E6895 Advanced Big Data Analytics Lecture 3:

*Next Generation Hardware (I) -- GPU*

**Ching-Yung Lin**, Ph.D.
Adjunct Professor, Dept. of Electrical Engineering and Computer Science
IBM Chief Scientist, Graph Computing Research
CUDA: Compute Unified Device Architecture
2001: NVIDIA’s GeForce 3 series made probably the most breakthrough in GPU technology
— the computing industry’s first chip to implement Microsoft’s then-new Direct 8.0 standard;
— which required that the compliant hardware contain both programmable vertex and programmable pixel shading stages

Early 2000s: The release of GPUs that possessed programmable pipelines attracted many researchers to the possibility of using graphics hardware for more than simply OpenGL or DirectX-based rendering.

— The GPUs of the early 2000s were designed to produce a color for every pixel on the screen using programmable arithmetic units known as pixel shaders.

— The additional information could be input colors, texture coordinates, or other attributes
2006: GPU computing starts going for prime time
— Release of CUDA
— The CUDA Architecture included a unified shader pipeline, allowing each and every arithmetic logic unit (ALU) on the chip to be marshaled by a program intending to perform general-purpose computations.

Example of CUDA processing flow
1. Copy data from main mem to GPU mem
2. CPU instructs the process to GPU
3. GPU execute parallel in each core
4. Copy the result from GPU mem to main mem
Examples

- Medical Imaging
- Computational Fluid Dynamics
- Environmental Science
GT 750M:
- 2 * 192 CUDA cores
- max thread number: 2 * 2048
Announcing New Amazon EC2 GPU Instance Type

Posted On: Nov 4, 2013

We are excited to announce G2 instances, a new Amazon Elastic Compute Cloud (EC2) instance type designed for applications that require 3D graphics capabilities. The new instance is backed by a high-performance NVIDIA GPU, making it ideally suited for video creation services, 3D visualizations, streaming graphics-intensive applications, and other server-side workloads requiring massive parallel processing power. With this new instance type, customers can build high-performance DirectX, OpenGL, CUDA, and OpenCL applications and services without making expensive up-front capital investments.

Customers can launch G2 instances using the AWS console, Amazon EC2 command line interface, AWS SDKs and third party libraries. Customers can launch the new instances in the US East (N. Virginia), US West (N. California), US West (Oregon), and EU (Ireland). In addition to On-Demand Instances, customers can also purchase instances as Reserved and Spot Instances. To learn more about G2 instances, visit http://aws.amazon.com/ec2. To get started immediately, visit the AWS Marketplace for GPU machine images from NVIDIA and other Marketplace sellers.
GPU on iOS devices
PowerVR GPU has been used.
A4 => SGX 535 (1.6 GFLOPS)
A5 => SGX 543 MP2 (12.8 GLOPS)
A6 => SGX 543 MP3 (25.5 GFLOPS)
A7 => G6430 (quad core) (230.4 GFLOPS)
A8 => GX6450 (quad core) (332.8 GLOPS)
GPU in Apple A8 SoC

A8 — iPhone 6 and iPhone 6 Plus

GPU: PowerVR Quad-core GX6450

- 4 Unified Shading Cluster (USC)

- # of ALUs: 32 (FP32) or 64 (FP16) per USC

- GFLOPS: 166.4 (FP32)/ 332.8 (FP16) @ 650 MHz

Supports OpenCL 1.2

source: http://www.imgtec.com/powervr/series6xt.asp

manufactured by TSMC
GPU Programming in iPhone/iPad - Metal

Metal provides the lowest-overhead access to the GPU, enabling developers to maximize the graphics and compute potential of iOS 8 app.*

Metal could be used for:

- Graphic processing ➔ openGL
- General data-parallel processing ➔ open CL and CUDA

Fundamental Metal Concepts

- Low-overhead interface
- Memory and resource management
- Integrated support for both graphics and compute operations
- Precompiled shaders
GPU Programming in iPhone/iPad - Metal

Programming flow is similar to CUDA
- Copy data from CPU to GPU
- Computing in GPU
- Send data back from GPU to CPU

Example: kernel code in Metal, sigmoid function:

```
kernel void sigmoid(const device float *inVector [[ buffer(0) ]],
                    device float *outVector [[ buffer(1) ]],
                    uint id [[ thread_position_in_grid ]]) {
    // This calculates sigmoid for _one_ position (=id) in a vector per call on the GPU
    outVector[id] = 1.0 / (1.0 + exp(-inVector[id]));
}
```

source: http://memkite.com
CUDA supports most Windows, Linux, and Mac OS compilers

For Linux:
- Red Hat
- OpenSUSE
- Ubuntu
- Fedora
Hello World!!

```c
#include "../common/book.h"

int main( void ) {
    printf( "Hello, World!\n" );
    return 0;
}
```

Host: CPU and its memory
Device: GPU and its memory
A Kernel Call

```cpp
#include <iostream>

__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

nvcc handles compiling the function `kernel()`
it feeds `main()` to the host compiler
#include <iostream>
#include "book.h"

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

int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );

    add<<<1,1>>>( 2, 7, dev_c );

    HANDLE_ERROR( cudaMemcpy( &c,
                                dev_c,
                                sizeof(int),
                                cudaMemcpyDeviceToHost ) );

    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );

    return 0;
}
Figure 4.1 Summing two vectors

CPU Vector Sums
Traditional C way

```c
#include "../common/book.h"

#define N 10

void add(int *a, int *b, int *c) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;    // we have one CPU, so we increment by one
    }
}

int main(void) {
    int a[N], b[N], c[N];

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    add(a, b, c);

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

    return 0;
}
```
Executing on each of the two CPU cores

```c
void add( int *a, int *b, int *c )
{
    int tid = 0;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```

```c
void add( int *a, int *b, int *c )
{
    int tid = 1;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```
#include "../common/book.h"

#define N 10

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

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

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                                cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                                cudaMemcpyHostToDevice ) );
```c
add<<N,1>>( dev_a, dev_b, dev_c );

// copy the array 'c' back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );

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

// free the memory allocated on the GPU
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );

return 0;
```
### Blocks and Threads

<table>
<thead>
<tr>
<th>Block 0</th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1</td>
<td>Thread 0</td>
<td>Thread 1</td>
<td>Thread 2</td>
<td>Thread 3</td>
</tr>
<tr>
<td>Block 2</td>
<td>Thread 0</td>
<td>Thread 1</td>
<td><strong>Thread 2</strong></td>
<td>Thread 3</td>
</tr>
<tr>
<td>Block 3</td>
<td>Thread 0</td>
<td>Thread 1</td>
<td>Thread 2</td>
<td>Thread 3</td>
</tr>
</tbody>
</table>

```cpp
int tid = threadIdx.x + blockIdx.x * blockDim.x;
```
**Figure 5.2** A 2D hierarchy of blocks and threads that could be used to process a 48 x 32 pixel image using one thread per pixel.
```c
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x; // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```
GPU Blocks

**BLOCK 1**

```c
__global__ void add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

**BLOCK 2**

```c
__global__ void add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

**BLOCK 3**

```c
__global__ void add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

**BLOCK 4**

```c
__global__ void add( int *a, int *b, int *c ) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```
#include "../common/book.h"

#define N 10

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

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

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i * i;
    }
}
// copy the arrays 'a' and 'b' to the GPU
HANDLE_ERROR( cudaMemcpy( dev_a, 
a, 
N * sizeof(int),
cudaMemcpyHostToDevice ) );

HANDLE_ERROR( cudaMemcpy( dev_b, 
b, 
N * sizeof(int),
cudaMemcpyHostToDevice ) );

add<<<1,N>>>( dev_a, dev_b, dev_c );

// copy the array 'c' back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( c, 
dev_c, 
N * sizeof(int),
cudaMemcpyDeviceToHost ) );

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

// free the memory allocated on the GPU
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );

return 0;
CUDA on Mac OS X

CUDA Toolkit v7.0
Getting Started Mac OS X

1. Introduction

CUDA® is a parallel computing platform and programming model invented by NVIDIA. It enables harnessing the power of the graphics processing unit (GPU).

CUDA was developed with several design goals in mind:

- Provide a small set of extensions to standard programming languages, like C, that enable
  With CUDA C/C++, programmers can focus on the task of parallelization of the algorithms
- Support heterogeneous computation where applications use both the CPU and GPU. Serial
  portions are offloaded to the GPU. As such, CUDA can be incrementally applied to existin
  devices that have their own memory spaces. This configuration also allows simultaneous
  memory resources.

CUDA-capable GPUs have hundreds of cores that can collectively run thousands of computing th
register file and a shared memory. The on-chip shared memory allows parallel tasks running on
system memory bus.
This guide will show you how to install and check the correct operation of the CUDA developme

1.1. System Requirements

To use CUDA on your system, you need to have:

- a CUDA-capable GPU
- Mac OS X 10.9 or later
- the Clang compiler and toolchain installed using Xcode
- the NVIDIA CUDA Toolkit (available from the CUDA Download page)