CSC3150 Assignment 3

CHEN Ang (*118010009*)

In this assignment, we are asked to implement a CUDA kernel function that simulates the mechanism of Virtual Memory (VM). We treat the Global (large-sized, high access latency) and Shared memories (small-sized, fast access speed) in the CUDA GPU as the External Storage and Physical Memory in a traditional CPU VM setup respectively. The Least Recently Used (LRU) page swapping algorithm is implemented via a counter method.

Environment

OS: Windows 10

CUDA version: CUDA 9.2

GPU model: Nvidia GeForce GTX 1050

Visual Studio version: VS 2017 (VS 2015 CUDA solution)

How to Run the Program

The program takes `~ 1 minute` to terminate in the above environment.

Select `user_program.cu`, `virtual_memory.cu`, and `main.cu` and compile them in Visual Studio via `ctrl + F7`. Then run the program via `ctrl + F5`.

Program Design

The core of this program is the implementation of the page table. We choose an inverted page table mapping virtual pages to physical pages using an array `short pt[N_PHYSICAL_PAGES]` instead of a regular page table that maps physical pages to virtual ones. The reason for this choice is that there are fewer physical pages than there are virtual pages, and so this mapping helps saving very limited shared memory space. However this mechanism makes the table lookup quite inefficient. To know if a virtual page is in the table, we have to scan through the whole page table and check one by one if any entry has been mapped to the same virtual page number of interest (`vm_translate_vp_to_pp`). This uses $O(n)$ time compared to, say, $O(1)$ with a hash map. The LRU page swapping mechanism is implemented by maintaining a counter working like a timestamp for each page table entry (stored in `vm->counter[0..N_PHYSICAL_PAGES-1]`), as well as a global counter which serves as a global clock (stored at `vm->counter[N_PHYSICAL_PAGES]`). At each time some data is accessed, the global counter is incremented by one and is assigned to the counter of the subject page. We can then find the LRU page by scanning through the pages and finding the page with the minimum count (function `vm_find_LRU_pp`). This implementation has some obvious drawbacks performance-wise compared to the doubly-linked list approach. However it saves more than half of the space (each entry has one `u32 count` data field) compared to the double linked list approach (each entry has two `Node*` data fields).

Each time a `vm_read` or `vm_write` is called, the program translates the virtual address provided by the caller into a virtual page number, then into a physical page number using `vm_translate_vp_to_pp`. The program then updates the corresponding byte at the physical address (physical page + offset). If the virtual page is not present in the page table. We use `vm_find_LRU_pp` to either find an empty page or, in the case of a full table, an LRU page to place the virtual page. In either case, the data page corresponding to the virtual address is moved from the disk to data (RAM), effectively "kicking out" the original page.

Page Fault Number and Explanation

The page fault number outputted is `8193`. The `vm_write` in the first loop produces one page fault every time it comes across a new page, creating `128 KB / 32 B == 4096` page faults. The `vm_read` in the second loop starts from `i == input_size - 1` backwards for `32769 bytes == 32 KB + 1 byte`. The first `32 KB` access of bytes is through the page table. However the last byte is not, which produces `1` page fault. Finally the `vm_snapshot` function calls `vm_read` from the first byte to the `input_size`-th byte, which is effectively the same as `vm_write` in the first loop, creating an additional `4096` page faults. Combined, there are exactly `4096 + 1 + 4096 == 8193` page faults.

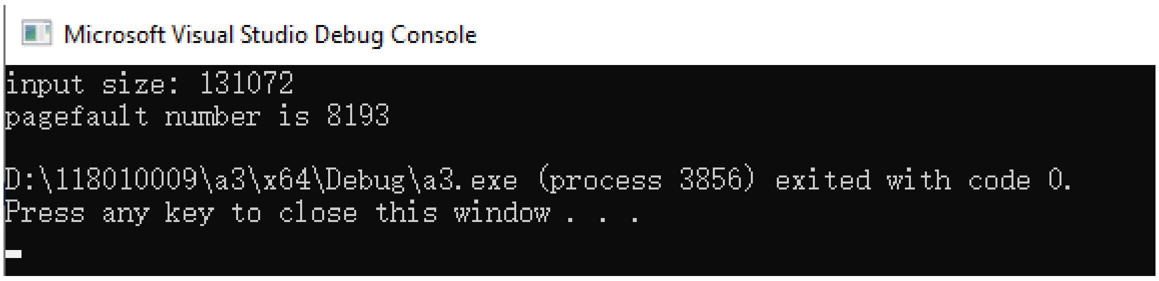
```
/* input_size == 131072 bytes == 128 KB == 4 * RAM */

for (int i = 0; i < input_size; i++)     // 4096
    vm_write(vm, i, input[i]);

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

vm_snapshot(vm, result, 0, input_size); // 4096
```

Problems I met

The first plan of mine was to implement the LRU paging system using a hashed doubly-linked list (mapping virtual pages to physical pages), which would enable $O(1)$ table lookup. However, CUDA kernel function does not support C++ STL. Considering the implementation difficulty of a hash map, I decided to compromise some speed at table lookup and use a regular array that of `Node`'s storing mapping from virtual pages to physical pages (or `-1` if the virtual page is yet mapped). The nodes also have data fields `prev` and `next`, forming a doubly-linked list for the swapping mechanism. However, this mechanism was very space-demanding (considering all the `prev` and `next` fields) and easily exceeded the limit for shared memory. For reducing memory consumption, I was forced to use the inversed page table that maps virtual pages to physical pages (using an array), and a similar doubly-linked list for LRU. The inversed mapping deteriorates the lookup performance to $O(n)$ as one has to scan through the elements of the array to figure out if a virtual page is in the table. Unfortunately this was still exceeding the memory limit. Finally I settled down to the counting method, which, despite the $O(n)$ performance of finding out the LRU page, proved to be very space-efficient and did not exceed shared memory available.
Another big problem I met was that, when I first finished the program I tested it on the CPU on my own PC, and it worked out fine. However when I transferred it to the GPU weird behavior such as random freezing and screen flashing kept popping up again and again. After three nights of frustrating debugging, I could not figure out why. That was when I started to doubt (my life seriously) if it had been a problem of my PC. So I went to C301 and tested my program on the computer there. Yes, miracles did happen. It ran perfectly.

Screenshots

- Screenshot of program output



```
input size: 131072
pagefault number is 8193

D:\118010009\a3\x64\Debug\a3.exe (process 3856) exited with code 0.
Press any key to close this window . . .
```

What I learned

I learned the basic working principles of virtual memory and the basics of CUDA programming. I also learned the LRU page swapping mechanism and knew various ways to implement it. I learned from the comparison of those different implementations the tradeoff between space and speed, and why compromise must be made from time to time. Finally, I learned to never trust my own PC too much and to always find an alternative when things don't seem to work out the way it should.