Assignment 3 Project Report

Project Introduction

With NVIDIA's Compute Unified Device Architecture, this programming project attempts to acquaint us with the workings of virtual memory and GPU programming (CUDA). With regard to this work, the inverted page table and Least Recently Used (LRU) method must be used to emulate virtual memory. The High-Performance Cluster (HPC) is used to compile and execute the task. The Kernel Version used is **3.10.0-862.el7.x86_64** with CUDA version **V11.7.99**.

Project Implementation

The software is made to mimic the GPU's virtual memory in an operating system. According to the assignment requirements, the physical memory should have 48KB and the secondary memory 128KB. The size of a table entry is 32 bytes. The page's valid/invalid bits are represented by the first 8 bytes, while the page number is represented by the next 8 bytes. This is what the assignment template dictates. We employ an additional 8 bytes to construct a counting system to designate for the least recently used (LRU) algorithm (Shown in Figure 1) when choosing a page victim because we have additional unused bytes. So, a total of 24 bytes are utilized. The addition of the counting bits makes it easy and straightforward to implement the necessary LRU algorithm. No matter whether the memory accesses the page for reading or writing, the counting variable is always raised by one. This will set the most recent page entry to zero. The algorithm will scan the table for the page with the highest count if a memory swap is necessary, choose it, and then write its contents to the disk while loading the requested page from the disk into the frame/memory. In order to mimic how an operating system (OS) will manage virtual memory, the vm read and vm write functions were created. We search for the page that the address is on and determine whether the page is there in the main memory.

```
__device__ int recently_used(VirtualMemory *vm, bool page_check, int index){
    if (page_check)
        return index;
    u32 used = vm->invert_page_table[2 * vm->PAGE_ENTRIES];
    int final_index = 0;
    int i = 1;

    while (i < vm->PAGE_ENTRIES){
        if (used < vm->invert_page_table[i + (2 * vm->PAGE_ENTRIES)])
        {
        used = vm->invert_page_table[i + (2 * vm->PAGE_ENTRIES)];
        final_index = i;
        }
        i++;
    }
    return final_index;
}
```

Figure 1. Implemented LRU Algorithm

```
__device__ int page_search(VirtualMemory *vm, u32 page_number){ // Function to search and update page
    int i = 0;
    int index = -1;
    bool check = false;

    while(i < vm->PAGE_ENTRIES){  // Page search in main memory
        if (vm->invert_page_table[i + vm->PAGE_ENTRIES] == page_number){
            check = true;
            index = i;
            break;
        }
        i++;
    }

    // Update page
    index = update(vm, check, index, page_number);
    return index;
}
```

Figure 2. Page Search Algorithm

To determine if a certain page is present in the main memory or not, we loop through the inverted page table. If it is located, the frame's validity in the main memory is then checked (valid/invalid bit). If this is true, we can simply read from and write to the address. If not (invalid), we would need to load the page into main memory from the disk. We also add one more page fault to the counter.

```
__device__ int update(VirtualMemory *vm, bool check, int index, u32 page_number){
    if (check)
    {
        if (vm->invert_page_table[index] != 0x80000000)
        return index;

        *(vm->pagefault_num_ptr) += 1;
        for (int i = 0; i < vm->PAGESIZE; i++)
            vm->buffer[(index * vm->PAGESIZE) + i] = vm->storage[(page_number * vm->PAGESIZE) + i];
        vm->invert_page_table[index] = 0x00000000;
        return index;
    }
    else {
        bool page_check = false;

        *(vm->pagefault_num_ptr) += 1; // Increase the pagefault number
        for (int i = 0; i < vm->PAGE_ENTRIES; i++)
        {
            if (vm->invert_page_table[i] == 0x80000000)
            {
            page_check = true;
            index = i;
            break;
            }
        }
        index = recently_used(vm, page_check, index);
    }

    // Write data from main memory to disk
    for (int i = 0; i < vm->PAGESIZE; i++)
        vm->storage[vm->invert_page_table[index + vm->PAGE_ENTRIES] * vm->PAGESIZE + i] =
            vm->buffer[index * vm->PAGESIZE + i];

    // Load data from disk to main memory
    for (int i = 0; i < vm->PAGESIZE; i++)
        vm->buffer[(index * vm->PAGESIZE) + i] = vm->storage[(page_number * vm->PAGESIZE) + i];
    vm->invert_page_table[index] = 0x00000000;

    // Update the invert page table
    vm->invert_page_table[index + vm->PAGE_ENTRIES] = page_number;
    return index;
```

Figure 3. Update Function

In Figure 3, if the requested page cannot be found in the main memory, we must access the disk and load the memory into the main memory from there. When this happens, we will additionally raise the page fault counter. The procedure will initially determine whether there is room in the main memory to load the page from the disk. If so, then we can just write the disk page directly into main memory. Otherwise, we would need to update the inverted page table and swap the main memory page selected by the LRU method with the disk page. After that, the requested page to be read from or written to should now be present in the main memory. Then, we only carry out the related tasks. When writing, we would write the value into that memory space or address while reading would simply include taking the value from main memory and returning it. Then, we update the page counter variable by setting the page we accessed to 0 and increasing the other pages we accessed by one.

Simply perform vm read on all of the input data for vm snapshot. We'll just run it through a for loop and have it display the information from main memory.

```
__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 (int i = 0; i < input_size; i++)
        results[i + offset] = vm_read(vm, i);
}
```

Figure 4. Vm_Snapshot Function

```
__device__ uchar vm_read(VirtualMemory *vm, u32 addr){
    /* Complate vm_read function to read single element from data buffer */
    u32 page_number = addr / vm->PAGESIZE;
    u32 page_offset = addr % vm->PAGESIZE;

    int index = page_search(vm, page_number);
    uchar data = vm->buffer[(index * vm->PAGESIZE) + page_offset]; // Get data from main memory

    for (int i = 0; i < vm->PAGE_ENTRIES; i++)
        vm->invert_page_table[i + (2 * vm->PAGE_ENTRIES)] += 1;
    vm->invert_page_table[index + (2 * vm->PAGE_ENTRIES)] = 0; // Set the accessed page's count ot 0

    return data;
}
```

Figure 5. Vm_Read Function

Project Summary and Learnings

This assignment improved my knowledge of the computer's paging mechanism and provided insight into virtual memory and GPU programming. Prior to now, I had trouble separating the ideas of secondary storage, physical memory, and virtual memory. But now that I've finished this project, I fully comprehend this paging notion. Stimulating virtual memory also aids in better understanding how the OS maintains memory and the strategies it employs to do so. We also learn more about and get experience with the CUDA API for NVIDIA GPUs. We had the opportunity to put fundamentals of general GPU programming to use.

Appendix

```
[119010507@node21 CSC3150_Assignment3]$ sh slurm.sh
main.cu(89): warning #2464-D: conversion from a string literal to "char *" is deprecated

main.cu(108): warning #2464-D: conversion from a string literal to "char *" is deprecated

main.cu(89): warning #2464-D: conversion from a string literal to "char *" is deprecated

main.cu(108): warning #2464-D: conversion from a string literal to "char *" is deprecated

srun: job 27418 queued and waiting for resources
srun: job 27418 has been allocated resources
input size: 131072
pagefault number is 8193
[119010507@node21 CSC3150_Assignment3]$
```

Figure 6. Project Result