Report of project 3

Manual and abstract

I worte project 3 which created virtual memory for NVIDIA GPU. What I mainly created is in virtual_memory.cu. I wrote makefile. And you just need to change working directory to Source and type make in terminal, then executable file main.cu comes out. Type main in terminal, then you can test this program.

Design of program

In this project, I mainly design virtual memory for NVIDIA GPU architecture. In virtual_memory.cu, I mainly implement three functions: vm_read, vm_write, vm_snapshot. I will mainly elaborate on these three functions. Before the elaboration, I will talk about some important functions created by myself.

__device__ u32 vm_find_vaddr_pp_num(VirtualMemory *vm, u32 addr)

Get_pp_num() gets virtual address (32 bit), returns the physical page number of that virtual memory address. The mechanism is that I created an inverted page table, which is implanted by an array. The index of inverted page table represents physical page number (pp_num) and the value of inverted page table are virtual page numbers. The get_pp_num function iterates all entries and find corresponding index or pp_num for vpn. If there is no vpn for input virtual address, then page fault occurs and return -1 which indicates page fault occurs.

Corresponding code is below:

```
__device__ u32 vm_find_vaddr_pp_num(VirtualMemory *vm, u32 addr)
{
    u32 i;
    u32 vp_num = vm_vpage_num(vm, addr);

    for (i = 0; i < vm->PAGE_ENTRIES; i++) {
        if (vm->invert_page_table[i] == vp_num) {
            return i;
        }
    }

    return (u32)-1;
}
```

__device__ void vm_copy_page_data(VirtualMemory *vm, u32 pp_num, u32 vp_num)

We use LRU algorithm to do the page swap. We use counter way to implement LRU algorithm. Each physical memory entry has an age. The age represents its activity. The larger age is, the more active this physical memory entry is. Once the physical memory entry is hit, this physical memory entry's age will be updated to the eldest. The update operation just like stack operation implemented by array. The value of physical memory entry represents its age. So once the physical memory entry is hit, its age becomes the eldest. And the entries whose age is larger than it has decrease their age by one each. The update operation is just like stack operation. The hit entry removes and put it on the top. The code is below:

```
__device__ void vm_copy_page_data(VirtualMemory *vm, u32 pp_num, u32 vp_num)
{
    int i;
    u32 poffest = pp_num * vm->PAGESIZE;
    u32 voffest = vp_num * vm->PAGESIZE;

    for (i = 0; i < vm->PAGESIZE; i++){
        vm->buffer[poffest + i] = vm->storage[voffest + i];
    }

    return;
}
```

__device__ u32 vm_do_page_fault(VirtualMemory *vm, int vp_num)

__device__ void vm_do_lru(VirtualMemory *vm, u32 pp_num)

I designed the two functions for page fault problem. When page fault occurs, thes function checks to find the youngest physical memory entry and swap the physical memory with the content of input virtual memory address. To find the youngest physical memory entry, we iterate all physical memory entry's age, to find the youngest and corresponding replaced physical memory, swap pages and overwrite the virtual memory for that replaced physical memory.

Code is below.

```
__device__ u32 vm_do_page_fault(VirtualMemory *vm, int vp_num)
{
    int pp_num;

    pp_num = vm_find_physical_page_num(vm);
    pp_num -= vm->PAGE_ENTRIES;

    vm_copy_page_data(vm, pp_num, vp_num);

    vm->invert_page_table[pp_num] = vp_num;

    *vm->pagefault_num_ptr = *vm->pagefault_num_ptr + 1;

    return pp_num;
}

__device__ void vm_do_lru(VirtualMemory *vm, u32 pp_num)
{
    u32 i;
    u32 pp_index = pp_num + vm->PAGE_ENTRIES;
    u32 max = vm->PAGE_ENTRIES << 1;

    for (i = vm->PAGE_ENTRIES; i < max; i++) {
        if (vm->invert_page_table[i] > vm->invert_page_table[pp_index]){
            vm->invert_page_table[i] = vm->invert_page_table[i] - 1;
        }
    }

    vm->invert_page_table[pp_index] = vm->PAGE_ENTRIES - 1;

    return;
}
```

__device__ uchar vm_read(VirtualMemory *vm, u32 addr)

In this function, the virtual address is input, we use get_pp_num() to get physical page number and then to get physical memory address. Then we find the memory content in buffer. If there is a page fault, just call swap function. Remember that everytime the physical page is hit, then update the date by calling age_update().

Codes are below:

```
__device__ uchar vm_read(VirtualMemory *vm, u32 addr)
{
    u32 vp_num = vm_vpage_num(vm, addr);
    u32 page_offest = vm_addr_page_offest(vm, addr);
    u32 pp_num = vm_find_vaddr_pp_num(vm, addr);
    u32 physical_addr;
    uchar read_data;

    //not find
    if (pp_num == (u32)-1) {
        //find a physical page, do page table mapping
        pp_num = vm_do_page_fault(vm, vp_num);

        physical_addr = vm_get_physical_addr(vm, pp_num, page_offest);
        //do lru update
        vm_do_lru(vm, pp_num);

        return vm->buffer[physical_addr];
    }

    physical_addr = vm_get_physical_addr(vm, pp_num, page_offest);
    vm_do_lru(vm, pp_num);
    return vm->buffer[physical_addr];
}
```

__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value)

In this part, we write the value into physical memory buffer. Just like in vm_read(), we use vm_read() to get the physical memory of input virtual address. If meeting page fault, then just call swap() function. Codes are below:

```
__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value)
{
    u32 vp_num = vm_vpage_num(vm, addr);
    u32 page_offest = vm_addr_page_offest(vm, addr);
    u32 pp_num = vm_find_vaddr_pp_num(vm, addr);
    u32 physical_addr;

    //not find
    if (pp_num == (u32)-1) {
        //find a physical page, do page table mapping
        pp_num = vm_do_page_fault(vm, vp_num);
        physical_addr = vm_get_physical_addr(vm, pp_num, page_offest);
        //do lru update
        vm_do_lru(vm, pp_num);

        vm->buffer[physical_addr] = value;
        vm->storage[addr] = value;

        return;
    }

    physical_addr = vm_get_physical_addr(vm, pp_num, page_offest);
```

__device__ void vm_snapshot(VirtualMemory *vm, uchar *results, int offset, int input_size)

In this part, we just load elements of vm buffer to results buffer.

```
__device__ void vm_snapshot(VirtualMemory *vm, uchar *results, int offset,
                            int input_size)
{
    int i;
    uchar read_data;

    printf("snapshot:\n");
    for (i = 0; i < input_size; i++) {
        read_data = vm_read(vm, i+offset);
        results[i] = read_data;
    }

    return;
}
```