Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

Lab 6
Course: Parallel Processing and Distributed Systems

Duc Hai Nguyen, Manh-Thin Nguyen
Email: haind@cse.hcmut.edu.vn

October 2, 2016

Objective This lab introduces the application of GPU in massive parallel computing.

Content Information included in the lab

- Introduction to GPGPU programming
- GPU architecture and CUDA platform for GPGPU
- CUDA programming

Prerequisite Student should has basic knowledge of SIMD architecture, GPU and C programming.

1 Introduction

1.1 General-purpose computing on GPU

For decades, Central Processing Unit (CPU) has been considered as the brain of computer and the most important methods for the improvement of the performance of computing is to increase CPU's speed. However, in recent years, manufactures are forced to look for alternative solutions because the power and heat restriction together with physical limit of transistor size prevent processing cores from reaching higher clock rate. One of promising approach is to put more processors into a computer instead of increasing the computing power of existing cores.
General-purpose computing on GPU (GPGPU) is one of the most noticeable solutions of the new approach. GPU is a co-processor designed to work together with CPU. Originally, this device is designed to cope with video and graphical applications since CPU alone does not handle those application efficiently. The primary principle underlying the design of GPU is to use a massive number of cheap (and weak) processors to solve problem rather than affording a powerful but expensive single processor. This strategy works very well. It even leads to a trend of using GPU for solving other problems which do not related to video or graphic but share the same features: require massive parallelization.

1.2 GPU architecture

GPU is designed based on SIMD architecture. All processing cores share the same instruction stream but work on different chunks of data. Therefore, the performance of GPU strongly depends on the nature of the problem: it only work well on applications which do not require much of communication and synchronization. The power of GPU based on combining the computing resources of multiple cores so inside a single GPU devices, there is a lot of (typically more than 100) processing core. Those cores share a large global memory integrated inside the device. Each of them also has their own private memory known as registers. Processor are organized by groups of multiple processor sharing the same cache region. The deployment of cache is to reduce data read and write latency from and to the large but slow global shared memory.

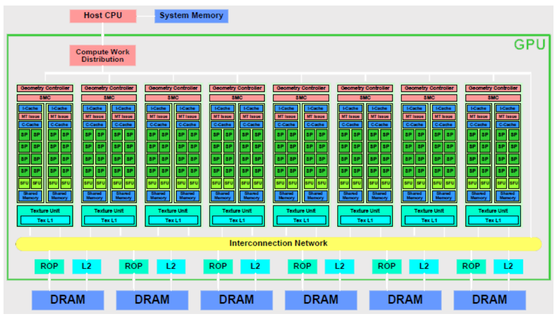


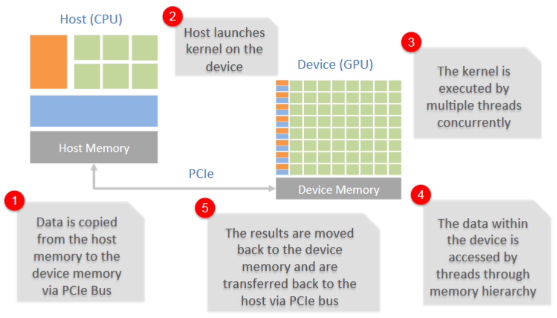
Figure 1.1: CUDA architecture.

GPU can work as an independent system beside the computer. However, it cannot directly interact with other devices in the system (e.g. I/O devices) so it still need the CPU to helps it cope with those problems. Particularly, in GPGPU model, GPU acts as a computing device whose main task is to computing and processing data while CPU acts as a control device who helps monitor and assign tasks to different parts of the system as well as fetch input data to and pull output data from GPU.
Among GPU programming platform, CUDA is one of the most popular and currently used by a lot of scientific applications. CUDA is a compiler and toolkit for programming Nvidia GPUs introduced by Nvidia in late 2006. It enables dramatic increases computing performance by harnessing the power of the GPU. CUDA is designed to work with programming language such as C++, C, and FORTRAN. This accessibility makes it easier to use GPU resources for parallel computing.

2 Programming Examples

2.1 A Simple GPU Vector Sum

The dataflow of a CUDA program is illustrated in Figure 2.1. This is introduced through the vector sum program.



Host (CPU)
Host Memory
2
Host launches kernel on the device
Device (GPU)
3
The kernel is executed by multiple threads concurrently
PCIe
Device Memory
1
Data is copied from the host memory to the device memory via PCIe Bus
5
The results are moved back to the device memory and are transferred back to the host via PCIe bus
4
The data within the device is accessed by threads through memory hierarchy

Figure 2.1: Dataflow

```
#define N 10

// STEP 3: Multiple threads are created with
// unique ID. Parameter (a, b, c) must be
// variables located on device memory.
__global__ void add(int *a, int *b, int *c) {
        int tid = threadIdx.x; // Get Thread ID
        // STEP 4: Preform computation on data in device memory
        if (tid < N) {
                c[tid] = a[tid] + c[tid];
        }
}

int main() {
        // Host's variables: Vector A, B, and C
        int a[N], b[N], c[N];
        // Counterpart of host's variable on the device
        int *dev_a, *dev_b, *dev_c;

        // Initialize vector A and B
        for (int i = 0; i < N; i++) {
                a[i] = -i;
                b[i] = i * i;
        }

        // Allocate the memory on the GPU
        cudaMalloc((void**)&dev_a, N * sizeof(int));
        cudaMalloc((void**)&dev_b, N * sizeof(int));
        cudaMalloc((void**)&dev_c, N * sizeof(int));


        // STEP 1: Copy data from host to device
        cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemoryHostToDevice);
        cudaMemcpy(dev_b, b, N * sizeof(int), cudaMemoryHostToDevice);

        // STEP 2: Launch the kernel code from the host
        add <<<N, 1>>> (dev_a, dev_b, dev_c);

        // STEP 5: Move the results from device back to host' memory
        cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost);

        // Display the results
        for (int i = 0; i < N; i++) {
                printf("%d + %d = %d", a[i], b[i], c[i]);
        }

        // Free memory allocated from the GPU
        cudaFree(dev_a);
        cudaFree(dev_b);
        cudaFree(dev_c);
}
```

Clearly, a CUDA program is very similar to standard C program. The only different part is the kernel which is the function that must be run on the device (GPU) instead of CPU like the rest of the program. In this example, we use *__global__* qualifier before *add* function definition to indicate that it is a kernel function. To launch kernel code, we must use the following form:

```
        kernel<<<N, M>>>(parameter_1, parameter_2, ...);
```

where kernel is the name of kernel we specified through its definition. The parameters used by the kernel must be variables located inside GPU's memory since the GPU's cores cannot access host memory directly. After being launched, the kernel code is shared and run concurrently by processing cores inside GPU. In CUDA platform, the kernel code is not assigned directly to cores but through an execution unit called "*thread*". Particularly, when the kernel function is invoked, GPU create multiple threads to execute this kernel. Those threads are treated as a stream of processing jobs and effective assigned to real processing core. The abstraction of thread make the hardware specification invisible from the outside and thus significantly reduce the amount of work for programmers.
In CUDA platform, threads are grouped into blocks. Each block is identified by its block Index. Also, in each block, threads are identified by their local thread index. The

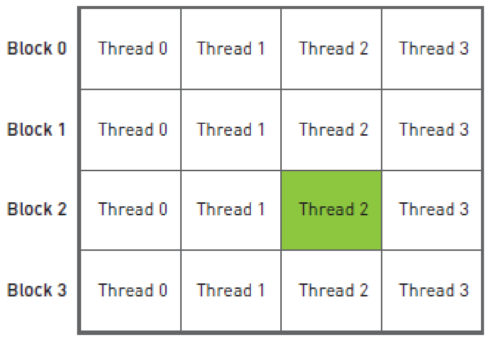


Figure 2.2: The arrangement of a collection of threads and blocks.

number of threads in each block and the number of blocks used for executing kernel code are specified through M and N of the kernel invocation, respectively. Inside the kernel code, the special variables threadIdx.x and blockIdx.x are used to identify thread Index and the index of its block, respectively. To identify the number of thread in a block , we could use blockDim.x variable. Figure 2.2 shows the arrangement of a collection of threads and blocks.
Generally, thread Index is only valid inside a block. To make the index globally valid, the following statement is widely used:

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

2.2 Practice

Write a CUDA program that calculate the sum of two matrices.

3 Exercise

Solve the matrix multiplication problem using CUDA platform.