GPU and CUDA overview

COL 730 class 2

Evolution of GPUs

RIVA 128
3M xtors

GeForce® 256
23M xtors

GeForce 3
60M xtors

GeForce FX
125M xtors

GeForce 8800
681M xtors

"Kepler"
7B xtors

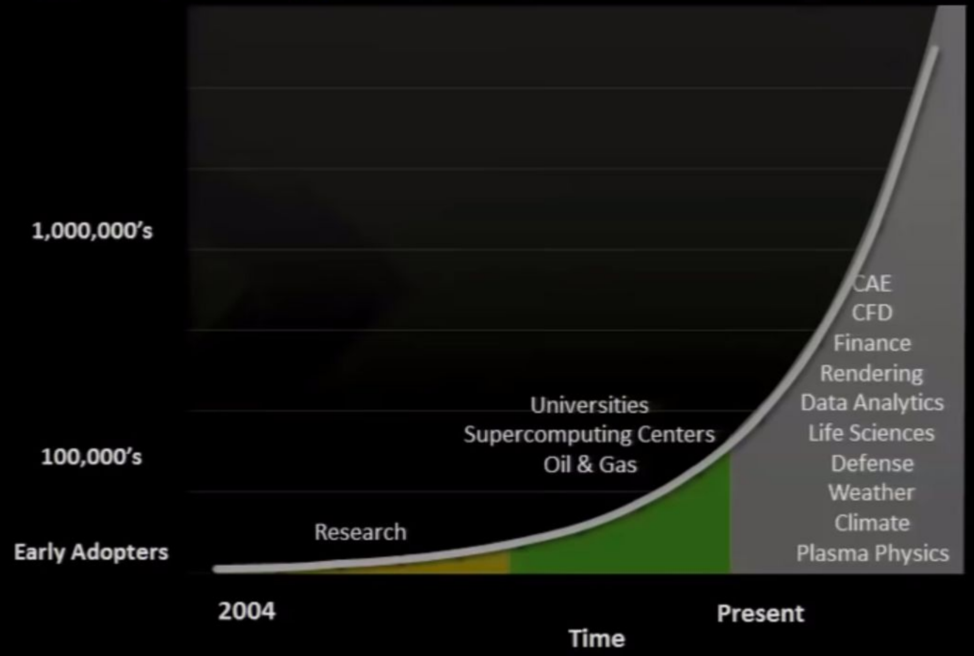
1995 2000 2005 2012

Fixed function

Programmable pixel and vertex shaders
Geforce 3 series
2001

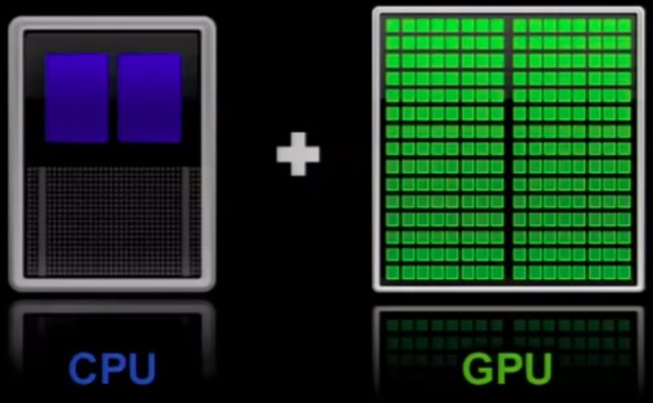
CUDA capable
Geforce 8 series
2006

GPUs Reaching Broader Set of Developers



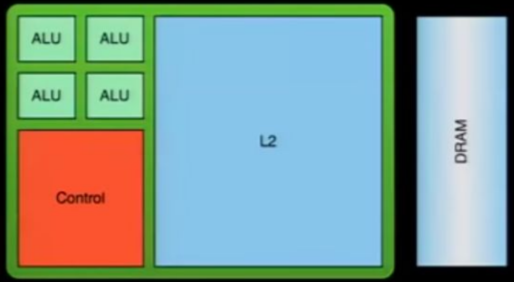
GPGPU Revolutionizes Computing

Latency Processor + Throughput processor



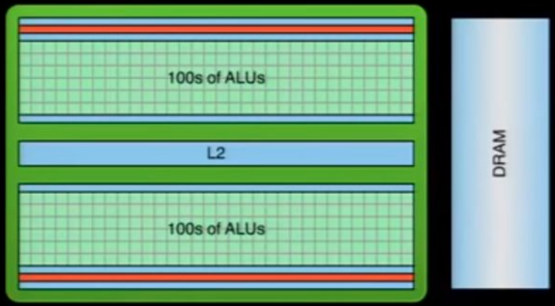
Latency and throughput

Low Latency or High Throughput?



CPU

- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

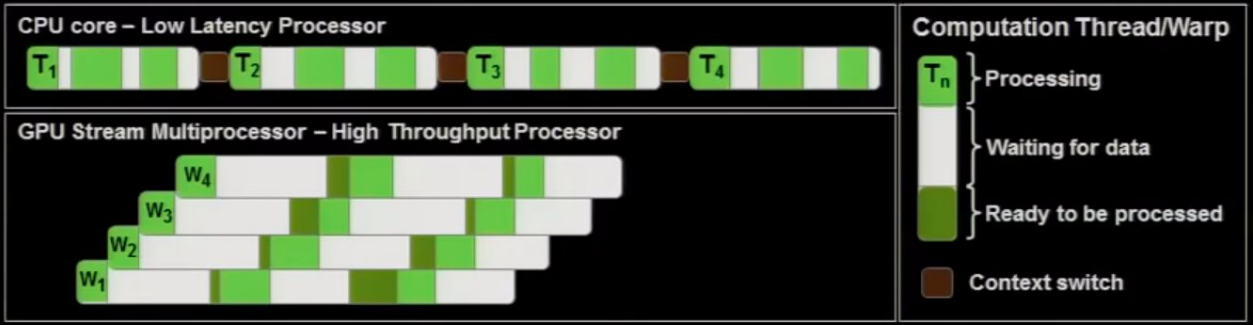


GPU

- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

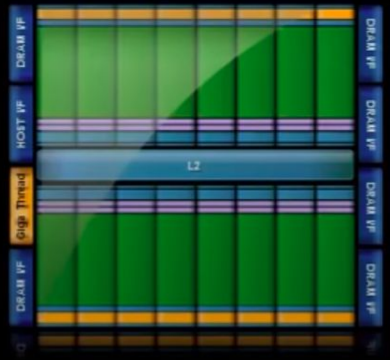
Low Latency or High Throughput?

- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other thread warps

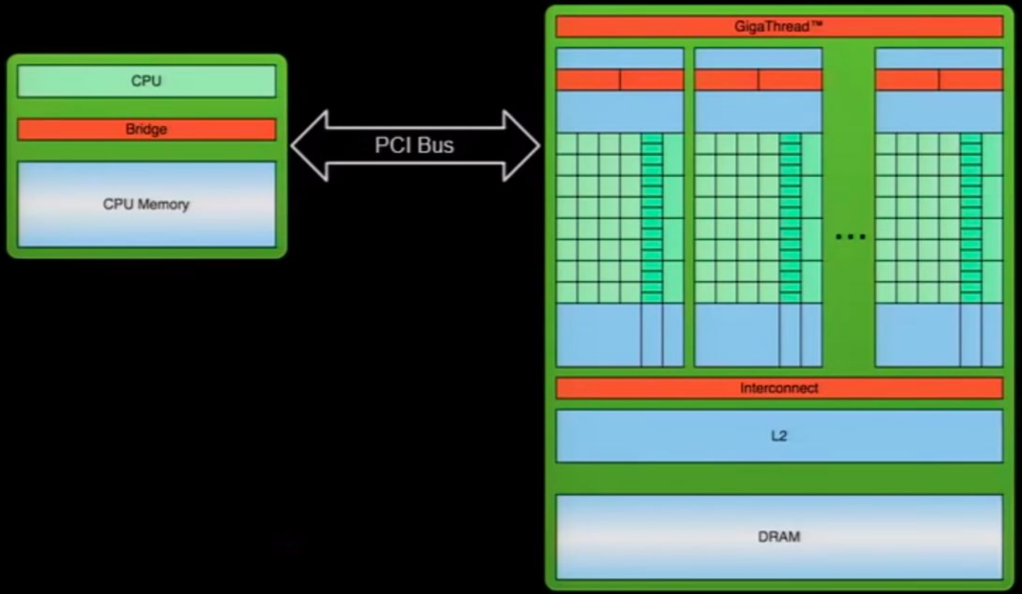


GPU Architecture:
Two Main Components

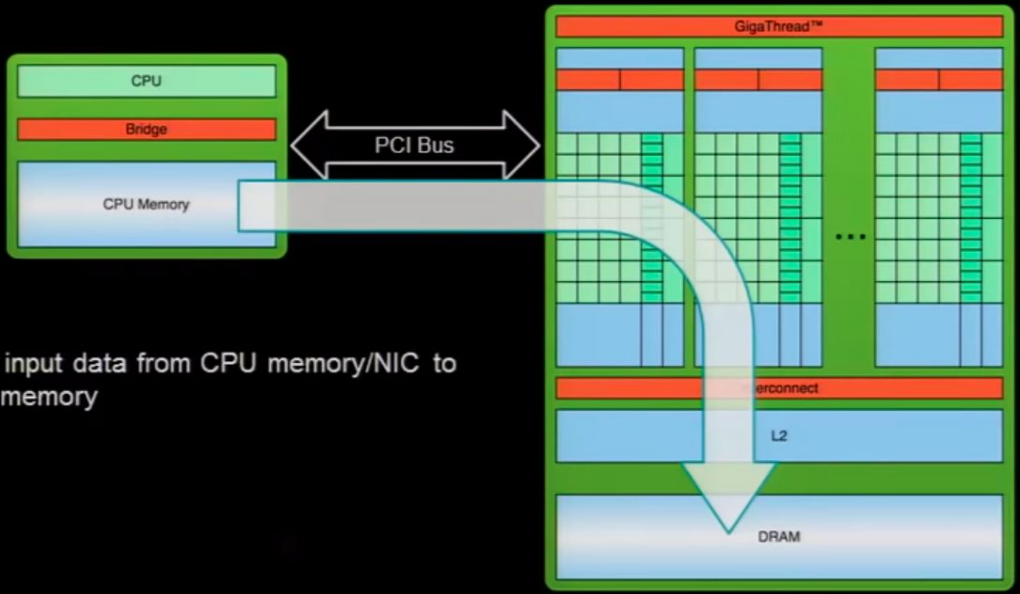
- **Global memory**
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 6 GB per GPU
  - Bandwidth currently up to ~180 GB/s (Tesla products)
  - ECC on/off (Quadro and Tesla products)
- **Streaming Multiprocessors (SMs)**
  - Perform the actual computations
  - Each SM has its own:
    - Control units, registers, execution pipelines, caches



Simple Processing Flow

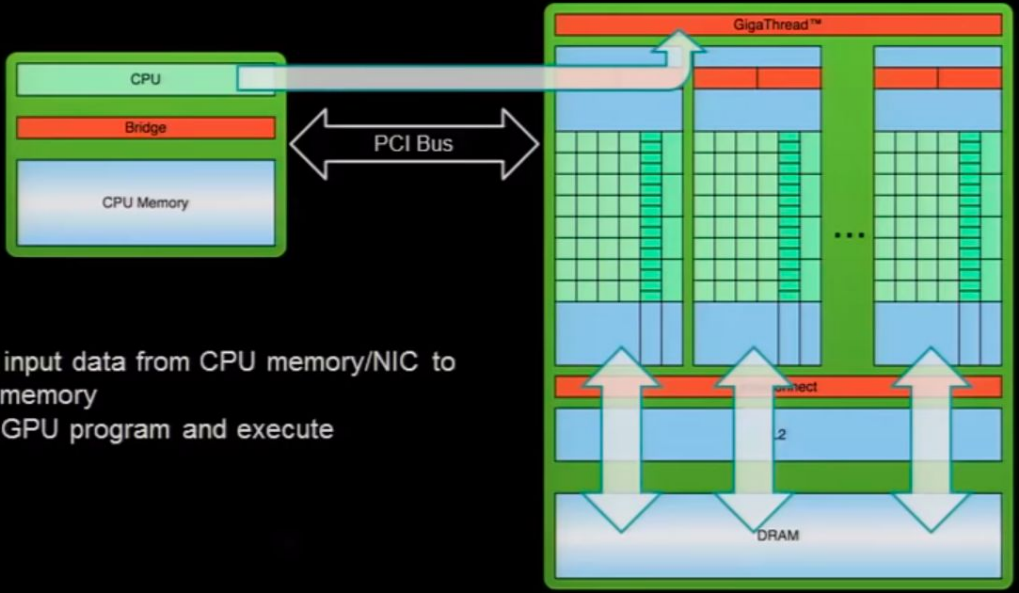


Simple Processing Flow



1. Copy input data from CPU memory/NIC to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute

Kepler GK110 Block Diagram

Architecture

- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5

GPU Architecture – Kepler GK110: Streaming Multiprocessor (SMX)

- Up to 2048 threads concurrently
  - 192 fp32 ops/clock
  - 64 fp64 ops/clock
  - 160 int32 ops/clock

- 48KB shared mem
- 64K 32-bit registers

Ways to accelerate programs with GPU

CUDA Math Libraries

High performance math routines for your applications:

- cuFFT – Fast Fourier Transforms Library
- cuBLAS – Complete BLAS Library
- cuSPARSE – Sparse Matrix Library
- cuRAND – Random Number Generation (RNG) Library
- NPP – Performance Primitives for Image & Video Processing
- Thrust – Templated C++ Parallel Algorithms & Data Structures
- math.h - C99 floating-point Library

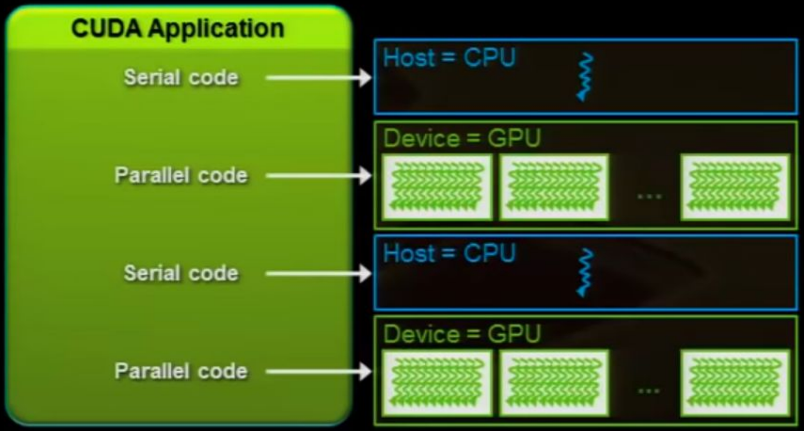
Included in the CUDA Toolkit Free download @ www.nvidia.com/getcuda

What is CUDA?

- C++ with extensions
  - Fortran support via e.g. PGI's CUDA Fortran
- CUDA goals:
  - Scale to 100's of cores, 1000's of parallel threads
  - Let programmers focus on parallel algorithms
  - Enable heterogeneous systems (i.e., CPU+GPU)
- CUDA defines:
  - Programming model
  - Memory model

Anatomy of a CUDA Application

- Serial code executes in a Host (CPU) thread
- Parallel code executes in many Device (GPU) threads across multiple processing elements



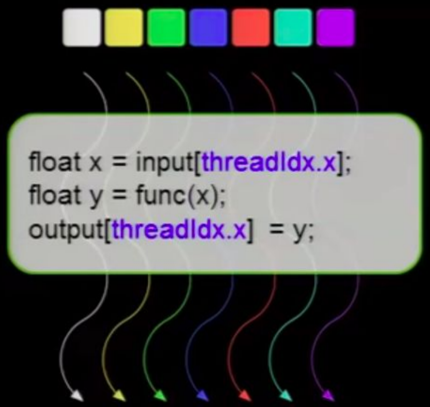
CUDA Kernels

- Parallel portion of application: execute as a kernel
  - Entire GPU executes kernel, many threads
- CUDA threads:
  - Lightweight
  - Fast switching
  - 1000s execute simultaneously

CPU	Host	Executes functions
GPU	Device	Executes kernels

CUDA Kernels: Parallel Threads

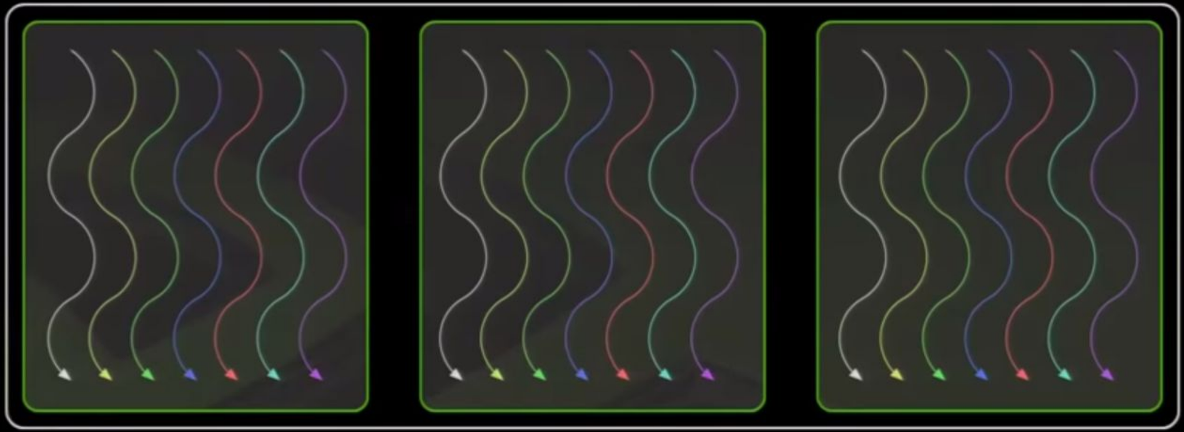
- A kernel is a function executed on the GPU as an array of threads in parallel
- All threads execute the same code, can take different paths
- Each thread has an ID
  - Select input/output data
  - Control decisions



CUDA Kernels: Subdivide into Blocks

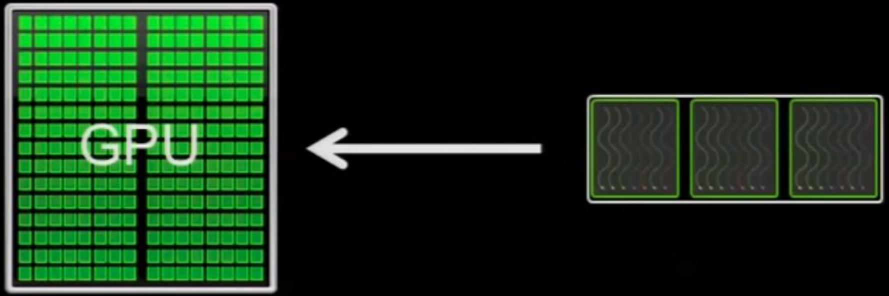
- Threads are grouped into blocks

CUDA Kernels: Subdivide into Blocks



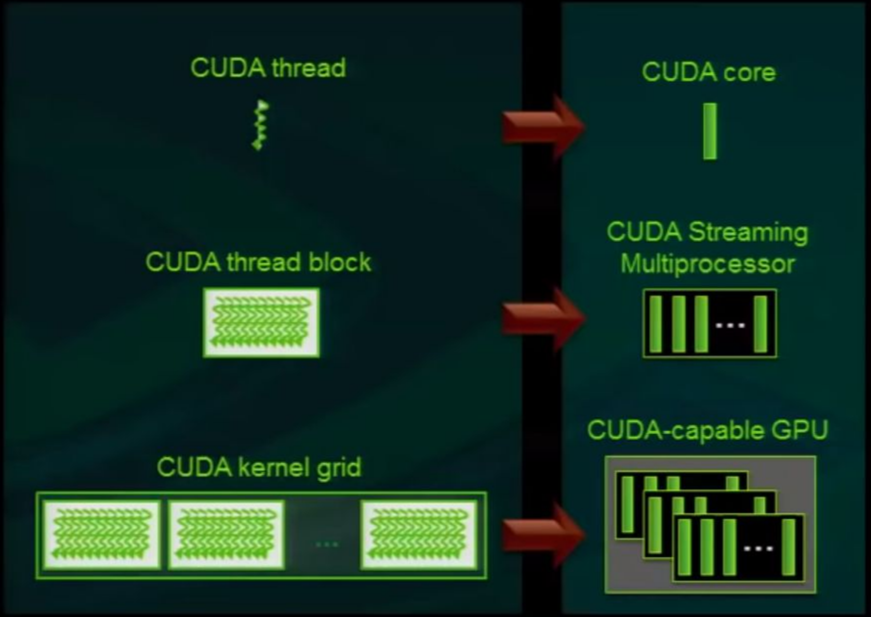
- Threads are grouped into blocks
- Blocks are grouped into a grid

CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks
  - Note: Adjacent threads execute in lock-step scheduling groupings called warps; a block comprises one or more warps
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Kernel Execution



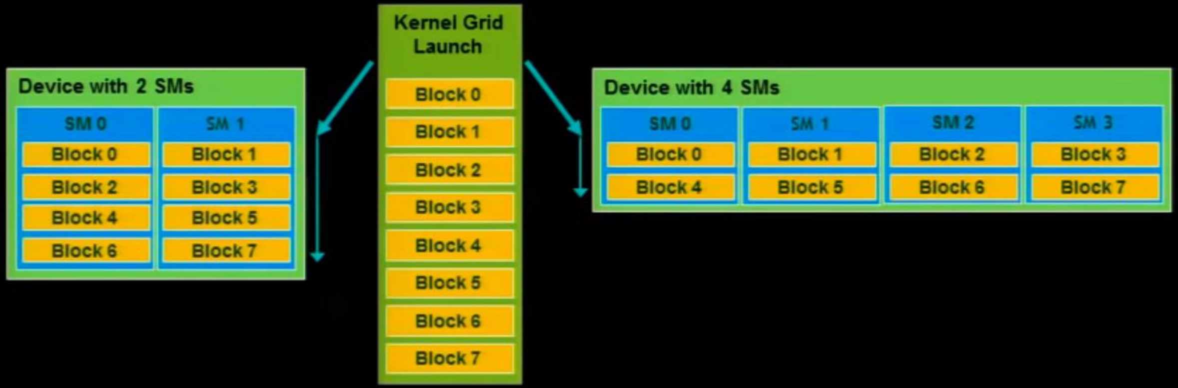
- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store blocks of memory that they all use
  - Share results with each other or cooperate to produce a single result
  - Synchronize with each other

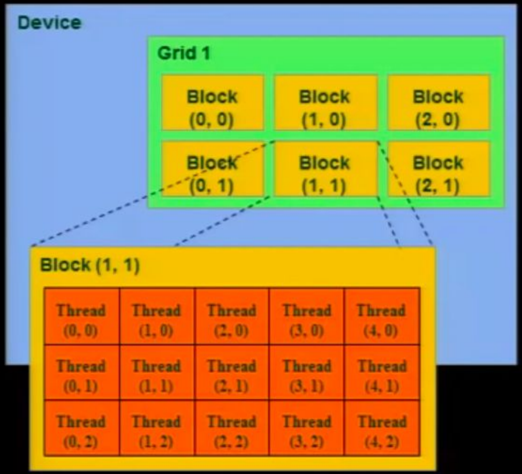
Thread blocks allow scalability

- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs



IDs and Dimensions

- Threads:
  - 3D IDs, unique within a block
- Blocks:
  - 3D IDs, unique within a grid
- Dimensions set at launch time
  - Can be unique for each section
- Built-in variables:
  - threadIdx, blockIdx
  - blockDim, gridDim



Launching kernels

- Modified C function call syntax:

```
kernel<<<dim3 grid, dim3 block>>>(…)
```

- Execution Configuration ("<<< >>>"):
  - grid dimensions: x , y, and z
  - thread-block dimensions: x, y, and z

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

Minimal Kernels

```
__global__ void minimal( int* d_a)
{
    *d_a = 13;
}

__global__ void assign( int* d_a, int value)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_a[idx] = value;
}
```

Common Pattern!

Example: Increment Array Elements

Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4 -> 4 blocks

blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

```
int idx = blockDim.x * blockId.x + threadIdx.x;
```

will map from local index threadIdx to global index

NB: blockDim should be >= 32 in real code, this is just an example

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{

        for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}



void main()
{
   .....
      increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      if (idx < N)
            a[idx] = a[idx] + b;
}


void main()
{
   .....
      dim3 dimBlock (blocksize);
      dim3 dimGrid( ceil( N / (float)blocksize) );
      increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Minimal Kernel for 2D data

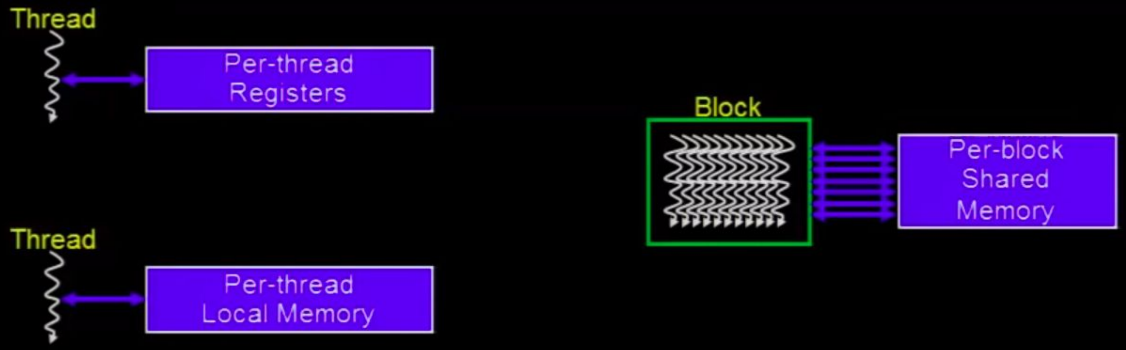
```
__global__ void assign2D(int* d_a, int w, int h, int value)
{
   int iy = blockDim.y * blockIdx.y + threadIdx.y;
   int ix = blockDim.x * blockIdx.x + threadIdx.x;
   int idx = iy * w + ix;

   d_a[idx] = value;
}
...
assign2D<<<dim3(64, 64), dim3(16, 16)>>>(...);
```

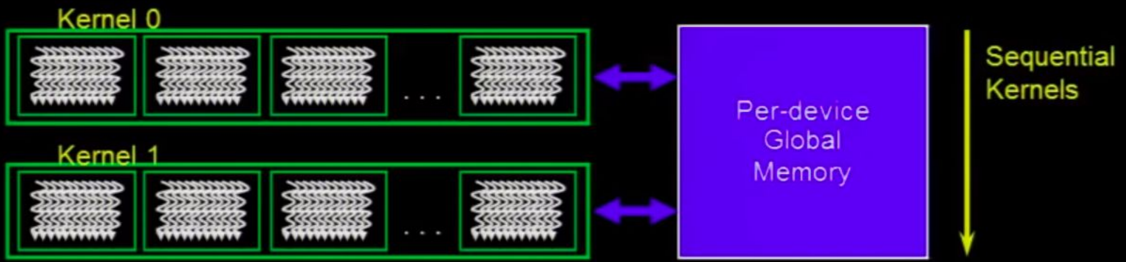
Memory Model

- Registers
  - Per thread
  - Data lifetime = thread lifetime
- Local memory
  - Per thread off-chip memory (physically in device DRAM)
  - Data lifetime = thread lifetime
- Shared memory
  - Per thread block on-chip memory
  - Data lifetime = block lifetime
- Global (device) memory
  - Accessible by all threads as well as host (CPU)
  - Data lifetime = from allocation to deallocation
- Host (CPU) memory
  - Not directly accessible by CUDA threads

Memory Model



Memory Model



Memory Model

GPU Memory Allocation / Release

- Host (CPU) manages GPU memory:
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Data Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - returns after the copy is complete
  - blocks CPU thread
  - doesn't start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
- Non-blocking memcopies are provided

Kernels: Code executed on GPU

- **C++ function with some restrictions:**
  - Can only access GPU memory (with some exceptions)
  - No variable number of arguments
  - No static variables
- **Must be declared with a qualifier:**
  - __global__ : launched by CPU,
    cannot be called from GPU
    must return void
  - __device__ : called from other GPU functions,
    cannot be launched by the CPU
  - __host__ : can be executed by CPU
  - __host__ and __device__ qualifiers can be combined
- **Built-in variables:**
  - gridDim, blockDim, blockIdx, threadIdx

Code Example 1

```
#include <stdio.h>
#define N 10

int main(int argc, char** argv){
   int vec_in[N] = {6, 1, 7, 3, 2, 9, 10, 5, 4, 8};
   int vec_out[N];
   int* d_vec;              // vector on the device

   cudaMalloc(&d_vec, N*sizeof(int));
   cudaMemcpy(d_vec, vec_in, N*sizeof(int), cudaMemcpyHostToDevice);

   cudaMemcpy(vec_out, d_vec, N*sizeof(int), cudaMemcpyDeviceToHost);

   printf("vec_out[3] = %d\n", vec_out[3]);
   cudaFree(d_msg);
   return 0;
}
```

Code Example 2

```
#include <string.h>
#include <stdio.h>
#define N 10
__global__ void kernel(int* d_vec, int n){
        int tid = threadIdx.x;
        if(threadIdx.x < n) {
          int i = d_vec[tid];
          d_vec[tid] = i >  5 ? -i : i;
         }
}
int main(int argc, char** argv){
…
        cudaMemcpy(d_msg, msg_in, N*sizeof(int), cudaMemcpyHostToDevice);
        kernel<<<1, 100>>>(d_msg, N);
        cudaMemcpy(msg_out, d_msg, N*sizeof(int), cudaMemcpyDeviceToHost);
..
}
```

Outline of CUDA Basics

- Basics to setup and execute CUDA code:
  - GPU memory management
  - Extensions to C++ for kernel code
  - GPU kernel launches
- Some additional basic features:
  - Checking CUDA errors
  - CUDA event API
- See the Programming Guide for the full API
  http://docs.nvidia.com/cuda/cuda-c-programming-guide