Module 3 Part 1
Application Design for Manycore
Introduction to CUDA

Carnegie Mellon University
18-645

Jike Chong
Ian Lane

What we discussed last time:

Fast Platforms

- Multicore platforms
- Manycore platforms
- Cloud platforms

+

Good Techniques

- Data structures
- Algorithms
- Software Architecture

- Highlighted the difference between multicore and manycore platforms
- **Exposing** concurrency in k-means, **Exploiting** parallelism by **exploring** mappings from application to platform
- Multicore platforms: OpenMP, performance metrics using Roofline model
- Manycore platforms: GPUs and CUDA

Answers you should know after this…

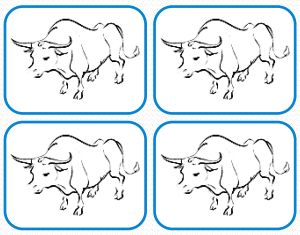
- What's the Difference between Multicore and Manycore?
- When does using a GPU make sense?
- What is the memory hierarchy inversion? And why is it there?
- What is the memory wall? How to get around it?
- Why warps?
- How do we deal with GPUs of different sizes?
- What are the implications of the thread block abstraction?
- How do threads communicate with each other?
- What is the caveat in synchronizing threads in a thread block?

Outline

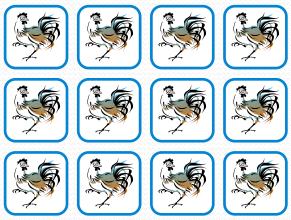
- Multicore and Manycore Differences
- Hardware and Software Mental Models
- Anatomy of an Application
- Synchronization
- The CUDA Platform

Multicore and Manycore Differences

- What's the difference between Multicore vs Manycore?



Multicore



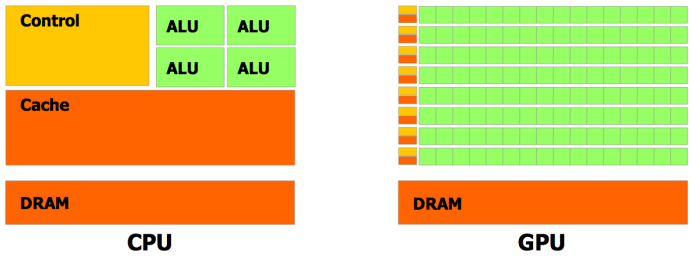
Manycore

- Multicore: yoke of oxen
- Manycore: flock of chickens

Slide by Bryan Catanzaro

Multicore and Manycore Differences

- Fundamentally different design philosophy

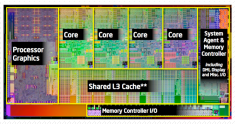


Kirk, Hwu, Programming Massively Parallel Processors, Figure 1.2

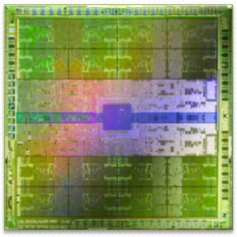
- **Multicore**: Optimized for reducing execution latency of a few threads
  - Sophisticated instruction controls, large caches
  - Each core optimized for executing a single thread
- **Manycore**: Assumes 1000-way concurrency readily available in applications
  - More resources dedicated to compute
  - Cores optimized for aggregate throughput, deemphasizing individual performance

Significant Architectural Difference

Specifications	Core i7 2600K	GTX580	
Processing Elements	4 cores, 8 way SIMD @2.5-3.4 GHz	16 cores, 16 way SIMD, dual issue @1.55 GHz	0.46x - 0.62x
Resident Threads (max)	4 cores, 2 threads, 8 width SIMD 64 strands	16 cores, 48 SIMD vectors, 32 width SIMD 24,576 strands	384x
SP GFLOP/s	160-218	1587	7.3x - 9.9x
Memory Bandwidth	21.4GB/s – 42.7GB/s	192.4 GB/s	4.5x - 9.0x
Die info	995M Transistors 32nm process 216mm²	3B Transistors 40nm process 520mm²	

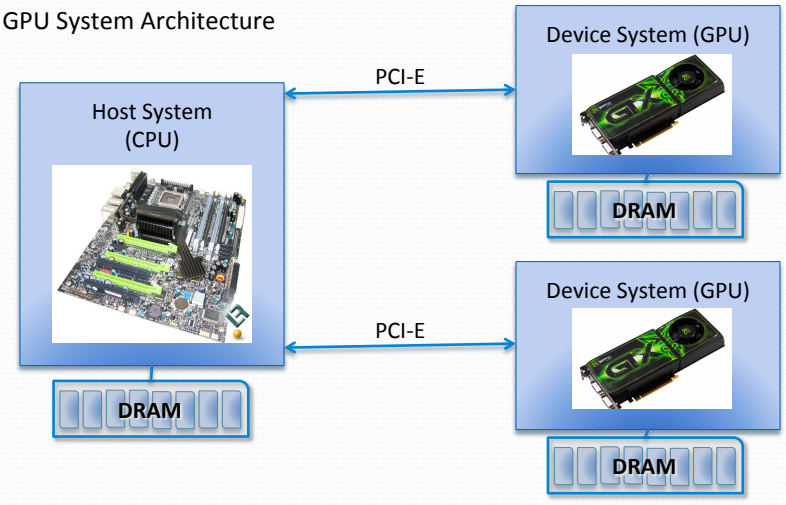


Intel Core i7-2600K


NVIDIA GTX580

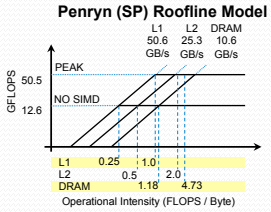
GPU is an Accelerator

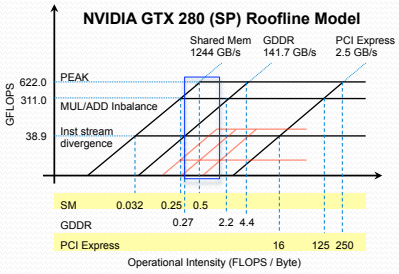
- GPU System Architecture



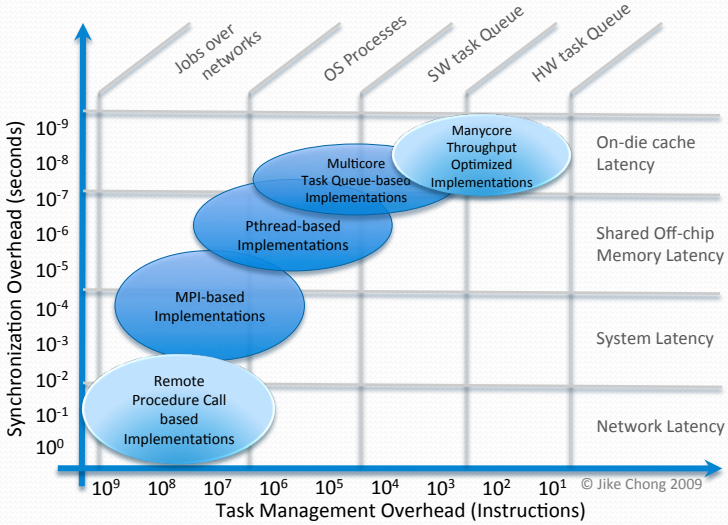
When Does Using a GPU Make Sense?

- Applications with a lot of concurrency
  - 1000-way, fine-grained concurrency
- Some memory intensive applications
  - Aggregate memory bandwidth is higher on the GPU
- ***Advantage diminishes when task granularity becomes to large to fit in shared memory***





Coarse-grained 1000-way concurrency?



Outline

- Multicore and Manycore Differences
- Hardware and Software Mental Models
- Anatomy of an Application
- Synchronization
- The CUDA Platform

Hardware and Software Mental Models

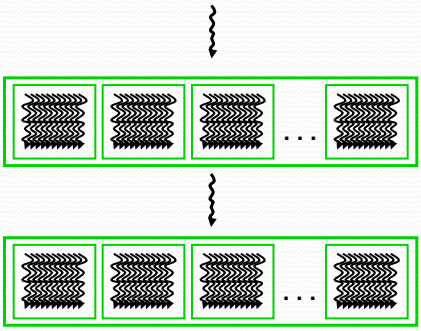
- **Mental models** → help us make design trade-offs
- Use CUDA terminologies to introduce Manycore programming
- CUDA is designed to be "functionally forgiving"
  - Easy to get correct program running
  - Can invest more time to improve performance
- Achieving good performance requires good understanding of hardware constraints

CUDA: Compute Unified Device Architecture

- A parallel computing architecture developed by NVIDIA
- Integrated host+device app C program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);



Serial Code (host)

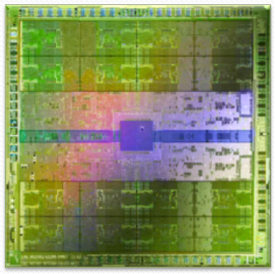
Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);

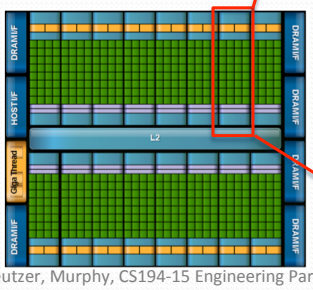
Kirk, Hwu, Programming Massively Parallel Processors, Figure 3.2

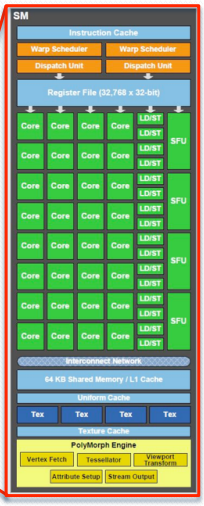
NVIDIA Fermi Architecture

- 16-way many-core, 48-way multi-threaded (warps)
- 32-wide SIMD
- 2 MB (16 x 128 KB) registers
- 1 MB (16 x 64 KB) L1 cache
- 0.75 MB L2 Cache

Why the inversion?





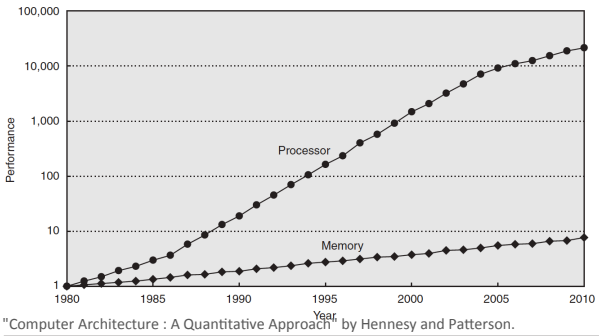


Keutzer, Murphy, CS194-15 Engineering Parallel SW, Lecture 7

Why Inversion in Mem Hierarchy?

	Task	Data	Synchronization
Latency	Time from task start to task finish (seconds)	**Time from request to receiving data (seconds)**	Time from start of synchronization to completion (seconds)
Throughput	# of tasks executed per unit time (tasks/second)	# of Bytes transferred per unit time (Bytes/second)	# of sync operations per unit time (sync ops/second)
Concurrency = (Latency * Throughput)	**# of Tasks concurrently managed (tasks)**	# of memory operations concurrently managed (Memory instructions)	# of sync operation in flight at the same time (sync operations)

Memory Wall

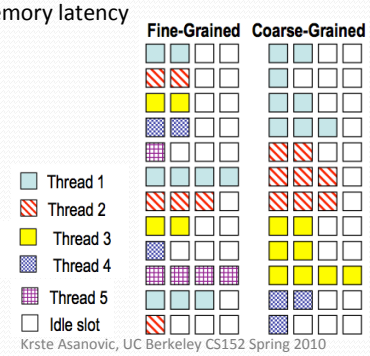


"Computer Architecture : A Quantitative Approach" by Hennesy and Patterson.

Figure 5.2 Starting with 1980 performance as a baseline, the gap in performance between memory and processors is plotted over time. Note that the vertical axis must be on a logarithmic scale to record the size of the processor-DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 5.13 on page 313). The processor line assumes a 1.25 improvement per year until 1986, and a 1.52 improvement until 2004, and a 1.20 improvement thereafter; see Figure 1.1 in Chapter 1.

Getting Around the Memory Wall

- "Memory Wall":
  Increasing gap between Processor and DRAM performance
- Manycore Processors:
  - Utilize application concurrency to hide memory latency
- How?

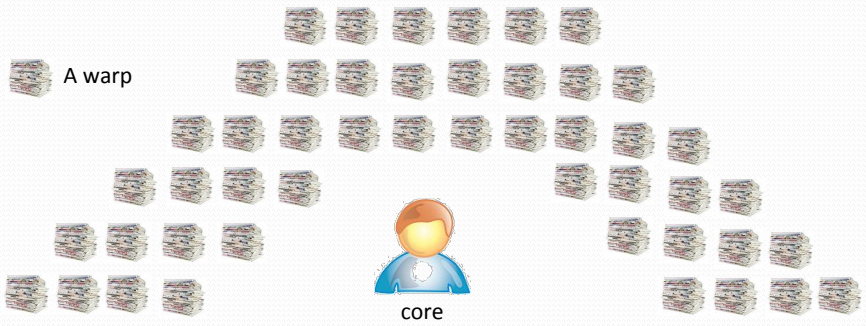


Krste Asanovic, UC Berkeley CS152 Spring 2010

Fine-Grained Multi-Threading

- Each Fermi core can maintain **48 warps** of architectural context
  - Each warp manages a 32-wide SIMD vector worth of computation
- With ~20 registers for each thread:

4 (Bytes/register) **x 20** (Registers) **x 32** (SIMD lanes) **x 48** (Warps)
➔ **128KB** per core **x 16** (core) ➔ **2MB** total of register files



Why Warps?

- **Software** abstraction to hide an extra level of architectural complexity
- A 128KB register file is a large memory
  - It takes more than one clock cycle to retrieve information
- **Hardware** provide 16-wide physical SIMD units, half-pump register files:
  - Provide half the operand per clock cycle, then the other half the following cycle
- To simplify the **programming model**:
  - Assume we are only working with 32-wide SIMD unit, where each 32-bit instruction has a bit more latency

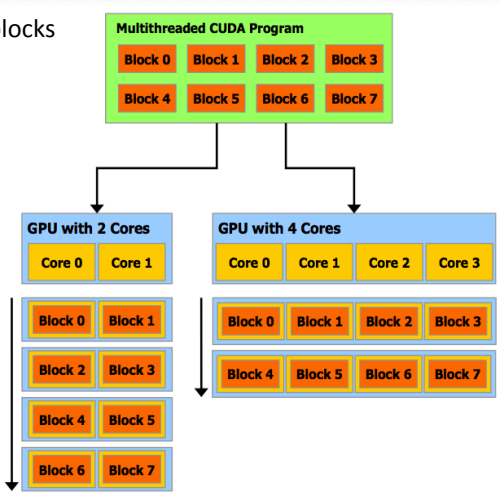
How to Deal with GPUs of Different Sizes?

	Compute Capability	Number of Multiprocessors	Number of SIMD Lanes
GeForce GTX 580	2	16	512
GeForce GTX 570, GTX 480	**2**	**15**	**480**
GeForce GTX 470	2	14	448
GeForce GTX 560 Ti	2.1	8	384
GeForce GTX 460	2.1	7	336
GeForce GTX 470M	2.1	6	288
GeForce GTX 285, GTX 280, GTX 275	1.3	30	240
GeForce GTS 450, GTX 460M	2.1	4	192
GeForce GTX 260	1.3	24	192
GeForce GT 445M	2.1	3	144
GeForce 8800 Ultra, 8800 GTX	1	16	128
GeForce 9800 GT, 8800 GT,	1.1	14	112
GeForce GT 415M	2.1	1	48

- We would like the same program to run on both GTX 580 and GT 415M
- **CUDA** provides an abstraction for concurrency to be fully **exposed**
- HW/Runtime provides capability to schedule the computation

Thread Blocks

- Computation is grouped into blocks of **independent, concurrently executable** work
- **Fully exposes** the concurrency in the application
- The HW/Runtime makes the decision to selectively sequentialize the execution as necessary
- What are some implications or limitations?



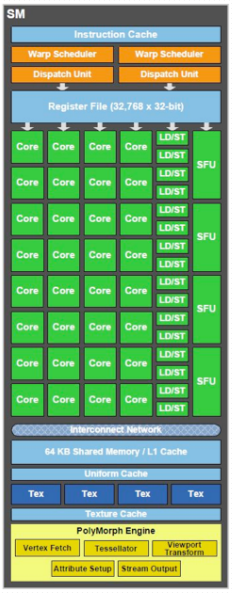
NVIDIA CUDA C Programming Guide Version 4.0, Chapter 1.3

Threads

- Threads are the computation performed in each SIMD lane in a core
  - CUDA provides a SIMT programming abstraction to assist users
- **SIMT: Single Instruction Multiple Threads**
  - A single instruction controls multiple processing element
  - Different from SIMD – **SIMD exposes the SIMD width** to the programmer
  - **SIMT abstract the # threads in a thread block** as a user-specified parameter
- SIMT enables programmers to write thread-level parallel code for
  - Independent, scalar threads
  - Data-parallel code for coordinated threads
- For functional correctness, programmers can ignore SIMT behavior
- For performance, programmers can tune applications with SIMT in mind

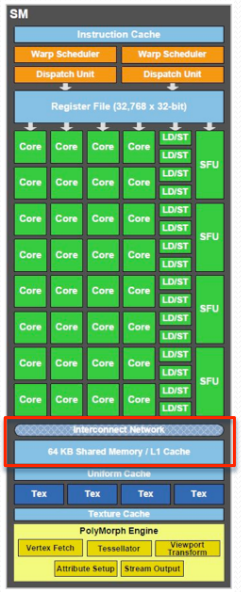
What About Data?

- SIMD (or SIMT) style programming can be **very restrictive** for communication between SIMD lanes
- On the same chip, in the same core, computations in SIMD lanes (physically) takes places very close to each other
- How can we exploit this **closeness of proximity in the hardware** while providing a **generalizable construct in software**?



Shared Memory/L1 Cache

- Manycore processors provide memory **local to each core**
- Computations in SIMD-lanes in the same core can **communicate via memory read/write**
- Two types of memory:
  - **Programmer-managed** scratch pad memory
  - **HW-managed** L1 cache
- For NVIDIA Fermi architecture, you get **64KB** per core with two configurations
  - **48KB** scratch pad (Shared Memory), **16kB** L1 cache
  - **16KB** scratch pad (Shared Memory), **48kB** L1 cache



How Many Threads per Thread Block?

- If I can efficiently communicate between threads in a thread block, **why not just put all my work in one thread block**?
- A few reasons:
  - A manycore processor has more than one core
    - If one uses only one core, one is not fully utilizing available HW
  - Hardware must maintain the context of all threads a thread block
    - There is a limited amount of resources on-chip
    - In Fermi, 48 warps of context are maintained per core
- In Fermi, each thread block can have up to 1024 threads
  - This changes between generations (used to be max of 768 threads/block)
  - One can often achieve higher performance to have **less threads/block**, but **multiple blocks concurrently running on the same core**

Hardware and Software Mental Models

- Summary of what we have discussed:

Hardware Compute Units	Software Abstractions	Memory Resources
SIMD Lanes	Threads	Register File
SIMD Execution Granularity	Warps	Registers/Shared Memory, L1 Cache
Cores (Streaming Multiprocessors – SM)	Thread Blocks	Shared Memory, L1 Cache
Multiple SMs	Grids	L2 Cache

Outline

- Multicore and Manycore Differences
- Hardware and Software Mental Models
- Anatomy of an Application
- Synchronization
- The CUDA Platform

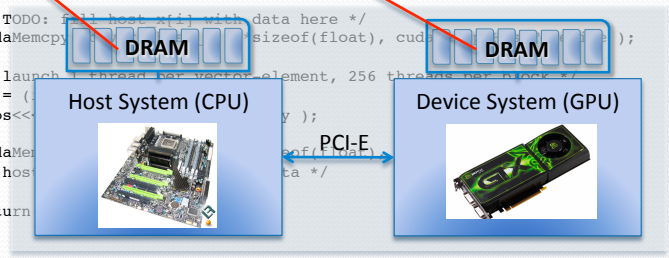
Anatomy of a CUDA Program

```
__global__ void vcos( int n, float* x, float* y ) {
     int ix = blockIdx.x*blockDim.x + threadIdx.x;
     y[ix] = cos( x[ix] );
}

int main() {
     float *host_x, *host_y; float *dev_x, *dev_y; int n = 1024;

     host_x = (float*)malloc( n*sizeof(float) );
     host_y = (float*)malloc( n*sizeof(float) );
     cudaMalloc( &dev_x, n*sizeof(float) );
     cudaMalloc( &dev_y, n*sizeof(float) );

     /* TODO: fill host_x[i] with data here */
     cudaMemcpy( dev_x, host_x, n*sizeof(float), cudaMemcpyHostToDevice );

     /* launch 1 thread per vector-element, 256 threads per block */
     bk = (int)( n / 256 );
     vcos<<<bk,256>>>( n, dev_x, dev_y );

     cudaMemcpy( host_y, dev_y, n*sizeof(float), cudaMemcpyDeviceToHost );
     /* host_y now contains cos(x) data */

     return( 0 );
}
```

Example from "Intro to CUDA Programming", by John Pormann, Ph.D. jbp1@duke.edu

Data Structure on Different Devices

```
__global__ void vcos( int n, float* x, float* y ) {
     int ix = blockIdx.x*blockDim.x + threadIdx.x;
     y[ix] = cos( x[ix] );
}

int main() {
     float *host_x, *host_y; float *dev_x, *dev_y; int n = 1024;

     host_x = (float*)malloc( n*sizeof(float) );
     host_y = (float*)malloc( n*sizeof(float) );
     cudaMalloc( &dev_x, n*sizeof(float) );
     cudaMalloc( &dev_y, n*sizeof(float) );

     /* TODO: fill host_x[i] with data here */
     cudaMemcpy( ... sizeof(float), cud ... );

     /* launch 1 thread per vector element, 256 threads per block */
     bk = (
     vcos<<<
                                          y );

     cudaMe
     /* hos
                                          ta */

     return
}
```



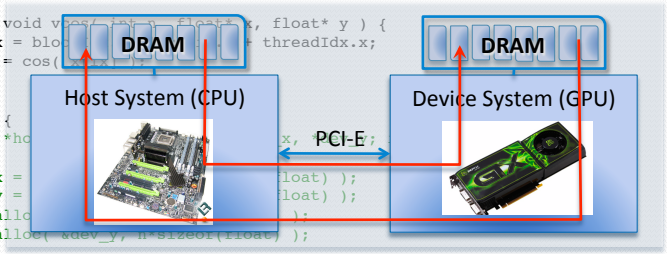
CUDA Memory Operations

- 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 *a_d = 0;
cudaMalloc( (void**)&a_d, nbytes );
cudaMemset( a_d, 0, nbytes);
cudaFree(a_d);
```

NVIDIA CUDA C Programming Guide Version 4.0, Chapter 3.3.4

Data Transfers: Host ←→ Device




```
__global__ void vcos( int n, float* x, float* y ) {
     int ix = blo                    + threadIdx.x;
     y[ix] = cos(
}

int main() {
     float *ho                                x,
     host_x =                                float) );
     host_y =                                float) );
     cudaMallo                                  );
     cudaMalloc( &dev_y, n*sizeof(float) );

     /* TODO: fill host_x[i] with data here */
     cudaMemcpy( dev_x, host_x, n*sizeof(float), cudaMemcpyHostToDevice );

     /* launch 1 thread per vector-element, 256 threads per block */
     bk = (int)( n / 256 );
     vcos<<<bk,256>>>( n, dev_x, dev_y );

     cudaMemcpy( host_y, dev_y, n*sizeof(float), cudaMemcpyDeviceToHost );
     /* host_y now contains cos(x) data */

     return( 0 );
}
```


CUDA Memory Copy Operations

```
cudaMemcpy( void *dst, void *src, size_t nbytes,
            enum cudaMemcpyKind direction);
```

- **`direction`** specifies locations (host or device) of src and dst
- Blocks CPU thread: returns after the copy is complete
- Doesn't start copying until previous CUDA calls complete
- **`enum cudaMemcpyKind`**
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`
  - `cudaMemcpyPeer`

NVIDIA CUDA C Programming Guide Version 4.0, Chapter 3.2.2, Chapter 3.2.6.5

Calling a __global__ Function

```
__global__ void vcos( int n, float* x, float* y ) {
    int ix = blockIdx.x*blockDim.x + threadIdx.x;
    y[ix] = cos( x[ix] );
}

int main()
{
    float *host_x, *host_y, *dev_x, *dev_y; int n = 1024;

    host_x = (float*)malloc( n*sizeof(float) );
    host_y = (float*)malloc( n*sizeof(float) );
    cudaMalloc( &dev_x, n*sizeof(float) );
    cudaMalloc( &dev_y, n*sizeof(float) );

    /* TODO: fill host_x[i] with data here */
    cudaMemcpy( dev_x, host_x, n*sizeof(float), cudaMemcpyHostToDevice );

    /* launch 1 thread per vector-element, 256 threads per block */
    bk = (int)( n / 256 );
    vcos<<<bk,256>>>( n, dev_x, dev_y );

    cudaMemcpy( host_y, dev_y, n*sizeof(float), cudaMemcpyDeviceToHost );
    /* host_y now contains cos(x) data */

    return( 0 );
}
```

The **__global__** qualifier declares a function as being a kernel. Such a function is:

- Executed on the device
- Callable from the host only

Built in variables:

- **blockIdx, threadIdx,** for thread self identification
- **blockDim,** where dimension of the thread block is accessible within the kernel

<<< num_blks, num_thrd >>>

CUDA Function Types

Function Type Qualifiers

- Function type qualifiers specify whether a function executes on the host or on the device and whether it is callable from the host or from the device.

__global__

- Executed on the device
- Callable from the host only
- **__global__** functions must have **void** return type
- Any call to a **__global__** function must specify its execution configuration <<< >>>
- A call to a **__global__** function is asynchronous

__device__

- Executed on the device
- Callable from the device only

__host__

- Executed on the host
- Callable from the host only

CUDA Built-in Variables

- Built-in variables specify the ***grid*** and ***block*** dimensions and the ***block and thread indices***
  - Only valid within functions that are executed on the device

gridDim

- This variable is of type **dim3** (see Section B.3.2) and contains the dimensions of the grid.

blockIdx

- This variable is of type **uint3** (see Section B.3.1) and contains the block index within the grid.

blockDim

- This variable is of type **dim3** (see Section B.3.2) and contains the dimensions of the block.

threadIdx

- This variable is of type **uint3** (see Section B.3.1) and contains the thread index within the block.

warpSize

- This variable is of type **int** and contains the warp size in threads (see Section 4.1 for the definition of a warp).

NVIDIA CUDA C Programming Guide Version 4.0, Chapter B.4

CUDA: Simple Extension of C

```
__global__ void vcos( int n, float* x, float* y ) {
     int ix = blockIdx.x*blockDim.x + threadIdx.x;
     y[ix] = cos( x[ix] );
}

int main() {
     float *host_x, *host_y; float *dev_x, *dev_y; int n = 1024;

     host_x = (float*)malloc( n*sizeof(float) );
     host_y = (float*)malloc( n*sizeof(float) );
     cudaMalloc( &dev_x, n*sizeof(float) );
     cudaMalloc( &dev_y, n*sizeof(float) );

     /* TODO: fill host_x[i] with data here */
     cudaMemcpy( dev_x, host_x, n*sizeof(float), cudaMemcpyHostToDevice );

     /* launch 1 thread per vector-element, 256 threads per block */
     bk = (int)( n / 256 );
     vcos<<<bk,256>>>( n, dev_x, dev_y );

     cudaMemcpy( host_y, dev_y, n*sizeof(float), cudaMemcpyDeviceToHost );
     /* host_y now contains cos(x) data */

     return( 0 );
}
```

CUDA: Simple Extension of C

- Type Qualifiers
  - global, device, shared, local, constant
- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads
- Runtime API
  - Memory, symbol, execution management
- Function launch

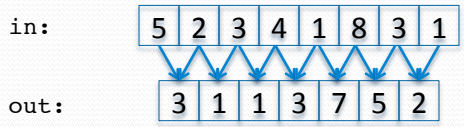
```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```


Outline

- Multicore and Manycore Differences
- Hardware and Software Mental Models
- Anatomy of an Application
- Synchronization
- The CUDA Platform

When to Use Shared Memory?

- Find absolute differences in neighboring elements in an array




```
__global__ void absolute_diff( int n, float* in, float* out) {
     int idx = blockIdx.x*256 + threadIdx.x;

     if ((idx != 0) && (idx < n)) {
          out[idx] = abs(in[idx] – in[idx-1]);
     }
}
```

- Opportunity to not load elements of **`in`** twice in the loop

Synchronization: Caveats

__syncthreads()

- waits until ***all threads in the thread block have reached this point*** and all global and shared memory accesses made by these threads prior to **__syncthreads()** are visible to all threads in the block
- used to coordinate communication between the threads of the same block

What's wrong with this code?

```
__global__ void absolute_diff( int n, float* in, float* out) {
     int idx = blockIdx.x*256 + threadIdx.x;
     __shared__ local_mem[256];

     if ((threadIdx.x != 0) && (idx < n)) {
          local_mem[threadIdx.x] = in[idx];
          __syncthread();
          out[idx] = abs(local_mem[threadIdx.x] - local_mem [threadIdx.x-1]);
     } else if (threadIdx.x == 0){
          out[idx] = abs(local_mem[threadIdx.x] – in[idx-1]);
     }
}
```

NVIDIA CUDA C Programming Guide Version 4.0, Chapter B.6

Synchronization

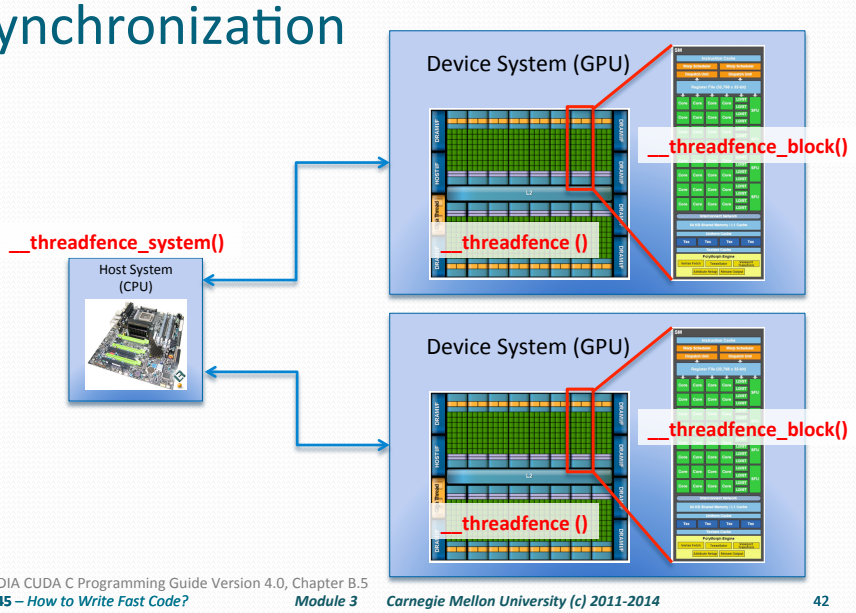
__syncthreads()

- waits until ***all threads in the thread block have reached this point*** and all global and shared memory accesses made by these threads prior to **__syncthreads()** are visible to all threads in the block
- used to coordinate communication between the threads of the same block

Corrected code:

```
__global__ void absolute_diff( int n, float* in, float* out) {
     int idx = blockIdx.x*256 + threadIdx.x;
     __shared__ local_mem[256];

     if ((threadIdx.x != 0) && (idx < n)) {
          local_mem[threadIdx.x] = in[idx];
     }
     __syncthread();
     if ((threadIdx.x != 0) && (idx < n)) {
          out[idx] = abs(local_mem[threadIdx.x] - local_mem [threadIdx.x-1]);
     } else if (threadIdx.x == 0){
          out[idx] = abs(local_mem[threadIdx.x] – in[idx-1]);
     }
}
```


Synchronization



NVIDIA CUDA C Programming Guide Version 4.0, Chapter B.5

Fence Synchronization

__threadfence_block()

- waits until all global and shared memory accesses made by the calling thread prior to **__threadfence_block()** are visible to all threads in the thread block

__threadfence()

- waits until all global and shared memory accesses made by the calling thread prior to **__threadfence()** are visible to:
  - All threads in the thread block for shared memory accesses
  - All threads in the device for global memory accesses

__threadfence_system()

- waits until all global and shared memory accesses made by the calling thread prior to **__threadfence_system()** are visible to
  - All threads in the thread block for shared memory accesses
  - All threads in the device for global memory accesses
  - Host threads for page-locked host memory accesses (see Section 3.2.4.3).
- only supported by devices of compute capability 2.x.

NVIDIA CUDA C Programming Guide Version 4.0, Chapter B.5

Atomics

- An atomic function performs a read-modify-write atomic operation a word
  - 32-bit or 64-bit word
  - Residing in global or shared memory

atomicCAS()	atomicInc()
atomicAdd()	atomicDec()
atomicSub()	atomicAnd()
atomicExch()	atomicOr()
atomicMin()	atomicXor()
atomicMax()	

NVIDIA CUDA C Programming Guide Version 4.0, Chapter B.11

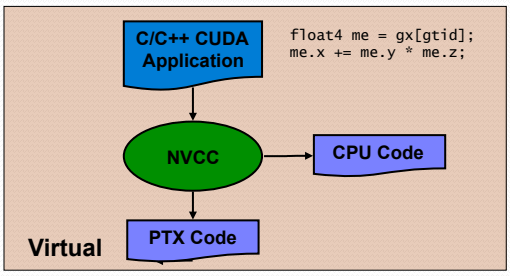
Outline

- Multicore and Manycore Differences
- Hardware and Software Mental Models
- Anatomy of an Application
- Synchronization
- The CUDA Platform

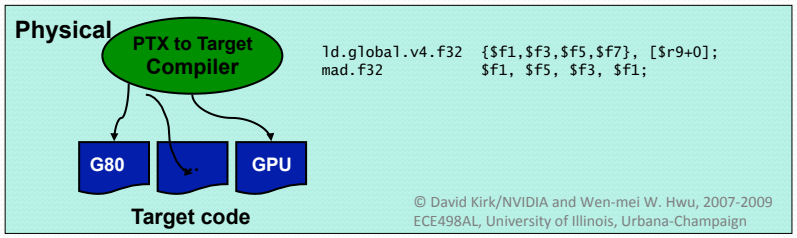
Compilation

- Any source file containing CUDA language extensions must be compiled with NVCC
  - NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC outputs:
  - C code (host CPU Code)
    - Must then be compiled with the rest of the application using another tool
  - PTX
    - Object code directly
    - Or, PTX source, interpreted at runtime

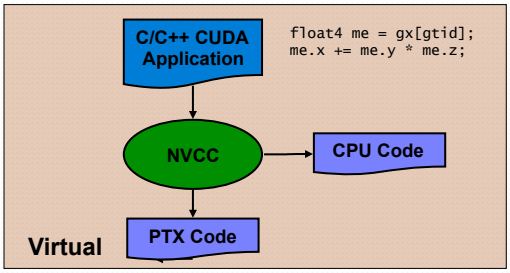
The CUDA Platform



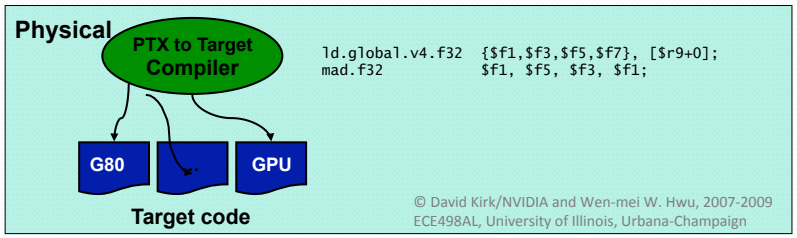
- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state



The CUDA Platform



- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state



Compilation

- Any source file containing CUDA language extensions must be compiled with NVCC
  - NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC outputs:
  - C code (host CPU Code)
    - Must then be compiled with the rest of the application using another tool
  - PTX
    - Object code directly
    - Or, PTX source, interpreted at runtime

How is this relevant to writing fast code?

Fast Platforms

- **Multicore platforms**
- **Manycore platforms**
- **Cloud platforms**

+

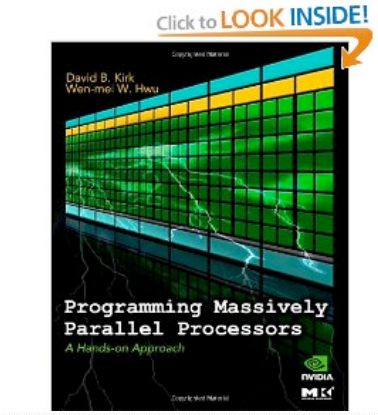
Good Techniques

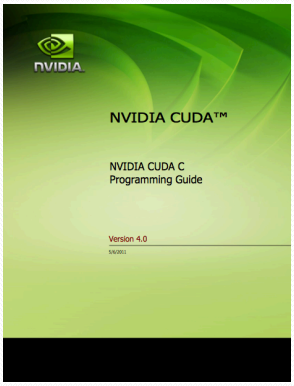
- **Data structures**
- **Algorithms**
- **Software Architecture**

- Introduced the manycore platform HW and SW mental models
- Introduced the terminologies for you to start FLIRTing with the technology

- Next part of this module:
  - Focus on the data structure, algorithms and software architecture

Resources for More Details





http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

Can You Answer These Questions Now?

- What's the Difference between Multicore and Manycore?
- When does using a GPU make sense?
- What is the memory hierarchy inversion? And why is it there?
- What is the memory wall? How to get around it?
- Why warps?
- How do we deal with GPUs of different sizes?
- What are the implications of the thread block abstraction?
- How do threads communicate with each other?
- What is the caveat in synchronizing threads in a thread block?