Secondary memory (global memory)
- - 128KB (131072 bytes)
- Physical memory (share memory)
- - 48KB (32768 bytes)
    - 32KB for data access
    - 16KB for page table setting
- Memory replacement policy for page fault:
- - If shared memory space is available, place data to the available page, otherwise, replace the **LRU** set. Pick the **least indexed** set to be the victim page in case of tie.
- We have to map virtual address (VA) to physical address (PA).
- The valid bit of each page table block is initialized as false before first data access in shared memory.

- Page size
-
    - 32 bytes
- Page table entries
-
    - 1024 (32KB / 32 bytes)
- 

- 

-

Template structure:

- At first, load the binary file, named `data.bin` to input buffer before kernel launch and return the size of input buffer: `input_size`.
- Launch to GPU kernel with single thread, and dynamically allocate 16KB of share memory, which will be used for variables declared as `extern shared`.

```
  /* 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, 1, INVERT_PAGE_TABLE_SIZE>>>(input_size);
```

- Initialize the virtual memory.

```
__device__ void vm_init(VirtualMemory *vm, uchar *buffer, uchar *storage,
                        u32 *invert_page_table, int *pagefault_num_ptr,
                        int PAGESIZE, int INVERT_PAGE_TABLE_SIZE,
                        int PHYSICAL_MEM_SIZE, int STORAGE_SIZE,
                        int PAGE_ENTRIES) {
  // init variables
  vm->buffer = buffer;
  vm->storage = storage;
  vm->invert_page_table = invert_page_table;
  vm->pagefault_num_ptr = pagefault_num_ptr;

  // init constants
  vm->PAGESIZE = PAGESIZE;
  vm->INVERT_PAGE_TABLE_SIZE = INVERT_PAGE_TABLE_SIZE;
  vm->PHYSICAL_MEM_SIZE = PHYSICAL_MEM_SIZE;
  vm->STORAGE_SIZE = STORAGE_SIZE;
  vm->PAGE_ENTRIES = PAGE_ENTRIES;

  // before first vm_write or vm_read
  init_invert_page_table(vm);
}
```

- Initialize the page table. (Considering the page entries is limited, we use invert page table)

```
__device__ void init_invert_page_table(VirtualMemory *vm) {
  for (int i = 0; i < vm->PAGE_ENTRIES; i++) {
    vm->invert_page_table[i] = 0x80000000; // invalid := MSB is 1
    vm->invert_page_table[i + vm->PAGE_ENTRIES] = i;
  }
}
```

- Under `vm_write`, you should implement the function to write data into vm buffer.
- Under `vm_read`, you should implement the function to read data from vm buffer.
- Under `vm_snapshot`, together with `vm_read`, you should implement the program to load the elements of vm buffer (in shared memory, as physical memory) to results buffer (in global memory).

```
__device__ uchar vm_read(VirtualMemory *vm, u32 addr) {
  /* Complate vm_read function to read single element from data buffer */
  return 123; //TODO
}

__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value) {
  /* Complete vm_write function to write value into data buffer */
}

__device__ void vm_snapshot(VirtualMemory *vm, uchar *results, int offset,
                            int input_size) {
  /* Complete snapshot function togther with vm_read to load elements from
data
   * to result buffer */
}
```

- For user_program (operations on vm_read/vm_write/vm_snapshot), you should strictly follow the name and input parameters as:

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

  We will replace `user_program` for testing, please do not change any symbol of these parameters.

- Count the page fault number when executing paging replacement.

- In Host, dump the contents of binary file into `snapshot.bin`.
- Print out page fault number when the program finishs execution. For the given case, "pagefault number is **8193**" should be last sentence your program prints out.

Functional Requirements (90 points):

- In addition to the test case already given, we will do similar tests to see if the program can print out correct page fault number and correctly dump the contents to `snapshot.bin`. (Pass the given test case will obtain 75% grades, i.e. 0.75x90 = 67.5 points out of 100)
- Moreover, if some test cases fail, we will check if the following modules of your program are correct.
- Implement `vm_write` to write data to vm buffer (shared memory, as physical memory)
- Implement `vm_read` to read data from vm buffer (shared memory, as physical memory)
- Implement `vm_snapshot` together with vm_read to load the elements of vm buffer (in shared memory, as physical memory) to results buffer (in global memory, as secondary storage).
- Implement with LRU paging algorithm for swapping memory.
- *Hashed page table is not required to implement in this project, which is actually designed for address spaces larger than 32 bits. But you can still give a theoretical analysis in report. (0 points)*

Bonus (15 points)

- Background:
  - We used only one page-table in basic task, if we want to launch multiple threads and each thread use the mechanism of paging, we should design a new page table for managing multiple threads.
  - Usually, each thread has an associated page-table, but we don't have enough memory size (shared memory) to setup.
  - To solve this problem, we can use an inverted page table (refers to Chapter 8).
- Requirement:
  - Based on Assignment 3, launch 4 threads in kernel function, all threads concurrently execute it. (2 point)

- To avoid the race condition, threads execute `vm_read()` / `vm_write()` should be a **non-preemptive priority** scheduling. Maintain the scheduling when operating on `vm_read()` / `vm_write()` / `vm_snapshot()`. (5 points)
- Modify your paging mechanism to manage multiple threads. (5 points)
- Print the times of page fault of whole system before the program end. (2 point)
- Correctly dump the contents to `snapshot.bin`. (1 point)

Report (10 points)

Write a report for your assignment, which should include main information as below:

- Environment of running your program. (E.g., OS, VS version, CUDA version, GPU information etc.)
- Execution steps of running your program.
- How did you design your program?
- What's the page fault number of your output? Explain how does it come out.
- What problems you met in this assignment and what are your solution?
- Screenshot of your program output.
- What did you learn from this assignment?

Grading rules

Here is a sample grading scheme. Different from the points specified above, this is the general guide when TA's grading.

Completion	**Marks**
Bonus	10 points
Report	10 points
Pass all the test cases	90
Pass the given test case	67.5 +
Given case failed	60 ~ 67.5
Partial submitted	0 ~ 60
No submission	0
Late submission	**10% off within 6h, 15% off within 12h, 25% off within 24h**
Plagiarism	**0 (We will check the duplication rate of your codes along with those of previous years' students.)**