EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 14: October 13

Lecturer: Vijay Garg *Scribe: Nishanth Shanmugham*

14.1 Topics

The topics covered in this lecture are:

- GPUs and CUDA Introduction
- CUDA Hello World
- Parallel Computations in CUDA
- Terminology
- CUDA Threads & Identification
- GitHub Links

14.2 GPUs and CUDA Introduction

Heterogeneous computing is programming using both a CPU and a GPU. The following terminology is commonly used to describe the CPU and GPU.

- **Host**: The CPU and its memory (host memory)
- **Device**: The GPU and its memory (device memory)

The general programming process is:

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute,caching data on chip for performance
3. Copy results from GPU memory to CPU memory

The contents of this lecture can be found in `Introduction_to_CUDA_C.pptx` on Canvas [Canvas]. This document aims to capture the essential information from class—not to repeat the contents in the slides.

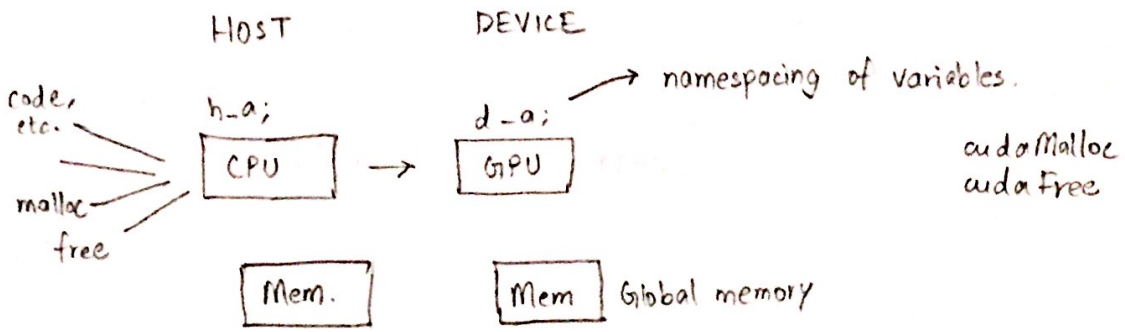

Figure 14.1: Host and Device

14.3 Cuda Hello World

```
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<1,1>>();
    printf(``hello, world\n'');
    return 0;
}
```

The above code is a simple *Hello World* in CUDA in C. First, we declare a function called `mykernel`. The function is then invoked in `main`.

```
mykernel<<N,M>>();
```

The first argument (N) in the angular brackets is the number of *blocks* and the second argument (M) is the number of *threads per block*.

The `__global__` indicates that the function should be run of the device; that is, `mykernel` will be executed on the GPU. In this example, the function does nothing. In future section, we write more useful programs.

14.4 Parallel Computations in CUDA

To run a CUDA function in parallel, we set the argument N (described above) to a value greater than 1. This runs the function in parallel on N blocks on the GPU. Making such a CUDA call can be considered the equivalent of OpenMP's `parallel for loop` construct.

14.4.1 Addition

The following code performs parallel addition on the GPU.

```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}


int main(void) {
    int a, b, c;             // host copies of a, b, c
    int *d_a, *d_b, *d_c;     // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Setup input values
    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Since the function runs on the device the variables used in the function should reside in device memory. Hence we use the `cudaMalloc` and `cudaFree` functions to allocate and release memory on the device. Moreover, after allocating the memory, the values should be copied from the host to the device using `cudaMemcpy`. Similarly, after the computation is complete and before releasing the device memory, the result must be copied to the host using `cudaMemcpy`. The direction that `cudaMemcpy` should use is specified using the last argument; in this case `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost`.

14.5 Terminology

When a GPU is started, a `grid` is started. Figure 13.2 displays the logical construction in a GPU. A grid is made of a set of `blocks`. A block consists of a set of `threads`.

Figure 14.2: Logical construction

14.5.1 Hardware

As show in Figure 13.1, the blocks of a grid are enumerated and distributed to multiprocessors. Shared memory s local to a block. Accessing shared memory is faster than going to global memory. A streaming multiprocessor is composed of multiple streaming processes.

14.6 CUDA Threads & Identification

A block can be split into parallel threads. However, there is a restriction on the maximum number of threads that can run in a single block. If such a situation arises, an option is to use multiple blocks.

Unlike blocks, threads can communicate and synchronize.

In addition, the following tips are useful:

1. If the work is dependent between parallel workers, use the same block. This provides the advantage of shared memory access and communication between threads in the block.
2. If the work is not dependent, then using multiple blocks is faster.

14.6.1 Identifying blocks and threads

The predefined variables `blockIdx`, `blockDim`, and `threadIdx` can be used to designate IDs to threads running in parallel. This allows programmers to assign tasks to each thread based on their ID.

For instance, the index that a particular thread should work on can be computed using:

```
int index = threadIdx.x + (blockIdx.x * blockDim.x);
```

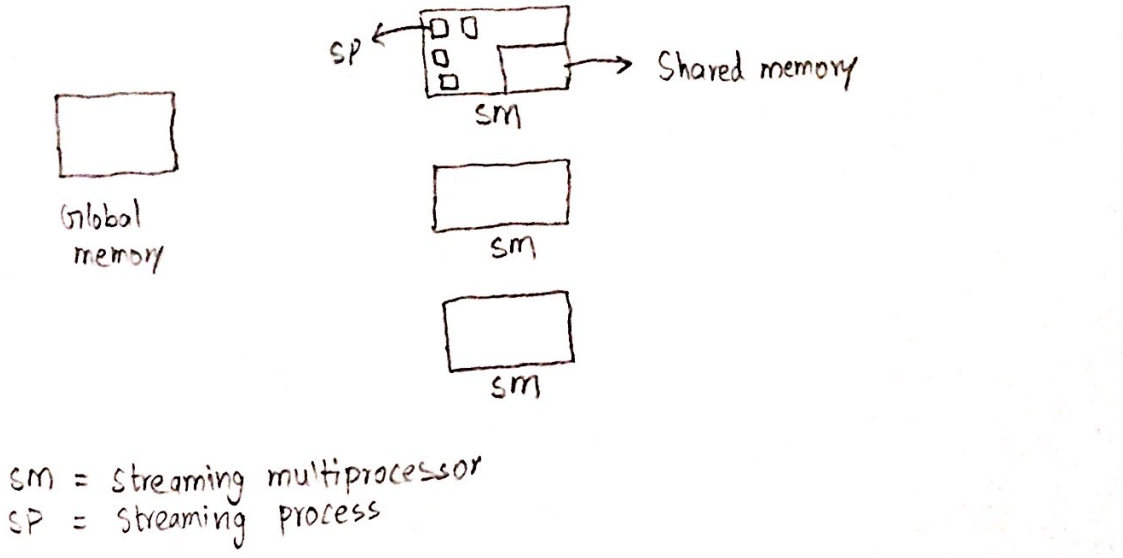

Figure 14.3: Hardware

In addition to the `.x` field, `.y` and `z` fields are also available to facilitate working with multi-dimensional data.

14.6.2 More Terms

- **__shared__**: Declares a variable/array as shared memory..
  - Data is shared between threads in a block.
  - Data is not visible to threads in other blocks.
- **__syncthreads()**: Used as a barrier to wait for threads to prevent data hazards. Similar to the Java construct Threadsjoin.

14.7 GitHub links and Conclusion

The files on GitHub have Cuda examples [GitHub]. See the directory named `cuda`. The code documented with comments. For additional explanation of `reduce.cu`, see the next lecture.

References

[GitHub] Multicore Computing source code, *https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore*

[Canvas] Introduction to CUDA, *https://utexas.instructure.com/courses/1174768/modules/items/8255450*

[Cuda] Cuda Toolkit Documentation*http://docs.nvidia.com/cuda/index.htmlaxzz4NfgY7FeW*