E6893 Big Data Analytics Lecture 13:

*GPU Introduction and Mobile Cognition*

Ching-Yung Lin, Ph.D.
Adjunct Professor, Dept. of Electrical Engineering and Computer Science
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.
CUDA:
Compute Unified Device Architecture

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.
CUDA Compiler

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

```
#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
", 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;
    }
}
```
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.
**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;
```
```c
#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 ) );
}```
add$$\ll N,1$$\rr( 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;
```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
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

BLOCK 2
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

BLOCK 3
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

BLOCK 4
__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;
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
// 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) 
}

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());
```
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`: Block width
- `blockDim.y`: Block height
- `threadIdx.x`: Thread index in x direction
- `threadIdx.y`: Thread index in y direction

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`: The number of thread blocks along the x-axis.
- `threadIdx.x`: The thread index along the x-axis.
- `ny`: The number of blocks along the y-axis.
- `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 sumMatrixOnGPU(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  nx
1

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

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

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
// case 4 transpose kernel: read in rows and write in columns + unroll 4 blocks
__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];
    }
}

// case 5 transpose kernel: read in columns and write in rows + unroll 4 blocks
__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 * ny] = in[to + blockDim.x * ny];
        out[ti + 2 * blockDim.x * ny] = in[to + 2 * blockDim.x * ny];
        out[ti + 3 * blockDim.x * ny] = 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
Reference

Professional CUDA C Programming
source code are available on the above website
CUDA on Mac OS X

NVIDIA CUDA Getting Started Guide for 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 programmers to write parallel code.
- 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 access to memory resources.

CUDA-capable GPUs have hundreds of cores that can collectively run thousands of computing threads in parallel. The on-chip shared memory allows parallel tasks running on the 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)
GPU Architecture

built by several streaming multiprocessors (SMs)

In each SM:
- CUDA cores
- Shared Memory/L1 Cache
- Register File
- Load/Store Units
- Special Function Units
- Warp Scheduler

In each device:
- L2 Cache
**Example: deviceQuery**

Understand the hardware constraint via deviceQuery (in example code of CUDA toolkit)

```
Device 0: "GeForce GT 650M"
CUDA Driver Version / Runtime Version: 7.0 / 7.0
CUDA Capability Major/Minor version number: 3.0
Total amount of global memory: 1024 MBytes (10734144 bytes)
(2) Multiprocessors, (192) CUDA Cores/MP:
GPU Max Clock rate: 900 MHz (0.90 GHz)
Memory Clock rate: 2508 Mhz
Memory Bus Width: 128-bit
L2 Cache Size: 262144 bytes
Maximum Texture Dimension Size (x,y,z): 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size: 32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block: 1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch: 2147483647 bytes
Texture alignment:
Concurrent copy and kernel execution: Yes with 1 copy engine(s)
Run time limit on kernels: Yes
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Disabled
Device supports Unified Addressing (UVA): Yes
Device PCI Domain ID / Bus ID / location ID: 0 / 1 / 0
```
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:

\[ S(t) = \frac{1}{1 + e^{-t}} \]

```c
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](http://memkite.com)
Metal Programming Model

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
- Prepare data
- Put your function into pipeline
- Command encoder
- Put command into command buffer
- Commit it to command queue
- Execute the command
- Get result back
Metal Programming, Kernel Function

Compute command

- Two parameters, threadsPerGroup and numThreadgroups, determines number of threads. \( \Rightarrow \) equivalent to grid and thread block in CUDA. They are all 3-D variable.
- The total of all threadgroup memory allocations must not exceed 16 KB.

- Kernel function: sigmoid function

```c
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]));
}
```
Reference

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


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/parallelforkall

NVIDIA Developer Zone Forums:

http://devtalk.nvidia.com
GPU on iPhone/iPad

Characteristics of different GPUs on iDevice (iPhone 6S, iPad Air 2, iPad Pro)

<table>
<thead>
<tr>
<th></th>
<th>iPhone 6S</th>
<th>iPhone Air 2</th>
<th>iPad Pro</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2x Twister @ 1.85 GHz</td>
<td>3x Typhone @ 1.5 GHz</td>
<td>2x Twister @ 2.26 GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>PVR GT7600</td>
<td>PVR GXA6850</td>
<td>PVR 12 Cluster Series 7</td>
</tr>
<tr>
<td>RAM</td>
<td>2GB LDDR4</td>
<td>2GB LDDR3</td>
<td>4GB LDDR4</td>
</tr>
<tr>
<td>Memory bus width</td>
<td>64-bit</td>
<td>128-bit</td>
<td>128-bit</td>
</tr>
<tr>
<td>Max # of threads per group</td>
<td>512</td>
<td>512</td>
<td>512</td>
</tr>
</tbody>
</table>

iDevice’s GPU is embedded in SoC → CPU and GPU share the same physical memory.
No memory transfers between host and device memory.
Note: one memory buffer can not exceed **256 MB.**
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
Metal Programming, Example in Sobel operator

Sobel operator for detecting edges in Images

\[ \begin{array}{ccc}
-1 & 0 & +1 \\
-2 & 0 & +2 \\
-1 & 0 & +1 \\
\end{array} \]

\[ \begin{array}{ccc}
+1 & +2 & +1 \\
0 & 0 & 0 \\
-1 & -2 & -1 \\
\end{array} \]

\[ \text{G}_x \quad \text{G}_y \]
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.

GPU compute a (4096,4096) matrix, spent 0.819480 secs
CPU compute a (4096,4096) matrix, spent 1.527501 secs
Using GPU on Mobile Phone (iPhone)
GPU on iOS devices
Early iPhone models and their GPUs

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)

FLOPS = sockets \times \frac{cores}{socket} \times \frac{clock}{cycle} \times \frac{FLOPs}{cycle}

Most microprocessors today can carry out 4 FLOPs per clock cycle;\(^1\) thus a single-core 2.5 GHz processor has a theoretical performance of 10 billion FLOPS = 10 GFLOPS.
GPU in Apple A11 SoC

A11 Bionic — iPhone 8 and iPhone X

Apple designed a three-core GPU: GPU Family 4

=> 30% faster than A10 or the same performance at half the power consumption.

source: http://www.imgtec.com/powervr/series6xt.asp
**GPU Programming in iPhone/iPad - Metal / Metal 2**

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

Metal could be used for:

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

Metal

iOS GPU Language

Project Name
Programming Model of Apple GPU

• Programming by Metal language
  • Most of syntaxes are compatible with C++14

• A unified programming language interface for **graphics** and **data-parallel computation** workloads.

• Single Instruction Multiple Threads (SIMT) programming fashion
  • Every thread performs **simple** and **identical** instruction.
Execution Model of Apple GPU

- It integrates the support for both **graphics** and **compute** operations.
  - Three command encoder:
    - Render Command Encoder: Graphics Rendering
    - **Compute Command Encoder**: Data-Parallel Compute Processing
    - Blitting Command Encoder: Transfer Data between Resource
  - Multi-threading in encoding command is supported

*Put identical type of commands together as possible.*
Diagram of iOS GPU Execution
Metal Programming, Kernal Function

• Compute command:
  • Two parameters, `threadsPerGroup` and `numThreadgroups`, determines number of threads. (They are all 3-D variable.)
  • The total of all threadgroup memory allocations must not exceed 16 KB (shared memory within one threadgroup.)
    - Shared memory used to sync data across different threads.

• Four attribute qualifiers are used to access data in kernel function:
  • `thread_position_in_grid`, `threadgroup_position_in_grid`
  • `thread_position_in_threadgroup`, `thread_index_in_threadgroup`

Max number of threads in a group is 512 (device-dependent).

For a 48x48 image
```
threadGroupCount = MTLSIZEMAKE(16, 16, 1)
threadGroups = MTLSIZEMAKE(3, 3, 1)
```
Sigmoid Function — Speedup

Syntax might be changed in swift3.

```swift
// pin CPU memory location to GPU; note that the memory should be aligned with 16KB
let inVectorBufferNoCopy = device.newBufferWithBytesNoCopy(inMemory, length: Int(size), options: [], deallocator: nil)
computeCommandEncoder.setBuffer(inVectorBufferNoCopy, offset: 0, atIndex: 0) // index is very important, you will access the data in kernel with that index.

let outVectorBufferNoCopy = device.newBufferWithBytesNoCopy(outMemory, length: Int(size), options: [], deallocator: nil)
computeCommandEncoder.setBuffer(outVectorBufferNoCopy, offset: 0, atIndex: 1)

let threadsPerGroup = MTLSize(width:maxThreadExeWidth!, height:1, depth:1)
let numThreadgroups = MTLSize(width:(Int(maxCount)+maxThreadExeWidth!-1)/maxThreadExeWidth!, height:1, depth:1)

Perform sigmoid function on $2^{24}$ data points on iPad Pro:

**SpeedGPU:** runtime : 0.01544429166666667 seconds
**CPU:** runtime : 39.5845222083333 seconds
**Speedup Ratio:** 2563.05197173713
Summary of Acceleration of Neural network on Mobile Devices

- Porting Deep Convolution Neural Network on iOS Device with near real-time computation
  - Reduce algorithmic complexity with ignorable performance degradation.
  - Utilize computational hardware to achieve better performance.

<table>
<thead>
<tr>
<th></th>
<th>iPhone 7</th>
<th>iPhone 7+</th>
<th>iPhone 6s</th>
<th>iPad Pro 12.9&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex Net</td>
<td>70 ms</td>
<td>70 ms</td>
<td>130 ms</td>
<td>69 ms</td>
</tr>
<tr>
<td>p-Alex Net</td>
<td>N/A</td>
<td>35 ms</td>
<td>N/A</td>
<td>28 ms</td>
</tr>
<tr>
<td>GoogLeNet</td>
<td>130 ms</td>
<td>128 ms</td>
<td>195 ms</td>
<td>110 ms</td>
</tr>
<tr>
<td>p-GoogLeNet</td>
<td>N/A</td>
<td>80 ms</td>
<td>N/A</td>
<td>70 ms</td>
</tr>
<tr>
<td>VGG16 Net</td>
<td>880 ms</td>
<td>883 ms</td>
<td>1450 ms</td>
<td>725 ms</td>
</tr>
</tbody>
</table>
Challenges
How to do Machine Reasoning?

*Five Layers of Understanding.*

**Deterministic Classification + Learning Inference**

- **Cognition Layer**
- **Semantics Layer**
- **Concept Layer**
- **Feature Layer**
- **Sensor Layer**

HR records, Travel records, Badge/Location records, Phone records, Mobile records

Transmitted images, speech content, video content

TTY: observations, hidden states
Multi-Scale Deep Convolutional Neural Network for Fast Object Detection
Where are Pedestrians?  — Deep Learning + Graph Contextual Analysis
Demo: Detecting Cars and Pedestrians in complex scenario
Comparison to the State-of-the-Art on KITTI benchmark test set

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>Cars</th>
<th></th>
<th>Pedestrians</th>
<th></th>
<th>Cyclists</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Easy</td>
<td>Mod</td>
<td>Hard</td>
<td>Easy</td>
<td>Mod</td>
<td>Hard</td>
</tr>
<tr>
<td>LSVM-MDPM-sv [35]</td>
<td>10s</td>
<td>68.02</td>
<td>56.48</td>
<td>44.18</td>
<td>47.74</td>
<td>39.36</td>
<td>35.95</td>
</tr>
<tr>
<td>DPM-VOC-VP [36]</td>
<td>8s</td>
<td>74.95</td>
<td>64.71</td>
<td>48.76</td>
<td>59.48</td>
<td>44.86</td>
<td>40.37</td>
</tr>
<tr>
<td>SubCat [16]</td>
<td>0.7s</td>
<td>84.14</td>
<td>75.46</td>
<td>59.71</td>
<td>54.67</td>
<td>42.34</td>
<td>37.95</td>
</tr>
<tr>
<td>3DVP [37]</td>
<td>40s</td>
<td>87.46</td>
<td>75.77</td>
<td>65.38</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AOG [38]</td>
<td>3s</td>
<td>84.80</td>
<td>75.94</td>
<td>60.70</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Faster-RCNN [4]</td>
<td>2s</td>
<td>86.71</td>
<td>81.84</td>
<td>71.12</td>
<td>78.86</td>
<td>65.90</td>
<td>61.18</td>
</tr>
<tr>
<td>CompACT-Deep [15]</td>
<td>1s</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>70.69</td>
<td>58.74</td>
<td>52.71</td>
</tr>
<tr>
<td>DeepParts [39]</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>70.49</td>
<td>58.67</td>
<td>52.78</td>
</tr>
<tr>
<td>FilteredICF [40]</td>
<td>2s</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>67.65</td>
<td>56.75</td>
<td>51.12</td>
</tr>
<tr>
<td>pAUCEnsT [41]</td>
<td>60s</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>65.26</td>
<td>54.49</td>
<td>48.60</td>
</tr>
<tr>
<td>Regionlets [20]</td>
<td>1s</td>
<td>84.75</td>
<td>76.45</td>
<td>59.70</td>
<td>73.14</td>
<td>61.15</td>
<td>55.21</td>
</tr>
<tr>
<td>3DOP [5]</td>
<td>3s</td>
<td>93.04</td>
<td>88.64</td>
<td>79.10</td>
<td>81.78</td>
<td>67.47</td>
<td>64.70</td>
</tr>
<tr>
<td>Ours</td>
<td>0.4s</td>
<td>89.62</td>
<td>87.05</td>
<td>70.76</td>
<td>82.02</td>
<td>71.47</td>
<td>66.02</td>
</tr>
</tbody>
</table>

Single CPU core (2.40GHz) of an Intel Xeon E5-2630 server with 64GB of RAM. An NVIDIA Titan GPU was used for CNN computations.
Demo: Multi-Scale Deep Convolutional Neural Network for Fast Object Detection
Mobile Cognition

- Created novel graph computing and deep learning framework on iOS devices and NAOqi robots including:
  - generic object recognition, event recognition, face recognition, visual sentiment recognition, and document recognition
  - graph database

Novel Deep Learning works that Speed Up image computation utilizing the GPUs on iOS devices: 195x or 1657x faster

<table>
<thead>
<tr>
<th>Classification rate</th>
<th>iPad Pro</th>
<th>iPhone 6s</th>
</tr>
</thead>
<tbody>
<tr>
<td>(on ~1000 classes)</td>
<td>~13 frames/sec</td>
<td>~7 frames/sec</td>
</tr>
</tbody>
</table>
Example - Generic Object Recognition running natively on mobile device (iPhone)
Demo - Document Detection (Warning!! Potential Sensitive Info Leakage!)
EventNet

— Generic Event Recognition on Videos
Demo — Video event recognition and search example

**Ontology**
A hierarchical structure organizing all events and concepts based on ontology defined by WikiHow.

**Search**
Given a text query, return the matched items in EventNet ontology.

**Upload**
Upload a video, predict its events and concepts, visualize the results on EventNet ontology.
Event Detection Baseline

Training Videos
- Attempting board trick
- Feeding an animal
- Landing a fish

Feature Extractions
- **Low-level feature**
  - SIFT (Visual)
  - STIP (Motion)
  - MFCC (Audio)
- **Mid-level Concept**

Classifiers
- Deep Learning
- Decision Tree
- SVM

Fusion
- Late Fusion
- Early Fusion

Output
Mid-level Feature Representation

- Decompose an event into concepts
Events Classification Framework

- **Event Classifier**
  - Pair-Activity Event Classifier
    - Embrace Classifier
    - PeopleMeet Classifier
    - PeopleSplitUp Classifier
    - PersonRuns Classifier

- **Feature Extracting**

- **Detected**
  - Embrace
  - PeopleMeet
  - PeopleSplitUp
  - PersonRuns

- **Event Identifying**
  - Backwards Search
  - Forwards Search

- **Event Merging**

- **Post-processing**

- **Key frames**

- **Preliminary Events**
Automatic Video Event Tagging

4,490 trained event models and a library of 95K videos

Event detected: cut_a_watermelon

Reasoned Event Concepts

Timestamp: 00:00:00
Concepts detected:
knife
watermelon
melon

Timestamp: 00:00:10
Concepts detected:
knife
watermelon
melon

Timestamp: 00:00:20
Concepts detected:
watermelon
melon

Timestamp: 00:00:30
Concepts detected:
knife
watermelon
melon

Timestamp: 00:00:40
Concepts detected:
knife
melon
watermelon

Timestamp: 00:00:50
Concepts detected:
food
knife
fruit

Timestamp: 00:01:00
Concepts detected:
watermelon
melon

Timestamp: 00:01:10
Concepts detected:
food
knife
home

Timestamp: 00:01:20
Concepts detected:
knife
watermelon
melon

Timestamp: 00:01:30
Concepts detected:
ingredient
knife
home

THanks for watching
Demo: EventNet

Large Video Event Ontology Browsing, Search and Tagging (EventNet Demo)
Examples of our Previous work on Abnormal Video Event Analysis

Event: Abnormal Behavior
(Surveillance Video)


Event: Making a bomb
(Consumer Video)

TRECVID Multimedia Event Detection (MED) Evaluation 2010-2016
Detection and Tracking of Head, Shoulder, and Body

- Fusion of Head-shoulder and Body detection
- Adjust the detector searching scales
Detection Results
Complex Scenario — how to predict people action?
Since 2009, U.S. Justice Department lawyers have pursued at least 19 cases of corporate espionage, including GM, Ford, Motorola, DuPont, ….

“Impacted economic and jobs”
– WSJ Feb 21, 2013

危険予測
Reasoning and Predictive Pipeline is composed of hundreds of cognitive analyzers
Game Theory may help decision making in complex scenario

- Forecasting what will happen based on our and others potential actions.
Acceleration of Neural network on Mobile Devices
Outline

- Background

- Full Network Acceleration and Compression
  - Kernel Importance Measurement
  - Algorithmic Performance Evaluation
  - Computation Complexity Assessment

- Acceleration on iOS Mobile Devices
  - Metal API for GPU Programming
  - Computation Speed Evaluation
Methods for Running CNNs on Mobile Devices

Sending CNN jobs to cloud

Acceleration CNN on Local Device

Apple A9X SoC, 12-core GPU

• How to trade off between algorithmic complexity and performance?
• How to utilize hardware effectively
## Problems for Running CNNs on Mobile Devices

<table>
<thead>
<tr>
<th>Model Size</th>
<th>Weights</th>
<th>Mult.s</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AlexNet</strong></td>
<td>243MB</td>
<td>61M</td>
</tr>
<tr>
<td><strong>VGG-S</strong></td>
<td>393MB</td>
<td>103M</td>
</tr>
<tr>
<td><strong>VGG-16</strong></td>
<td>552MB</td>
<td>138M</td>
</tr>
<tr>
<td><strong>GoogLeNet</strong></td>
<td>51MB</td>
<td>6.9M</td>
</tr>
</tbody>
</table>

Statistics of some popular CNNs

Reference:
Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications
Computational Resource on iPhone and iPad

<table>
<thead>
<tr>
<th></th>
<th>iPhone 6S (Plus)</th>
<th>iPad Air 2</th>
<th>iPad Pro (12.9/9.7)</th>
<th>iPhone 7 (Plus)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SoC</strong></td>
<td>A9</td>
<td>A8X</td>
<td>A9X</td>
<td>A10 Fusion</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>2x Twister @ 1.85 GHz</td>
<td>3x Typhone @ 1.5 GHz</td>
<td>2x Twister @ 2.26 GHz</td>
<td>4-core</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td>PVR GT7600 (6 cluster)</td>
<td>PVR GXA6850 (8 cluster)</td>
<td>PVR 12 Cluster Series 7</td>
<td>6 cluster GPU?</td>
</tr>
<tr>
<td><strong>RAM (shared memory)</strong></td>
<td>2GB LDDR4</td>
<td>2GB LDDR3</td>
<td>4GB LDDR4</td>
<td>3GB on Plus?</td>
</tr>
<tr>
<td><strong>Memory bus width</strong></td>
<td>64-bit</td>
<td>128-bit</td>
<td>128-bit</td>
<td>?</td>
</tr>
<tr>
<td><strong>Max # of threads per group</strong></td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>?</td>
</tr>
</tbody>
</table>
Methods for Running CNNs on Mobile Devices

- Sending CNN jobs to cloud
- Compression (pruning) of CNN
- Speeding up CNN
- Sending CNN jobs to cloud

Sending CNN jobs to cloud

Compression (pruning) of CNN

Speeding up CNN
Think Differently

- All existing methods can be viewed as approximations of an overly-redundant CNN, but do we really need such a CNN as the starting point?
Good to be Slim

- Slim CNN leads to:
  - less storage space
  - less memory usage
  - less computation
  - less power consumption

- Compare with others:
  - Full-Network Acceleration and Compression
  - We think one step forward
Be Slim is Hard…

Train a small model from scratch

Randomly Pruning
Feature Selection on CNN

- CNNs can be viewed as a set of "overly-redundant" feature extractors
A method for Pruning Redundant Neurons and Kernels of Deep Convolutional Neural Networks

A pre-trained CNN → Extract CNN Responses → Measure the Importance of Feature Extractors → Prune Model → Fine-tuning
A method for Pruning Redundant Neurons and Kernels of Deep Convolutional Neural Networks

- Intractable → tractable
- Inconsistent → consistent
Questions?