CS 677: Parallel Programming for Many-core Processors
Lecture 1

Instructor: Philippos Mordohai
Webpage: www.cs.stevens.edu/~mordohai
E-mail: Philippos.Mordohai@stevens.edu
Objectives

• Learn how to program massively parallel processors and achieve
  – High performance
  – Functionality and maintainability
  – Scalability across future generations

• Acquire technical knowledge required to achieve above goals
  – Principles and patterns of parallel programming
  – Processor architecture features and constraints
  – Programming API, tools and techniques
Important Points

• This is an elective course. You chose to be here.
• Expect to work and to be challenged.
• If your programming background is weak, you will probably suffer.

• This course will evolve to follow the rapid pace of progress in GPU programming. It is bound to always be a little behind...
Important Points II

• At any point ask me WHY?
• You can ask me anything about the course in class, during a break, in my office, by email.
  – If you think a homework is taking too long or is wrong.
  – If you can’t decide on a project.
Logistics

• Class webpage:
  http://www.cs.stevens.edu/~mordohai/classes/cs677_s17.html
• Office hours: Tuesdays 5-6pm and by email
• Evaluation:
  – Homework assignments (40%)
  – Quizzes (10%)
  – Midterm (15%)
  – Final project (35%)
Project

• Pick topic BEFORE middle of the semester
• I will suggest ideas and datasets, if you can’t decide
• Deliverables:
  – Project proposal
  – Presentation in class
  – Poster in CS department event
  – Final report (around 8 pages)
Project Examples

• k-means
• Perceptron
• Boosting
  – General
  – Face detector (group of 2)
• Mean Shift
• Normal estimation for 3D point clouds
More Ideas

- Look for parallelizable problems in:
  - Image processing
  - Cryptanalysis
  - Graphics
    - GPU Gems
  - Nearest neighbor search

<table>
<thead>
<tr>
<th>Version</th>
<th>Time Elapsed*</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>C# CPU Version w/ GUI and CPU-only solver</td>
<td>~900 seconds</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>C CPU Version Command-line only CPU solver</td>
<td>236.65 seconds</td>
<td>Reference</td>
<td>Reference</td>
</tr>
<tr>
<td>Kernel1 Working solver on GPU</td>
<td>16.07 seconds</td>
<td>14.73x</td>
<td>14.73x</td>
</tr>
<tr>
<td>Kernel2 Added reduction kernel</td>
<td>9.18 seconds</td>
<td>1.75x</td>
<td>25.78x</td>
</tr>
<tr>
<td>Kernel4 Changed data structure to array instead of AoS</td>
<td>8.47 seconds</td>
<td>1.00x</td>
<td>27.94x</td>
</tr>
<tr>
<td>Kernel5 Simple caching w/ shared memory</td>
<td>7.25 seconds</td>
<td>1.17x</td>
<td>32.64x</td>
</tr>
</tbody>
</table>

GPU: Shared Memory
512 Zombies
Average FPS: 45.9
Even More...

- Particle simulations
- Financial analysis
- MCMC
- Games/puzzles
Resources

• Textbook
  – (Third edition was published on Dec. 21, 2016.)

• Slides and more
  – Textbook’s companion site
    http://booksite.elsevier.com/9780124159921/
Online Resources

• NVIDIA. The NVIDIA CUDA Programming Guide.

• NVIDIA. CUDA Reference Manual.

• CUDA Toolkit
  – ...
Lecture Overview

• Scaling up computational power
• GPUs
• Introduction to CUDA
• CUDA programming model
Moore’s Law (paraphrased)

“The number of transistors on an integrated circuit doubles every two years.”

- Gordon E. Moore
Moore’s Law (Visualized)

Serial Performance Scaling is Over

• **Cannot** continue to scale processor frequencies
  – no 10 GHz chips

• **Cannot** continue to increase power consumption
  – cannot melt chip

• **Can** continue to increase transistor density
  – as per Moore’s Law
How to Use Transistors?

• Instruction-level parallelism
  – out-of-order execution, speculation, ...
  – vanishing opportunities in power-constrained world

• Data-level parallelism
  – vector units, SIMD execution, ...
  – increasing ... SSE, AVX, Cell SPE, Clearspeed, GPU

• Thread-level parallelism
  – increasing ... multithreading, multicore, manycore
  – Intel Core2, AMD Phenom, Sun Niagara, STI Cell, NVIDIA Fermi, ...
Why Massively Parallel Processing?

- A quiet revolution and potential build-up
  - Computation: TFLOPs vs. 100 GFLOPs
  - GPU in every PC - massive volume & potential impact
Why Massively Parallel Processing?

• A quiet revolution and potential build-up
  – Bandwidth: ~10x

  – GPU in every PC - massive volume & potential impact
The “New” Moore’s Law

• Computers no longer get faster, just wider

• You *must* re-think your algorithms to be parallel!

• Data-parallel computing is most scalable solution
  – Otherwise: refactor code for 2 cores 4 cores 8 cores 16 cores…
  – You will always have more data than cores - build the computation around the data
The von Neumann Model

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2011
ECE408/CS483, University of Illinois, Urbana-Champaign
Generic Multicore Chip

- Handful of processors each supporting ~1 hardware thread
- On-chip memory near processors (cache, RAM, or both)
- Shared global memory space (external DRAM)
Generic Manycore Chip

- Many processors each supporting many hardware threads
- On-chip memory near processors (cache, RAM, or both)
- Shared global memory space (external DRAM)
Enter the GPU

- Massive economies of scale
- Massively parallel
Graphics in a Nutshell

• Make great images
  – intricate shapes
  – complex optical effects
  – seamless motion

• Make them fast
  – invent clever techniques
  – use every trick imaginable
  – build monster hardware

Eugene d’Eon, David Luebke, Eric Enderton
In Proc. EGSR 2007 and GPU Gems 3
The Graphics Pipeline

- Vertex Transform & Lighting
- Triangle Setup & Rasterization
- Texturing & Pixel Shading
- Depth Test & Blending
- Framebuffer
The Graphics Pipeline

- Vertex Transform & Lighting
- Triangle Setup & Rasterization
- Texturing & Pixel Shading
- Depth Test & Blending
- Framebuffer
The Graphics Pipeline

- Vertex Transform & Lighting
- Triangle Setup & Rasterization
- Texturing & Pixel Shading
- Depth Test & Blending
- Framebuffer
The Graphics Pipeline

1. Vertex Transform & Lighting
2. Triangle Setup & Rasterization
3. Texturing & Pixel Shading
4. Depth Test & Blending
5. Framebuffer
The Graphics Pipeline

- Key abstraction of real-time graphics
- Hardware used to look like this
- One chip/board per stage
- Fixed data flow through pipeline
The Graphics Pipeline

- Everything had fixed function, with a certain number of modes
- Number of modes for each stage grew over time
- Hard to optimize HW
- Developers always wanted more flexibility
The Graphics Pipeline

- Remains a key abstraction
- Hardware used to look like this
- Vertex & pixel processing became programmable, new stages added
- GPU architecture increasingly centers around shader execution
The Graphics Pipeline

- Exposing an (at first limited) instruction set for some stages
- Limited instructions & instruction types and no control flow at first
- Expanded to full Instruction Set Architecture
Why GPUs scale so nicely

• Workload and Programming Model provide lots of parallelism

• Applications provide large groups of vertices at once
  – Vertices can be processed in parallel
  – Apply same transform to all vertices

• Triangles contain many pixels
  – Pixels from a triangle can be processed in parallel
  – Apply same shader to all pixels

• Very efficient hardware to hide serialization bottlenecks
With Moore’s Law...
More Efficiency

- Note that we do the same thing for lots of pixels/vertices

A warp = 32 threads launched together
- Usually, execute together as well
Early GPGPU

- All this performance attracted developers
- To use GPUs, re-expressed their algorithms as graphics computations
- Very tedious, limited usability
- Still had some very nice results

- This was the lead up to **CUDA**
GPU Evolution

• High throughput computation
  – GeForce GTX 280: 933 GFLOPS
  – GeForce 600 series (Kepler): 22811 GFLOPS
  – GTX Titan Z with 5760 cores: 8000 GFLOPS

• High bandwidth
  – GeForce GTX 280: 140 GB/s
  – GeForce 600 series (Kepler): 2192 GB/s
  – GTX Titan Z with 5760 cores: 672 GB/s

• High availability to all
Lessons from Graphics Pipeline

• Throughput is paramount
  – must paint every pixel within frame time
  – scalability
  – video games have strict time requirements: bare minimum: 2 Mpixels * 60 fps * 2 = 240 Mthread/s

• Create, run, & retire lots of threads very rapidly
  – measured 14.8 Gthread/s on increment() kernel (2010)

• Use multithreading to hide latency
  – 1 stalled thread is OK if 100 are ready to run
Why is this different from a CPU?

• Different goals produce different designs
  – GPU assumes work load is highly parallel
  – CPU must be good at everything, parallel or not

• CPU: minimize latency experienced by 1 thread
  – big on-chip caches
  – sophisticated control logic

• GPU: maximize throughput of all threads
  – # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
  – multithreading can hide latency => skip the big caches
  – share control logic across many threads
Design Philosophies

**GPU**
Throughput Oriented Cores

**CPU**
Latency Oriented Cores

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2012
ECE408/CS483, University of Illinois, Urbana-Champaign
CPUs: Latency Oriented Design

- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
- Powerful ALU
  - Reduced operation latency
GPUs: Throughput Oriented Design

• Small caches
  – To boost memory throughput

• Simple control
  – No branch prediction
  – No data forwarding

• Energy efficient ALUs
  – Many, long latency but heavily pipelined for high throughput

• Require massive number of threads to tolerate latencies
SMs and SPs

- **SM**: Streaming Multiprocessor
- **SP**: Streaming Processor (core)
SM Multiprocessor

- 32 CUDA Cores per SM (512 total)
- Direct load/store to memory
  - High bandwidth (Hundreds GB/sec)
- 64KB of fast, on-chip RAM
  - Software or hardware-managed
  - Shared amongst CUDA cores
  - Enables thread communication
Key Architectural Ideas

• **SIMT** (Single Instruction Multiple Thread) execution
  – threads run in groups of 32 called **warps**
  – threads in a warp share instruction unit (IU)
  – HW automatically handles divergence

• Hardware multithreading
  – HW resource allocation & thread scheduling
  – HW relies on threads to hide latency

• Threads have all resources needed to run
  – any warp not waiting for something can run
  – context switching is (basically) free
Enter CUDA

• Scalable parallel programming model

• Minimal extensions to familiar C/C++ environment

• Heterogeneous serial-parallel computing
CUDA: Scalable parallel programming

• Augment C/C++ with minimalist abstractions
  – let programmers focus on parallel algorithms
  – *not* mechanics of a parallel programming language

• Provide straightforward mapping onto hardware
  – good fit to GPU architecture
  – maps well to multi-core CPUs too

• Scale to 100s of cores & 10,000s of parallel threads
  – GPU threads are lightweight – create / switch is free
  – GPU needs 1000s of threads for full utilization
Key Parallel Abstractions in CUDA

• Hierarchy of concurrent threads

• Lightweight synchronization primitives

• Shared memory model for cooperating threads
Hierarchy of concurrent threads

• Parallel kernels composed of many threads
  – all threads execute the same sequential program

• Threads are grouped into thread blocks
  – threads in the same block can cooperate

• Threads/blocks have unique IDs
CUDA Model of Parallelism

- CUDA virtualizes the physical hardware
  - a thread is a virtualized scalar processor (registers, PC, state)
  - a block is a virtualized multiprocessor (threads, shared memory)

- Scheduled onto physical hardware without pre-emption
  - threads/blocks launch & run to completion
  - blocks should be independent
NOT: Flat Multiprocessor

- Global synchronization isn’t cheap
- Global memory access times are expensive
NOT: Distributed Processors

- Distributed computing is a different setting

- cf. BSP (Bulk Synchronous Parallel) model, MPI
Control Flow Divergence
Heterogeneous Computing

Multicore CPU

Manycore GPU
CUDA Programming Model
Overview

• CUDA programming model - basic concepts and data types

• CUDA application programming interface - basic

• Simple examples to illustrate basic concepts and functionalities

• Performance features will be covered later
CUDA - C

- 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);
CUDA Devices and Threads

• A compute **device**
  – Is a coprocessor to the CPU or **host**
  – Has its own DRAM (**device memory**)
  – Runs many **threads in parallel**
  – Is typically a **GPU** but can also be another type of parallel processing device

• **Data-parallel portions of an application are expressed as device kernels** which run on many threads

• **Differences between GPU and CPU threads**
  – GPU threads are extremely lightweight
    • Very little creation overhead
  – GPU needs 1000s of threads for full efficiency
    • Multi-core CPU needs only a few
Arrays of Parallel Threads

- A CUDA kernel is executed by a grid (array) of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions

```
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2011
ECE408/CS483, University of Illinois, Urbana-Champaign
Thread Blocks: Scalable Cooperation

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  - Threads in different blocks cannot cooperate

\[
i = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x};
\]
\[
C_d[i] = A_d[i] + B_d[i];
\]
Block IDs and Thread IDs

• Each thread uses IDs to decide what data to work on
  – Block ID: 1D or 2D
  – Thread ID: 1D, 2D, or 3D

• Simplifies memory addressing when processing multidimensional data
  – Image processing
  – Solving PDEs on volumes
  – …
CUDA Memory Model Overview

- **Global memory**
  - Main means of communicating R/W Data between **host** and **device**
  - Contents visible to all threads
  - Long latency access

- **We will focus on global memory for now**
  - Constant and texture memory will come later
CUDA API Highlights: Easy and Lightweight

- The API is an extension to the ANSI C programming language
  - Low learning curve

- The hardware is designed to enable lightweight runtime and driver
  - High performance
Extended C

• Declspecs
  – global, device, shared, local, constant

```c
__device__ float filter[N];
__global__ void convolve (float *image) {
  __shared__ float region[M];
  ...
  region[threadIdx] = image[i];
  __syncthreads()
  ...
  image[j] = result;
}
```

• Keywords
  – threadIdx, blockIdx

• Intrinsics
  – __syncthreads

• Runtime API
  – Memory, symbol, execution management

• Function launch

```c
void *myimage; cudaMalloc(myimage, bytes)

// Allocate GPU memory

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

- **cudaMalloc()**
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object

- **cudaFree()**
  - Frees object from device **Global Memory**
    - **Pointer to freed object**

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE 498AL, University of Illinois, Urbana-Champaign
CUDA Device Memory Allocation (cont.)

• Code example:
  – Allocate a 64 * 64 single precision float array
  – Attach the allocated storage to Md
  – “d” is often used to indicate a device data structure

```c
int TILE_WIDTH = 64;
float* Md;
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);
cudaMalloc((void**)&Md, size);
cudaFree(Md);
```
CUDA Host-Device Data Transfer

- `cudaMemcpy()`
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device

- Asynchronous transfer

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE 498AL, University of Illinois, Urbana-Champaign
CUDA Host-Device Data Transfer (cont.)

• Code example:
  – Transfer a 64 * 64 single precision float array
  – M is in host memory and Md is in device memory
  – cudaMemcpyHostToDevice and cudaMemcpyDeviceToDevice are symbolic constants

\[
\text{cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice)};
\]
\[
\text{cudaMemcpy(M, Md, size, cudaMemcpyDeviceToDevice)};
\]

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE 498AL, University of Illinois, Urbana-Champaign
CUDA Keywords
CUDA Function Declarations

<table>
<thead>
<tr>
<th><strong>device</strong> float DeviceFunc()</th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global</strong> void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- __global__ defines a kernel function
  - Must return void
- __device__ and __host__ can be used together
CUDA Function Declarations (cont.)

• **__device__** functions cannot have their address taken

• For functions executed on the device:
  – No recursion
    • Recursion supported since CUDA Toolkit 3.1
  – No static variable declarations inside the function
  – No variable number of arguments
Calling a Kernel Function - Thread Creation

- A kernel function must be called with an execution configuration:

```c
__global__ void KernelFunc(...);
dim3 DimGrid(100, 50);  // 5000 thread blocks
dim3 DimBlock(4, 8, 8);  // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking
**Example:** vector_addition

```c
// compute vector sum \( \mathbf{c} = \mathbf{a} + \mathbf{b} \)
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // initialization code
    ...
    // Launch \( \frac{N}{256} \) blocks of 256 threads each
    vector_add<<< N/256, 256 >>>(d_A, d_B, d_C);
}
```
Example: `vector_addition`

```c
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // initialization code
    ...
    // launch N/256 blocks of 256 threads each
    vector_add<<<N/256, 256>>>(d_A, d_B, d_C);
}
```
Example: Initialization code for `vector_addition`

// allocate and initialize host (CPU) memory
float *h_A = …, *h_B = …;

// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc((void**) &d_A, N * sizeof(float));
cudaMalloc((void**) &d_B, N * sizeof(float));
cudaMalloc((void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice);

// launch N/256 blocks of 256 threads each
vector_add<<<N/256, 256>>>(d_A, d_B, d_C);
Running Example: Matrix Multiplication

• A simple matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  – Leave shared memory usage until later
  – Local, register usage
  – Thread ID usage
  – Memory data transfer API between host and device
  – Assume square matrix for simplicity
Programming Model: Square Matrix Multiplication Example

- $P = M \times N$ of size $\text{WIDTH} \times \text{WIDTH}$
- Without tiling:
  - One thread calculates one element of $P$
  - $M$ and $N$ are loaded $\text{WIDTH}$ times from global memory

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE 498AL, University of Illinois, Urbana-Champaign
Memory Layout of a Matrix in C
Matrix Multiplication
A Simple Host Version in C

// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
Step 1: Input Matrix Data Transfer

(Host-side Code)

```c
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float *Md, *Nd, *Pd;

    ... // Allocate and Load M, N to device memory
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMemcpy(Pd, P, size, cudaMemcpyHostToDevice);
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE 498AL, University of Illinois, Urbana-Champaign
Step 3: Output Matrix Data Transfer
(Host-side Code)

2. // Kernel invocation code – to be shown later
   …

3. // Read P from the device
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

   // Free device matrices
cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
Step 2: Kernel Function

// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;

Step 2: Kernel Function (cont.)

```c
for (int k = 0; k < WIDTH; ++k) {
    float Melement = Md[threadIdx.y*WIDTH+k];
    float Nelement = Nd[k*WIDTH+threadIdx.x];
    Pvalue += Melement * Nelement;
}
Pd[threadIdx.y*WIDTH+threadIdx.x] = Pvalue;
}```
Step 2: Kernel Invocation
(Host-side Code)

// Setup the execution configuration
    dim3 dimGrid(1, 1);
    dim3 dimBlock(Width, Width);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
Only One Thread Block Used

• One Block of threads computes matrix Pd
  – Each thread computes one element of Pd

• Each thread
  – Loads a row of matrix Md
  – Loads a column of matrix Nd
  – Performs one multiply and addition for each pair of Md and Nd elements
  – Compute to off-chip memory access ratio close to 1:1 (not very high)

• Size of matrix limited by the number of threads allowed in a thread block
Handling Arbitrary Sized Square Matrices (will cover later)

- Have each 2D thread block to compute a \((\text{TILE\_WIDTH})^2\) sub-matrix (tile) of the result matrix
  - Each has \((\text{TILE\_WIDTH})^2\) threads

- Generate a 2D Grid of \((\text{WIDTH}/\text{TILE\_WIDTH})^2\) blocks

You still need to put a loop around the kernel call for cases where \(\text{WIDTH}/\text{TILE\_WIDTH}\) is greater than max grid size (64K)!

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE 498AL, University of Illinois, Urbana-Champaign
Some Useful Information on Tools

Download CUDA Toolkit 8.0
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 (Parallel Thread eXecution)
    • Object code directly
    • Or, PTX source, interpreted at runtime

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE 498AL, University of Illinois, Urbana-Champaign