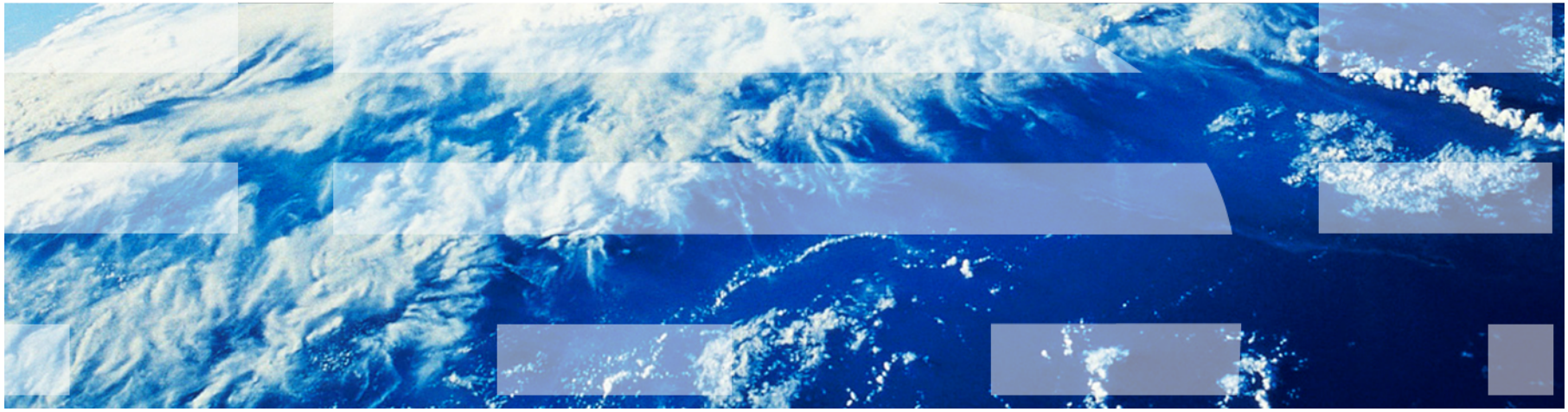
E6895 Advanced Big Data Analytics Lecture 7:

GPU and CUDA

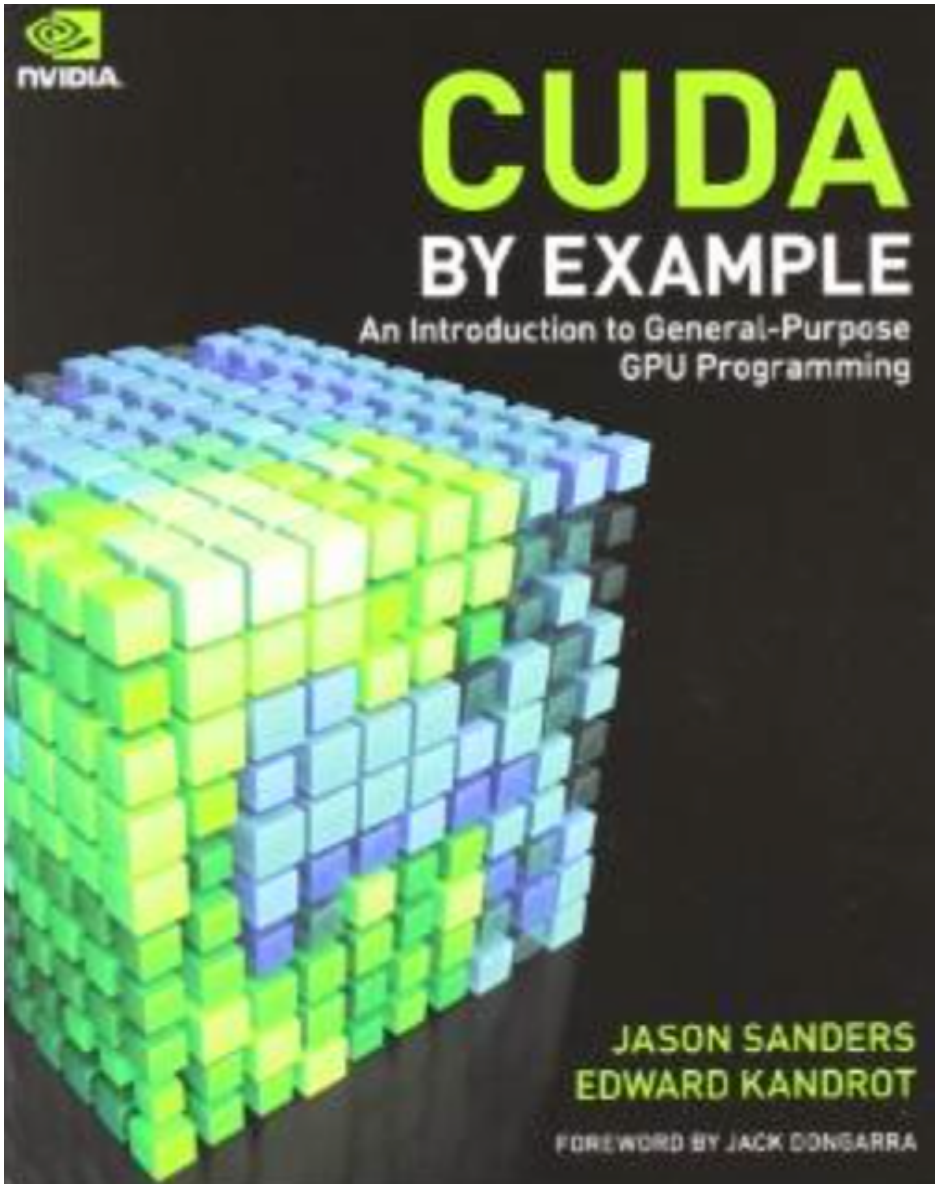
Ching-Yung Lin, Ph.D.

Adjunct Professor, Dept. of Electrical Engineering and Computer Science

IBM Chief Scientist, Graph Computing Research



Reference Book



CUDA:
Compute Unified Device Architecture

GPU

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.

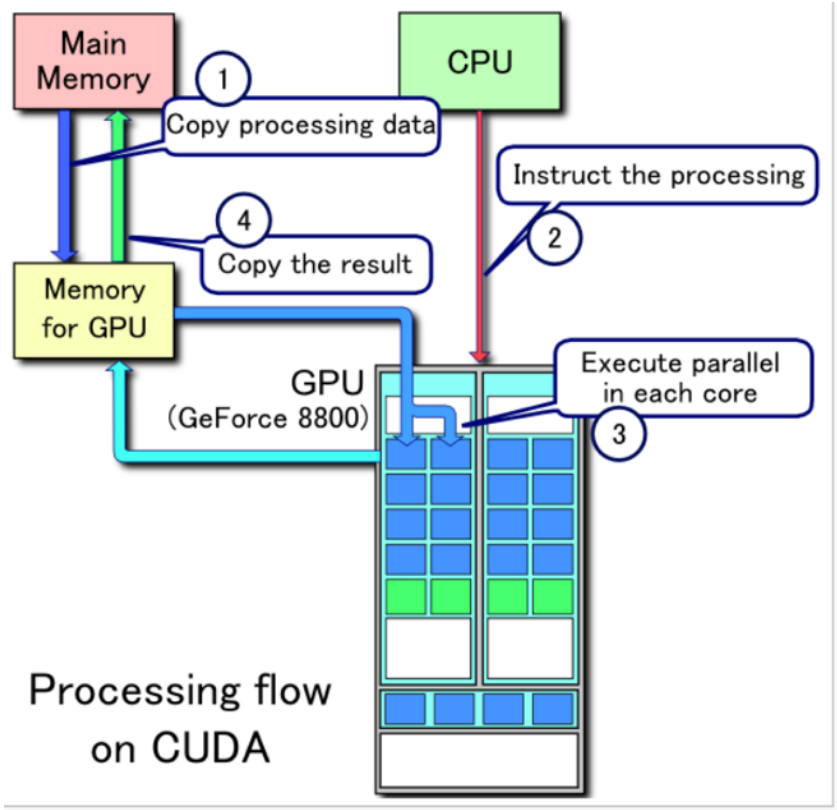
— The additional information could be input colors, texture coordinates, or other attributes

CUDA

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.



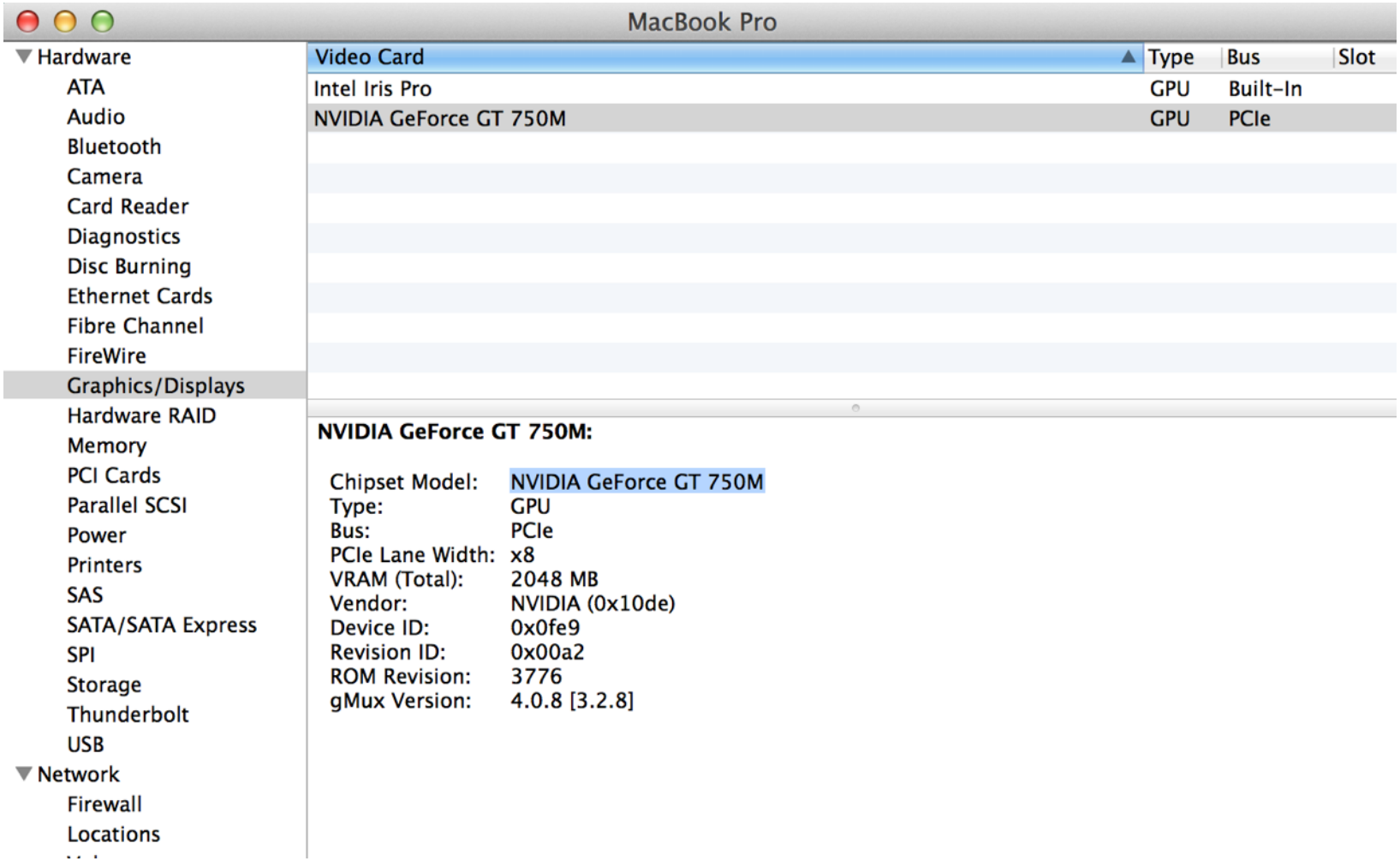
Example of CUDA processing flow

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

GPU on a MacBook



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.

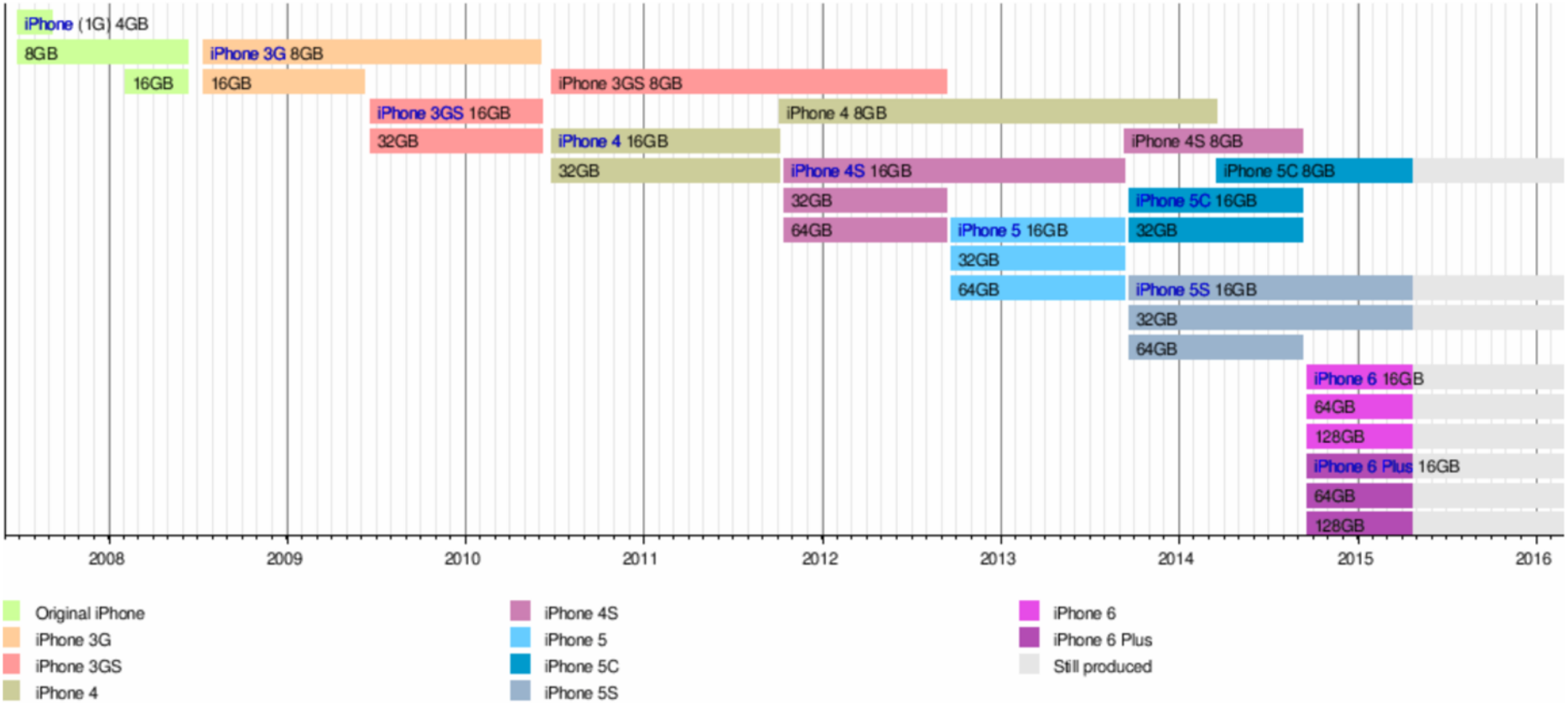
Customers can launch G2 instances using the AWS console, Amazon EC2 command line interface, AWS SDKs and third party libraries. Customers can launch the new instances in the US East (N. Virginia), US West (N. California), US West (Oregon), and EU (Ireland). In addition to On-Demand Instances, customers 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

iPhone models and their GPUs

Columbia University

Timeline of models [edit]



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)

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

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 A8 SoC

A8 — iPhone 6 and iPhone 6 Plus

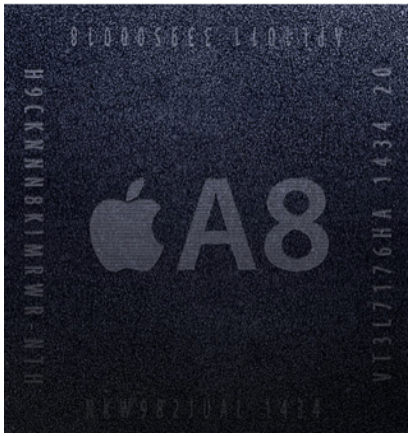
GPU: PowerVR Quad-core GX6450

4 Unified Shading Cluster (USC)

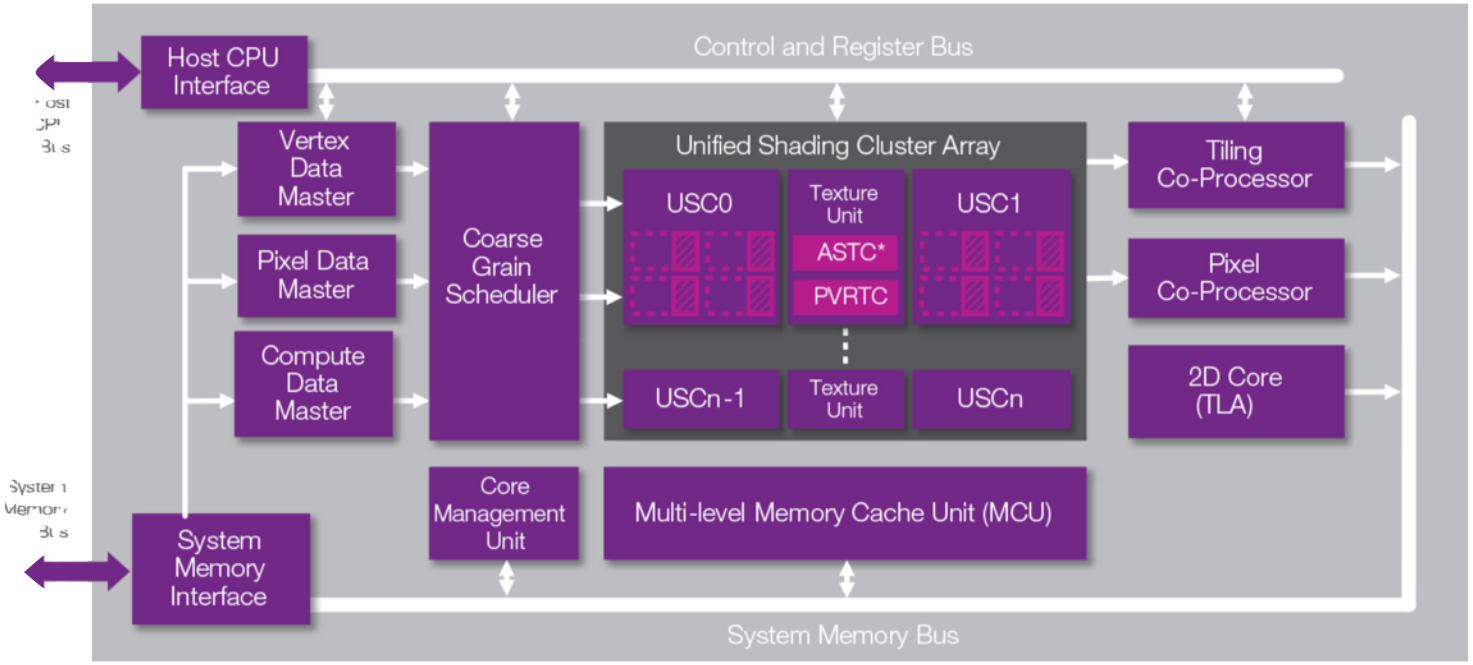
# of ALUs: 32 (FP32) or 64 (FP16) per USC

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

Supports OpenCL 1.2



manufactured by TSMC



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

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

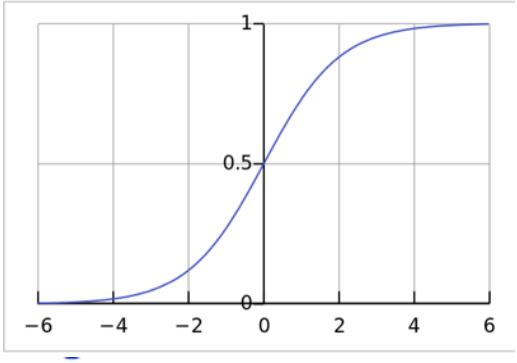
General data-parallel processing ➔ open CL and CUDA

*: https://developer.apple.com/metal/

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:

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

kernel code

device memory

thread_id for data parallelization

```
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 Compiler

CUDA supports most Windows, Linux, and Mac OS compilers

For Linux:

- Red Hat
- OpenSUSE
- Ubuntu
- Fedora

Hello World!!

```
#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

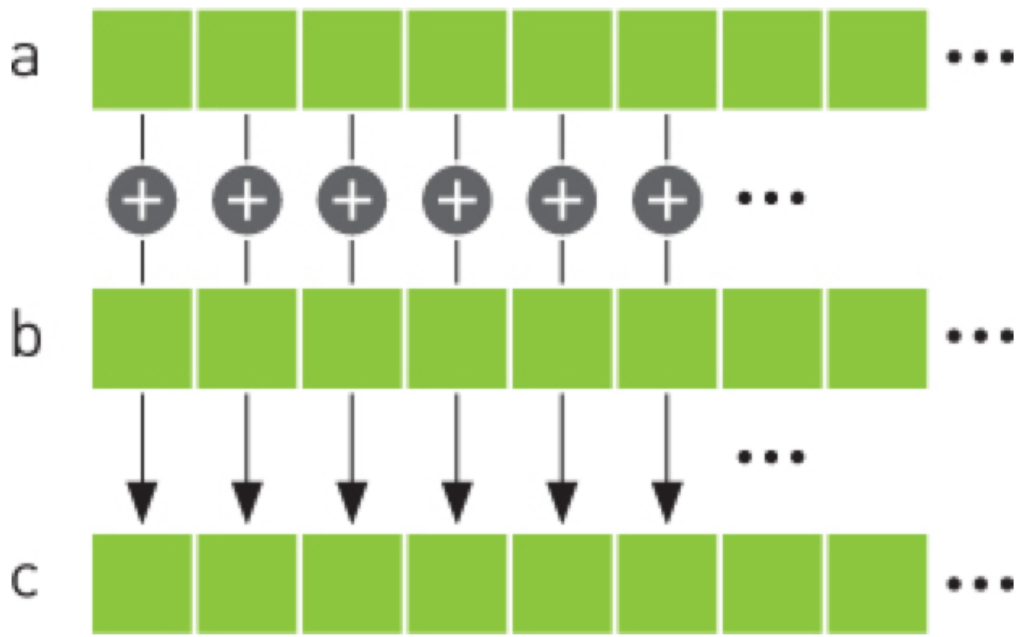
```
#include <iostream>

__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

Passing Parameters

```
#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

CPU CORE 1

```
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

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

GPU way — I

```
#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 ) );
```

GPU way — II

```
    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;
}
```

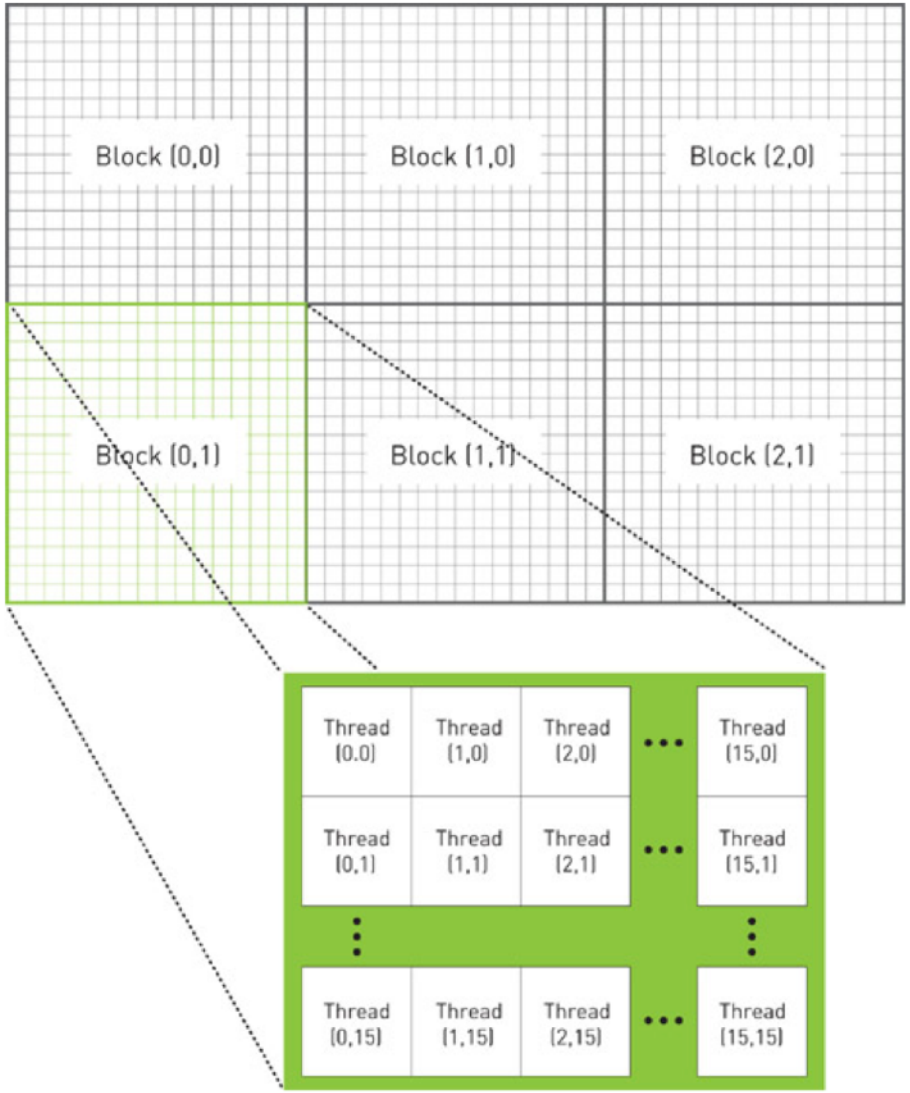
Blocks and Threads

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

2D hierarchy of blocks and threads

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



GPU way — III

```
__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];
}
```

GPU Threads — I

```
#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;
    }
```

GPU Threads — II

```
    // 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

NVIDIA CUDA TOOLKIT DOCUMENTATION

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 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. Seria 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)

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 (1073414144 bytes)
  ( 2) Multiprocessors, (192) CUDA Cores/MP:     384 CUDA Cores
  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:                             512 bytes
  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
```

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

```
void sumArraysOnHost(float *A, float *B, float *C, const int N)
{
    for (int idx = 0; idx < N; idx++)
    {
        C[idx] = A[idx] + B[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

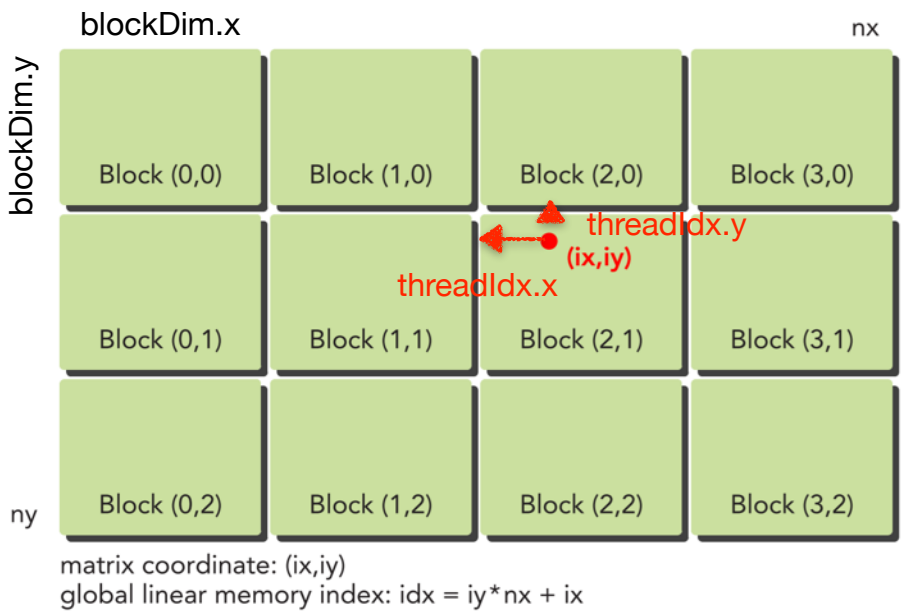
```
// grid 2D block 2D
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int nx,
                                 int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx +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

```
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)

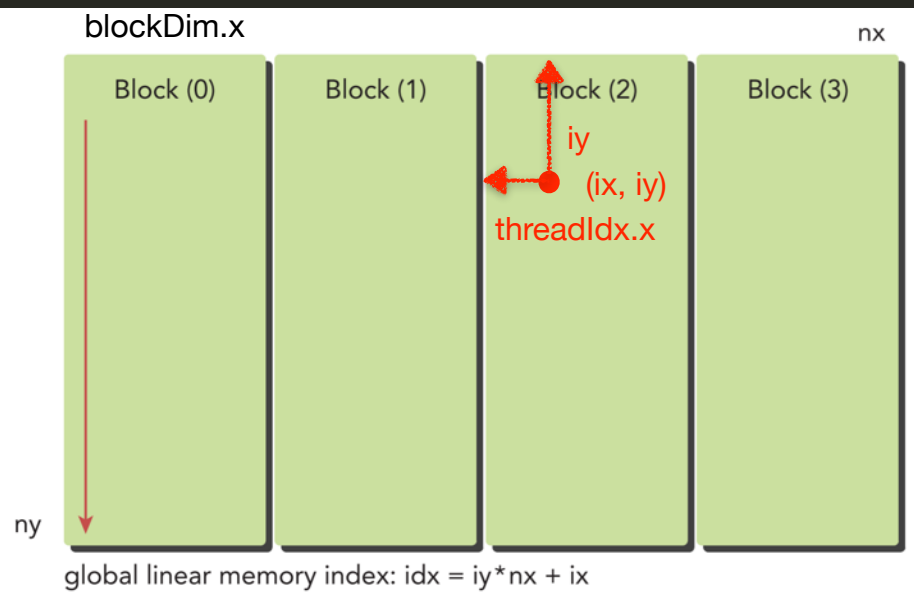
```
// grid 2D block 2D
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int nx,
                                 int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx +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



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)

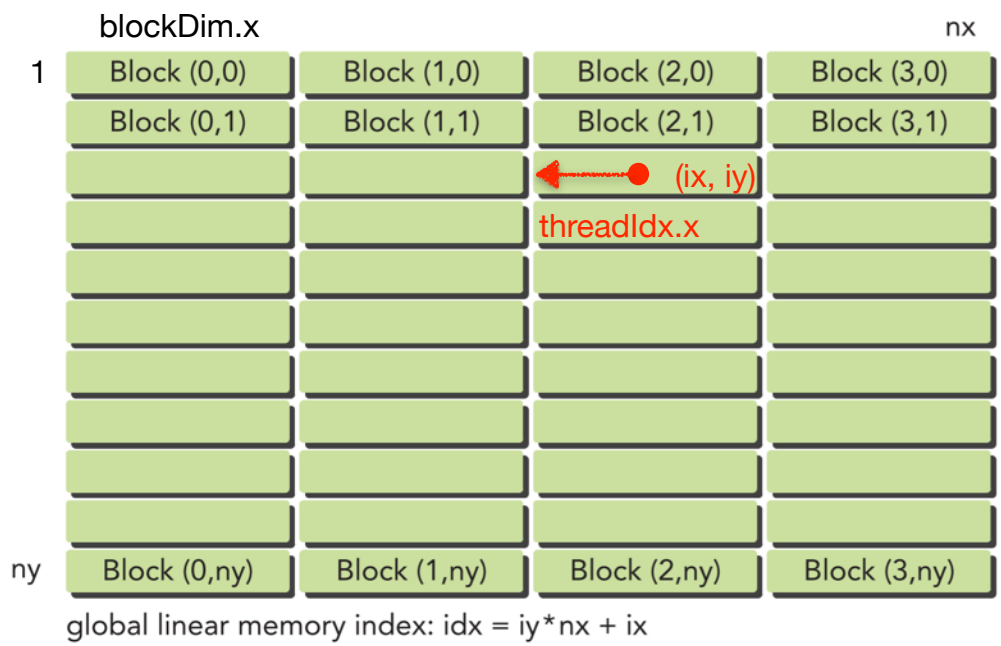
```
// grid 1D block 1D
__global__ void sumMatrixOnGPU1D(float *MatA, float *MatB, float *MatC, int nx,
                                 int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    if (ix < nx )
        for (int iy = 0; iy < ny; iy++) {
            int idx = iy * nx + ix;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



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)

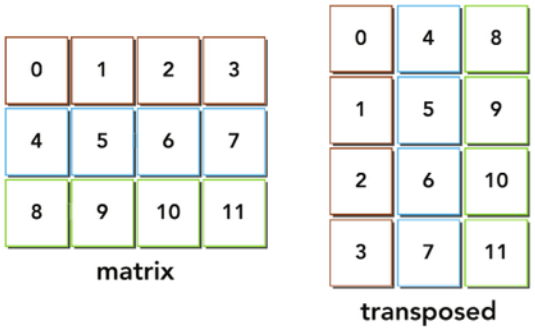
```
// grid 2D block 1D
__global__ void sumMatrixOnGPUMix(float *MatA, float *MatB, float *MatC, int nx,
                                  int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = blockIdx.y;
    unsigned int idx = iy * nx + ix;
    if (ix < nx && iy < ny)
        MatC[idx] = MatA[idx] + MatB[idx];
}
```



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

```
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

```
// 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

```
// 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] = 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

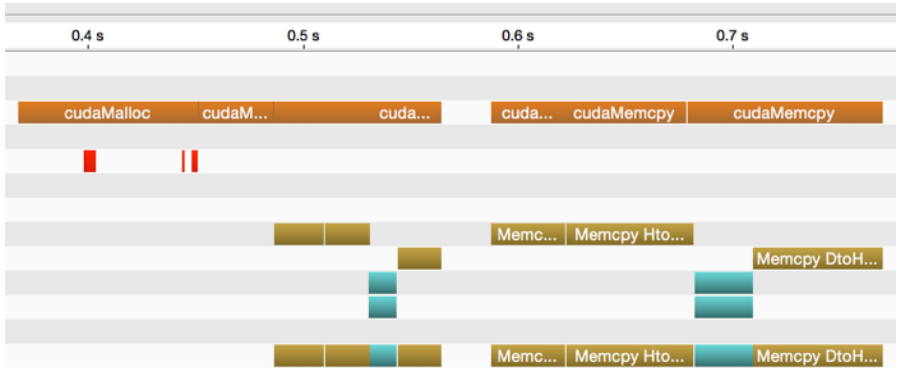
Concurrent handle ***data transfer*** and ***computation***

- For NVIDIA GT 650M (laptop GPU), there is one copy engine.
- For NVIDIA Tesla K40 (high-end GPU), there are two copy engines

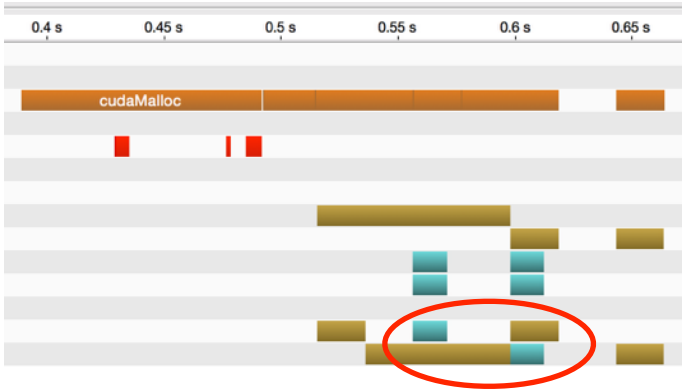
The latency in data transfer could be hidden during computing

To handle two tasks, which both are matrix multiplications.

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



No concurrent processing



Concurrent processing

Reference

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

Homework #3 (due March 31st)

Choose 2 of the algorithms you use in your homework #2. Convert them into a GPU version.

Show your code, and performance measurement comparing to your non-GPU version.

The more innovative/complex your algorithms are, the higher score you will get on your Homework #3.