CSC3150 Assignment 4

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, 30 Nov 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_4_<student_id>$ tree
.
├── bonus
|   ├── data.bin
|   ├── file_system.cu
|   ├── file_system.h
|   ├── main.cu
|   ├── slurm.sh
|   └── user_program.cu
├── report.pdf
└── source
    ├── data.bin
    ├── file_system.cu
    ├── file_system.h
    ├── main.cu
    ├── Makefile
    └── user_program.cu

2 directories, 13 files
```

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_4_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_4_<student_id>.zip
Assignment_4_<student_id>
main@ubuntu:~/Desktop$ ls
Assignment_4_<student_id>                    Assignment_4_<student_id>.zip
```

Task Description

In Assignment 4, you are required to implement a mechanism of file system management via GPU's memory.

Background:

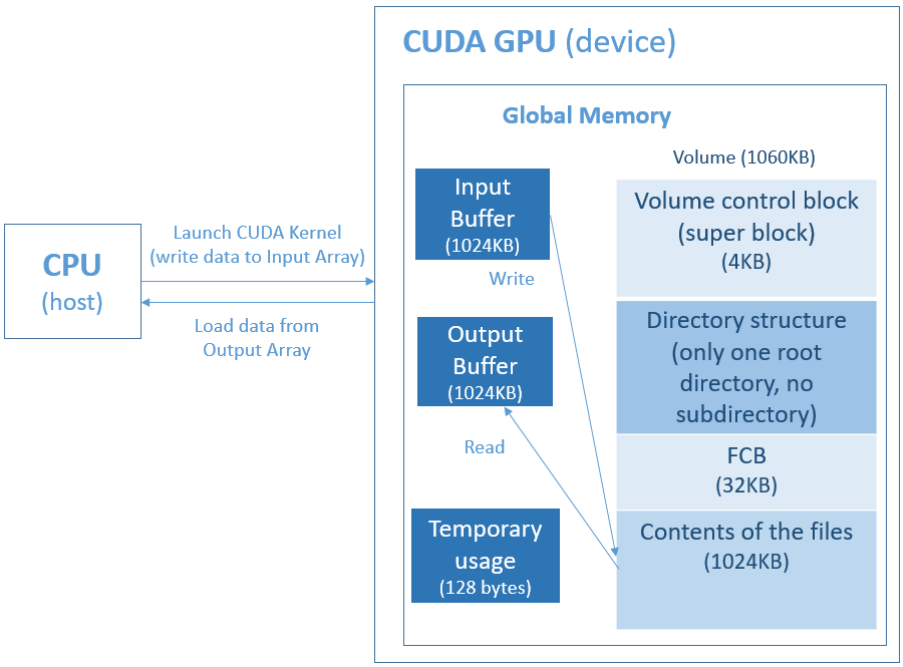
- **File systems** provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily.
- A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file with its attributes, the operations allowed on a file, and the directory structure for organizing files.
- The second problem is creating algorithms and data structures to map the logical file system on to the physical secondary-storage devices.
- The file-organization module knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block address to physical block address for the basic file system to transfer.
- Each file's logical blocks are numbered from 0 (or 1) through N. Since the physical blocks containing the data usually do not match the logical numbers, a translation is required to locate each block.
- The **logical file system** manages **metadata** information.
- **Metadata** includes all of the file-system structure except the actual data (or contents of the files).
- The **file-organization** module also includes the **free-space manager**, which tracks unallocated blocks and provides these blocks to the file-organization module when

requested.

- The **logical file system** manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. It maintains file structure via file-control blocks.
- A **file-control block** (**FCB**) (an inode in UNIX file systems) **contains information about the file**, including ownership, permissions, and location of the file contents.
- Because there have no OS in *GPU* to maintain the mechanism of the **logical file system**, we can try to implement a simple file system in CUDA GPU with single thread, and limit global memory as volume.

The GPU File System we need to design:

- We take the **global memory as a volume** (logical drive) from a hard disk.
- No directory structure stored in volume, **only one root directory**, no subdirectory in this file system.
- **A set of file operations should be implemented**.
- In this project, **we use only one of GPU memory, the global memory as a volume**. **We don't create the shared memory as physical memory for any data structures stored in**, **like system-wide open file table in memory**.
- In this simple file system, we just directly take the information from a volume (in global memory) by single thread.



Specification:

- The size of volume is 1085440 bytes (1060KB).
- The size of files in total is 1048576 bytes (1024KB).
- The maximum number of file is 1024.
- The maximum size of a file is 1024 bytes (1KB).
- The maximum size of a file name is 20 bytes.
- File name end with “\0”.
- FCB size is 32 bytes.
- FCB entries is 32KB/ 32 bytes = 1024.
- Storage block size is 32 bytes.
- `fs_open`:
  - Open a file
  - Give a file pointer to find the file’s location.
  - Space in the file system must be found for the file.
  - An entry for the new file must be made in the directory.
  - Also accept access-mode information: read/write
  - When to use write mode, if no such file name can be found, create a new zero byte file.
  - Return a write/read pointer.
  - Function definition:

    ```
    // s: File name, op: G_READ or G_WRITE
    __device__ u32 fs_open(FileSystem *fs, char *s, int op)
    ```

  - Demo usage:

```
static char file_0[] = "t.txt\0";
static char file_1[] = "b.txt\0";

// open a file in write mode
u32 fp = fs_open(fs, file_0, G_WRITE);
// open a file in read mode
fp = fs_open(fs, file_1, G_READ);
```

- `fs_write`:
  - To write a file.
  - There is a write pointer to identify the location in the file.
  - If the file has existed, cleanup the older contents of the file and write the new contents.
  - Take the **input** buffer to write bytes data to the file.
  - Function definition:

```
// input: the input buffer
// size: the size of the data(byte) to be written to the file
// fp: the write pointer
__device__ u32 fs_write(FileSystem *fs, uchar* input, u32 size, u32 fp)
```

  - Demo usage:

```
static char file_0[] = "t.txt\0";

u32 fp = fs_open(fs, file_0, G_WRITE);
// starting from input[0], write 64 bytes of data into t.txt
fs_write(fs, input, 64, fp);

// starting from input[32], write 64 bytes of data into t.txt
fs_write(fs, input+32, 64, fp);
```

- `fs_read`:
  - To read contents from a file.
  - There is a read pointer to identify the location in the file.
  - To read bytes data from the file to the **output** buffer.

- The offset of the opened file associated with the read pointer is 0 (always read the file from head).
- Function definition:

```
// output: the output buffer
// size: the size of the data(byte) to be written to the file
// fp: read pointer
__device__ void fs_read(FileSystem *fs, uchar *output, u32 size, u32
fp)
```

- Demo usage:

```
fp = fs_open(fs, file_0, G_READ);

// from the begining of the file, read 64 bytes of data into t.txt
fs_read(fs, output, 64, fp);
```

- `fs_gsys` (RM):
  - To delete a file and release the file space.
  - Search the directory for the named file.
  - Implement `gsys()` to pass the `RM` command.
  - Function definition:

```
// op: command (RM is denotes the DELETE Command)
// s: file name
__device__ void fs_gsys(FileSystem *fs, int op, char *s)
```

  - Demo usage:

```
fs_gsys(fs, RM, file_0);
```

- **fs_gsys (LS_D / LS_S):**
  - List information about files.
  - Implement `gsys()` to pass the `LS_D` / `LS_S` commands.

- `LS_D` list all files name in the directory and order by modified time of files.
- `LS_S` list all files name and size in the directory and order by size.
- If there are several files with the same size, then first create first print.
- Demo usage:

```
fs_gsys(fs, LS_S);
fs_gsys(fs, LS_D);
```

Demo output

```
===sort by modified time===
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
```

Template structure:

- The storage size of the file system is already pre-defined as:

```
#define SUPERBLOCK_SIZE 4096 //32K/8 bits = 4 K
#define FCB_SIZE 32 //32 bytes per FCB
#define FCB_ENTRIES 1024
#define VOLUME_SIZE 1085440 //4096+32768+1048576
#define STORAGE_BLOCK_SIZE 32

#define MAX_FILENAME_SIZE 20
#define MAX_FILE_NUM 1024
#define MAX_FILE_SIZE 1048576

#define FILE_BASE_ADDRESS 36864 //4096+32768

// data input and output
__device__ __managed__ uchar input[MAX_FILE_SIZE];
__device__ __managed__ uchar output[MAX_FILE_SIZE];

// volume (disk storage)
__device__ __managed__ uchar volume[VOLUME_SIZE];
```

- At first, load the binary file, named `data.bin` to **input** buffer (via `load_binarary_file()`) before kernel launch.
- Launch to GPU kernel with single thread.

```
// Launch mykernle function on GPU with single thread
mykernel<<<1, 1>>>(input, output);
```

- In kernel function, initialize the file system we constructed.

```
// Initilize the file system
FileSystem fs;
fs_init(&fs, volume, SUPERBLOCK_SIZE, FCB_SIZE, FCB_ENTRIES,
        VOLUME_SIZE,STORAGE_BLOCK_SIZE, MAX_FILENAME_SIZE,
        MAX_FILE_NUM, MAX_FILE_SIZE, FILE_BASE_ADDRESS);
```

```
__device__ void fs_init(FileSystem *fs, uchar *volume, int SUPERBLOCK_SIZE,
                        int FCB_SIZE, int FCB_ENTRIES, int VOLUME_SIZE,
                        int STORAGE_BLOCK_SIZE, int MAX_FILENAME_SIZE,
                        int MAX_FILE_NUM, int MAX_FILE_SIZE, int
FILE_BASE_ADDRESS)
{
  // init variables
  fs->volume = volume;

  // init constants
  fs->SUPERBLOCK_SIZE = SUPERBLOCK_SIZE;
  fs->FCB_SIZE = FCB_SIZE;
  fs->FCB_ENTRIES = FCB_ENTRIES;
  fs->STORAGE_SIZE = VOLUME_SIZE;
  fs->STORAGE_BLOCK_SIZE = STORAGE_BLOCK_SIZE;
  fs->MAX_FILENAME_SIZE = MAX_FILENAME_SIZE;
  fs->MAX_FILE_NUM = MAX_FILE_NUM;
  fs->MAX_FILE_SIZE = MAX_FILE_SIZE;
  fs->FILE_BASE_ADDRESS = FILE_BASE_ADDRESS;

}
```

- In kernel function, invoke user_program to simulate file operations for testing. **We will replace the user program with different test cases.**

```
// user program the access pattern for testing file operations
user_program(&fs, input, output);
```

- You should complete the file operations for `fs_open` / `fs_write` / `fs_read` / `fs_gsys(rm)` / `fs_gsys(ls_d)` / `fs_gsys(ls_s)`.

```
__device__ u32 fs_open(FileSystem *fs, char *s, int op)
{
    /* Implement open operation here */
}

__device__ void fs_read(FileSystem *fs, uchar *output, u32 size, u32 fp)
{
    /* Implement read operation here */
}

__device__ u32 fs_write(FileSystem *fs, uchar* input, u32 size, u32 fp)
{
    /* Implement write operation here */
}

__device__ void fs_gsys(FileSystem *fs, int op)
{
    /* Implement LS_D and LS_S operation here */
}

__device__ void fs_gsys(FileSystem *fs, int op, char *s)
{
    /* Implement rm operation here */
}
```

- In CPU(host) main function, the output buffer is copied to device, and it is written into “snapshot.bin” (via `write_binarary_file()`).

Functional Requirements (90 points):

- **Implement file volume structure. (10 points)**
- **Implement free space management. (For example, Bit-Vector / Bit-Map). (10 points)**
- **Implement contiguous allocation. (10 points)**
- **Implement `fs_open` operation (10 points)**

- **Implement `fs_write` operation (10 points)**
- **Implement `fs_read` operation (10 points)**
- **Implement `fs_gsys(RM)` operation (10 points)**
- **Implement `fs_gsys(LS_D)` operation (10 points)**
- **Implement `fs_gsys(LS_S)` operation (10 points)**

Demo Output

There are four test cases In the “user_program.cu”.

- Test Case 1

```
===sort by modified time===
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
===sort by file size===
t.txt 32
b.txt 12
===sort by modified time===
b.txt
t.txt
===sort by file size===
b.txt 12
```

- Test Case 2

- ```
  ===sort by modified time===
  t.txt
  b.txt
  ===sort by file size===
  t.txt 32
  b.txt 32
  ===sort by file size===
  t.txt 32
  b.txt 12
  ===sort by modified time===
  b.txt
  t.txt
  ===sort by file size===
  b.txt 12
  ===sort by file size===
  *ABCDEFGHIJKLMNOPQR 33
  )ABCDEFGHIJKLMNOPQR 32
  (ABCDEFGHIJKLMNOPQR 31
  'ABCDEFGHIJKLMNOPQR 30
  &ABCDEFGHIJKLMNOPQR 29
  %ABCDEFGHIJKLMNOPQR 28
  $ABCDEFGHIJKLMNOPQR 27
  #ABCDEFGHIJKLMNOPQR 26
  "ABCDEFGHIJKLMNOPQR 25
  !ABCDEFGHIJKLMNOPQR 24
  b.txt 12
  ===sort by modified time===
  *ABCDEFGHIJKLMNOPQR
  )ABCDEFGHIJKLMNOPQR
  (ABCDEFGHIJKLMNOPQR
  'ABCDEFGHIJKLMNOPQR
  &ABCDEFGHIJKLMNOPQR
  b.txt
  ```

- Test Case 3
  ```

```
===sort by modified time===
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
===sort by file size===
t.txt 32
b.txt 12
===sort by modified time===
b.txt
t.txt
===sort by file size===
b.txt 12
===sort by file size===
*ABCDEFGHIJKLMNOPQR 33
)ABCDEFGHIJKLMNOPQR 32
(ABCDEFGHIJKLMNOPQR 31
'ABCDEFGHIJKLMNOPQR 30
&ABCDEFGHIJKLMNOPQR 29
%ABCDEFGHIJKLMNOPQR 28
$ABCDEFGHIJKLMNOPQR 27
#ABCDEFGHIJKLMNOPQR 26
"ABCDEFGHIJKLMNOPQR 25
!ABCDEFGHIJKLMNOPQR 24
b.txt 12
===sort by modified time===
*ABCDEFGHIJKLMNOPQR
)ABCDEFGHIJKLMNOPQR
(ABCDEFGHIJKLMNOPQR
'ABCDEFGHIJKLMNOPQR
&ABCDEFGHIJKLMNOPQR
b.txt
===sort by file size===
~ABCDEFGHIJKLM 1024
}ABCDEFGHIJKLM 1023
......
......
......
=A 35
<A 34
*ABCDEFGHIJKLMNOPQR 33
;A 33
)ABCDEFGHIJKLMNOPQR 32
:A 32
(ABCDEFGHIJKLMNOPQR 31
9A 31
```

```
'ABCDEFGHIJKLMNOPQR 30
8A 30
&ABCDEFGHIJKLMNOPQR 29
7A 29
6A 28
5A 27
4A 26
3A 25
2A 24
b.txt 12
```

- Test Case 4

```
triggering gc
===sort by modified time===
1024-block-1023
1024-block-1022
1024-block-1021
1024-block-1020
1024-block-1019
1024-block-1018
1024-block-1017
1024-block-1016
1024-block-1015
1024-block-1014
1024-block-1013
1024-block-1012
...
1024-block-0008
1024-block-0007
1024-block-0006
1024-block-0005
1024-block-0004
1024-block-0003
1024-block-0002
1024-block-0001
1024-block-0000
```

Bonus (15 points)

- In basic task, there is only one root directory for the file system. In bonus, you must implement **tree-structured directories**. (3 points)
- A directory (or subdirectory) contains a set of files or subdirectories.

- A directory is simply another file.
- There are at most **50 files** (include subdirectories) in a directory.
- The size of a directory is the **sum of character bytes of all files name** (include subdirectories).

E.g., the directory root/ have these files:

“A.txt\0” “b.txt\0” “c.txt\0” “app\0”

The size of directory root/ is 22 bytes.

- The maximum number of files (include directory) is **1024.**
- The maximum depth of the tree-structured directory is **3**.
- File operations: (12 points)
- - `fs_gsys(fs, MKDIR, “app\0”);`

    Create a directory named ‘app’.
- - `fs_gsys(fs, CD, “app\0”);`

    Enter app directory (only move to its subdirectory).
- - `fs_gsys(fs, CD_P);`

    Move up to parent directory.
- - `fs_gsys(fs, RM_RF, “app\0”);`

    Remove the app directory and all its subdirectories and files recursively. You cannot delete a directory by fs_gsys(fs, RM, “app\0”), cannot remove `app' if it is a directory.
- - `fs, gsys(fs, PWD);`

    Print the path name of current, eg., “/app/soft”
- - `fs_gsys(fs, LS_D / LS_S);`

    Update this file list operation, to list the files as well as directories. For a file, list it name (with size) only. For a directory, add an symbol ‘d’ at the end.
- Demo test case:

- ```
  ////////////////////// Bonus Test Case ///////////////
  u32 fp = fs_open(fs, "t.txt\0", G_WRITE);
  fs_write(fs, input, 64, fp);
  fp = fs_open(fs, "b.txt\0", G_WRITE);
  fs_write(fs, input + 32, 32, fp);
  fp = fs_open(fs, "t.txt\0", G_WRITE);
  fs_write(fs, input + 32, 32, fp);
  fp = fs_open(fs, "t.txt\0", G_READ);
  fs_read(fs, output, 32, fp);
  fs_gsys(fs, LS_D);
  fs_gsys(fs, LS_S);
  fs_gsys(fs, MKDIR, "app\0");
  fs_gsys(fs, LS_D);
  fs_gsys(fs, LS_S);
  fs_gsys(fs, CD, "app\0");
  fs_gsys(fs, LS_S);
  fp = fs_open(fs, "a.txt\0", G_WRITE);
  fs_write(fs, input + 128, 64, fp);
  fp = fs_open(fs, "b.txt\0", G_WRITE);
  fs_write(fs, input + 256, 32, fp);
  fs_gsys(fs, MKDIR, "soft\0");
  fs_gsys(fs, LS_S);
  fs_gsys(fs, LS_D);
  fs_gsys(fs, CD, "soft\0");
  fs_gsys(fs, PWD);
  fp = fs_open(fs, "A.txt\0", G_WRITE);
  fs_write(fs, input + 256, 64, fp);
  fp = fs_open(fs, "B.txt\0", G_WRITE);
  fs_write(fs, input + 256, 1024, fp);
  fp = fs_open(fs, "C.txt\0", G_WRITE);
  fs_write(fs, input + 256, 1024, fp);
  fp = fs_open(fs, "D.txt\0", G_WRITE);
  fs_write(fs, input + 256, 1024, fp);
  fs_gsys(fs, LS_S);
  fs_gsys(fs, CD_P);
  fs_gsys(fs, LS_S);
  fs_gsys(fs, PWD);
  fs_gsys(fs, CD_P);
  fs_gsys(fs, LS_S);
  fs_gsys(fs, CD, "app\0");
  fs_gsys(fs, RM_RF, "soft\0");
  fs_gsys(fs, LS_S);
  fs_gsys(fs, CD_P);
  fs_gsys(fs, LS_S);
  ```
- Demo output:
  ```

```
Sorted 2 files by modification time:
t.txt    4
b.txt    3
Sorted 2 files by size:
t.txt    32
b.txt    32
Sorted 3 files by modification time:
app    5    d
t.txt    4
b.txt    3
Sorted 3 files by size:
t.txt    32
b.txt    32
app    0    d
......
```

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	15 points
Report	10 points
Pass the additional grading cases	90
Pass test case 4	85+
Pass test case 3	79+
Pass test case 2	73+
Pass test case 1	67+
Fully Submitted (compile successfully)	60 +
Partial submitted	0 ~ 60
No submission	0
Late submission	**Not allowed**