Report–CSC3150AS3

Author: Shao Jiaqi (119010256)

Introduction

- Environment

  I use the environment provided by CSC4005.

```
Operating System: CentOS Linux 7 (Core)
```

```
$ g++ --version
g++ (GCC) 4.8.5 20150623 (Red Hat 4.8.5-28)
Copyright © 2015 Free Software Foundation, Inc.

$ cmake --version
cmake version 3.21.2
```

  CUDA & GPU Device

```
Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce RTX 2080 Ti"
CUDA Driver Version / Runtime Version          11.4 / 11.4
CUDA Capability Major/Minor version number:    7.5
Total amount of global memory:                 11019 MBytes
(11554717696 bytes)
(068) Multiprocessors, (064) CUDA Cores/MP:    4352 CUDA Cores
GPU Max Clock rate:                            1620 MHz (1.62 GHz)
Memory Clock rate:                             7000 Mhz
Memory Bus Width:                              352-bit
L2 Cache Size:                                 5767168 bytes
Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=
(131072, 65536), 3D=(16384, 16384, 16384)
Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048
layers
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       49152 bytes
Total shared memory per multiprocessor:        65536 bytes
Total number of registers available per block: 65536
Warp size:                                     32
Maximum number of threads per multiprocessor:  1024
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
```

```
Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
65535)
Maximum memory pitch:                          2147483647 bytes
Texture alignment:                             512 bytes
Concurrent copy and kernel execution:          Yes with 3 copy
engine(s)
Run time limit on kernels:                     No
Integrated GPU sharing Host Memory:            No
Support host page-locked memory mapping:       Yes
Alignment requirement for Surfaces:            Yes
Device has ECC support:                        Disabled
Device supports Unified Addressing (UVA):      Yes
Device supports Managed Memory:                Yes
Device supports Compute Preemption:            Yes
Supports Cooperative Kernel Launch:            Yes
Supports MultiDevice Co-op Kernel Launch:      Yes
Device PCI Domain ID / Bus ID / location ID:   0 / 175 / 0
Compute Mode:
   < Default (multiple host threads can use ::cudaSetDevice() with
device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4, CUDA
Runtime Version = 11.4, NumDevs = 1
Result = PASS
```

- Background
  - Virtual Memory
    The virtual memory is efficient to execute partially-loaded program, so that the program is free of physical memory limited space. The virtual address is generated by the CPU, and after loading, the physical address is bound. The virtual memory space can be divided into pages, and the physical memory space can be divided into frames of the same size. If there is a reference to a page, OS needs to check the page table to find out its physical page number or it is not in memory, where the page fault may occur. The virtual memory management is significant in mapping virtual memory to physical memory, and load data from disk into physical memory only when necessary.
    In this project, we aim to simulate a virtual memory management in GPU, and the Fig.1.1 shows how the memory arrangement in the GPU. Moreover, I used Inverted Page Table(IPT) in this project, and LRU is implemented for swapping out when page faults.

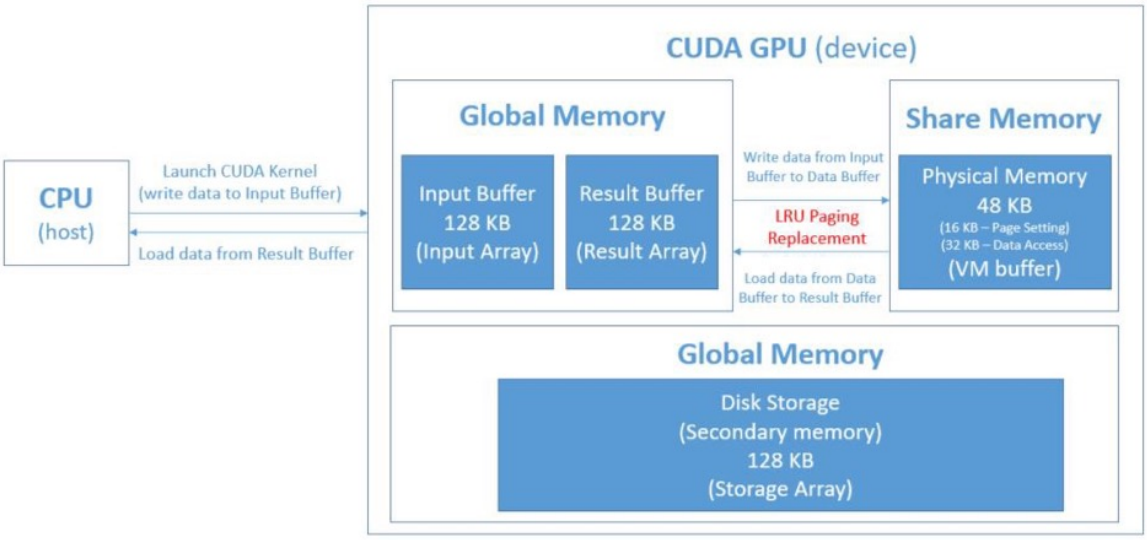

Fig.1.1 Memory Arrangement

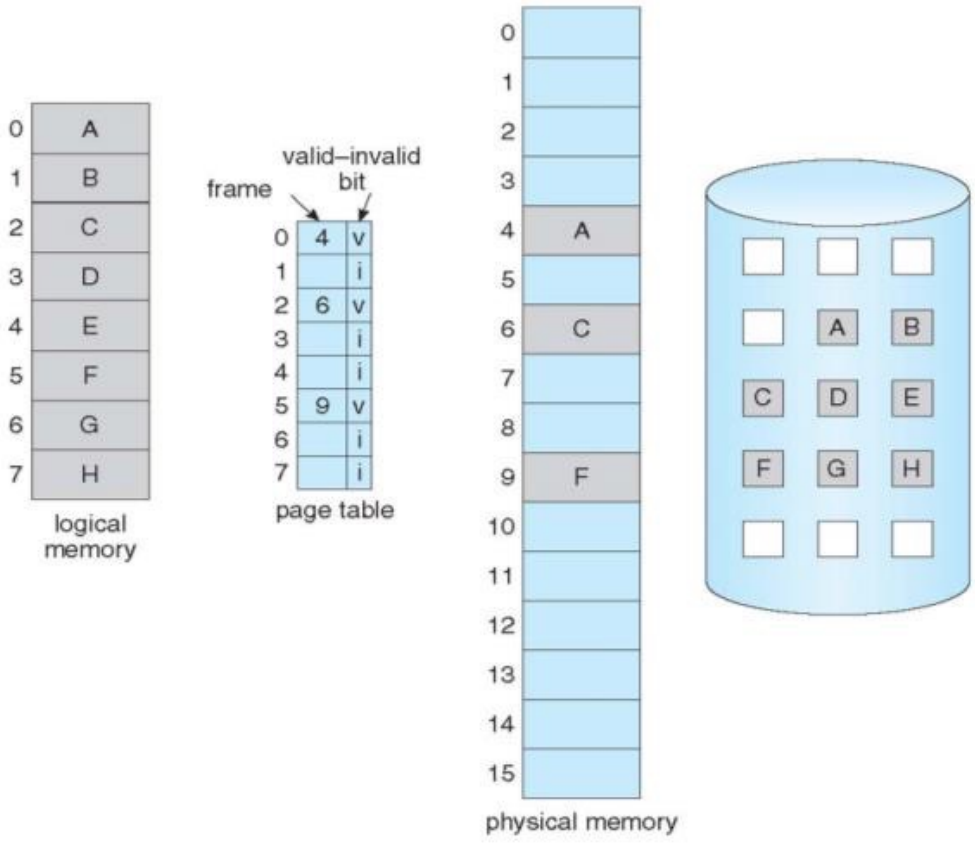

Fig.1.2 Overview of Virtual Memory Management

- Inverted Page Table (IPT)
  IPT is a fixed-size table with the number of entries equal to the number of frames in physical memory. In order to translate a virtual address, the virtual page number and current process ID are compared against each entry, traversing the array sequentially. When a match is found, the index of the match replaces the virtual page number in the address to obtain a physical address. If no match is found, a page fault occurs. In this project, process ID is exclusive since the program is single process.

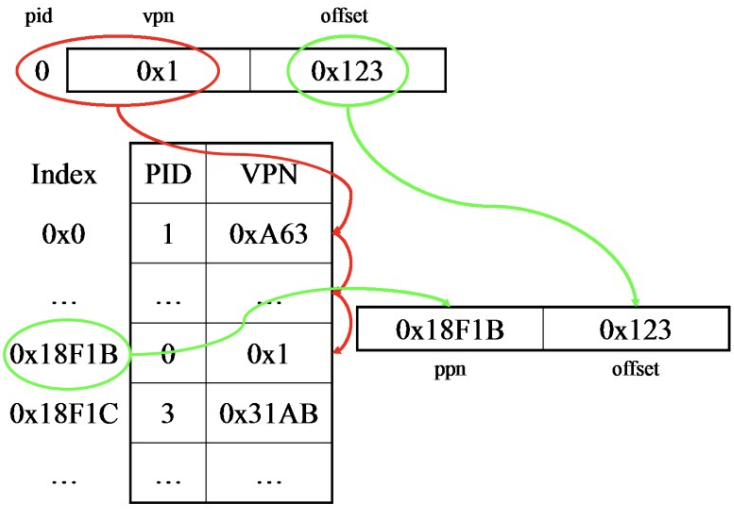

Fig.1.3 Inverted Page Table

Design methods

LRU

> Data Structure

```
struct Node
{
  u16 prev;
  u16 next;
};

struct LRU
{
  Node* head;
  Node* tail;
  Node* nodes;

  u16 count = 0;

};
```

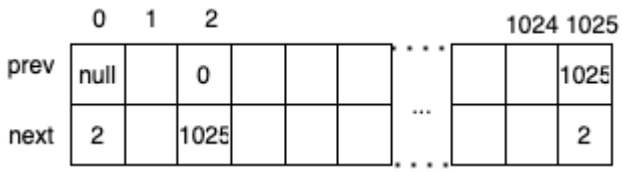

Fig.2.1 The data structure used in LRU

The LRU is a array based doubly linked list. There are 1026 nodes, where the first node and the last node are sentinels.

For each node, it contains `prev` and `next` (16bits for each). The value of `prev` and `next` is the index of the LRU array, which also indicates the frame number. Since the frame number starts from 0, we can use `index-1` to get `frame_number`.

Once the `frame_number k` is accessed, the LRU array will be updated by calling `update_LRU()`. The *least recently unused* `frame_number` can be obtained from `tail->prev - 1`.

> Functions

```
/*virtual_memory.h*/
__device__ void update_LRU(VirtualMemory* vm, u16 frame_number);
__device__ u16 get_LRU_frame_number(VirtualMemory* vm);
```

```
/*virtual_memory.cu*/
__device__ void init_LRU(VirtualMemory *vm);
__device__ void add_to_LRU(VirtualMemory *vm, u16 frame_number);
__device__ void update_LRU(VirtualMemory *vm, u16 frame_number);
__device__ u16 get_LRU_frame_number(VirtualMemory *vm)
```

`init_LRU` is used to initialize the LRU–putting the LRU into the virtual memory.
`add_to_LRU` is used to add node in the front of the doubly linked list.
`update_LRU` is used to update the LRU once a frame is accessed.
`get_LRU_frame_number` is used to get the *least recently unused* `frame_number`, which will be swapped out later.

---

vm_read

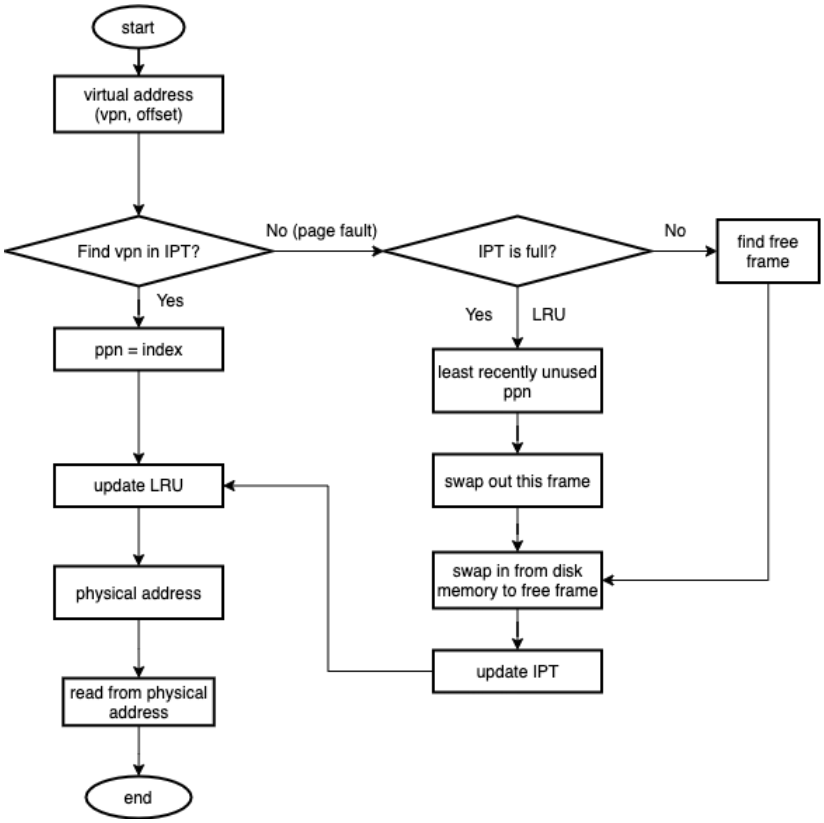

Fig.2.2 Flowchart of wm_read

vm_write

The design of `vm_write` is similar to `vm_read`.
Therefore, I simply my function `vm_write` and `vm_read` as following:

```
__device__ uchar vm_read(VirtualMemory *vm, u32 addr)
{
  u32 phy_addr = get_physical_addr(vm, addr);
  return vm->buffer[phy_addr];
}

__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value)
{
  u32 phy_addr = get_physical_addr(vm, addr);
  vm->buffer[phy_addr] = value;
}
```

vm_snapshot

The `vm_snapshot` uses `vm_read` to put data in buffer to `results`.

```
__device__ void vm_snapshot(VirtualMemory *vm, uchar *results, int offset,
                            int input_size)
{
  for (int i = 0; i < input_size; i++)
  {
    results[i + offset] = vm_read(vm, i);
  }
}
```

Problem encountered

- How to implement LRU algorithm?
  If using counter to record the *least recently unused* frame, it need to iterate all to get the target frame, and the time complexity is O(n). However, using doubly linked list (DLL), it can get the target frame in O(1), and update the LRU in O(1). Therefore, using DLL is more efficient.

- How to know the IPT is full?
  A: The `count` of `LRU` indicates the number of valid frames in IPT. Thus, if `count == PAGE_ENTRIES`, the IPT is full, so we need to swap out the *least recently unused* frame.

  ```
  // swap out
  free_frame = get_LRU_frame_number(vm);
  u32 old_vpn = vm->invert_page_table[free_frame];
  for (int i = 0; i < vm->PAGESIZE; i++)
  {
    vm->storage[old_vpn * vm->PAGESIZE + i] = vm->buffer[free_frame *
  vm->PAGESIZE + i];
  }
  ```

- How to handle page fault?
  A:

  1. The IPT is not full: `free_frame = count`;
  2. The IPT is full: find the frame to swap out => `free_frame`.

  In the end, swap disk data into `free_frame`

Execution

- The submitted folder tree is like:

```
.
└── src
    ├── 3150-cuda.sln
    ├── 3150-cuda.vcxproj
    ├── 3150-cuda.vcxproj.user
    ├── CMakeLists.txt
    ├── build.sh
    ├── data.bin
    ├── kernel.cu
    ├── main.cu
    ├── test.bin
    ├── user_program.cu
    ├── virtual_memory.cu
    └── virtual_memory.h

1 directory, 13 files
```

- Run `build.sh` to build the project

```
$ build.sh
```

- After build finished, a *build* folder will be generate, which contains an executable file `cuda`
- Run and Execute `src/build/cuda`

  The execution only takes seconds to get the output...

```
input size: 131072
pagefault number is 8193
```

  In the `user_program.cu`:

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

Since the input_size is 131072, thus, there will be 131072/32 = 4096 page faults in `vm_write`. In the end, the physical memory contains 32768 bytes data.

Moreover, in the second for loop, it reads data from 32769 addresses, where the last address with offset `input_size-32769` causes one page fault.

In `vm_snapshot`, there will be 131072/32 = 4096 page faults.

Therefore, the total page fault is 4096 + 1 + 4096 = 8193

- After execution, it generates an output file `snapshot.bin`
- Compare the difference between `snapshot.bin` and `data.bin`

```
$ diff ./data.bin ./snapshot.bin
Binary files ./data.bin and ./snapshot.bin differ
```

That indicate there is no difference between the two files.

Bonus

There are 4 threads executing the user program separately and concurrently. To avoid race condition, IPT is useful, The IPT leaves two bits for thread_id, which is used along with page number to get the frame number. In this project, the priority is thread 0 > thread 1 > thread 2 > thread3. When the thread 0 is executing the user program, the other threads will wait for synchronization. After thread 0 finished, the thread 1 will executing user program, the thread 2 and thread 3 will be waiting... In the end, the snapshot will be saved into the "snapshot_1.bin","snapshot_2.bin","snapshot_3.bin","snapshot_4.bin". The Fig3.1 shows the page fault number for each thread. Then, I compare the snapshot_k.bin with data_bin, the Fig.3.2 claims that the consistency.

```
Thread Id: 0
input size: 131072
pagefault number is 8193
Thread Id: 1
input size: 131072
pagefault number is 8193
Thread Id: 2
input size: 131072
pagefault number is 8193
Thread Id: 3
input size: 131072
pagefault number is 8193
```

Fig.3.1 Page fault number

```
-bash-4.2$ diff snapshot_1.bin data.bin
Binary files snapshot_1.bin and data.bin differ
-bash-4.2$ diff snapshot_2.bin data.bin
Binary files snapshot_2.bin and data.bin differ
-bash-4.2$ diff snapshot_3.bin data.bin
Binary files snapshot_3.bin and data.bin differ
-bash-4.2$ diff snapshot_4.bin data.bin
Binary files snapshot_4.bin and data.bin differ
-bash-4.2$
```

Fig.3.2 Compare snapshot with input file

Conclusion

In this project, I have implement the program simulating the virtual memory. During handling memory management, I review the knowledge about memory management and draw the workflow chart. Thus, I have a better understanding of the memory mapping, page table's functionality. I know how to deal with page fault, and how to realize the page replacement strategy (LRU). Although I have encountered some problems during this project, I overcame them with my efforts.