Assignment Report

CSC 3150 Virtual Memory Management

Wei Wu (吴畏)

118010335

November 7, 2020

The School of Data Science

香 港 中 文 大 學（深 圳）
The Chinese University of Hong Kong, Shenzhen

1. Environment of Running My Program. (e.g., OS, VS Version, CUDA Version, GPU Information etc.)

a. OS: Microsoft Windows [Version 10.0.19041.610]

b. VS: Microsoft Visual Studio Community 2017 [Version 15.9.28]

c. VS Toolkit: Visual Studio 2015 (v140)

d. CUDA: NVIDIA CUDA 11.1

e. GPU: NVIDIA GeForce GTX 1070

f. GPU Driver: NVIDIA GeForce Game Ready Driver 456.81

2. Execution Steps of Running My Program.

a. Open CSC3150_A3\CSC3150_A3.vcxproj using Visual Studio 2017

b. Do NOT change the Target Platform Version or the Toolkit.

c. Right click on and compile the cuda files (.cu). (Or use Ctrl+F7).

d. Press Ctrl+F5 to run the program.

3. How Did I Design My Program?

3.1. The Original Program (Source)

a. The main program (main.cu) follows the template. It loads the binary file named “data.bin” to the input buffer before the kernel launches. It also shows the size of input buffer. Then, it launches the GPU kernel with a single thread, and dynamically allocates 16KB of the share memory, which will be used for variables declared as “extern

__shared__".

```
int main() {
  cudaError_t cudaStatus;
  int input_size = load_binaryFile(DATAFILE, input, STORAGE_SIZE);

/* Launch kernel function in GPU, with single thread
and dynamically allocate INVERT_PAGE_TABLE_SIZE bytes of share memory,
which is used for variables declared as "extern __shared__" */
mykernel<<<1, 1, INVERT_PAGE_TABLE_SIZE>>>(input_size);
```

b. The GPU kernel creates and initializes a Virtual Memory instance. Then it calls the user program.

```
__global__ void mykernel(int input_size) {

  // memory allocation for virtual_memory
  // take shared memory as physical memory
  __shared__ uchar data[PHYSICAL_MEM_SIZE];

  VirtualMemory vm;
  vm_init(&vm, data, storage, pt, &pagefault_num, PAGE_SIZE,
          INVERT_PAGE_TABLE_SIZE, PHYSICAL_MEM_SIZE, STORAGE_SIZE,
          PHYSICAL_MEM_SIZE / PAGE_SIZE);

  // user program the access pattern for testing paging
  user_program(&vm, input, results, input_size);
}
```

```
// initialize the virtual memory
__device__ void vm_init(VirtualMemory *vm, uchar *buffer, uchar *storage,
    u32 *invert_page_table, int *pagefault_num_ptr,
    int PAGESIZE, int INVERT_PAGE_TABLE_SIZE,
    int PHYSICAL_MEM_SIZE, int STORAGE_SIZE,
    int PAGE_ENTRIES) {
    // init variables
    vm->buffer = buffer;
    vm->storage = storage;
    vm->invert_page_table = invert_page_table;
    vm->pagefault_num_ptr = pagefault_num_ptr;

    // init constants
    vm->PAGESIZE = PAGESIZE;
    vm->INVERT_PAGE_TABLE_SIZE = INVERT_PAGE_TABLE_SIZE;
    vm->PHYSICAL_MEM_SIZE = PHYSICAL_MEM_SIZE;
    vm->STORAGE_SIZE = STORAGE_SIZE;
    vm->PAGE_ENTRIES = PAGE_ENTRIES;

    // before first vm_write or vm_read
    init_invert_page_table(vm);
}
```

```
// initialize the inverted page table
__device__ void init_invert_page_table(VirtualMemory *vm) {
    for (int i = 0; i < vm->PAGE_ENTRIES; i++) {
        // invert_page_table[i] (from 0 to PAGE_ENTRIES - 1) stores valid-invalid bit (initialized as false)
        vm->invert_page_table[i] = 0x80000000; // invalid := MSB is 1
        // invert_page_table[i] (from PAGE_ENTRIES to 2 * PAGE_ENTRIES) stores page number
        vm->invert_page_table[i + vm->PAGE_ENTRIES] = i;
    }
}
```

c. The user program (user_program.cu of A3_template) does vm_write() from logical address 0 to logical address input_size sequentially. Then, it does vm_read() in the reverse order. And then, it calls vm_snapshot() to move the content of the data buffer to the result buffer.

```
__device__ void user_program(VirtualMemory *vm, uchar *input, uchar *results,
                             int input_size) {
  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);
}
```

d. In the virtual memory class (virtual_memory.cu), there are two major

methods: vm_read(), vm_write(). vm_read() first calls valid_addr() to check whether the logical address is valid. If valid, it calls get_phy_addr() to translate the logical address into a physical address and return the value in that physical address. Similarly, vm_write() does the same thing as vm_read() except that it writes a value to the derived physical address.

```
// read single element from data buffer
__device__ uchar vm_read(VirtualMemory *vm, u32 addr) {
    if (!valid_addr(vm, addr))
        return NULL;
    int phy_addr = get_phy_addr(vm, addr);
    return vm->buffer[phy_addr];
}

// write value into data buffer
__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value) {
    if (!valid_addr(vm, addr))
        return;
    int phy_addr = get_phy_addr(vm, addr);
    vm->buffer[phy_addr] = value;
}
```

e. The function valid_addr() checks if the logical address is valid. If the logical address is greater than the storage size, it is out of bound.

```
// check if the address is valid (in bound)
__device__ bool valid_addr(VirtualMemory *vm, u32 addr) {
    if (addr > (u32)vm->STORAGE_SIZE) {
        printf("Logical address out of bound!\n");
        return false;
    }
    return true;
}
```

f. The function get_phy_addr() translates a logical address into a physical address. There are two possible cases. The first is a page hit, which

means the desired page number is already as a valid entry in the inverted page table. All we need to do is update the LRU status and return the physical address. The other case is a page miss, which means the desired page is not in the primary memory but the secondary memory. In this case, we call get_frame_number_on_page_fault() to swap in the page and return the physical address.

```
// translate a logical address into a physical address
__device__ int get_phy_addr(VirtualMemory *vm, u32 logic_addr) {
    u32 page_offset = logic_addr % vm->PAGESIZE;
    u32 page_number = logic_addr / vm->PAGESIZE;
    int phy_addr;
    /* Case 1: PAGE HIT */
    // search the inverted page table
    for (int i = 0; i < vm->PAGE_ENTRIES; i++) {
        // if "page_number" is in the inverted page table
        if (vm->invert_page_table[i + vm->PAGE_ENTRIES] == page_number) {
            if (vm->invert_page_table[i] != 0x80000000) {
                // update the LRU status
                update_lru(vm, i);
                // return the physical address
                phy_addr = (i * vm->PAGESIZE) + (int)page_offset;
                return phy_addr;
            }
        }
    }
    /* Case 2: PAGE MISS */
    // swap in the desired page from the secondary memory
    int frame_number = get_frame_number_on_page_fault(vm, page_number);
    // return the physical address
    phy_addr = (frame_number * vm->PAGESIZE) + (int)page_offset;
    return phy_addr;
}
```

g. The function get_frame_number_on_page_fault() returns a frame number on page misses. It starts with counting the page miss scenario. There are two possible cases. Case 1 is that the table is not full. That is, there exists an invalid entry. In this case, we only need to swap in,

update the table and the LRU status, and return this frame number. Case 2 is that the table is full. We need to find the LRU frame number, do swap in and swap out, update the table and the LRU status, and then return the frame number.

```
// get a frame number on page fault
__device__ int get_frame_number_on_page_fault(VirtualMemory *vm, u32 page_number) {
    // count the page fault
    (*vm->pagefault_num_ptr)++;
    int frame_number_to_swap = 0;
    /* Case 1: the table is not full */
    // search the inverted page table for space
    for (int i = 0; i < vm->PAGE_ENTRIES; i++) {
        // if there exists an invalid entry, just swap in
        if (vm->invert_page_table[i] == 0x80000000) {
            // swap in
            swap(vm, page_number, vm->invert_page_table[i + vm->PAGE_ENTRIES], i, false);
            // update the inverted page table
            vm->invert_page_table[i + vm->PAGE_ENTRIES] = page_number;
            // update the LRU status
            update_lru(vm, i);
            // return this frame number
            return i;
        }
        // find the LRU frame number
        // pick the least indexed entry to be the victim page in case of tie
        else if (vm->invert_page_table[i] >
            vm->invert_page_table[frame_number_to_swap])
            frame_number_to_swap = i;
    }
    /* Case 2: the table is full */
    // i.e. no invalid entry
    // swap out and swap in
    swap(vm, page_number, vm->invert_page_table[frame_number_to_swap + vm->PAGE_ENTRIES], frame_number_to_swap, true);
    // update the inverted page table
    vm->invert_page_table[frame_number_to_swap + vm->PAGE_ENTRIES] = page_number;
    // update the LRU status
    update_lru(vm, frame_number_to_swap);
    // return the frame number
    return frame_number_to_swap;
}
```

h. The function swap() swaps out the specified page from data buffer to secondary memory if the boolean argument swap_out is asserted. It then swaps in the specified page from secondary memory to data buffer.

```
// swap out and swap in pages
__device__ void swap(VirtualMemory *vm, int in_page_number, int out_page_number, int frame_number, bool swap_out) {
    // move from data buffer to secondary memory
    if (swap_out) {
        for (int i = 0; i < vm->PAGESIZE; i++)
            vm->storage[out_page_number * vm->PAGESIZE + i] = vm->buffer[frame_number * vm->PAGESIZE + i];
    }
    // move from secondary memory to data buffer
    for (int i = 0; i < vm->PAGESIZE; i++)
        vm->buffer[frame_number * vm->PAGESIZE + i] = vm->storage[in_page_number * vm->PAGESIZE + i];
}
```

i. The function update_lru() updates the LRU status. Note that increasing an LRU status of 0x7FFFFFFF by 1 will result in 0x80000000 (invalid).

In a sense, this is similar to an overflow. Therefore, we should not increase an LRU status with a value of 0x7FFFFFFF.

```
// update the LRU status
__device__ void update_lru(VirtualMemory *vm, int frame_number) {
    for (int i = 0; i < vm->PAGE_ENTRIES; i++) {
        // if valid, increase the LRU status by 1
        // if the LRU status == 0x7FFFFFFF, do not increase
        if (vm->invert_page_table[i] != 0x80000000
            && vm->invert_page_table[i] != 0x7FFFFFFF)
            vm->invert_page_table[i]++;
    }
    // reset the LRU status to 0
    vm->invert_page_table[frame_number] = 0;
}
```

j. The vm_snapshot() function simply dumps the content of the data buffer to the result buffer by iteratively calling vm_read().

```
// load elements from data to result buffer
__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);
}
```

k. The main program dumps the data in the result buffer into the binary file named “snapshot.bin” and outputs the page fault number.

```
write_binaryFile(OUTFILE, results, input_size);

printf("pagefault number is %d\n", pagefault_num);
```

3.2. The Bonus Program

a. In the virtual_memory.h, "device_functions.h" is included for the __syncthreads() function, which is used for thread synchronization.

```
// BONUS
#include "device_functions.h"
```

b. In the VirtualMemory structure, two additional data fields are added,

namely THREAD_NUM and THREAD_ID. The former represents the total number of threads while the latter represents the thread ID of the current thread.

```
// BONUS
int THREAD_NUM;
int THREAD_ID;
```

c. The initialization function vm_init() is modified to pass two additional arguments, namely THREAD_NUM and THREAD_ID.

```
// BONUS
vm->THREAD_NUM = THREAD_NUM;
vm->THREAD_ID = THREAD_ID;
```

d. In the very beginning of vm_read() and vm_write(), __syncthreads() is called for thread synchronization. Then, each of the threads will check whether it should deal with the given logical address. If it is others' duty to deal with the logical address, it simply returns.

```
// read single element from data buffer
__device__ uchar vm_read(VirtualMemory *vm, u32 addr) {
    // synchronize the threads
    __syncthreads();
    if (!valid_addr(vm, addr))
        return NULL;
    // BONUS: skip the job of the other threads
    if (addr % vm->THREAD_NUM != vm->THREAD_ID)
        return NULL;
    int phy_addr = get_phy_addr(vm, addr);
    return vm->buffer[phy_addr];
}

// write value into data buffer
__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value) {
    // synchronize the threads
    __syncthreads();
    if (!valid_addr(vm, addr))
        return;
    // BONUS: skip the job of the other threads
    if (addr % vm->THREAD_NUM != vm->THREAD_ID)
        return;
    int phy_addr = get_phy_addr(vm, addr);
    vm->buffer[phy_addr] = value;
}
```

e. Similarly, in the very beginning of vm_snapshot(), __syncthreads() is also called for thread synchronization. Then, each thread only deals with its corresponding logical addresses.

```
// load elements from data to result buffer
__device__ void vm_snapshot(VirtualMemory *vm, uchar *results, int offset, int input_size) {
    // BONUS: skip the job of the other threads
    for (int i = vm->THREAD_ID; i < input_size; i += vm->THREAD_NUM) {
        // synchronize the threads
        __syncthreads();
        results[i + offset] = vm_read(vm, i);
    }
}
```

f. In main.cu, the number of threads is defined as 4. In main(), THREAD_NUM is asserted instead of 1 when initializing the kernel.

```
// the number of threads used is 4
#define THREAD_NUM (1 << 2)
```

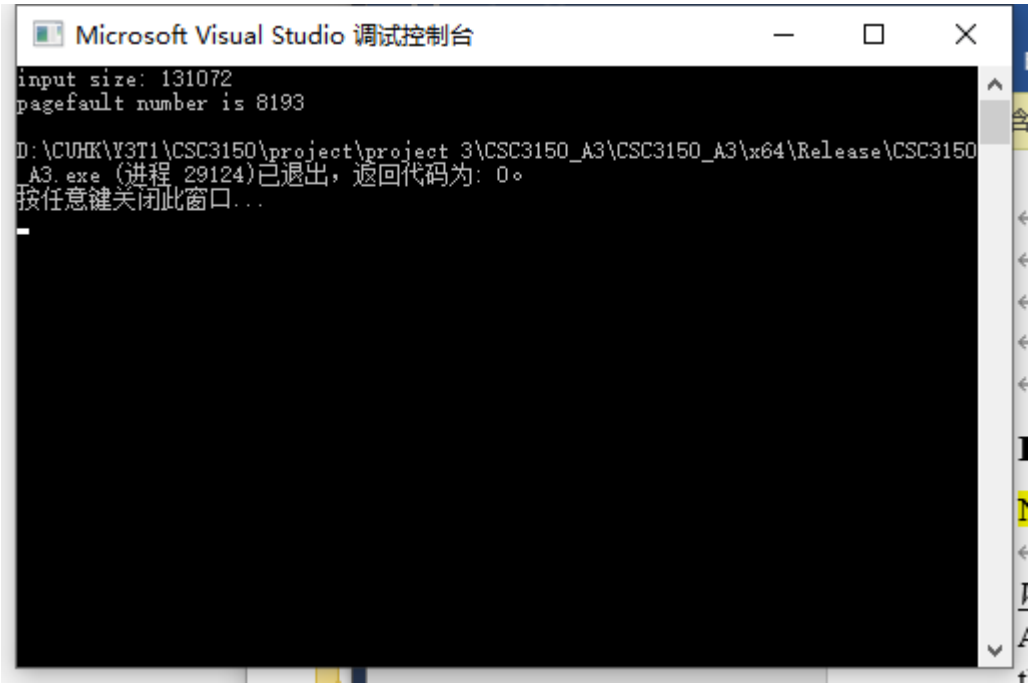
```
/* Launch kernel function in GPU, with MULTIPLE thread
and dynamically allocate INVERT_PAGE_TABLE_SIZE bytes of share memory,
which is used for variables declared as "extern __shared__" */
mykernel<<<1, THREAD_NUM, INVERT_PAGE_TABLE_SIZE>>>(input_size);
```

g. In mykernel(), the number of threads and the current thread ID are passed as arguments to vm_init().

```
// BONUS: thread ID
int tid = blockDim.x * blockIdx.x + threadIdx.x;

VirtualMemory vm;
vm_init(&vm, data, storage, pt, &pagefault_num, PAGE_SIZE,
        INVERT_PAGE_TABLE_SIZE, PHYSICAL_MEM_SIZE, STORAGE_SIZE,
        PHYSICAL_MEM_SIZE / PAGE_SIZE, THREAD_NUM, tid);
```

4. What's the Page Fault Number of My Output? Explain How Does It Comeout.



input size: 131072

pagefault number is 8193

This number can be separated into three parts, mathematically 8193 = 4096 + 1 + 4096.

The first 4096 corresponds to the writing stage. In the beginning, the data buffer and the inverted page table can be viewed as "empty". Whenever a page is accessed for the first time, there will be a page fault.

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

As in the user_program.cu, in total (input_size / page_size) = (131072 / 32) = 4096 pages are accessed for the first time, resulting in 4096 page faults.

The 1 corresponds to the reading stage. After the writing operations, the last PAGE_ENTRIES = (PHYSICAL_MEM_SIZE / PAGE_SIZE) = (32768 / 32) = 1024 pages are in the physical memory.

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

As in the user_program.cu, the program reads the last (32769 / 32 + 1) = (1024 + 1) = 1025 pages. The last 1024 pages are in the physical memory while the 1025th page from the bottom is not. Hence, there will be exactly 1 page fault.

The second 4096 corresponds to the snapshot stage. After the reading stage, the 2nd to the 1025th pages from the bottom should be in the physical memory.

```
// load elements from data to result buffer
__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);
}
```

As in the snapshot function, the logical pages are accessed in the ascending order. Since there are in total 4096 logical pages, and in the beginning only the 2nd to the 1025th pages from the bottom are in the physical memory, by the LRU algorithm, every 8 consecutive read operations should trigger a page fault. So, the snapshot stage will cause 4096 page faults.

5. What Problems I Met in This Assignment and What Are My Solution?

a. How to make the code more concise? That is, how to minimize the length of the code?

   I noticed that the read operation and the write operation had a lot of common procedures, such as determining page hits and page faults, swapping in and out, updating the inverted page table, and updating the LRU status, etc. If I wrote these logic separately in the vm_write() and vm_read() function, there would be large blocks of duplicated code, which would not be desirable. So, I extracted the common procedures as shared functions: swap() that would swap out and in pages, get_frame_number_on_page_fault() that would return a frame number on page fault, update_lru that would update LRU

status, and get_phy_addr that would translate a logical address into a physical address. In this design, the length of the code is greatly reduced, as in the virtual_memory.cu.

b. On page fault, how to swap in the desired page?

   If there is an invalid page table entry, then we only need to swap in the desired page. Otherwise, we must first swap out the page in the physical memory and then swap in the desired page.

c. How to implement the LRU algorithm?

   Although I could have implemented an LRU algorithm with time complexity O(1) using a doubly linked list and a hash map, I chose an O(N) approach. The reason was that the O(1) algorithm had a much larger space complexity: the doubly linked list with valid-invalid bits could cost 32 bytes per entry and the hash map is in some sense equivalent to a (forward) page table. As the CUDA specified in the assignment description only provided 16KB for page settings, the O(1) algorithm was not affordable in terms of space.

```
// update the LRU status
__device__ void update_lru(VirtualMemory *vm, int frame_number) {
    for (int i = 0; i < vm->PAGE_ENTRIES; i++) {
        // if valid, increase the LRU status by 1
        // if the LRU status == 0x7FFFFFFF, do not increase
        if (vm->invert_page_table[i] != 0x80000000
            && vm->invert_page_table[i] != 0x7FFFFFFF)
            vm->invert_page_table[i]++;
    }
    // reset the LRU status to 0
    vm->invert_page_table[frame_number] = 0;
}
```

Therefore, I used the invalid-invalid bit (which was actually implemented as a 32-bit variable) to record the “unused time”, or namely “LRU status”. On every page access, whether page hit or page fault, the LRU status of every valid page table entry would be incremented by 1. Then, the LRU status of the accessed page would be reset to 0. Note that increasing an LRU status of 0x7FFFFFFF by 1 will result in 0x80000000 (invalid). In a sense, this is similar to an overflow. Therefore, we should not increase an LRU status with a value of 0x7FFFFFFF.

d. How to design the multi-thread execution?

Each thread only deals with its corresponding logical addresses. The logical addresses are distributed to threads according to their modulo of THREAD_NUM. This scheme ensures that the multi-thread operation is both correct and efficient.

6. Screenshot of My Program Output.

a. Source

```
input size: 131072
pagefault number is 8193

D:\CUHK\Y3T1\CSC3150\project\project 3\CSC3150_A3\CSC3150_A3\x64\Release\CSC3150
_A3.exe (进程 22124)已退出，返回代码为: 0。
按任意键关闭此窗口...
```

```
C:\Users\WuWei>fc "D:\CUHK\Y3T1\CSC3150\project\project 3\CSC3150_A3\CS
C3150_A3\data.bin" "D:\CUHK\Y3T1\CSC3150\project\project 3\CSC3150_A3\C
SC3150_A3\snapshot.bin"
正在比较文件 D:\CUHK\Y3T1\CSC3150\PROJECT\PROJECT 3\CSC3150_A3\CSC3150_
A3\data.bin 和 D:\CUHK\Y3T1\CSC3150\PROJECT\PROJECT 3\CSC3150_A3\CSC315
0_A3\SNAPSHOT.BIN
FC: 找不到差异
```

b. Bonus

```
input size: 131072
pagefault number is 8193

D:\CUHK\Y3T1\CSC3150\project\project 3\Bonus\CSC3150_A3\CSC3150_A3\x64\Release\C
SC3150_A3.exe (进程 27032)已退出，返回代码为: 0。
按任意键关闭此窗口...
```

```
C:\Users\WuWei>fc "D:\CUHK\Y3T1\CSC3150\project\project 3\Bonus\CSC3150
_A3\CSC3150_A3\data.bin" "D:\CUHK\Y3T1\CSC3150\project\project 3\Bonus\
CSC3150_A3\CSC3150_A3\snapshot.bin"
正在比较文件 D:\CUHK\Y3T1\CSC3150\PROJECT\PROJECT 3\BONUS\CSC3150_A3\CS
C3150_A3\data.bin 和 D:\CUHK\Y3T1\CSC3150\PROJECT\PROJECT 3\BONUS\CSC31
50_A3\CSC3150_A3\SNAPSHOT.BIN
FC: 找不到差异
```

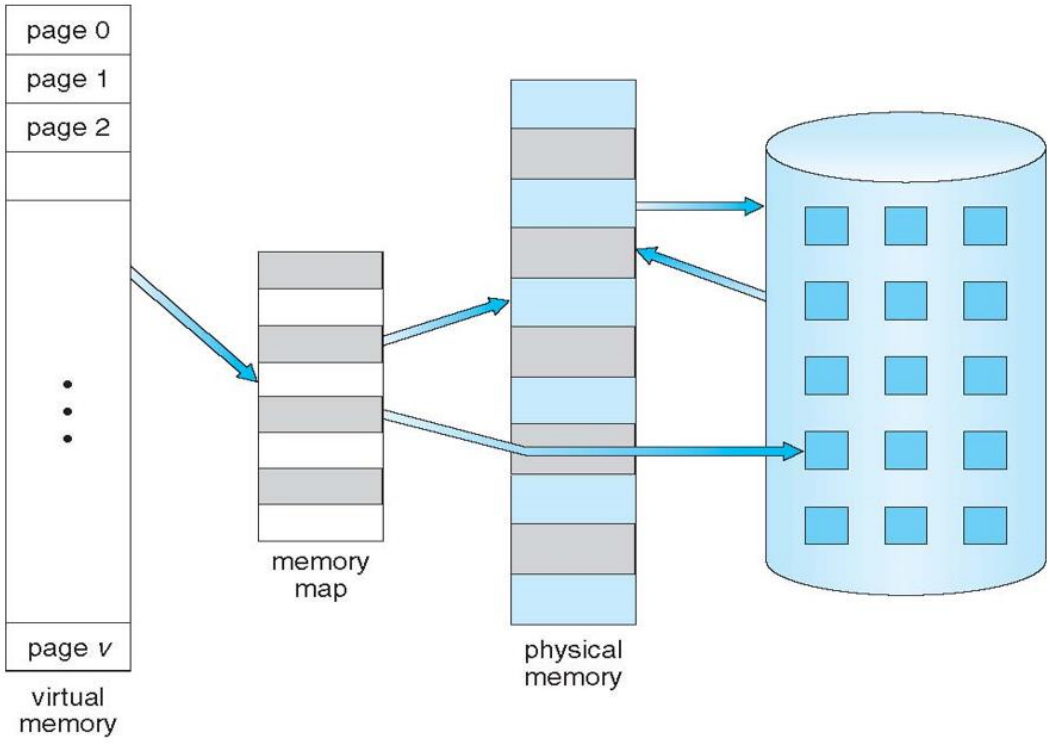
7. What Did I Learn from This Assignment?

a. Virtual Memory

The virtual memory separates the user logical memory from the physical memory. With virtual memory, only part of the program needs to be in memory for execution, and logical address space can be much larger than physical address space. It can be implemented by demand paging or demand segmentation.

b. Demand Paging

Demand paging only transfers a page into memory only when it is needed. This approach can effectively reduce the I/O and physical memory needed, and thus enables faster response and more concurrent users.



c. The Computer Unified Device Architecture

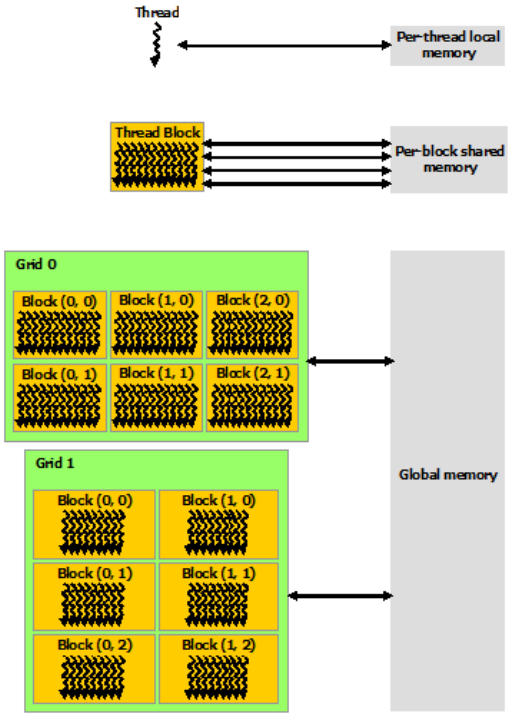
CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.

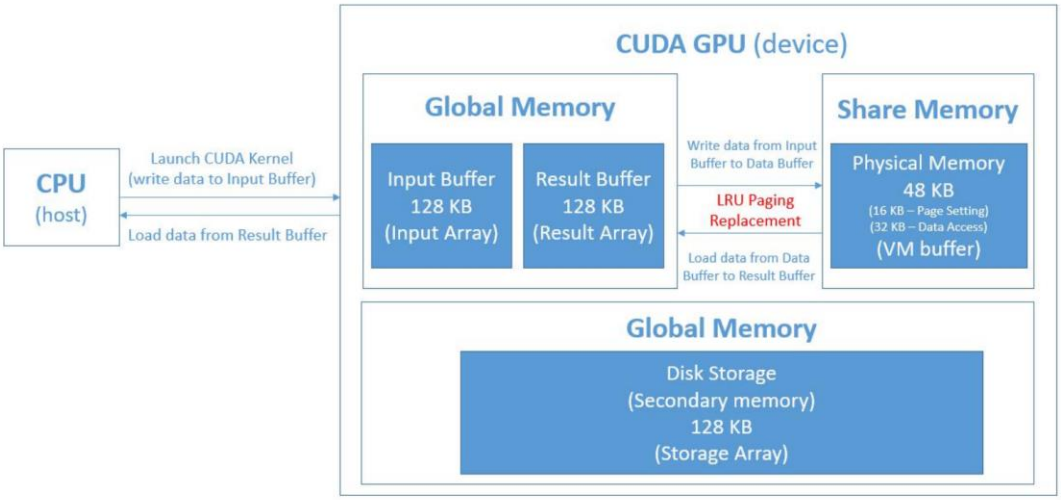
In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel. When using CUDA, developers

program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords.

d. The CUDA memory hierarchy

CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.





e. How to properly declare functions in CUDA programming?

__global__

Kernels designated by function qualifier

Function called from host and executed on device

Must return void and cannot be a member of class.

__device__

Other CUDA function qualifiers

Functions called from device and run on device

Cannot be called from host code

__host__

Function called from host and executed on host (default)

f. How to properly declare variables in CUDA programming?

__device__

Stored in global memory (large, high latency, no cache)

Allocated with cudaMalloc (__device__ qualifier implied)

Accessible by all threads

Lifetime: application

__shared__

Stored in on-chip shared memory (very low latency)

Specified by execution configuration or at compile time

Accessible by all threads in the same thread block

Lifetime: thread block

__managed__

Memory space specifier, optionally used together with __device__

Can be referenced from both device and host code, e.g., its address can be taken or it can be read or written directly from a device or host function.

Lifetime: application.

Unqualified variables:

Scalars and built-in vector types are stored in registers

What doesn’t fit in registers spills to “local” memory