Lecture Outline

- General Purpose Graphics Processing Unit (GPGPU)
- Programming model overview (SPMD, BSP)
- Hardware features (SIMT)
- Programming environment
• “General purpose graphics processing unit”
  – Harness GPU shader cores for general computation
  – Poor name: oxymoron
• History
  – GPU shader cores provide increasing FP computation throughput
  – Only accessible through graphic device driver (e.g. DirectX/OpenGL)
  – Early 2000’s “hackers” learn to write programs
  – Mid 2000’s Nvidia decides to create CUDA framework
  – ATI follows with OpenCL
• Compute Unified Device Architecture == CUDA
  – Heavily-multithreaded multiprocessor, NCC-UMA (sort of)
  – Software toolkit: C-like source/APIs => PTX binary
Speedup of Applications

- GeForce 8800 GTX vs. 2.2GHz Opteron 248
- 10× speedup in a kernel is typical, as long as the kernel can occupy enough parallel threads
- 25× to 400× speedup if the function’s data requirements and control flow suit the GPU and the application is optimized
- Keep in mind that the speedup also reflects how suitable the CPU is for executing the kernel

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
Single-Program Multiple-Data (SPMD)

- CUDA integrated CPU + GPU application C program
  - Serial C code executes on CPU
  - Parallel Kernel C code executes on GPU thread blocks

```
CPU Serial Code

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

CPU Serial Code

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

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

• A kernel is executed as a grid of thread blocks
  – All threads share data memory space

• A thread block is a batch of threads that can cooperate with each other by:
  – Synchronizing their execution
    • For hazard-free shared memory accesses
  – Efficiently sharing data through a low latency shared memory
  – Two threads from two different blocks cannot cooperate

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

- Programmer declares (Thread) Block:
  - Block size 1 to 512 concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads

- All threads in a Block execute the same thread program
- Threads have thread id numbers within Block
- Threads share data and synchronize while doing their share of the work
- Thread program uses thread id to select work and address shared data

Courtesy: John Nickolls, NVIDIA
GeForce-8 Series HW Overview

Texture Processor Cluster

Streaming Multiprocessor

- Instruction L1
- Data L1
- Instruction Fetch/Dispatch
- Shared Memory
- SFU
- SP
- SP
- SFU

Streaming Processor Array

TPC

TEX

SM

SM
CUDA Processor Terminology

- **SPA**
  - Streaming Processor Array (variable across GeForce 8-series, 8 in GeForce8800)

- **TPC**
  - Texture Processor Cluster (2 SM + TEX)

- **SM**
  - Streaming Multiprocessor (8 SP)
  - Multi-threaded processor core
  - Fundamental processing unit for CUDA thread block

- **SP**
  - Streaming Processor
  - Scalar ALU for a single CUDA thread

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
Streaming Multiprocessor (SM)

• Streaming Multiprocessor (SM)
  – 8 Streaming Processors (SP)
  – 2 Super Function Units (SFU)
• Multi-threaded instruction dispatch
  – 1 to 512 threads active
  – Shared instruction fetch per 32 threads
  – Cover latency of texture/memory loads
• 20+ GFLOPS
• 16 KB shared memory
• DRAM texture and memory access
Thread Life Cycle in HW

- Grid is launched on the SPA
- Thread Blocks are serially distributed to all the SM’s
  - Potentially >1 Thread Block per SM
- Each SM launches Warps of Threads
  - 2 levels of parallelism
- SM schedules and executes Warps that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - SPA can distribute more Thread Blocks
SM Executes Blocks

- Threads are assigned to SMs in Block granularity
  - Up to 8 Blocks to each SM as resource allows
  - SM in G80 can take up to 768 threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently
  - SM assigns/maintains thread id #s
  - SM manages/schedules thread execution
Thread Scheduling/Execution

- Each Thread Blocks is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.
CUDA Device Memory

• Each thread can:
  – R/W per-thread registers
  – R/W per-thread local memory
  – R/W per-block shared memory
  – R/W per-grid global memory
  – Read only per-grid constant memory
  – Read only per-grid texture memory

• The host can R/W global, constant, and texture memories

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
Global, Constant, and Texture Memories
(Long Latency Accesses)

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

- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads
Parallel Memory Sharing

- **Local Memory**: per-thread
  - Private per thread
  - Auto variables, register spill
- **Shared Memory**: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- **Global Memory**: per-application
  - Shared by all threads
  - Inter-Grid communication

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

- **Register File (RF)**
  - 32 KB
  - Provides 4 operands/clock
- **TEX pipe can also read/write RF**
  - 2 SMs share 1 TEX
- **Load/Store pipe can also read/write RF**

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

- There are 8192 registers in each SM in G80
  - This is an implementation decision, not part of CUDA
  - Registers are dynamically partitioned across all Blocks assigned to the SM
  - Once assigned to a Block, the register is NOT accessible by threads in other Blocks
  - Each thread in the same Block only access registers assigned to itself
Constants

- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
  - L1 per SM
- A constant value can be broadcast to all threads in a Warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block!
Shared Memory

• Each SM has 16 KB of Shared Memory
  – 16 banks of 32bit words
• CUDA uses Shared Memory as shared storage visible to all threads in a thread block
  – read and write access
• Not used explicitly for pixel shader programs
  – we dislike pixels talking to each other 😊
Parallel Memory Architecture

• In a parallel machine, many threads access memory
  – Therefore, memory is divided into banks
  – Essential to achieve high bandwidth

• Each bank can service one address per cycle
  – A memory can service as many simultaneous accesses as it has banks

• Multiple simultaneous accesses to a bank result in a bank conflict
  – Conflicting accesses are serialized
Bank Addressing Examples

- **No Bank Conflicts**
  - Linear addressing
  - stride == 1

- **No Bank Conflicts**
  - Random 1:1 Permutation
Extended C

- **Declspecs**
  - global, device, shared, local, constant

- **Keywords**
  - threadIdx, blockIdx

- **Intrinsics**
  - __syncthreads

- **Runtime API**
  - Memory, symbol, execution management

- **Function launch**

```c
__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);
```
Extended C

Integrated source
(foo.cu)

cudacc
EDG C/C++ frontend
Open64 Global Optimizer

GPU Assembly
(foo.s)

OCG

G80 SASS
(foo.sass)

CPU Host Code
(foo.cpp)

gcc / cl

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
CUDA Programming Model:
A Highly Multithreaded Coprocessor

• The GPU is viewed as a compute device that:
  – Is a coprocessor to the CPU or host
  – Has its own DRAM (device memory)
  – Runs many threads in parallel

• Data-parallel portions of an application are executed on the device as kernels which run in parallel 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
Thread Batching: Grids and Blocks

- A kernel is executed as a **grid** of thread blocks
  - All threads share data memory space
- A **thread block** is a batch of threads that can **cooperate** with each other by:
  - Synchronizing their execution
  - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency **shared memory**
- Two threads from two different blocks cannot **cooperate**
Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can 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
  - ...
A Simple Running Example
Matrix Multiplication

• A straightforward 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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
Programming Model: 
Square Matrix Multiplication Example

- \( P = M \times N \) of size \( WIDTH \times WIDTH \)
- Without tiling:
  - One thread handles one element of \( P \)
  - \( M \) and \( N \) are loaded \( WIDTH \) times from global memory

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
Step 1: Matrix Data Transfers

// Allocate the device memory where we will copy M to Matrix Md;
Md.width  = WIDTH;
Md.height = WIDTH;
Md.pitch  = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);

// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);

// Read M from the device to the host into P
cudaMemcpy(P.elements, Md.elements, size, cudaMemcpyDeviceToHost);

...// Free device memory
cudaFree(Md.elements);
Step 2: Matrix Multiplication

A Simple Host Code in C

// Matrix multiplication on the (CPU) host in double precision
// for simplicity, we will assume that all dimensions are equal

void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i)
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k];
                double b = N.elements[k * N.width + j];
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
}
Multiply Using One Thread Block

- One Block of threads compute matrix $P$
  - Each thread computes one element of $P$

- Each thread
  - Loads a row of matrix $M$
  - Loads a column of matrix $N$
  - Perform one multiply and addition for each pair of $M$ and $N$ 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
Step 3: Matrix Multiplication Host-side Main Program Code

int main(void) {
    // Allocate and initialize the matrices
    Matrix M = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix N = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix P = AllocateMatrix(WIDTH, WIDTH, 0);

    // M * N on the device
    MatrixMulOnDevice(M, N, P);

    // Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);
    return 0;
}
Step 3: Matrix Multiplication

Host-side code

// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);
    CopyToDeviceMatrix(Pd, P); // Clear memory
Step 3: Matrix Multiplication  
Host-side Code (cont.)

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

// Launch the device computation threads!  
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

// Read P from the device  
CopyFromDeviceMatrix(P, Pd);

// Free device matrices  
FreeDeviceMatrix(Md);  
FreeDeviceMatrix(Nd);  
FreeDeviceMatrix(Pd);


Step 4: Matrix Multiplication
Device-side Kernel Function

// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P) {

    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
}
Step 4: Matrix Multiplication
Device-Side Kernel Function (cont.)

for (int k = 0; k < M.width; ++k) 
{
    float Melement = M.elements[ty * M.pitch + k];
    float Nelement = Nd.elements[k * N.pitch + tx];
    Pvalue += Melement * Nelement;
}

// Write the matrix to device memory;
// each thread writes one element
P.elements[ty * P.pitch + tx] = Pvalue;

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
Step 5: Some Loose Ends

// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M)
{
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void**)&Mdevice.elements, size);
    return Mdevice;
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
Step 5: Some Loose Ends (cont.)

// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost)
{
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size,
                cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice)
{
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size,
                cudaMemcpyDeviceToHost);
}
Granularity Considerations

• For Matrix Multiplication, should I use 8X8, 16X16 or 32X32 tiles?

  — For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, it can take up to 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!

  — For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.

  — For 32X32, we have 1024 threads per Block. Not even one can fit into an SM!
What’s new since G80?

• More of everything
  – Registers, threads, blocks, SPs per SM, SMs per chip, DPFP, DRAM bandwidth & capacity

• Caches
  – Shared L2 cache
  – L1 cache vs. L1 scratchpad memory (configurable in ratios)
    • L1 caches not coherent

• Clock frequency: hasn’t changed much (power-limited)

• Fusion/APU organization: CPU & GPU on same die
  – GPU L2 vs. CPU LLC may be shared or coherent
  – DRAM shared (kernel launch need not memcpy to GPU memory)

• ???
Lecture Outline

• General Purpose Graphics Processing Unit (GPGPU)
• Programming model overview (SPMD, BSP)
• Hardware features (SIMT)
• Programming environment