Note: These slides are truncated from a longer version which is publicly available on the web
Enter the GPU

- GPU = *Graphics Processing Unit*
- Chip in computer video cards, PlayStation 3, Xbox, etc.
- Two major vendors: NVIDIA and ATI (now AMD)
Enter the GPU

- GPUs are massively multithreaded manycore chips
  - NVIDIA Tesla products have up to 128 scalar processors
  - Over 12,000 concurrent threads in flight
  - Over 470 GFLOPS sustained performance

- Users across science & engineering disciplines are achieving 100x or better speedups on GPUs

- CS researchers can use GPUs as a research platform for manycore computing: arch, PL, numeric, …
CUDA is a scalable parallel programming model and a software environment for parallel computing

- Minimal extensions to familiar C/C++ environment
- Heterogeneous serial-parallel programming model

NVIDIA’s TESLA GPU architecture accelerates CUDA

- Expose the computational horsepower of NVIDIA GPUs
- Enable general-purpose GPU computing

CUDA also maps well to multicore CPUs!
CUDA Programming Model
Heterogeneous Programming

CUDA = serial program with parallel kernels, all in C
- Serial C code executes in a host thread (i.e. CPU thread)
- Parallel kernel C code executes in many device threads across multiple processing elements (i.e. GPU threads)

<table>
<thead>
<tr>
<th>Serial Code</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Kernel KernelA (args);</td>
<td>Device</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Serial Code</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Kernel KernelB (args);</td>
<td>Device</td>
</tr>
</tbody>
</table>
Kernel = Many Concurrent Threads

- One kernel is executed at a time on the device
- Many threads execute each kernel
  - Each thread executes the same code...
  - ... on different data based on its `threadID`

CUDA threads might be
- **Physical** threads
  - As on NVIDIA GPUs
  - GPU thread creation and context switching are essentially free
- Or **virtual** threads
  - E.g. 1 CPU core might execute multiple CUDA threads

```c
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...```

© NVIDIA Corporation 2008
Hierarch of Concurrent Threads

- Threads are grouped into **thread blocks**
- **Kernel** = grid of thread blocks

By definition, threads in the same block may **synchronize with barriers**

```
scratch[threadID] = begin[threadID];
__syncthreads();
int left = scratch[threadID - 1];
```

Threads wait at the barrier until all threads in the same block reach the barrier
Transparent Scalability

- Thread blocks cannot synchronize
  - So they can run in any order, concurrently or sequentially
- This independence gives scalability:
  - A kernel scales across any number of parallel cores

Implicit barrier between dependent kernels

```c
vec_minus<<<nblocks, blksize>>>(a, b, c);
vec_dot<<<nblocks, blksize>>>(c, c);
```
Memory Hierarchy

Thread

Per-thread Local Memory

Block

Per-block Shared Memory

Sequential Kernels

Kernel 0

Kernel 1

Per-device Global Memory
Heterogeneous Memory Model

Host memory

`cudaMemcpy()`

Device 0 memory

Device 1 memory
CUDA Language: C with Minimal Extensions

Philosophy: provide minimal set of extensions necessary to expose power

Declaration specifiers to indicate where things live

__global__ void KernelFunc(...);  // kernel function, runs on device
__device__ int  GlobalVar;       // variable in device memory
__shared__ int  SharedVar;       // variable in per-block shared memory

Extend function invocation syntax for parallel kernel launch

KernelFunc<<<500, 128>>>(...);    // launch 500 blocks w/ 128 threads each

Special variables for thread identification in kernels

dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;  dim3 gridDim;

Intrinsics that expose specific operations in kernel code

__syncthreads();                  // barrier synchronization within kernel
CUDA Runtime

- **Device management:**
  - `cudaGetDeviceCount()`, `cudaGetDeviceProperties()`

- **Device memory management:**
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`

- **Graphics interoperability:**
  - `cudaGLMapBufferObject()`, `cudaD3D9MapResources()`

- **Texture management:**
  - `cudaBindTexture()`, `cudaBindTextureToArray()`
Example: Increment Array Elements

CPU program

```c
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx < N; idx++)
        a[idx] = a[idx] + b;
}
```

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

CUDA program

```c
__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;
}
```

```c
void main()
{
    ..... 
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize) );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```
Example: Increment Array Elements

Increment N-element vector a by scalar b

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

```
blockIdx.x=0
blockDim.x=4
threadIdx.x=0,1,2,3
idx=0,1,2,3

blockIdx.x=1
blockDim.x=4
threadIdx.x=0,1,2,3
idx=4,5,6,7

blockIdx.x=2
blockDim.x=4
threadIdx.x=0,1,2,3
idx=8,9,10,11

blockIdx.x=3
blockDim.x=4
threadIdx.x=0,1,2,3
idx=12,13,14,15
```

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

will map from local index threadIdx to global index

\{ Common Pattern! \}

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

// allocate host memory
unsigned int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**) &d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
More on Thread and Block 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
More on Memory Spaces

Each thread can:
- Read/write per-thread registers
- Read/write per-block shared memory
- Read/write per-grid global memory
- Most important, commonly used

Each thread can also:
- Read/write per-thread local memory
- Read only per-grid constant memory
- Read only per-grid texture memory
- Used for convenience/performance
- More details later

The host can read/write global, constant, and texture memory (stored in DRAM)
Features Available in Device Code

- **Standard mathematical functions**
  - `sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtf`, etc.

- **Texture accesses in kernels**
  ```cpp
  texture<float,2> my_texture;  // declare texture reference
  float4 texel = texfetch(my_texture, u, v);
  ```

- **Integer atomic operations in global memory**
  - `atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.
  - e.g., increment shared queue pointer with `atomicInc()`
  - Only for devices with **compute capability 1.1**
    - 1.0 = Tesla, Quadro FX5600, GeForce 8800 GTX, etc.
    - 1.1 = GeForce 8800 GT, etc.
CUDA Implementation on the GPU
CUDA is Easy and Fast

- CUDA can provide large speedups on data-parallel computations *straight out of the box*!

- Even higher speedups are achievable by understanding hardware implementation and tuning for it
  - What the rest of the presentation is about
Hardware Implementation: A Set of SIMT Multiprocessors

Each multiprocessor is a set of 32-bit processors with a Single-Instruction Multi-Thread architecture

- 16 multiprocessors on G80
- 8 processors per multiprocessors

At each clock cycle, a multiprocessor executes the same instruction on a group of threads called a warp

- The number of threads in a warp is the warp size (= 32 threads on G80)
- A half-war is the first or second half of a warp
The global, constant, and texture spaces are regions of device memory.

Each multiprocessor has:

- A set of 32-bit registers per processor (8192 on G80)
- **On-chip shared memory** (16 K on G80)
  - Where the shared memory space resides
- A read-only **constant cache**
  - To speed up access to the constant memory space
- A read-only **texture cache**
  - To speed up access to the texture memory space
Hardware Implementation: Execution Model

- Each multiprocessor processes batches of blocks one batch after the other
  - **Active blocks** = the blocks processed by one multiprocessor in one batch
  - **Active threads** = all the threads from the active blocks
- The multiprocessor’s registers and shared memory are split among the active threads
- Therefore, for a given kernel, the number of active blocks depends on:
  - The number of registers the kernel compiles to
  - How much shared memory the kernel requires
- If there cannot be at least one active block, the kernel fails to launch
Each active block is split into warps in a well-defined way

Warps are time-sliced

In other words:
- Threads within a warp are executed \textit{physically} in parallel
- Warps and blocks are executed \textit{logically} in parallel
Host Synchronization

All kernel launches are asynchronous
- control returns to CPU immediately
- kernel executes after all previous CUDA calls have completed

cudaMemcpy is synchronous
- control returns to CPU after copy completes
- copy starts after all previous CUDA calls have completed

cudaThreadSynchronize()
- blocks until all previous CUDA calls complete
Multiple CPU Threads and CUDA

CUDA resources allocated by a CPU thread can be consumed only by CUDA calls from the same CPU thread

Violation Example:
- CPU thread 2 allocates GPU memory, stores address in $p$
- thread 3 issues a CUDA call that accesses memory via $p$
Memory Latency and Bandwidth

Host memory
- Device ↔ host memory bandwidth is 4 GB/s peak (PCI-express x16)
- Test with SDK’s bandwidthTest

Global/local device memory
- High latency, not cached
- 80 GB/s peak, 1.5 GB (Quadro FX 5600)

Shared memory
- On-chip, low latency, very high bandwidth, 16 KB
- Like a user-managed per-multiprocessor cache

Texture memory
- Read-only, high latency, cached

Constant memory
- Read-only, low latency, cached, 64 KB
Performance Optimization

- Expose as much parallelism as possible
- Optimize memory usage for maximum bandwidth
- Maximize occupancy to hide latency
- Optimize instruction usage for maximum throughput
Expose Parallelism: GPU Thread Parallelism

- Structure algorithm to maximize independent parallelism
- If threads of same block need to communicate, use shared memory and __syncthreads()
- If threads of different blocks need to communicate, use global memory and split computation into multiple kernels
  - No synchronization mechanism between blocks
- High parallelism is especially important to hide memory latency by overlapping memory accesses with computation
Optimize Memory Usage: Basic Strategies

- Processing data is cheaper than moving it around
  - Especially for GPUs as they devote many more transistors to ALUs than memory
- And will be increasingly so
  - The less memory bound a kernel is, the better it will scale with future GPUs
- So you want to:
  - Maximize use of low-latency, high-bandwidth memory
  - Optimize memory access patterns to maximize bandwidth
  - Leverage parallelism to hide memory latency by overlapping memory accesses with computation as much as possible
    - Kernels with high arithmetic intensity (ratio of math to memory transactions)
  - Sometimes recompute data rather than cache it
Minimize CPU ↔ GPU Data Transfers

CPU ↔ GPU memory bandwidth much lower than GPU memory bandwidth

- Use page-locked host memory (`cudaMallocHost()`) for maximum CPU ↔ GPU bandwidth
  - 3.2 GB/s common on PCI-e x16
  - ~4 GB/s measured on nForce 680i motherboards (8GB/s for PCI-e 2.0)
  - Be cautious however since allocating too much page-locked memory can reduce overall system performance

Minimize CPU ↔ GPU data transfers by moving more code from CPU to GPU

- Even if that means running kernels with low parallelism computations
- Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to CPU memory

Group data transfers

- One large transfer much better than many small ones
Optimize Memory Access Patterns

Effective bandwidth can vary by an order of magnitude depending on access pattern

Optimize access patterns to get:
- **Coalesced** global memory accesses
- Shared memory accesses with *no or few bank conflicts*
- **Cache-efficient** texture memory accesses
- **Same-address** constant memory accesses
Global Memory Reads/Writes

- Global memory is not cached on G8x
- Highest latency instructions: 400-600 clock cycles
- Likely to be performance bottleneck
- Optimizations can greatly increase performance
Coalesced Global Memory Accesses

The simultaneous global memory accesses by each thread of a half-warp (16 threads on G80) during the execution of a single read or write instruction will be coalesced into a single access if:

- The size of the memory element accessed by each thread is either 4, 8, or 16 bytes
- The elements form a contiguous block of memory
- The N<sup>th</sup> element is accessed by the N<sup>th</sup> thread in the half-warp
- The address of the first element is aligned to 16 times the element’s size

Coalescing happens even if some threads do not access memory (divergent warp)
Coalesced Global Memory Accesses

Coalesced float memory access

Coalesced float memory access (divergent warp)
Non-Coalesced Global Memory Accesses

Non-sequential `float` memory access

Misaligned starting address
Non-Coalesced Global Memory Accesses

Non-contiguous float memory access

Non-coalesced float3 memory access

© NVIDIA Corporation 2008
Coalescing: Timing Results

Experiment:
- Kernel: read a float, increment, write back
- 3M floats (12MB)
- Times averaged over 10K runs

12K blocks x 256 threads:
- 356µs – coalesced
- 357µs – coalesced, some threads don’t participate
- 3,494µs – permuted/misaligned thread access

4K blocks x 256 threads:
- 3,302µs – float3 non-coalesced

Conclusion:
- Coalescing greatly improves throughput!
- Critical to small or memory-bound kernels
Maximize Use of Shared Memory

- Shared memory is hundreds of times faster than global memory
- Threads can cooperate via shared memory
  - Not so via global memory
- A common way of scheduling some computation on the device is to **block it up** to take advantage of shared memory:
  - Partition the data set into data subsets that fit into shared memory
  - Handle **each data subset with one thread block**:
    - Load the subset from global memory to shared memory
    - __syncthreads()
    - Perform the computation on the subset from shared memory
      - each thread can efficiently multi-pass over any data element
    - __syncthreads() (if needed)
    - Copy results from shared memory to global memory
Example: Square Matrix Multiplication

- $C = A \cdot B$ of size $N \times N$
- Without blocking:
  - One thread handles one element of $C$
  - $A$ and $B$ are loaded $N$ times from global memory

- Wastes bandwidth
- Poor balance of work to bandwidth
Example:
Square Matrix Multiplication Example

- **C = A \cdot B of size N \times N**
- **With blocking:**
  - One **thread block** handles one $M \times M$ sub-matrix $C_{sub}$ of $C$.
  - **A and B are only loaded** $(N / M)$ **times from global memory**
- **Much less bandwidth**
- **Much better balance of work to bandwidth**
Maximize Occupancy to Hide Latency

Sources of latency:
- Global memory access: 400-600 cycle latency
- Read-after-write register dependency
  - Instruction’s result can only be read 11 cycles later
- Latency blocks dependent instructions in the same thread
- But instructions in other threads are not blocked
- Hide latency by running as many threads per multiprocessor as possible!
- Choose execution configuration to maximize occupancy
  \[ \text{occupancy} = \frac{\# \text{ of active warps}}{\text{maximum } \# \text{ of active warps}} \]
  - Maximum \# of active warps is 24 on G8x
Execution Configuration: Constraints

- Maximum # of threads per block: 512
- # of active threads limited by resources:
  - # of registers per multiprocessor (register pressure)
  - Amount of shared memory per multiprocessor

Use --maxrregcount=N flag to NVCC
- N = desired maximum registers / kernel
- At some point “spilling” into LMEM may occur
  - Reduces performance – LMEM is slow
  - Check .cubin file for LMEM usage
Main performance concern with branching is **divergence**
- Threads within a single warp take different paths
- Different execution paths must be serialized

Avoid divergence when branch condition is a function of thread ID
- Example with divergence:
  ```c
  if (threadIdx.x > 2) { }
  ``
  Branch granularity < warp size
- Example without divergence:
  ```c
  if (threadIdx.x / WARP_SIZE > 2) { }
  ``
  Branch granularity is a whole multiple of warp size
Shared Memory Implementation: Banked Memory

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
Shared Memory Is Banked

- Bandwidth of each bank is 32 bits per 2 clock cycles
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
  - No bank conflicts between different half-warps, only within a single half-warp
Bank Addressing Examples

No bank conflicts
- Linear addressing
  stride == 1

No bank conflicts
- Random 1:1 permutation

© NVIDIA Corporation 2008
Bank Addressing Examples

2-way bank conflicts
- Linear addressing
  stride == 2

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4

Thread 8
Thread 9
Thread 10
Thread 11

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 15

8-way bank conflicts
- Linear addressing
  stride == 8

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 15

Bank 0
Bank 1
Bank 2

Bank 7
Bank 8
Bank 9
Bank 15
**Shared Memory Bank Conflicts**

- **Shared memory is as fast as registers if there are no bank conflicts**

- **The fast case:**
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp read the same word, there is no bank conflict (*broadcast*)

- **The slow case:**
  - Bank conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank
Conclusion

CUDA is a powerful parallel programming model
- Heterogeneous - mixed serial-parallel programming
- Scalable - hierarchical thread execution model
- Accessible - minimal but expressive changes to C

CUDA on GPUs can achieve great results on data-parallel computations with a few simple performance optimization strategies:
- Structure your application and select execution configurations to maximize exploitation of the GPU’s parallel capabilities
- Minimize CPU ↔ GPU data transfers
- Coalesce global memory accesses
- Take advantage of shared memory
- Minimize divergent warps
- Minimize use of low-throughput instructions
- Avoid shared memory accesses with high degree of bank conflicts