CSC3150 Assignment 4 Report

Chen Zhixin, 120090222

Running Environment

All my program run on CSC4005's HPC Cluster

Kernel Version

```
[120090222@node21 Assignment4]$ uname -r
3.10.0-862.el7.x86_64
```

OS version

```
Centos 7.x
```

CUDA version

```
[120090222@node21 ~]$ nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Jun__8_16:49:14_PDT_2022
Cuda compilation tools, release 11.7, V11.7.99
Build cuda_11.7.r11.7/compiler.31442593_0
```

GPU/CPU Information (From intro session of CSC4005)

```
• Configuration
1 login node
20 Intel CPU cores (40 logic cores)
100GB RAM
1 Nvidia Quadro RTX 4000 GPU
• 30+ compute node
Each compute node has:
20 Intel CPU cores (40 logic cores)
100GB RAM
1 Nvidia Quadro RTX 4000 GPU
```

Execution steps of running my program

File structure:

```
.
├── 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
```

source

Assert you run on CSC4005 HPC cluster.

```
cd ./source/
sh slurm.sh
```

bonus

Assert you run on CSC4005 HPC cluster.

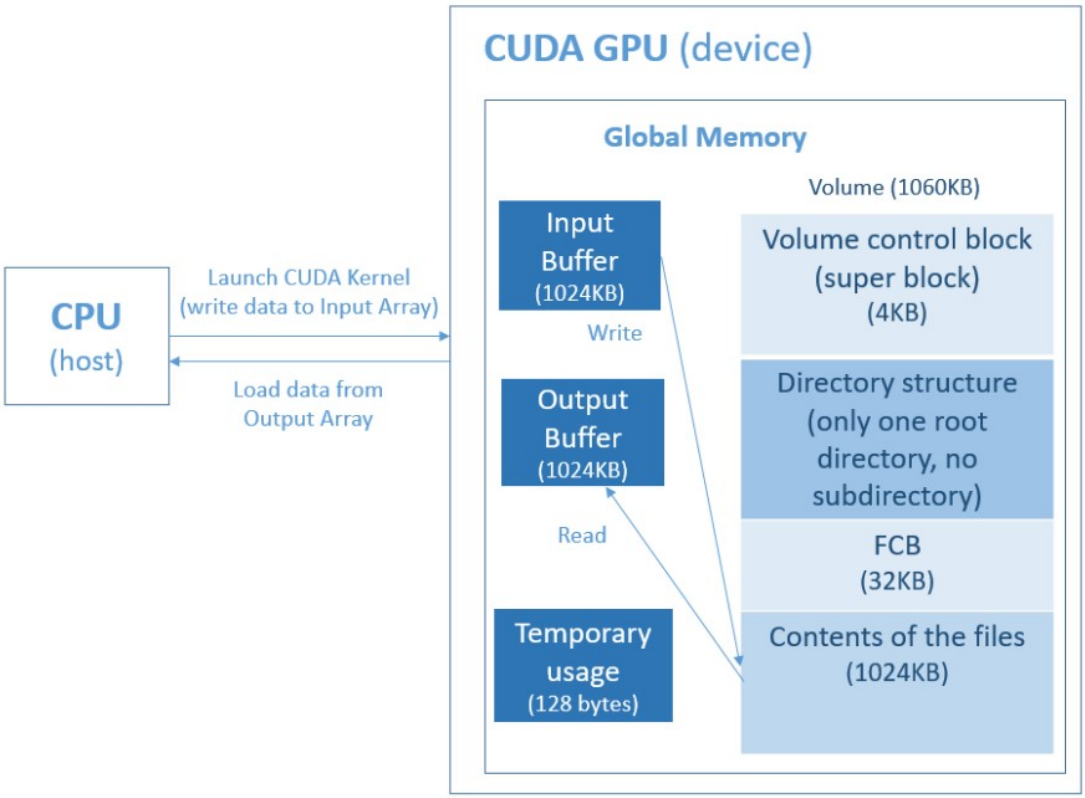
```
cd ./bonus/
sh slurm.sh
```

Design Idea

Main

In this project, we are going to implement a mechanism of file system management via GPU's memory. Since 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.

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. Each portion of the file system and their limitation in space can be found in the following figure:



Volume partition

We have a 1060KB volume, which contains 3 partitions: Volume control block(VCB, superblock, 4KB), FCB(32KB) and the contents of the files(storage, 1024 KB).

- VCB can indicate the available blocks on the storage.
- FCB contains each file's metadata, including but not limited to: Name, Starting block number, Size, Last modified time, Permission, Create time etc.
- Storage contains the content of each file. The **dynamically contiguous allocation** is used.

Statistics

- Data storage: 1024KB (2^20) = 32 bytes (block size) * 2^15 (32768 entries)
- Superblock (VCB): 4KB (4096) = 2^15 bits <--> 2^15 entries
- FCB: 1024 entries, 32 bytes / entry.
- FCB Entry (32 bytes):
    - 20 bytes Name
    - 2 bytes Starting block number (unit: block)
    - 2 bytes size(unit: bytes)
    - 1 byte Permission(2 bit) + Valid(1 bit) + Empty(5 bit)
    - 1 byte Empty
    - 3 bytes Last modified time(up to 2^24 ranking)
    - 3 bytes Create time(up to 2^24 ranking)

Design Philosophy & Highlights

- **Defensive programming** - Reduce unforeseen use of procedures.
- **Focus on designing function interfaces: Provide underlying capabilities + facilitate code reuse** - easy for maintenance, high readability.
- **Unit testing** - When debugging & testing, make sure every function interfaces work as expected to minimize bugs.
- **Time in exchange of space** - Strictly obey restrictions of up to 128 bytes temporary storage, low-level implementation of sorting algorithm with O(1) space complexity ("造轮子" in Chinese).

API Implementations (from low-level to high-level)

Function	Input	Output	Descriptions
fs_init	*Args*	*N/A*	*File system initialization*
memcpy	*uchar pointer (pointing to the source and target)*	*N/A*	*Copy content from source to target*
memcmp	*uchar pointer (pointing to the source and target)*	*1 -> different; 0-> identical*	*Compare the source and target with equal size (used in comparing file names)*
FCB_read_validbit	*FCB_address*	*Valid = 1, Invalid = 0.*	*Read the # FCB's valid bit.*
FCB_set_validbit	*Valid = 1, Invalid = 0, FCB_address*	*N/A*	*Set the FCB permission bit.
FCB_read_filename	*output pointer, FCB_address*	*N/A*	*Read the FCB filename*

Function	Input	Output	Descriptions
FCB_set_filename	*input pointer, FCB_address*	*N/A*	*Set the FCB filename*
FCB_read_start	*FCB_address*	*start block number*	*Read the FCB starting point address (Unit: block)*
FCB_set_start	*start block number, FCB_address*	*N/A*	*Set the FCB starting point address (Unit: block)*
FCB_read_size	*FCB_address*	*size*	*Read the FCB size (Unit: bytes)*
FCB_set_size	*size, FCB_address*	*N/A*	*Set the FCB size (Unit: bytes)*
FCB_read_ltime	*FCB_address*	*Last modified time*	*Read the FCB Last modified time*
FCB_set_ltime	*FCB_address, Last modified time*	*N/A*	*Set the FCB Last modified time*
FCB_read_ctime	*FCB_address*	*Create time*	*Read the FCB created time*
FCB_set_ctime	*FCB_address*	*N/A*	*Set the FCB created time*
print_FCB	*N/A*	*N/A*	*For testing only, printing out all FCB info.*
print_VCB	*N/A*	*N/A*	*For testing only, printing out all VCB info.*
VCB_Query	*blocks number*	*blocks number*	*Check if there is continuous n free blocks. (unit: 32-bytes-large block)*
cover	*uchar # of VCB block*	*N/A*	*bit-operation on masking, supporting VCB_modification ONLY*
VCB_modification	*start (Unit:block), size (Unit: block)*	*N/A*	*Set the [start, start+size] in VCB to be 0/1.*
memory_compaction	*N/A*	*N/A*	*Memory Compaction (a very time-comsuming job)*
fs_open	*filename*, *op*	*file descriptor*	*Implement read operation here*
fs_read	*file descriptor*, *size, output pointer*	*N/A*	*Implement read operation here*

Function	Input	Output	Descriptions
fs_write	*file descriptor, size, input pointer*	*N/A*	*Implement write operation here*
fs_gsys	*op*	*N/A*	*Implement LS_D and LS_S operation here*
fs_gsys	*op, filename*	*N/A*	*Implement rm operation here*

Notes:

- `fs_open`: Find whether the file already exists. If no, create a 0KB new file and initiate the metadata. Finally return the file descriptor. Clean the content before writing.
- `fs_read`: Use the file descriptor to read the file with size n.
- `fs_write`: Use the file descriptor to write the file with size n. If no continuous space for storage, do memory compaction.
- `fs_gsys(LS_D)`: Display all the file, with modified time from oldest to latest.
- `fs_gsys(LS_S)`: Display all the file, with size from largest to smallest. If multiple files has the same size, then rank with modified time from oldest to latest.
- `fs_gsys(RM)`: Remove the file, also update info in VCB and FCB.

Bonus

Based on source version, here we need to implement tree-structured directories, i.e., the directory (or subdirectory) can contains a set of files or subdirectories. And we need to support these functions:

- `fs_gsys(fs, MKDIR, “app\0”);`
- `fs_gsys(fs, CD, “app\0”);`
- `fs_gsys(fs, CD_P);`
- `fs_gsys(fs, RM_RF, “app\0”);`
- `fs, gsys(fs, PWD);`
- `fs_gsys(fs, LS_D / LS_S);`

Statistics

The corresponding FCB entry statistics change to:

- FCB Entry (32 bytes):
    - 20 bytes Name
    - 2 bytes Starting block number(unit: block)
    - 3 bytes size(unit: byte)
    - 1 byte Valid(1 bit) + Is_folder(1 bit) + Number of sub-files(6 bits)
    - 3 bytes Last modified time(up to 2^24 ranking)
    - 2 bytes create time(up to 2^16 ranking)
    - 1 bytes parent address(unit: FCB block)

API Implementations

Based on source version, new changes in implementations:

Function	Input	Output	Description
...	...	...	...
FCB_read_folder	*FCB_address*	Is_folder -> 1, Non_folder -> 0	Check if # FCB block is the folder's metadata
FCB_set_folder	*FCB_address, Is_folder -> 1 / Non_folder -> 0*	*N/A*	Set # FCB block to be folder/non-folder
FCB_read_childnum	*FCB_address*	Number of the children	Read the number of the children (folder only)
FCB_set_childnum	*FCB_address, number of children*	*N/A*	Set the number of the children (folder only)
rm	FCB_address	*N/A*	remove the folder/file recursively
FCB_read_parent	*FCB_address*	FCB address of the parent	Read the parent's FCB address of the current file
FCB_set_parent	FCB address of the parent	*N/A*	Set the parent's FCB address of the current file
...	...	...	...

Notes:

- Some of the implementations are integrated in the old function interfaces. For example:
  - `MKDIR`: To create a folder is similar to creating a file, but we need to make tags on its metadata, telling that "it is a folder".
  - `CD` and `CD_P`: The position of the current folder and the parent folder are preserved in the Filesystem state. If we need to change the position of the current path, just simply change the state in Filesystem.
  - `RM` and `RM_RF`: The "remove" and "remove folder" command is similar, so we do not distinguish them strictly, since all of them are treated as "file block", with the info stored in FCB entries. Here we use recursive removal method to implement these two commands. *To be notice that, only the file under the current folder can be removed.*
  - `PWD`: Record the current folder, and find parents along the tree structure, until it reached the root folder.
  - `LS_D` and `LS_S`: The sorting logic is similar to the source version, but at this time, only the file under the current folder are eligible as candidate.

Problems I met doing this project

It takes me approximately a week to finish source & bonus. There are many problems I met when doing this project:

- How to break down big problems into small ones - In my implementations, I used many function interfaces to implement some basic and low-level operations, like to read or write the attributes in FCB entries. It will take some effort to construct the overall structure, but after that, everything seems worthwhile.
- Pointer management in C++ - In our project, sometimes we need to do modification directly on the volume. Sometimes we need to fetch and write 4-bytes long word, sometimes it's 2-bytes long. So it's a difficult task to manage all the pointers, especially the number of pointers are large.
- Data type and unexpected behavior: unsigned vs. signed, type conversion - for example, we need some rounding in calculation, but there comes a bug the result are strange. After debugging, I found that the problem comes from that, we assign a -1 to an unsigned integer. So there comes unexpected behavior.
- Robustness - Sometimes, there may be some corner cases. For example, we need to implement memory compaction in case of external fragmentation. Also, we need to do similar compaction in case like the create time / last modified time reached the upper bound of the capability.
- Space limitations - there are merely 128 bytes extra space for temporary storage. That means sometimes we need some extra time in exchange of space. (Like sorting, finding the min / max value among something, we need to compare among each other)

Screenshot of the output

Source

Testcase 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===
```

Testcase 4

```
1024-block-0026
1024-block-0025
1024-block-0024
1024-block-0023
1024-block-0022
1024-block-0021
1024-block-0020
1024-block-0019
1024-block-0018
1024-block-0017
1024-block-0016
1024-block-0015
1024-block-0014
1024-block-0013
1024-block-0012
1024-block-0011
1024-block-0010
1024-block-0009
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

```
===sort by modified time===
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
===sort by modified time===
app d
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
app 0 d
===sort by file size===
===sort by file size===
a.txt 64
b.txt 32
soft 0 d
===sort by modified time===
soft d
b.txt
a.txt
/app/soft
===sort by file size===
B.txt 1024
C.txt 1024
D.txt 1024
A.txt 64
===sort by file size===
a.txt 64
b.txt 32
soft 24 d
/app
```

What I have learned in this project?

- Patience. - *Patience is the most important thing.*
- The ability to break down big problems into small ones.
- File system management: concept and simple implementation.
- Unit testing.
- Writing comment and docs for better understanding and readability.