E6895 Advanced Big Data Analytics Lecture 5:

*Parallel Computing — GPU and CUDA*

Ching-Yung Lin, Ph.D.
Adjunct Professor, Dept. of Electrical Engineering and Computer Science
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
GPU on a MacBook

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Video Card</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATA</td>
<td>Intel Iris Pro</td>
</tr>
<tr>
<td>Audio</td>
<td>NVIDIA GeForce GT 750M</td>
</tr>
<tr>
<td>Bluetooth</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td></td>
</tr>
<tr>
<td>Card Reader</td>
<td></td>
</tr>
<tr>
<td>Diagnostics</td>
<td></td>
</tr>
<tr>
<td>Disc Burning</td>
<td></td>
</tr>
<tr>
<td>Ethernet Cards</td>
<td></td>
</tr>
<tr>
<td>Fibre Channel</td>
<td></td>
</tr>
<tr>
<td>FireWire</td>
<td></td>
</tr>
<tr>
<td>Graphics/Displays</td>
<td>NVIDIA GeForce GT 750M</td>
</tr>
<tr>
<td>Hardware RAID</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>PCI Cards</td>
<td></td>
</tr>
<tr>
<td>Parallel SCSI</td>
<td></td>
</tr>
<tr>
<td>Power</td>
<td></td>
</tr>
<tr>
<td>Printers</td>
<td></td>
</tr>
<tr>
<td>SAS</td>
<td></td>
</tr>
<tr>
<td>SATA/SATA Express</td>
<td></td>
</tr>
<tr>
<td>SPI</td>
<td></td>
</tr>
<tr>
<td>Storage</td>
<td></td>
</tr>
<tr>
<td>Thunderbolt</td>
<td></td>
</tr>
<tr>
<td>USB</td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td></td>
</tr>
<tr>
<td>Firewall</td>
<td></td>
</tr>
<tr>
<td>Locations</td>
<td></td>
</tr>
</tbody>
</table>

NVIDIA GeForce GT 750M:

- Chipset Model: NVIDIA GeForce GT 750M
- Type: GPU
- Bus: PCIe
- PCIe Lane Width: x8
- VRAM (Total): 2048 MB
- Vendor: NVIDIA (0x10de)
- Device ID: 0x0fe9
- Revision ID: 0x00a2
- ROM Revision: 3776
- gMux Version: 4.0.8 [3.2.8]

GT 750M:
- 2 * 192 CUDA cores
- max thread number: 2 * 2048
CUDA DRIVERS FOR MAC ARCHIVE

CUDA Mac Driver
Latest Version: CUDA 418.163 driver for MAC
Release Date: 05/10/2019

Previous Releases:
CUDA 418.105 driver for MAC
Release Date: 02/27/2019

CUDA 410.130 driver for MAC
Release Date: 09/19/2018
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
Passing Parameters

```c
#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( cudaMemcpy( (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
```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

**CPU CORE 1**

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

**CPU CORE 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( cudaMemcpy( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMemcpy( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMemcpy( (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<<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>

```c
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;

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 existing 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 threads, a 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 development environment.

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)
Example: deviceQuery

Understand the hardware constraint via deviceQuery (in example code of CUDA toolkit)
Example: Matrix Addition on CPU

Problem: Sum two matrices with M by N size.

\[ C_{mxn} = A_{mxn} + B_{mxn} \]

In traditional C/C++ implementation:
- A, B are input matrix, N is the size of A and B.
- C is output matrix
- Matrix stored in array is row-major fashion

```c
void sumArraysOnHost(float *A, float *B, float *C, const int N) {
    for (int idx = 0; idx < N; idx++)
    {
    }
}
```
Example: Matrix Addition on GPU - 2D grid with 2D blocks

Problem: Sum two matrices with M by N size.

\[ C_{mxn} = A_{mxn} + B_{mxn} \]

CUDA C implementation:
- matA, matB are input matrix, nx is column size, and ny is row size
- matC is output matrix

```c
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int nx, int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx + ix;
    if (ix < nx && iy < ny)
}

int dimx = 32;
int dimy = 32;
dim3 block(dimx, dimy);
dim3 grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y);

iStart = seconds();
sumMatrixOnGPU2D<<<grid, block>>>(d_MatA, d_MatB, d_MatC, nx, ny);
CHECK(cudaDeviceSynchronize());
```
Example: Matrix Addition on GPU - 2D grid with 2D blocks

Data accessing in 2D grid with 2D blocks arrangement (one green block is one thread block)

```c
// grid 2D block 2D
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int nx, int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx + ix;
    if (ix < nx && iy < ny)
}
```

blockDim.x

blockDim.y

nx

ny

matrix coordinate: (ix,iy)
global linear memory index: idx = iy*nx + ix
Example: Matrix Addition on GPU - 1D grid with 1D blocks

Data accessing in 1D grid with 1D blocks arrangement (one green block is one thread block)

```c
// grid 1D block 1D
__global__ void sumMatrixOnGPU1D(float *MatA, float *MatB, float *MatC, int nx, int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    if (ix < nx )
        for (int iy = 0; iy < ny; iy++) {
            int idx = iy * nx + ix;
        }
}
```

```
blockDim.x

 Block (0)  Block (1)  Block (2)  Block (3)

     iy (ix, iy) threadIdx.x

ny

global linear memory index: idx = iy*nx + ix
```
Example: Matrix Addition on GPU - 2D grid with 1D blocks

Data accessing in 2D grid with 1D blocks arrangement (one green block is one thread block)

```c
__global__ void sumMatrixOnGPUMix(float *MatA, float *MatB, float *MatC, int nx, int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = blockIdx.y;
    unsigned int idx = iy * nx + ix;
    if (ix < nx && iy < ny)
}
```

blockDim.x

<table>
<thead>
<tr>
<th>1</th>
<th>Block (0,0)</th>
<th>Block (1,0)</th>
<th>Block (2,0)</th>
<th>Block (3,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Block (0,1)</td>
<td>Block (1,1)</td>
<td>Block (2,1)</td>
<td>Block (3,1)</td>
</tr>
<tr>
<td>ny</td>
<td>Block (0,ny)</td>
<td>Block (1,ny)</td>
<td>Block (2,ny)</td>
<td>Block (3,ny)</td>
</tr>
</tbody>
</table>

global linear memory index: idx = iy*nx + ix
Example: Matrix Transpose on CPU

Problem: Transpose one matrix with $M$ by $N$ to one matrix with $N$ by $M$.

\[ A_{mxn} = B_{nxm} \]

In traditional C/C++ implementation:

- `in` is input matrix, `nx` is column size, and `ny` is row size.
- `out` is output matrix
- Matrix stored in array is row-major fashion

```c
void transposeHost(float *out, float *in, const int nx, const int ny) {
    for (int iy = 0; iy < ny; ++iy) {
        for (int ix = 0; ix < nx; ++ix) {
            out[ix * ny + iy] = in[iy * nx + ix];
        }
    }
}
```
Example: Matrix Transpose on GPU

```c
// case 2 transpose kernel: read in rows and write in columns
__global__ void transposeNaiveRow(float *out, float *in, const int nx, const int ny) {
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
    if (ix < nx && iy < ny) {
        out[ix * ny + iy] = in[iy * nx + ix];
    }
}

// case 3 transpose kernel: read in columns and write in rows
__global__ void transposeNaiveCol(float *out, float *in, const int nx, const int ny) {
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
    if (ix < nx && iy < ny) {
        out[iy * nx + ix] = in[ix * ny + iy];
    }
}
```
Example: Matrix Transpose on GPU

```c
__global__ void transposeUnroll4Row(float *out, float *in, const int nx,
const int ny) {
    unsigned int ix = blockDim.x * blockIdx.x * 4 + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;

    unsigned int ti = iy * nx + ix; // access in rows
    unsigned int to = ix * ny + iy; // access in columns
    if (ix + 3 * blockDim.x < nx && iy < ny) {
        out[to] = in[ti];
        out[to + ny * blockDim.x] = in[ti + blockDim.x];
        out[to + ny * 2 * blockDim.x] = in[ti + 2 * blockDim.x];
        out[to + ny * 3 * blockDim.x] = in[ti + 3 * blockDim.x];
    }
}
```

```c
__global__ void transposeUnroll4Col(float *out, float *in, const int nx,
const int ny) {
    unsigned int ix = blockDim.x * blockIdx.x * 4 + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int ti = iy * nx + ix; // access in rows
    unsigned int to = ix * ny + iy; // access in columns
    if (ix + 3 * blockDim.x < nx && iy < ny) {
        out[ti] = in[to];
        out[ti + blockDim.x] = in[to + blockDim.x * ny];
        out[ti + 2 * blockDim.x] = in[to + 2 * blockDim.x * ny];
        out[ti + 3 * blockDim.x] = in[to + 3 * blockDim.x * ny];
    }
}
```
Example: Concurrent Processing

Concurrent handle **data transfer** and **computation**

For NVIDIA GT 650M (laptop GPU), there is one copy engine.

For NVIDIA Tesla K40 (high-end GPU), there are two copy engines.

The latency in data transfer could be hidden during computing.

To handle two tasks, which both are matrix multiplications.

Copy two inputs to GPU, copy one output from GPU.

No concurrent processing

Concurrent processing
In neural network, the most important operation is **inner-product**

\[ a = f(x^T w + b) \]

- \( x \) is a matrix that records the input which is fed to neural network
- \( w \) is a matrix that records the weights of network connection
- \( b \) is a matrix that records the bias of network connection
- \( f \) is an activation function that used to activate the neuron
- \( a \) is output

GPU is more suitable for such intensively regular operations.
Example, \( x^T w + b \)

**cuBLAS (GPU) vs. OpenBLAS (CPU)**

- GPU computation includes data transfer between host and device.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (GPU)</th>
<th>Time (CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU compute a (4096, 4096) matrix</td>
<td>0.819480 secs</td>
<td>1.527501 secs</td>
</tr>
</tbody>
</table>
Check all metrics and events for nvprof, it will also explain the meaning of options

    nvprof --query-metrics
    nvprof --query-events

Professional CUDA C Programming

    http://www.wrox.com/WileyCDA/WroxTitle/Professional-CUDA-C-
    Programming.productCd-1118739329,descCd-DOWNLOAD.html

    source code are available on the above website

GTC On-Demand:


Developer Zone:

    http://www.gputechconf.com/resource/developer-zone

NVIDIA Parallel Programming Blog:

    http://devblogs.nvidia.com/parallelforall

NVIDIA Developer Zone Forums:

    http://devtalk.nvidia.com
Metal — Programming GPU on iOS devices

Metal

Accelerating graphics and much more.

Metal provides near-direct access to the graphics processing unit (GPU), enabling you to maximize the graphics and compute potential of your apps on iOS, macOS, and tvOS. Building on an approachable, low-overhead architecture with precompiled GPU shaders, fine-grained resource control, and multithreading support, Metal further evolves support for GPU-driven command creation, simplifies working with the array of Metal-capable GPUs, and lets you tap into Pro power of Mac Pro and Pro Display XDR.
Xcode 11 includes everything you need to create amazing apps and to bring your apps to even more devices. Take advantage of SwiftUI, an all-new user interface framework with a declarative Swift syntax. Start bringing your iPad app to Mac with just a click. And with support for Swift packages, Xcode 11 lets you share code among all of your apps or use packages created by the community.
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
It integrates the support for both *graphics* and *compute* operations. Three command encoder:

- Graphics Rendering: Render Command Encoder
- Data-Parallel Compute Processing: Compute Command Encoder
- Transfer Data between Resource: Blitting Command Encoder

Multi-threading in encoding command is supported

Typical flow in compute command encoder:

1. Prepare data
2. Put your function into pipeline
3. Command encoder
4. Put command into command buffer
5. Commit it to command queue
6. Execute the command
7. Get result
Sobel operator for detecting edges in Images

\[
\begin{array}{ccc}
-1 & 0 & +1 \\
-2 & 0 & +2 \\
-1 & 0 & +1 \\
\end{array}
\quad + \quad
\begin{array}{ccc}
+1 & +2 & +1 \\
0 & 0 & 0 \\
-1 & -2 & -1 \\
\end{array}
\]
Metal Programming, Example in Sobel operator

Metal codes:

```metal
kernel void sobelOperator(const device float *inVector [[ buffer(0) ]],
                          const device int *inParams [[buffer(2)]],
                          device float *outVector [[ buffer(1) ]],
                          uint2 id [[ thread_position_in_grid ]]) {

    int row = inParams[0];
    int col = inParams[1];

    int row_index = id[0];
    int col_index = id[1];

    float3x3 horMask;
    horMask[0][0] = -1.0; horMask[0][1] = -2.0; horMask[0][2] = -1.0;
    horMask[1][0] = 0.0; horMask[1][1] = 0.0; horMask[1][2] = 0.0;
    horMask[2][0] = 1.0; horMask[2][1] = 2.0; horMask[2][2] = 1.0;

    float3x3 verMask;
    verMask[0][0] = -1.0; verMask[0][1] = 0.0; verMask[0][2] = 1.0;
    verMask[1][0] = -2.0; verMask[1][1] = 0.0; verMask[1][2] = 2.0;
    verMask[2][0] = -1.0; verMask[2][1] = 0.0; verMask[2][2] = 1.0;

    float horTemp = 0.0;
    float verTemp = 0.0;

    if (row_index >= 1 && row_index < row - 1 && col_index >= 1 && col_index < col - 1) {
        for (int j = -1; j <= 1; ++j) {
            for (int i = -1; i <= 1; ++i) {
                float pixelTemp = inVector[(col_index + j) * row + (row_index + i)];
                horTemp += (pixelTemp * horMask[j + 1][i + 1]);
                verTemp += (pixelTemp * verMask[j + 1][i + 1]);
            }
        }
        float grad = sqrt(horTemp * horTemp + verTemp * verTemp);
        outVector[col_index * row + row_index] = grad;
    } else {
        outVector[col_index * row + row_index] = 0.0;
    }
}
```
Metal Programming, Example in Sobel operator

CPU side: Swift language

```swift
var horMask: [[Float]] = [[-1.0, -2.0, -1.0],
                         [0.0, 0.0, 0.0],
                         [1.0, 2.0, 1.0]]

var verMask: [[Float]] = [[-1.0, 0.0, 1.0],
                         [-2.0, 0.0, 2.0],
                         [-1.0, 0.0, 1.0]]

start = mach_absolute_time()
for j in 0..<imageHeight {
    for i in 0..<imageWidth {
        if (j >= 1 && j < imageHeight - 1 && i >= 1 && i < imageWidth - 1) {
            var tempGradX: Float = 0.0
            var tempGradY: Float = 0.0
            for m in -1..<2 {
                for n in -1..<2 {
                    let pixelTemp: Float = inMemBuf[(j + m) * imageWidth + (i + n)]
                    tempGradX += (pixelTemp * horMask[m + 1][n + 1])
                    tempGradY += (pixelTemp * verMask[m + 1][n + 1])
                }
            }
            let grad: Double = sqrt(Double(tempGradX * tempGradX + tempGradY * tempGradY))
            h_outMemBuf[j*imageWidth + i] = Float(grad);
        } else {
            h_outMemBuf[j*imageWidth + i] = 0.0;
        }
    }
}
```
CPU/GPU Performance on iPhone 6 and iPad Pro:
Image size: 4096x3072 (image size captured by 12 MegaPixel Camera)

**iPhone 6**

GPU: runtime : 0.384229375 seconds  
CPU: runtime : 75.0961254166667 seconds  
Speedup Ratio: 195.446080656032

**iPad Pro**

GPU: runtime : 0.02371125 seconds  
CPU: runtime : 39.2896004166667 seconds  
Speedup Ratio: 1657.00249529935

Thanks for colleague, Dr. Larry Lai in helping the app UI.
GPU Programming with Python and CUDA
General Problem

- Many problem scenario parameters to address: different types, array dimensions, etc.
- Many possible hardware scenarios: number of threads, blocks, compute capability, etc.
- Python reduces (but *does not* eliminate) the need to think about computer architecture and hardware. Can it do the same for GPUs?
Solution
Runtime Code Generation

- With PyCUDA, code does not need to be fixed at compile time.
- Kernels may be constructed and tuned as Python strings before being launched.
- Classes for facilitating construction of certain types of kernels included.

Edit -> Run
GPU Source Module -> Run on GPU

Cache?

yes
no

GPU Compiler -> GPU Binary
Upload to GPU

PyCUDA
Unlike MATLAB, Python contains no native vector data type.

`numpy.ndarray`: can be used to define vectors, matrices, tensors, etc.

Binds multidimensional data with information about `dtype`, `shape`, and `strides`.

Supports operation broadcasting, e.g., `A+B`, `sin(A)`, `A**2`

Serves as basis for other scientific computing packages: scipy, matplotlib, etc.
GPUArray - Multidimensional Arrays in GPU Memory

- `pycuda.gpuarray.GPUArray` - ndarray-like class for managing GPU memory.
- Array info resides in PC memory, data in GPU memory.
- Similar attributes to `ndarray`: `dtype`, `shape`, `strides`
- Compatible with `ndarray`:

```python
import pycuda.gpuarray as gpuarray
x_gpu = gpuarray.to_gpu(numpy.random.rand(3))
y = x_gpu.get()
```

- `print x_gpu` works automatically.
- Implicit generation of kernels for vectorized (elementwise) operations, e.g., `x_gpu + y_gpu`. 
import atexit
import numpy as np
import pycuda.driver as drv
import pycuda.gpucarray as gpuarray

drv.init()
dev = drv.Device(0)  # initialize GPU 0
ctx = dev.make_context()
atexit.register(ctx.pop)  # clean up on exit

x = np.random.rand(2, 3).astype(np.double)
x_gpu = gpusarray.to_gpu(x)
Single-Pass Expressions

- Implicitly generated kernels are cached to improve performance. However..
- Elementwise expressions involving `GPUArray` instances (e.g., $x+y*z$) compile/launch new kernels for each intermediate step.
- To improve efficiency, PyCUDA enables construction of complex single-pass elementwise expressions computed using a single kernel.
- Classes also provided that facilitate construction of other types of kernels:
  - Reductions (e.g., sum, product, dot product)
  - Scans (e.g., cumulative sum)
Elementwise Operations

```python
import numpy as np
import pycuda.autoinit
import pycuda.gpusarray as gpuarray
from pycuda.elementwise import ElementwiseKernel
from pycuda.cumath import rand

x_gpu = rand(10, np.double); y_gpu = rand(10, np.double)
z_gpu = gpuarray.empty_like(x_gpu)
func = ElementwiseKernel("double x*, double y*, double z*",
                         "z[i] = 2*x[i]+3*y[i]"")
func(x_gpu, y_gpu, z_gpu)
print 'Success:', np.allclose(2*x_gpu.get()+3*y_gpu.get(),
                              z_gpu.get())
```
Reductions

```python
import numpy as np
import pycuda.autoinit
import pycuda.gpusarray as gpuarray
from pycuda.reduction import ReductionKernel
from pycuda.curandom import rand

x_gpu = rand(10, np.double); y_gpu = rand(10, np.double)
func = ReductionKernel(dtype_out=np.double,
    neutral="0",
    reduce_expr="a+b",
    map_expr="x[i]*y[i]",
    arguments="double *x, double *y")
result = func(x_gpu, y_gpu).get()
print 'Success: ', np.allclose(np.dot(x_gpu.get(), y_gpu.get()), result)
```
```python
import numpy as np
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
from pycuda.scan import InclusiveScanKernel
from pycuda.curandom import rand

x_gpu = rand(10, np.double); x = x_gpu.get()
func = InclusiveScanKernel(np.double, "a+b")
result = func(x_gpu).get()
print 'Success:', np.allclose(np.cumsum(x), result)
```
Creating Your Own Kernels

```python
import numpy as np
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
from pycuda.compiler import SourceModule
from pycuda.curandom import rand

x_gpu = rand(10, np.double); x = x_gpu.get()
mod = SourceModule(""
__global__ void func(double *x) {
    int idx = threadIdx.x;
    x[idx] *= 3;
}
""
)
func = mod.get_function('func')
func(x_gpu, block=(10, 1, 1))
print 'Success: ', np.allclose(3*x, x_gpu.get())
```
Using GPU-based Libraries

- Optimizing common algorithms for GPUs can be nontrivial - why reinvent the wheel?
- Increasing number of mathematical libraries available for GPUs: linear systems (CUBLAS, CUSOLVER), signal processing (CUFFT, CULA), sparse data (CUSPARSE) etc.
- Most of these libraries only have C/C++ interfaces, however.
- Can we use them from Python?
- Solution: CUDA SciKit
  (http://scikit-cuda.readthedocs.org)
- Provides both low level (C-like) and high level (numpy-like) interfaces to libraries.
CUDA SciKit Example

```python
import pycuda.gpuarray as gpuarray
import pycuda.autograd
import numpy as np
import scikits.cuda.linalg as linalg

linalg.init()
a = np.random.randn(9, 6) + 1j*np.random.randn(9, 6)
a = np.asarray(a, np.complex64)
a_gpu = gpuarray.to_gpu(a)
u_gpu, s_gpu, vh_gpu = linalg.svd(a_gpu, 'S', 'S')
print 'Success:', np.allclose(a, np.dot(u_gpu.get(),
    np.dot(np.diag(s_gpu.get()), vh_gpu.get())), 1e-4)
```
PyCUDA Resources

- http://mathema.tician.de/software/pycuda
- http://lists.tiker.net/listinfo/pycuda
- http://wiki.tiker.net/PyCuda
- http://scikit-cuda.readthedocs.org