GPU 1

CSCI 4850/5850 High-Performance Computing

Spring 2018

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University
Learning Objectives

- You will get started how to use GPU CUDA language.
GPU

- GPU: Graphics Processing Unit
  - Designed for real-time graphics
  - Present in almost every PC
  - Increasing realism and complexity
What is GPU?

- It is a processor optimized for 2D/3D graphics, video, visual computing, and display.
- It is highly parallel, highly multithreaded multiprocessor optimized for visual computing.
- It provides real-time visual interaction with computed objects via graphics images, and video.
- It serves as both a programmable graphics processor and a scalable parallel computing platform.
- Heterogeneous Systems: combine a GPU with a CPU
GEFORCE GTX 1070 Ti

10: GAMING PERFECTED

Take on today’s most challenging, graphics-intensive games without missing a beat. The GeForce GTX 1070 Ti and GeForce GTX 1070 graphics cards deliver the incredible speed and power of NVIDIA Pascal™, the most advanced gaming GPU architecture ever created. This is the ultimate gaming platform. #GameReady.

$449.00

OUT OF STOCK

Limit 2 per customer
## GPU Specs

### The Power of GeForce GTX 1070 Family

<table>
<thead>
<tr>
<th>Spec</th>
<th>GeForce GTX 1070 Ti</th>
<th>GeForce GTX 1070</th>
<th>GeForce GTX 970</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Pascal</td>
<td>Pascal</td>
<td>Maxwell</td>
</tr>
<tr>
<td>Cores</td>
<td>2432</td>
<td>1920</td>
<td>1664</td>
</tr>
<tr>
<td>Performance</td>
<td>3x</td>
<td>3x</td>
<td>1x</td>
</tr>
<tr>
<td>Frame Buffer</td>
<td>8 GB GDDR5</td>
<td>8 GB GDDR5</td>
<td>4 GB GDDR5</td>
</tr>
<tr>
<td>Memory Type-Speed</td>
<td>8 Gbps</td>
<td>8 Gbps</td>
<td>6 Gbps</td>
</tr>
<tr>
<td>Boost Clock</td>
<td>Relative 1.4x</td>
<td>1.4x</td>
<td>1x</td>
</tr>
<tr>
<td>Actual</td>
<td>1683 MHz</td>
<td>1683 MHz</td>
<td>1178 MHz</td>
</tr>
</tbody>
</table>

**View Full Specs**

### Product Info

**GeForce GTX 1070 Ti**

- Supported Technologies:
  - SLI
  - CUDA
  - 3D Vision
  - PhysX
  - NVIDIA G-SYNC™
  - GameStream
  - ShadowWorks
  - DirectX 12
  - Virtual Reality
  - Ansel
  - NVIDIA WhisperMode
GPU and CPU

- Typically GPU and CPU coexist in a heterogeneous setting
- "Less" computationally intensive part runs on CPU (coarse-grained parallelism), and more intensive parts run on GPU (fine-grained parallelism)
- NVIDIA's GPU architecture is called CUDA (Compute Unified Device Architecture) architecture, accompanied by CUDA programming model, and CUDA C/C++ language
What is CUDA?

● CUDA Architecture
  ▪ Expose GPU computing for general purpose
  ▪ Retain performance

● CUDA C/C++
  ▪ Based on industry-standard C/C++
  ▪ Small set of extensions to enable heterogeneous programming
  ▪ Straightforward APIs to manage devices, memory etc.

● This session introduces CUDA C/C++
Heterogeneous Computing

- **Terminology:**
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)
```cpp
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__
void stencil_1d(
    int* in,
    int* out) {

    __shared__
    int temp[BLOCK_SIZE + 2 * RADIUS];

    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(
    int* x,
    int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int* in, *out;

    // host copies of a, b, c
    int* d_in, *d_out;

    // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in  = (int*) malloc(size);
    fill_ints(in, N + 2*RADIUS);
    out = (int*) malloc(size);
    fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void**)&d_in, size);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>>>(
        d_in + RADIUS,
        d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in);
    cudaFree(d_out);
    return 0;
}
```
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
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
Develop, Optimize and Deploy GPU-accelerated Apps

The NVIDIA® CUDA® Toolkit provides a development environment for creating high performance GPU-accelerated applications. With the CUDA Toolkit, you can develop, optimize and deploy your applications on GPU-accelerated embedded systems, desktop workstations, enterprise data centers, cloud-based platforms and HPC supercomputers. The toolkit includes GPU-accelerated libraries, debugging and optimization tools, a C/C++ compiler and a runtime library to deploy your application.

GPU-accelerated CUDA libraries enable drop-in acceleration across multiple domains such as linear algebra, image and video processing, deep learning and graph analytics. For developing custom algorithms, you can use available integrations with commonly used languages and numerical packages as well as well-published development APIs. Your CUDA applications can be deployed across all NVIDIA GPU families available on premise and on GPU instances in the cloud. Using built-in capabilities for distributing computations across multi-GPU configurations, scientists and researchers can develop applications that scale from single GPU workstations to cloud installations with thousands of GPUs.

To get started, browse through online getting started resources, optimization guides, illustrative examples and collaborate with the rapidly growing developer community.

Download Now

CUDA 9.2: What’s New...
#include <stdio.h>

int main() {
    int nDevices;

    cudaGetDeviceCount(&nDevices);
    for (int i = 0; i < nDevices; i++) {
        cudaDeviceProp prop;
        cudaGetDeviceProperties(&prop, i);
        printf("Device Number: %d\n", i);
        printf("  Device name: %s\n", prop.name);
        printf("  Memory Clock Rate (KHz): %d\n", prop.memoryClockRate);
        printf("  Memory Bus Width (bits): %d\n", prop.memoryBusWidth);
        printf("  Peak Memory Bandwidth (GB/s): %f\n\n",
                2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
    }
}

$ nvcc -o cuda_dev_info cuda_dev_info.cu
Hello World!

Standard C that runs on the host

- NVIDIA compiler (nvcc) can be used to compile programs with no device code

```
#include <iostream>
using namespace std;

__global__ void mykernel(void)
{
}

int main(void) {
    mykernel<<<1,1>>>();
    cout << "Hello CUDA World!" << endl;
    return 0;
}
```

Output:

```
$ nvcc hello.cu
$ ./a.out
Hello World!
```

Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    cout << "Hello CUDA World!" << endl;
    return 0;
}

- Two new syntactic elements...
```
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc`, `cl.exe`
Hello World! with Device Code

```c
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment

- That’s all that is required to execute a function on the GPU!
• But wait... GPU computing is about massive parallelism!

• We’ll start by adding two integers and build up to vector addition
Addition on the Device

● A simple kernel to add two integers

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

● As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host
Addition on the Device

- Note that we use pointers for the variables

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

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory

- We need to allocate memory on the GPU
Memory Management

- Host and device memory are separate entities
  - **Device** pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - **Host** pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code

- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`
Addition on the Device: `add()`

- Returning to our `add()` kernel

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

- Let’s take a look at `main()`...
Addition on the Device: `main()`

```c
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;
}
```
Addition on the Device: \texttt{main()}

// Copy inputs to device
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;
}
#include <iostream>
#include <cstdlib>

using namespace std;

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

int main(int argc, char *argv[]) {
    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;
    c = 0;

    // Copy inputs to device
    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);

    cout << "c=" << c << endl;

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
Moving to Parallel

- GPU computing is about massive parallelism.
  - So how do we run code in parallel on the device?

```plaintext
add<<< 1, 1 >>>();
```

```plaintext
add<<< N, 1 >>>();
```

- Instead of executing `add()` once, execute N times in parallel.
Vector Addition on the Device

- With \texttt{add()} running in parallel we can do vector addition

- Terminology: each parallel invocation of \texttt{add()} is referred to as a block
  - The set of blocks is referred to as a \textit{grid}
  - Each invocation can refer to its block index using \texttt{blockIdx.x}

```c
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using \texttt{blockIdx.x} to index into the array, each block handles a different index
Vector Addition on the Device

```c
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- On the device, each block can execute in parallel:

  Block 0  Block 1  Block 2  Block 3
  
Vector Addition on the Device: \texttt{add()} 

- Returning to our parallelized \texttt{add()} kernel 

\begin{verbatim}
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
\end{verbatim}

- Let's take a look at \texttt{main()}...
#define N 512

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 = N * sizeof(int);

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

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

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

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

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}