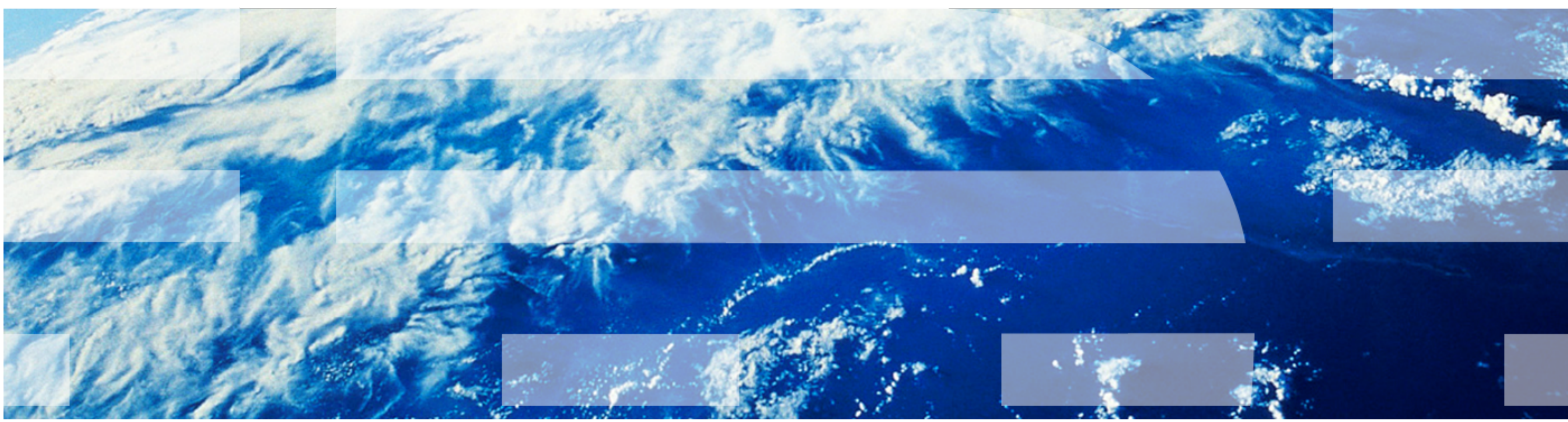
E6893 Big Data Analytics Lecture 9:

GPU Fundamentals for Massive Data Processing

Ching-Yung Lin, Ph.D.

Adjunct Professor, Dept. of Electrical Engineering and Computer Science



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.

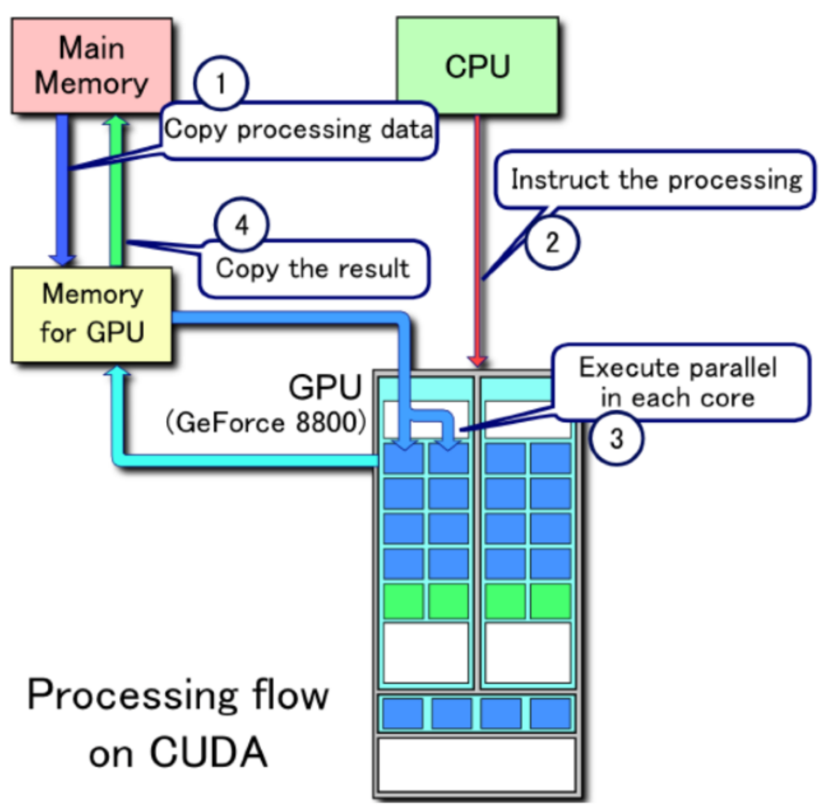
— The GPUs of the early 2000s were designed to produce a color for every pixel on the screen using programmable arithmetic units known as pixel shaders.

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

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.



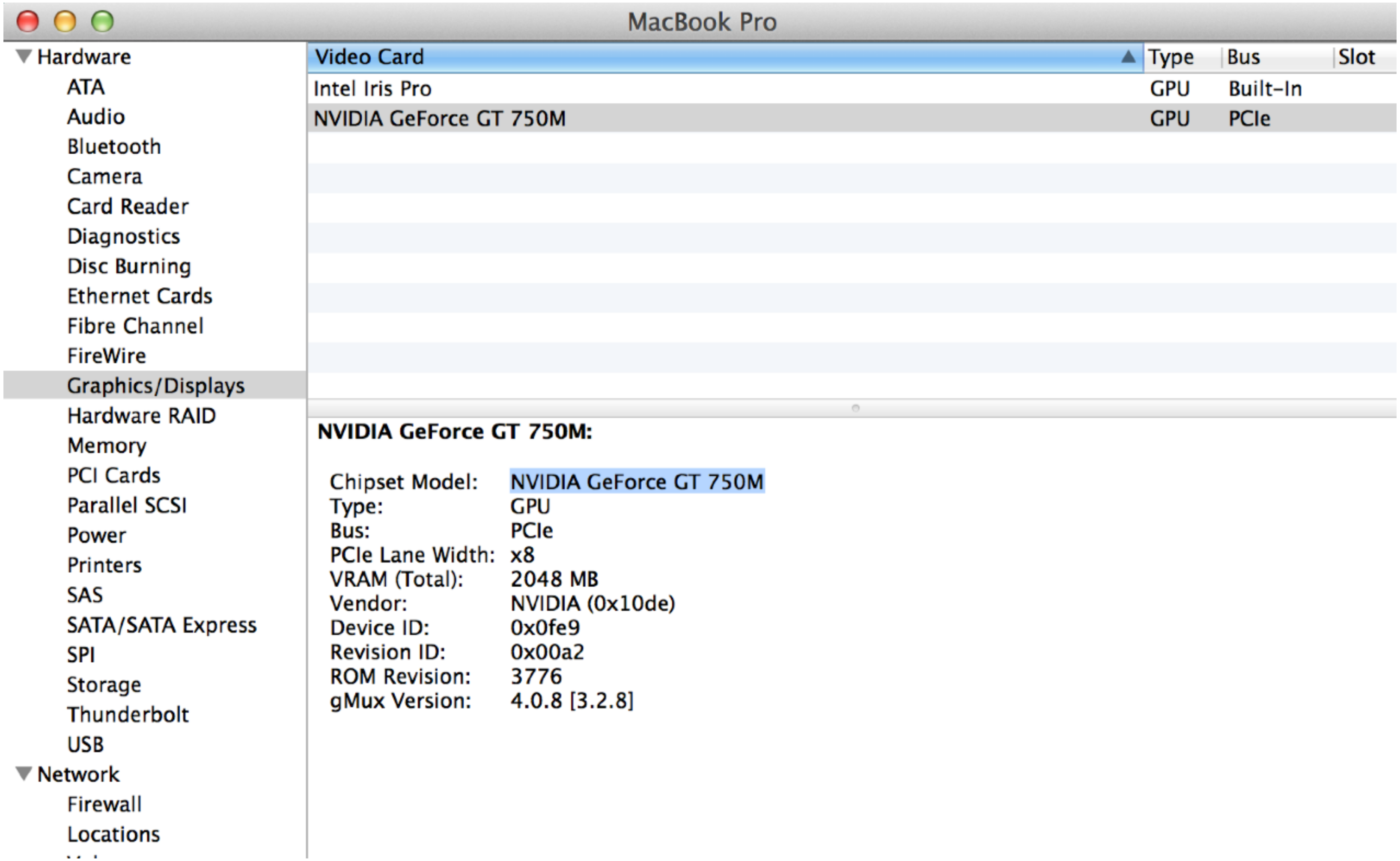
Example of CUDA processing flow

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

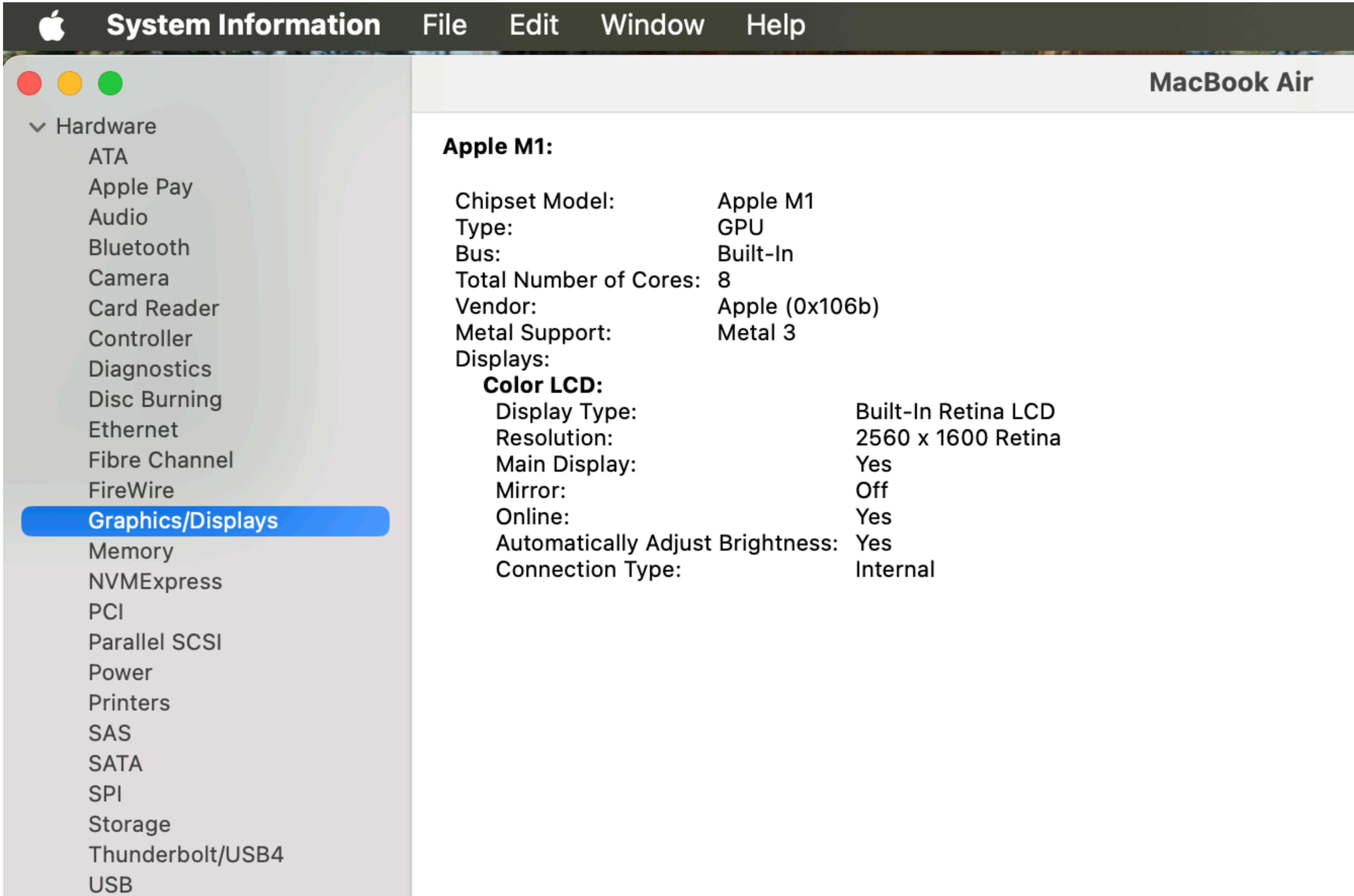
Medical Imaging
Computational Fluid Dynamics
Environmental Science

GPU on an Intel MacBook



GT 750M:
— 2 * 192 CUDA cores
— max thread number: 2 * 2048

GPU on an M1 MacBook



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.

Compiler

CUDA supports most Windows, Linux, and ~~Mac OS~~ compilers

For Linux:
- Red Hat
- OpenSUSE
- Ubuntu
- Fedora

GPU Architecture

- built by several streaming multiprocessors (SMs)

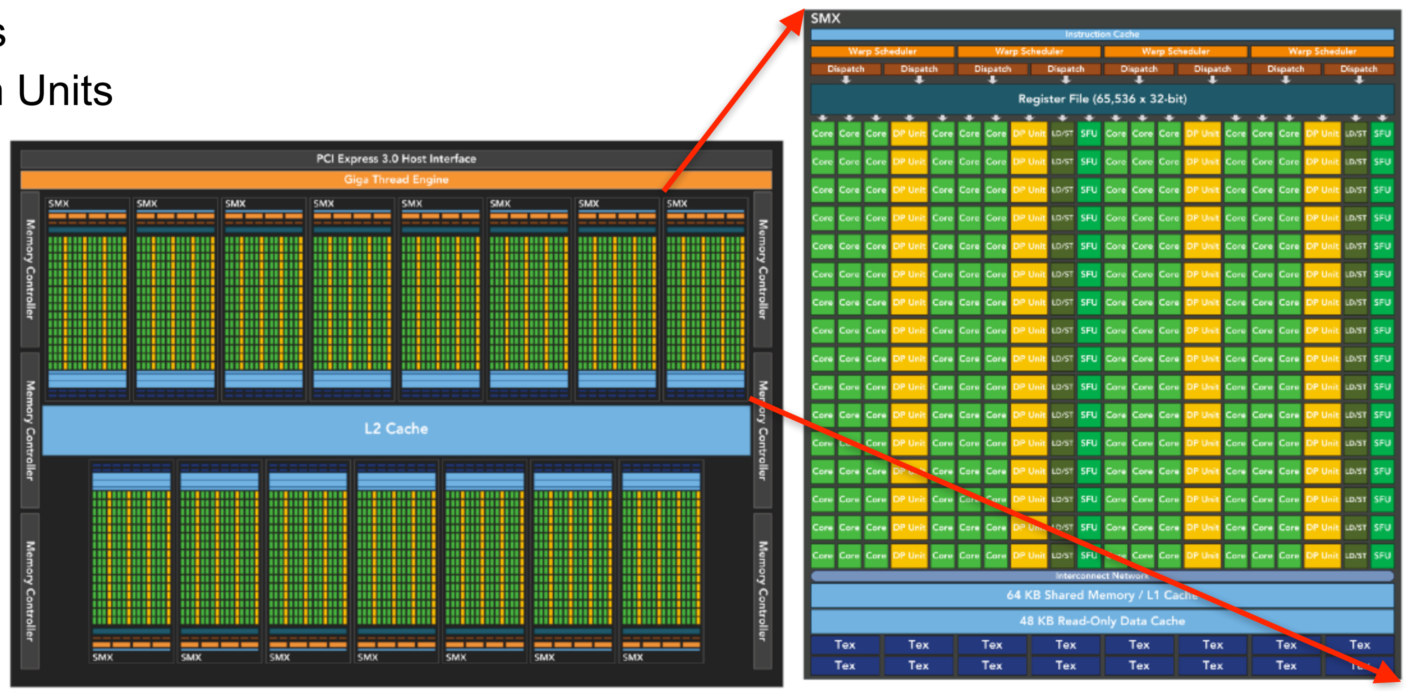
In each SM:

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

In each device:

- L2 Cache

Kepler Architecture, K20X



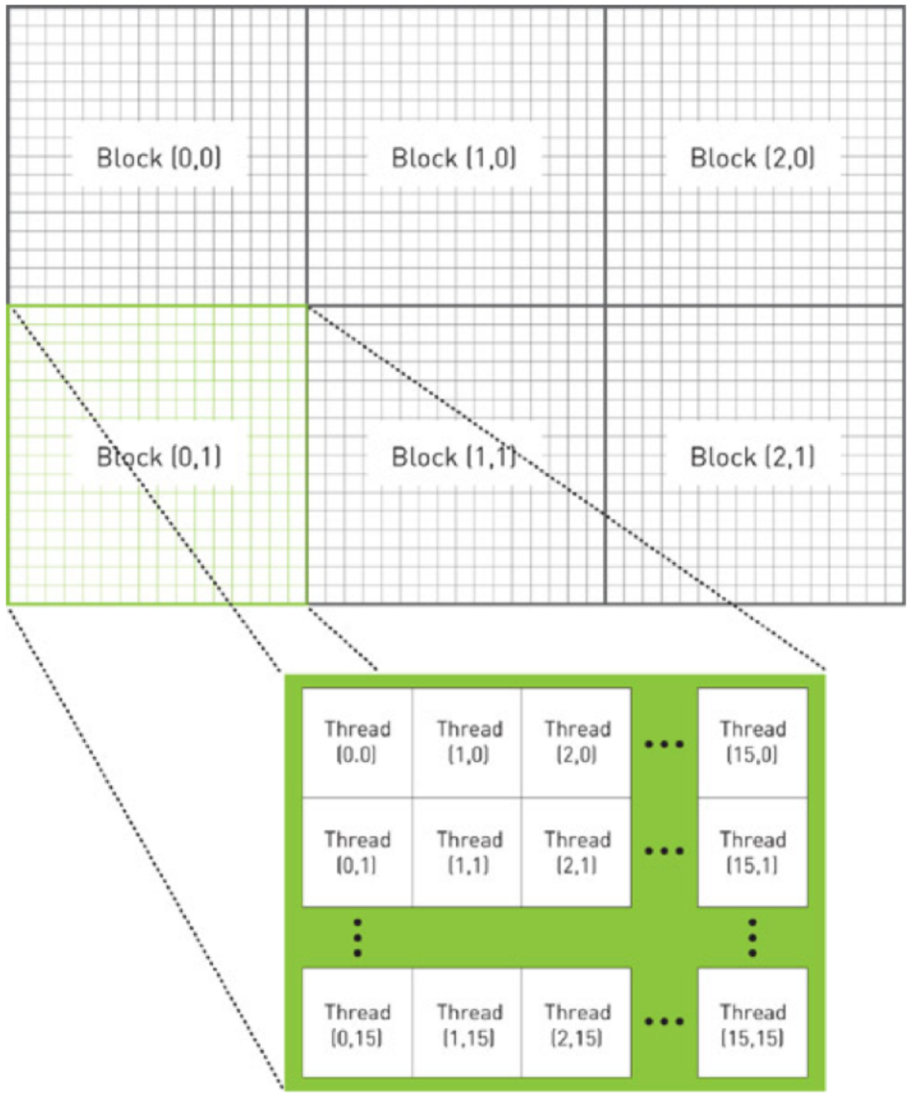
Blocks and Threads

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



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

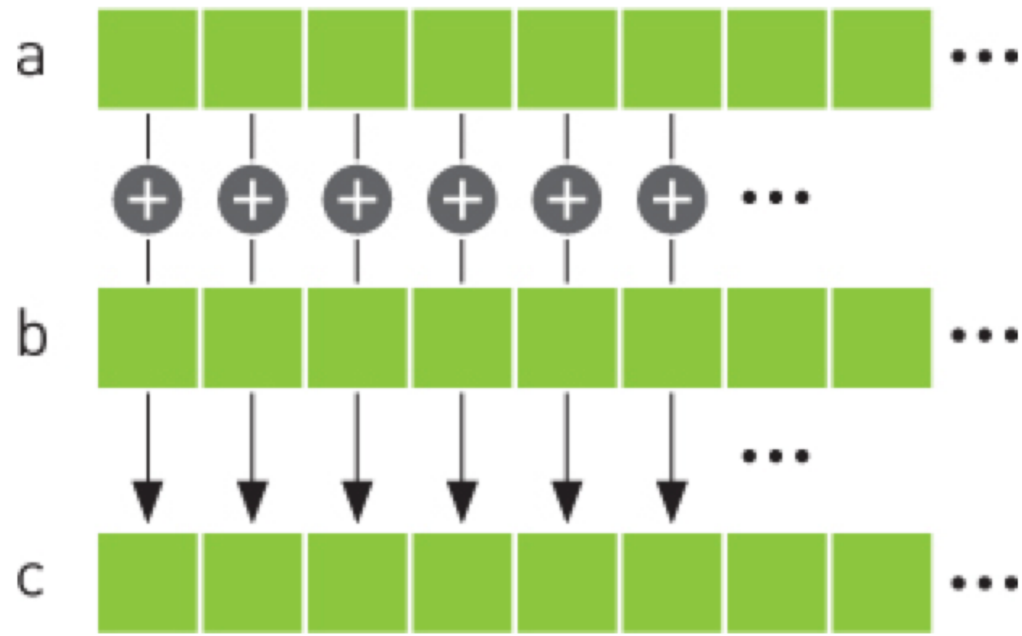
```
#include <iostream>

__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

```
#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;
}
```

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;
}
```

GPU Architecture

GPU Architecture

built by several streaming multiprocessors (SMs)

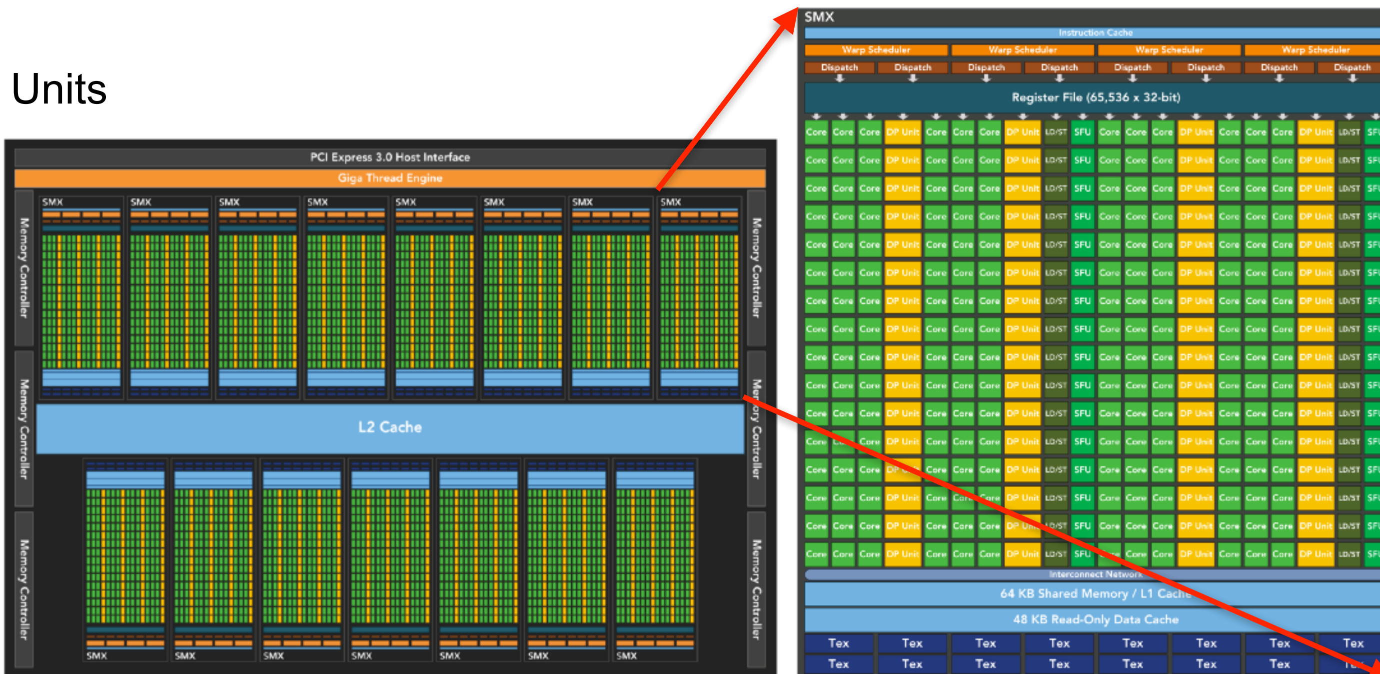
In each SM:

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

In each device:

- L2 Cache

Kepler Architecture, K20X



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

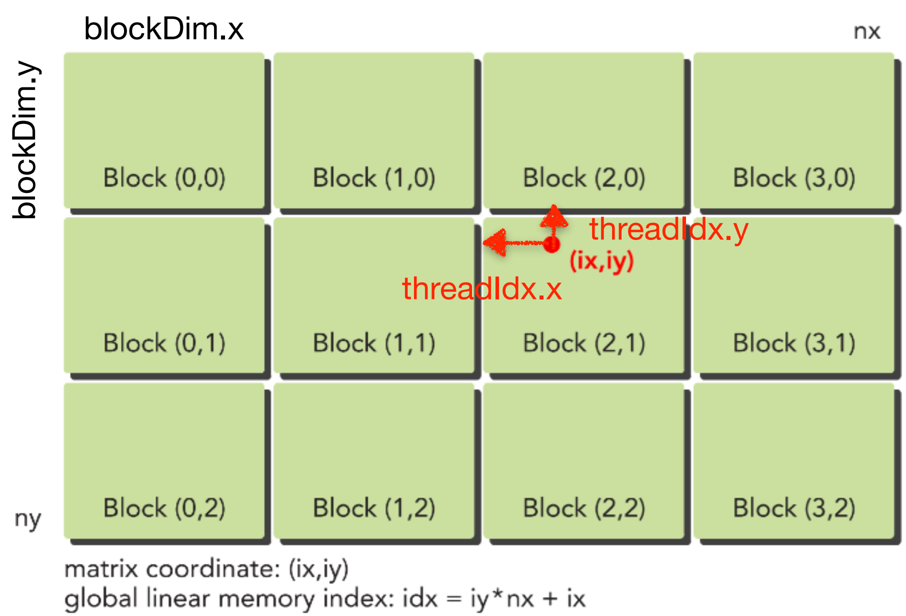
```
// grid 2D block 2D
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int nx,
                                 int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx +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

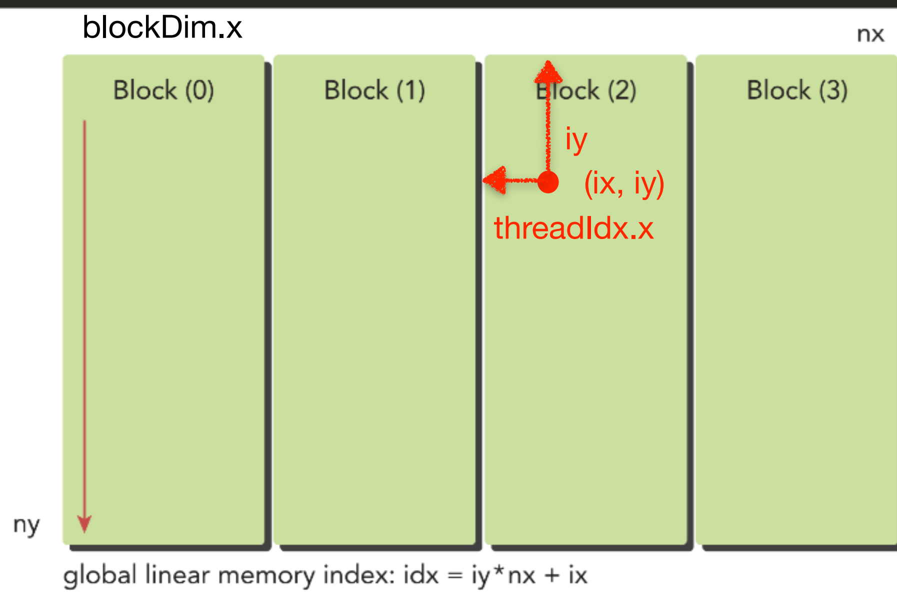
```
// grid 2D block 2D
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int nx,
                                  int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx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

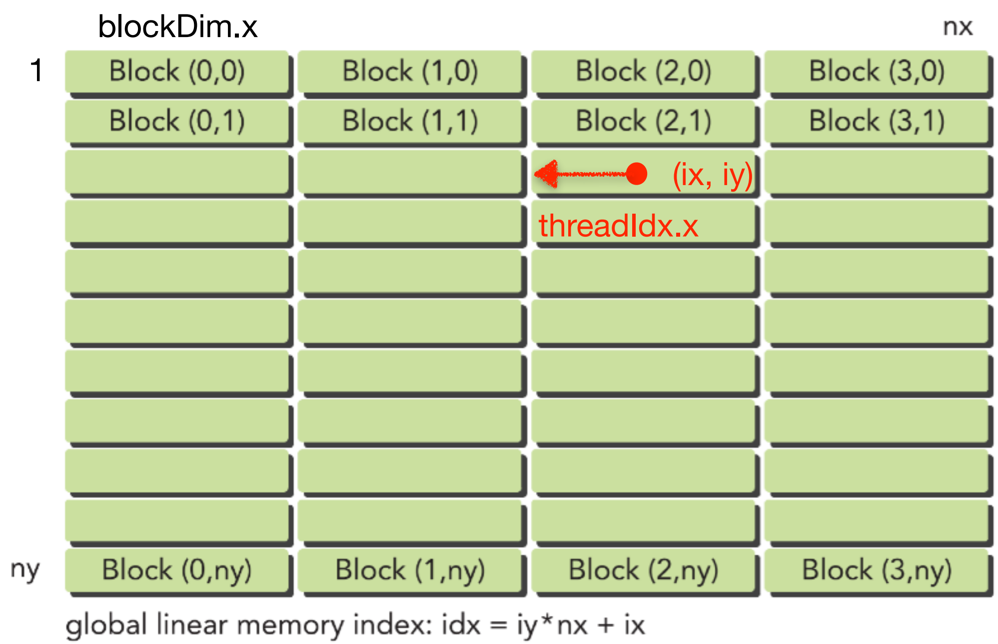
```
// grid 1D block 1D
__global__ void sumMatrixOnGPU1D(float *MatA, float *MatB, float *MatC, int nx,
                                  int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    if (ix < nx )
        for (int iy = 0; iy < ny; iy++) {
            int idx = iy * nx +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

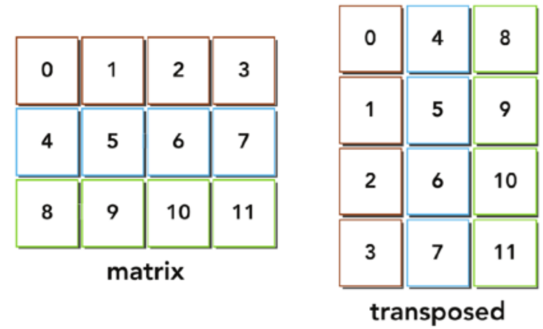
```
// grid 2D block 1D
__global__ void sumMatrixOnGPUMix(float *MatA, float *MatB, float *MatC, int nx,
                                  int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = blockIdx.y;
    unsigned int idx = iy * nx + ix;
    if (ix < nx && iy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

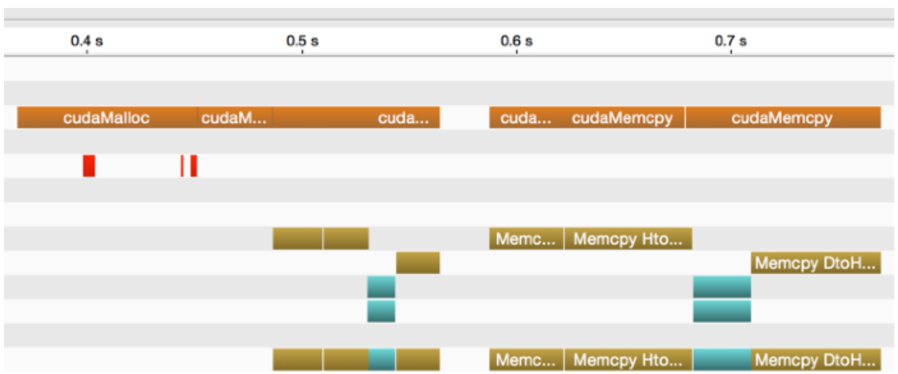
Concurrent handle ***data transfer*** and ***computation***

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

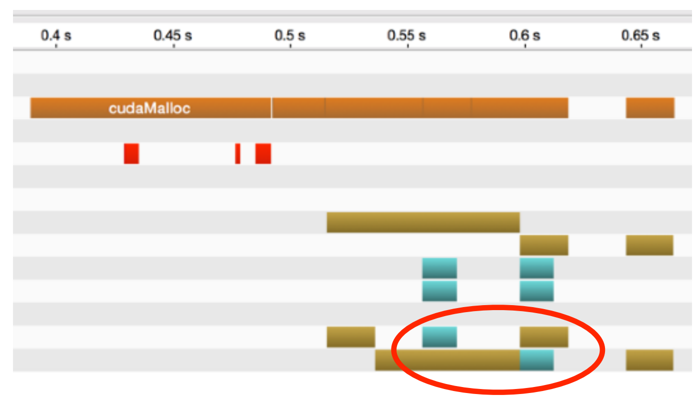
The latency in data transfer could be hidden during computing

To handle two tasks, which both are matrix multiplications.

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



No concurrent processing



Concurrent processing

Outline

- NVIDIA GPU Architecture
  - Execution Model
  - Resource Allocation
  - Memory Type
  - Concurrent Processing
- Applications on NVIDIA GPU:
  - Mandatory Component in Machine Learning:
    - Matrix Multiplication with Addition ($Y = A^T B + C$)

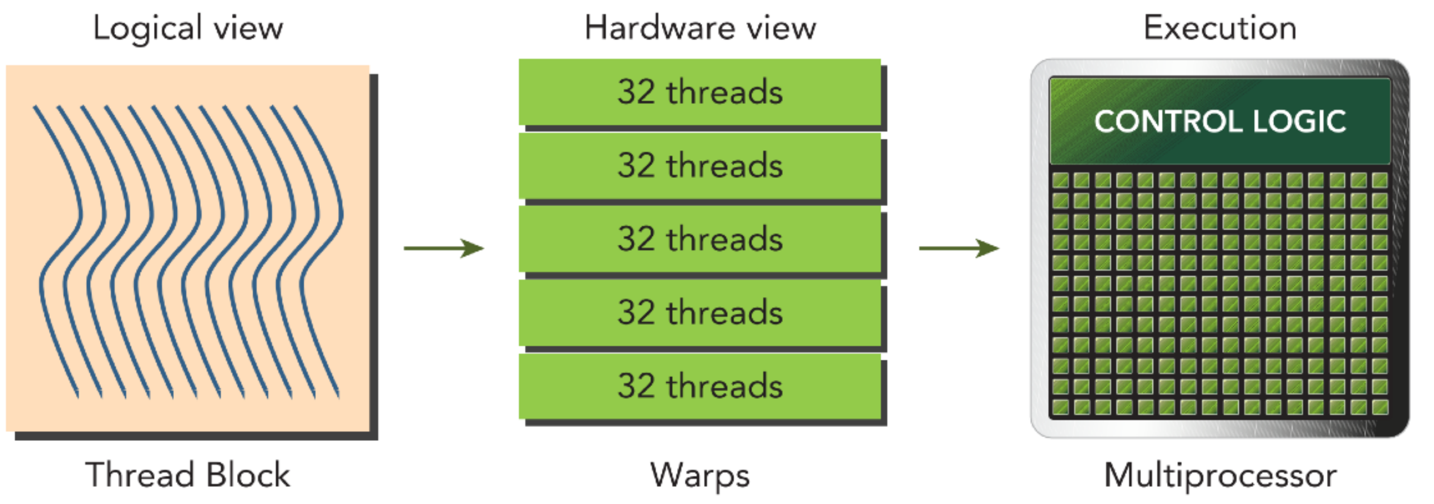
- GPU Architecture on iPhone/iPad
  - Execution Model
  - Metal Programming Examples for Data-Parallel Computation on GPU
    - Sigmoid Function
    - Sobel Operators for Image Processing

Execution Model - Warp Concept

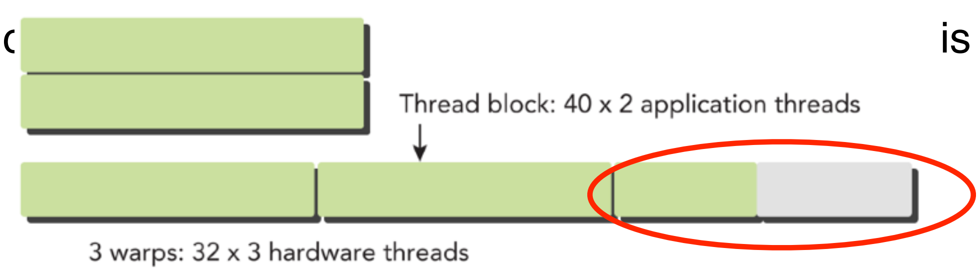
NVIDIA GPU groups 32 threads into one warp, and then execute warps ***sequentially***.

Number of **concurrent** warps are based on the number of warp scheduler. (Kepler has 4 warp scheduler)

Relationship between logical view and hardware view



Inefficient way to allc … is not multiples of 32.

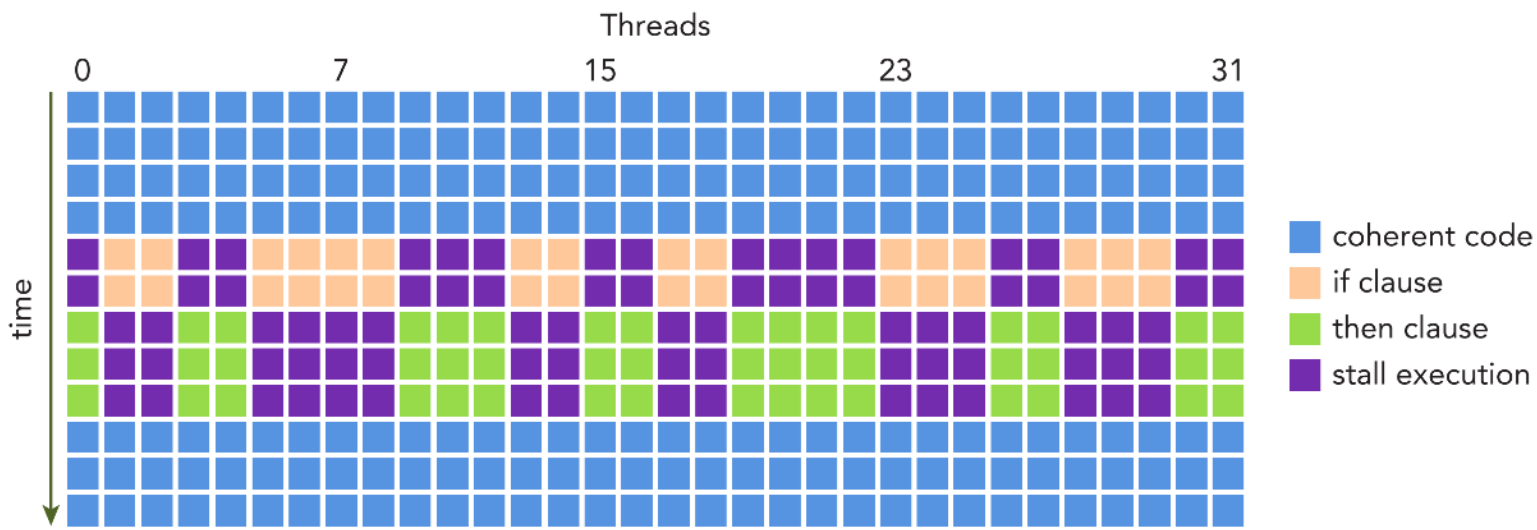


Execution Model - Warp Divergence

GPU has light-weight control module, complicated ***control flow*** will hurt the performance of GPU.

In the same warp, e.g., if you allocate 16 threads to do **A** task; and 16 threads to do **B** task.

A and B will be executed serially.



Example: simpleDivergence.cu

- Test with optimization (-O2) and without optimization (-g)

Execution Model - Warp Divergence

Compilation — turn off the optimization

nvcc -g -G -o simpleDivergence simpleDivergence.cu

Execute through nvprof to extract profiling information

nvprof **--metrics** branch_efficiency **--events** branch,divergent_branch ./simpleDivergence

```
Invocations                                Event Name         Min         Max         Avg
Device "GeForce GT 650M (0)"
        Kernel: mathKernel1(float*)
          1                                    branch          10          10          10
          1                           divergent_branch           2           2           2
        Kernel: mathKernel2(float*)
          1                                    branch          10          10          10
          1                           divergent_branch           0           0           0
        Kernel: mathKernel3(float*)
          1                                    branch          12          12          12
          1                           divergent_branch           4           4           4
        Kernel: mathKernel4(float*)
          1                                    branch           4           4           4
          1                           divergent_branch           0           0           0
        Kernel: warmingup(float*)
          1                                    branch          10          10          10
          1                           divergent_branch           0           0           0
```

```
==99767== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GT 650M (0)"
        Kernel: mathKernel1(float*)
          1                         branch_efficiency                         Branch Efficiency      80.00%      80.00%      80.00%
        Kernel: mathKernel2(float*)
          1                         branch_efficiency                         Branch Efficiency     100.00%     100.00%     100.00%
        Kernel: mathKernel3(float*)
          1                         branch_efficiency                         Branch Efficiency      66.67%      66.67%      66.67%
        Kernel: mathKernel4(float*)
          1                         branch_efficiency                         Branch Efficiency     100.00%     100.00%     100.00%
        Kernel: warmingup(float*)
          1                         branch_efficiency                         Branch Efficiency     100.00%     100.00%     100.00%
```

Resource Allocation

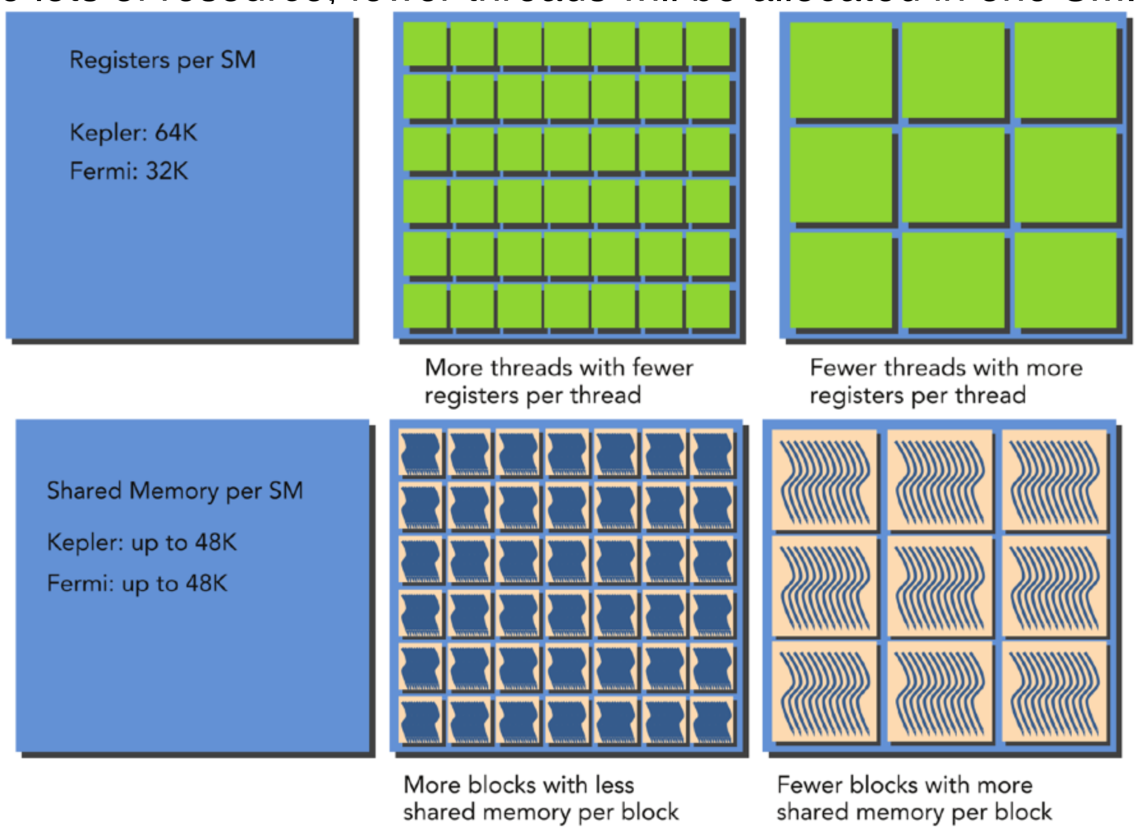
A warp is consisted of

Program counters

Registers

Shared memory

If a thread use lots of resource, fewer threads will be allocated in one SM.



Occupancy

Toolkit, CUDA Occupancy Calculator:

in /usr/local/cuda/tools/CUDA_Occupancy_Calculator.xls

It could assist in measuring the occupancy of your configuration

CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	3.0	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	

2.) Enter your resource usage:		
Threads Per Block	1024	(Help)
Registers Per Thread	32	
Shared Memory Per Block (bytes)	32	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	2048	(Help)
Active Warps per Multiprocessor	64	
Active Thread Blocks per Multiprocessor	2	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	3.0
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4

Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	32	64	2
Registers	(Warp limit per SM due to per-warp reg count)	32	64	2
Shared Memory (Bytes)		256	49152	192

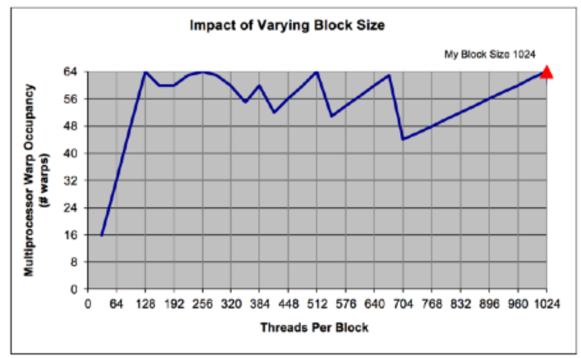
Note: SM is an abbreviation for (Streaming) Multiprocessor

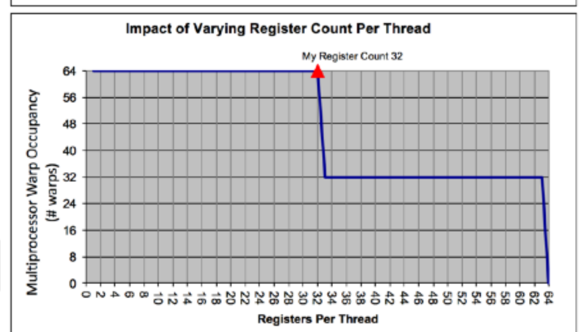
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	2	32	64

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.





Occupancy, Memory Load Efficiency, Memory Load Throughput

View number of registers, set the constraints on number of registers per thread

nvcc -g -G -arch=sm_30 --ptxas-options=-v --maxrregcount=31 -o sumMatrix sumMatrix.cu

Check *occupancy*, *memory load efficiency*, *memory load throughput* to explore the suitable configuration of size of thread block

nvprof --metrics gld_throughput,gld_efficiency,achieved_occupancy ./sumMatrix dimX dimY

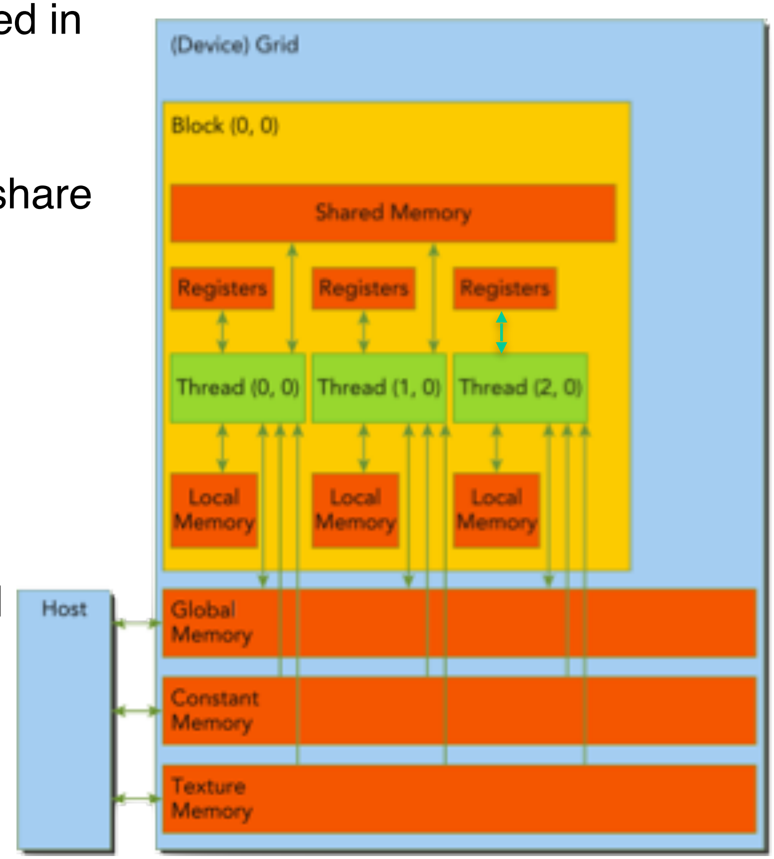
do example on sumMatrix.cu with dim of thread block {4,4}, {4,8}, {8,4}, {8,8}, {16,16}, {32,32}

{16,16} is the fast one.

```
⚙ cfchen@cfchen-MBP  ~/Downloads/CodeSamples/chapter03  ./sumMatrix 4 4
sumMatrixOnGPU2D <<<(1024,1024), (4,4)>>> elapsed 5.806647 s
⚙ cfchen@cfchen-MBP  ~/Downloads/CodeSamples/chapter03  ./sumMatrix 4 8
sumMatrixOnGPU2D <<<(1024,512), (4,8)>>> elapsed 3.085108 s
⚙ cfchen@cfchen-MBP  ~/Downloads/CodeSamples/chapter03  ./sumMatrix 8 4
sumMatrixOnGPU2D <<<(512,1024), (8,4)>>> elapsed 2.559193 s
⚙ cfchen@cfchen-MBP  ~/Downloads/CodeSamples/chapter03  ./sumMatrix 8 8
sumMatrixOnGPU2D <<<(512,512), (8,8)>>> elapsed 1.624973 s
⚙ cfchen@cfchen-MBP  ~/Downloads/CodeSamples/chapter03  ./sumMatrix 16 16
sumMatrixOnGPU2D <<<(256,256), (16,16)>>> elapsed 1.234786 s
⚙ cfchen@cfchen-MBP  ~/Downloads/CodeSamples/chapter03  ./sumMatrix 32 32
sumMatrixOnGPU2D <<<(128,128), (32,32)>>> elapsed 1.382378 s
```

Memory Type

1. Register
   - per thread, An automatic variable in kernel function, *low latency, high bandwidth*
2. Local memory
   - per thread, variable in a kernel but can not be fitted in register
3. Shared memory (__**shared**__)
   - all threads, faster than local and global memory, share among thread blocks
   - Use for **inter-thread communication**, 64KB, physically shared with L1 cache
4. Constant memory (__**constant**__)
   - per device, read-only memory
5. Texture memory
   - per SM, read-only cache, optimized for 2D spatial locality
6. Global memory



Memory Type

MEMORY	ON/OFF CHIP	CACHED	ACCESS	SCOPE	LIFETIME
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached only on devices of compute capability 2.x

Concurrent Processing

Concurrent handle

- ***Data transfer*** + ***computation***

  For NVIDIA GT 750M (laptop GPU), there is one copy engine.

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

- ***Computation*** + ***computation*** is possible if your computation resource is enough.

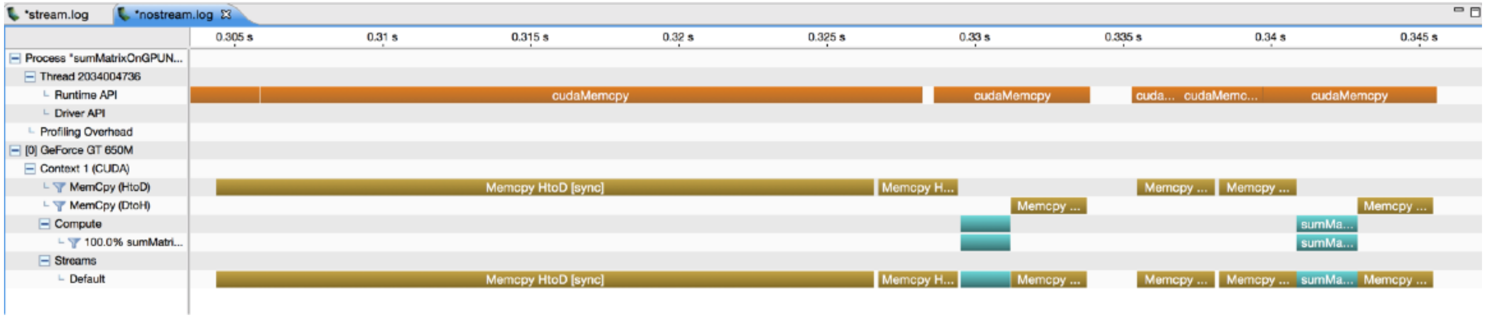
Check example, sumMatrixOnGPUStream.cu and sumMatrixOnGPUNoStream.cu

Goal: Compute two matrix addition (C1 = A1 + B1 and C2 = A2 + B2)

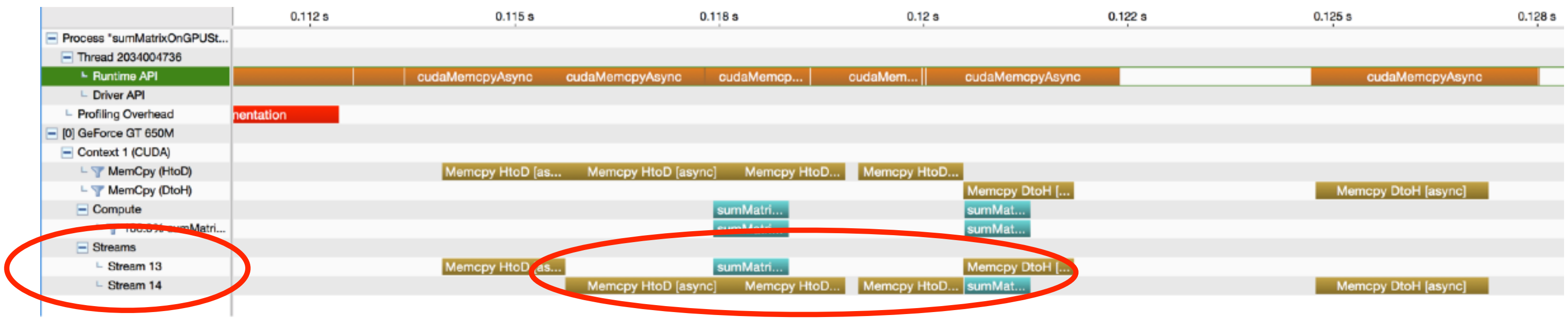
The latency in data transfer could be hidden during computing

or concurrent computation.

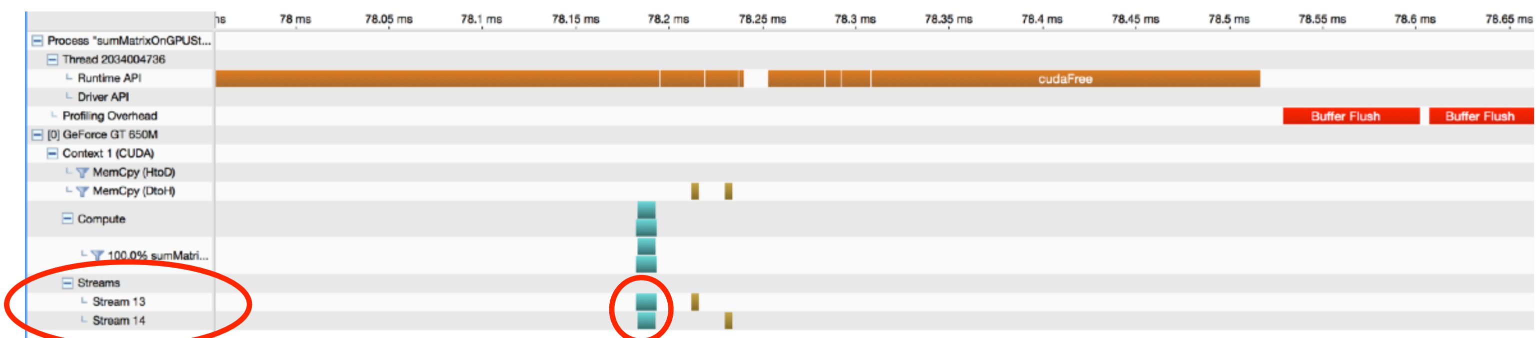
Concurrent Processing



Sequential processing



Concurrent processing (data transfer + computation)



Concurrent processing (computation + computation)

Application: Neural Network

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

$\mathbf{a}=f(\mathbf{x}^T\mathbf{w}+\mathbf{b})$

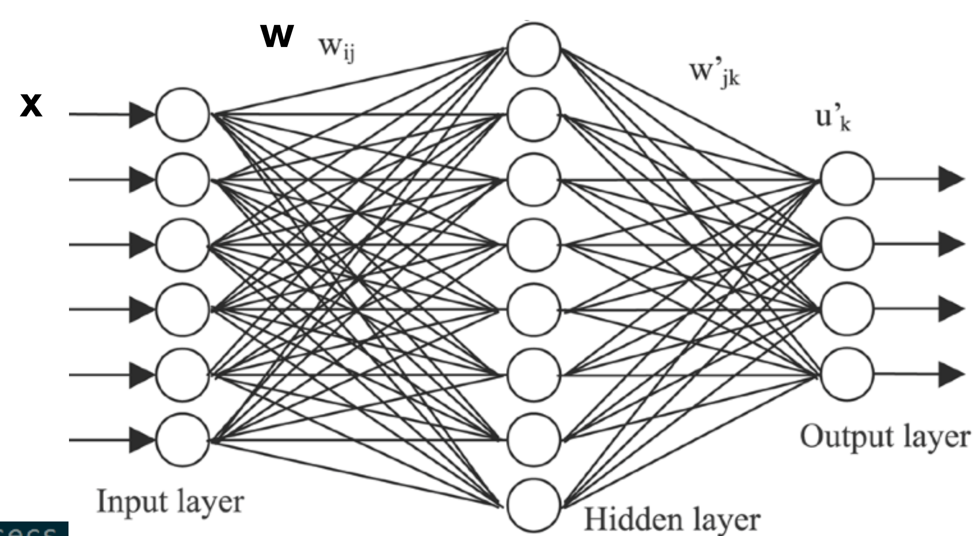
x is a matrix that records the input which is fed to neural network

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, $\mathbf{x}^T\mathbf{w}+\mathbf{b}$

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
```

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

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:

http://on-demand-gtc.gputechconf.com/gtcnew/on-demand-gtc.php

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

GPU on iOS devices

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 apps***.*

Metal could be used for:

Graphic processing ➔ openGL

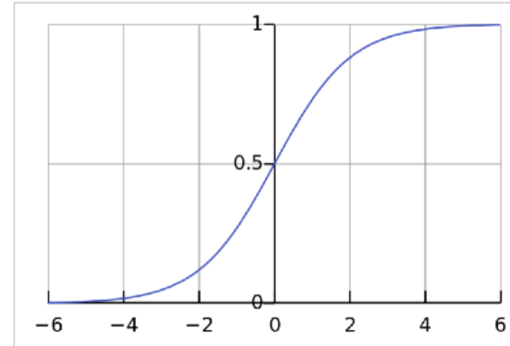
General data-parallel processing ➔ open CL and CUDA

*: 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 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]));
}
```

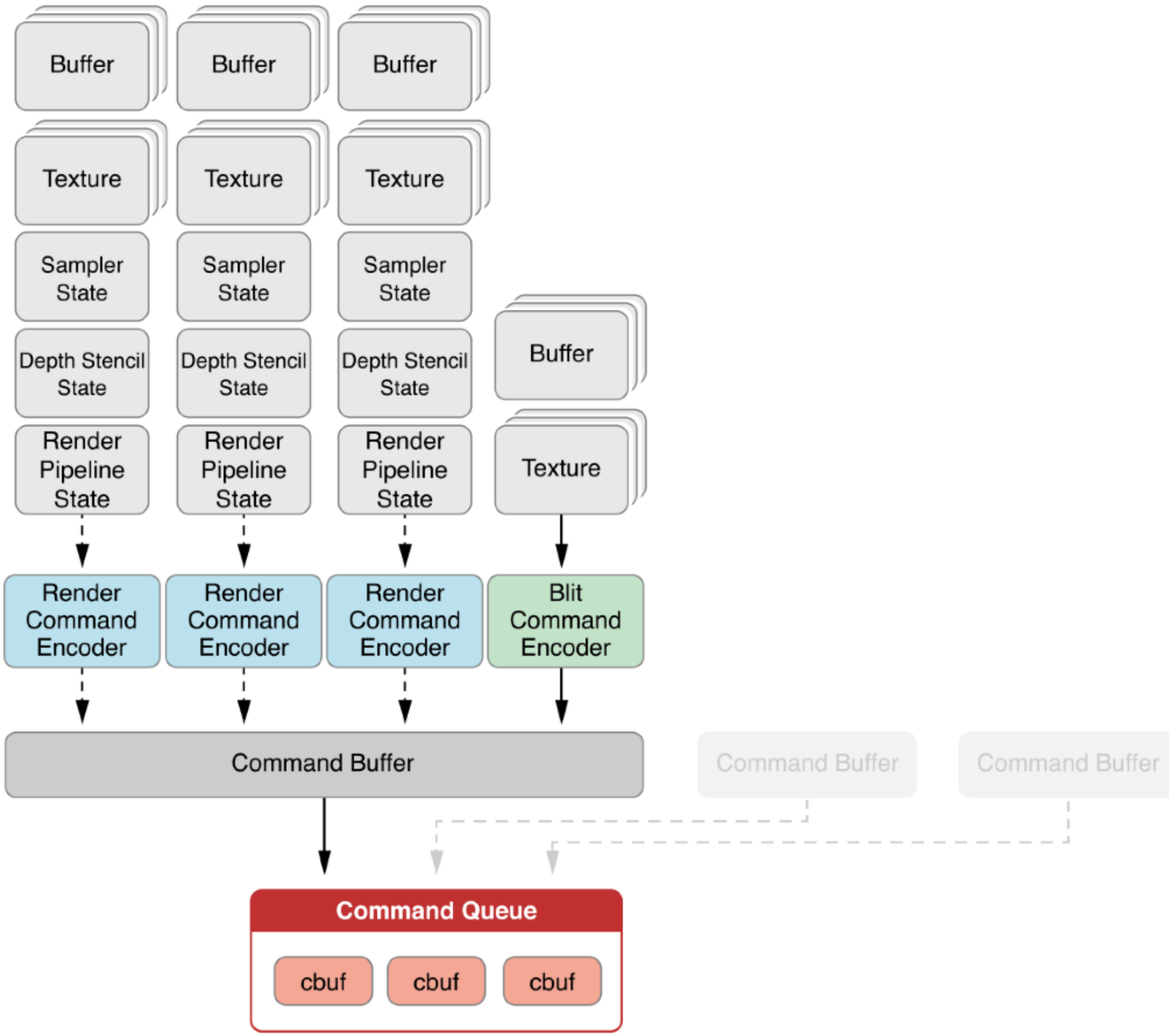
kernel code

device memory

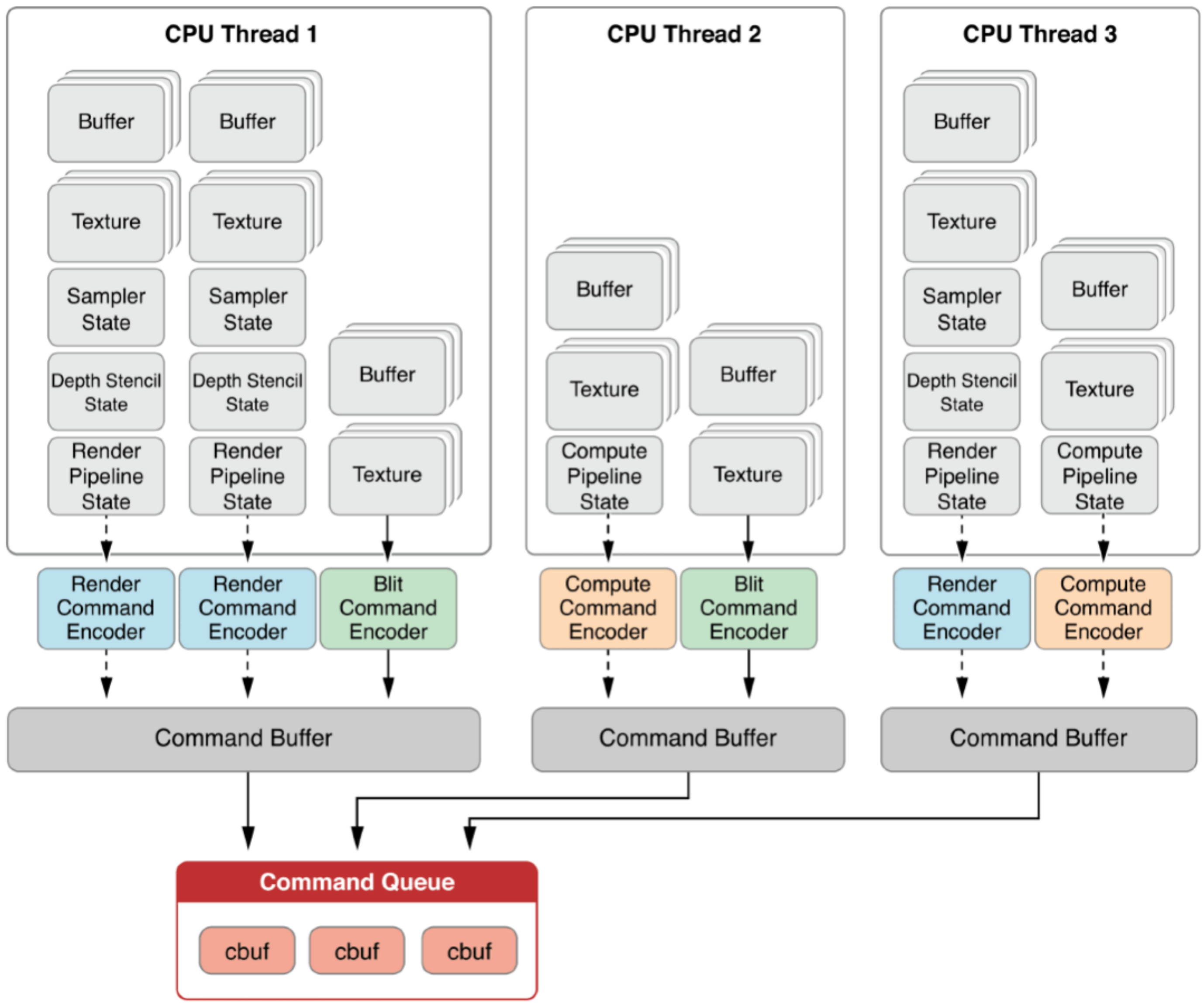
thread_id for data parallelization

source: http://memkite.com

Metal Object Relationships



Metal Command Buffers with Multiple Threads



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

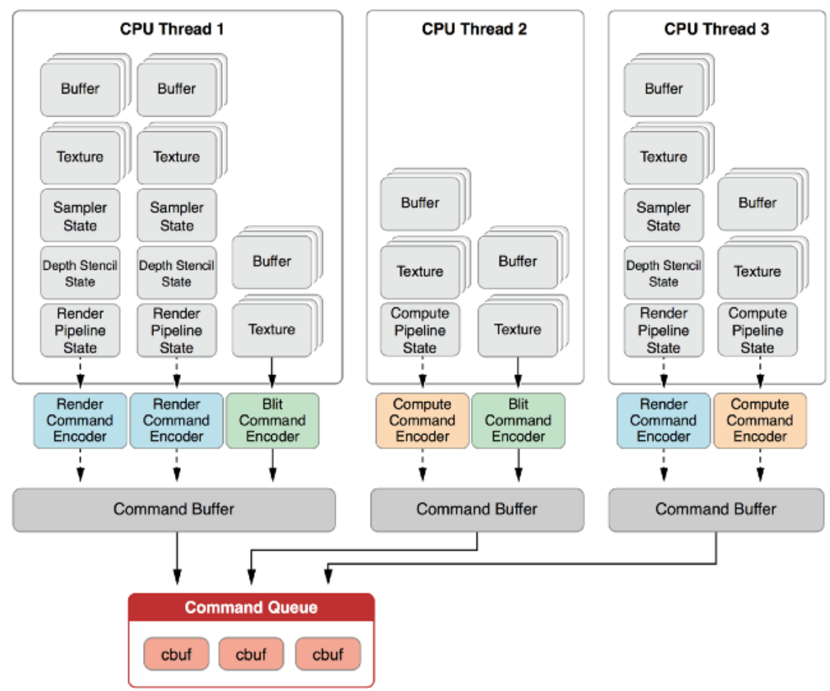
Command encoder

Put command into command buffer

Commit it to command queue

Execute the command

Get result back

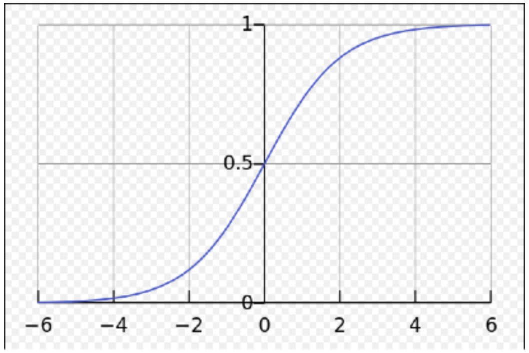


Metal Programming, Kernel Function

Compute command

- Two parameters, threadsPerGroup and numThreadgroups, determines number of threads. ➔ 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




```
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]));
}
```

Example

```swift
func initMetal() -> (MTLDevice, MTLCommandQueue, MTLLibrary, MTLCommandBuffer,
    MTLComputeCommandEncoder){
        // Get access to iPhone or iPad GPU
        var device = MTLCreateSystemDefaultDevice()

        // Queue to handle an ordered list of command buffers
        var commandQueue = device.newCommandQueue()

        // Access to Metal functions that are stored in Shaders.metal file, e.g. sigmoid()
        var defaultLibrary = device.newDefaultLibrary()

        // Buffer for storing encoded commands that are sent to GPU
        var commandBuffer = commandQueue.commandBuffer()

        // Encoder for GPU commands
        var computeCommandEncoder = commandBuffer.computeCommandEncoder()

        return (device, commandQueue, defaultLibrary!, commandBuffer, computeCommandEncoder)
}
```

```swift
// set up a compute pipeline with Sigmoid function and add it to encoder
let sigmoidProgram = defaultLibrary.newFunctionWithName("sigmoid")
var pipelineErrors = NSErrorPointer()
var computePipelineFilter = device.newComputePipelineStateWithFunction(sigmoidProgram!, error: pipelineErrors)
computeCommandEncoder.setComputePipelineState(computePipelineFilter!)
```

```swift
//   set the input vector for the Sigmoid() function, e.g. inVector
//    atIndex: 0 here corresponds to buffer(0) in the Sigmoid function
var inVectorBufferNoCopy = device.newBufferWithBytesNoCopy(memory, length: Int(size), options: nil, deallocator: nil)
computeCommandEncoder.setBuffer(inVectorBufferNoCopy, offset: 0, atIndex: 0)

// d. create the output vector for the Sigmoid() function, e.g. outVector
//    atIndex: 1 here corresponds to buffer(1) in the Sigmoid function
var outVectorBufferNoCopy = device.newBufferWithBytesNoCopy(outmemory, length: Int(size), options: nil, deallocator: nil)
computeCommandEncoder.setBuffer(outVectorBufferNoCopy, offset: 0, atIndex: 1)
```

Example

```
// hardcoded to 32 for now (recommendation: read about threadExecutionWidth)
var threadsPerGroup = MTLSize(width:32,height:1,depth:1)
var numThreadgroups = MTLSize(width:(Int(maxcount)+31)/32, height:1, depth:1)
computeCommandEncoder.dispatchThreadgroups(numThreadgroups, threadsPerThreadgroup: threadsPerGroup)
computeCommandEncoder.endEncoding()
```

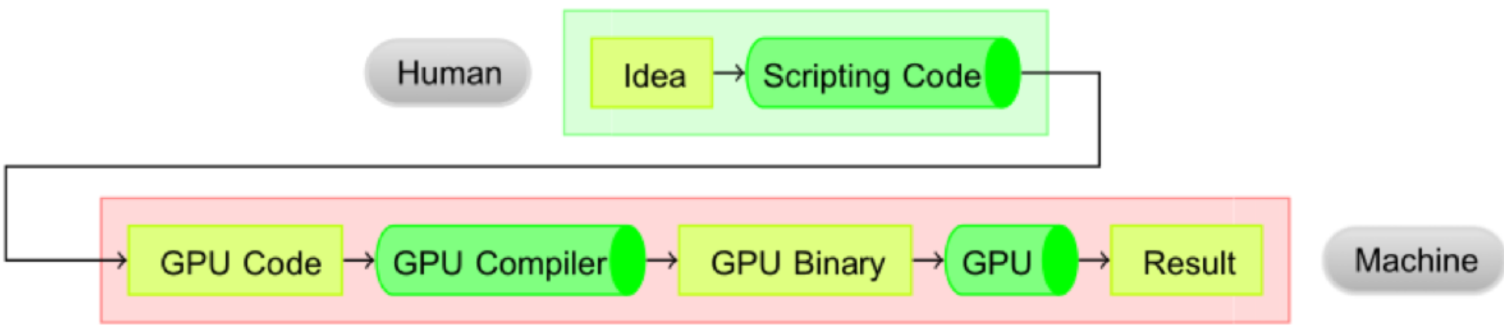
```
commandBuffer.commit()
commandBuffer.waitUntilCompleted()
```

GPU Programming with Python

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?

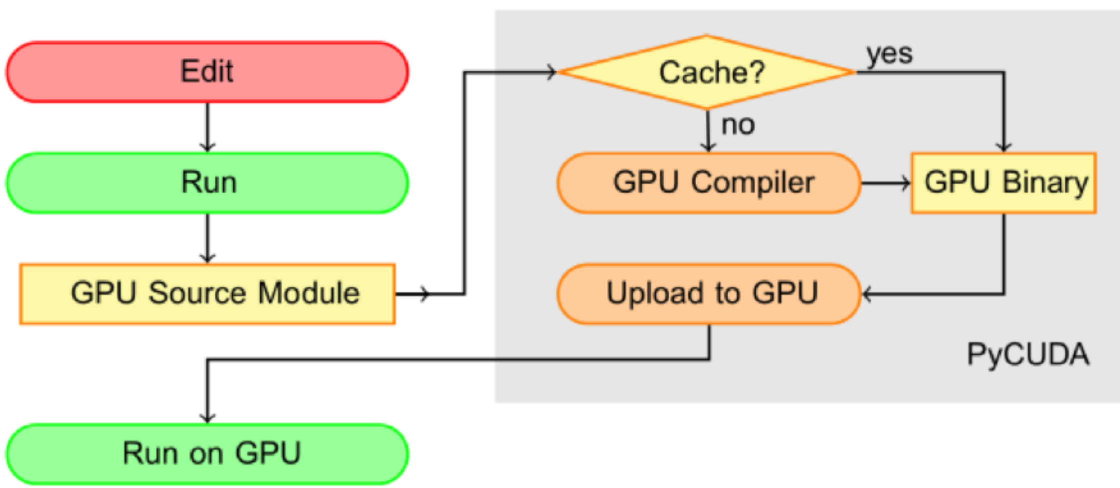
Solution



Human
Idea
Scripting Code
GPU Code
GPU Compiler
GPU Binary
GPU
Result
Machine

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.



`ndarray` - Multidimensional Arrays in Python

- 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:

```
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.

Example

```python
import atexit
import numpy as np
import pycuda.driver as drv
import pycuda.gpuarray 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 = gpuarray.to_gpu(x)
```

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.

PyCUDA Resources

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