Tutorial CUDA

Cyril Zeller
NVIDIA Developer Technology
Overview

- Introduction and motivation
  - GPU computing: the democratization of parallel computing
  - Why GPUs?
- CUDA programming model, language, and runtime
- Break
- CUDA implementation on the GPU
  - Execution model
  - Memory architecture and characteristics
  - Floating-point features
  - Optimization strategies
    - Memory coalescing
    - Use of shared memory
    - Instruction performance
    - Shared memory bank conflicts
- Q&A
GPU Computing: The Democratization of Parallel Computing
Parallel Computing’s Golden Age

1980s, early ‘90s: a golden age for parallel computing
   Particularly data-parallel computing

Architectures
   Connection Machine, MasPar, Cray
   True supercomputers: incredibly exotic, powerful, expensive

Algorithms, languages, & programming models
   Solved a wide variety of problems
   Various parallel algorithmic models developed
   P-RAM, V-RAM, circuit, hypercube, etc.
Parallel Computing’s Dark Age

But…impact of data-parallel computing limited
- Thinking Machines sold 7 CM-1s (100s of systems total)
- MasPar sold ~200 systems

Commercial and research activity subsided
- Massively-parallel machines replaced by clusters of ever-more powerful commodity microprocessors
- Beowulf, Legion, grid computing, …

Massively parallel computing lost momentum to the inexorable advance of commodity technology
Illustrated History of Parallel Computing
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, ...
Enter CUDA

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!
The Democratization of Parallel Computing

GPU Computing with CUDA brings data-parallel computing to the masses
- Over 46,000,000 CUDA-capable GPUs sold
- A “developer kit” costs ~$200 (for 500 GFLOPS)

Data-parallel supercomputers are everywhere!
- CUDA makes this power accessible
- We’re already seeing innovations in data-parallel computing

Massively parallel computing has become a commodity technology!
Why GPUs?

45X

17X

100X

13–457x

110-240X

35X
GPUs Are Fast

- Theoretical peak performance: 518 GFLOPS

- Sustained μbenchmark performance:
  - Raw math: 472 GFLOPS (8800 Ultra)
  - Raw bandwidth: 80 GB per second (Tesla C870)

- Actual application performance:
  - Molecular dynamics: 290 GFLOPS (VMD ion placement)
GPUs Are Getting Faster, Faster
G80 (launched Nov 2006 – GeForce 8800 GTX)
128 Thread Processors execute kernel threads
Up to 12,288 parallel threads active
Per-block shared memory (PBSM) accelerates processing
CUDA Programming Model
Some Design Goals

- **Enable heterogeneous systems** (i.e., CPU+GPU)
  - CPU & GPU are separate devices with separate DRAMs

- **Scale** to 100’s of cores, 1000’s of parallel threads

- Let programmers focus on parallel algorithms
  - *not* mechanics of a parallel programming language
  - Use C/C++ with minimal extensions
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)
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
Hierarchy 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**

```c
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

Device 0 memory

cudaMemcpy()

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
```

© NVIDIA Corporation 2008
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;
}

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;
}

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

\[
\begin{align*}
\text{blockIdx.x} &= 0 \\
\text{blockDim.x} &= 4 \\
\text{threadIdx.x} &= 0, 1, 2, 3 \\
\text{idx} &= 0, 1, 2, 3
\end{align*}
\]

\[
\begin{align*}
\text{blockIdx.x} &= 1 \\
\text{blockDim.x} &= 4 \\
\text{threadIdx.x} &= 0, 1, 2, 3 \\
\text{idx} &= 4, 5, 6, 7
\end{align*}
\]

\[
\begin{align*}
\text{blockIdx.x} &= 2 \\
\text{blockDim.x} &= 4 \\
\text{threadIdx.x} &= 0, 1, 2, 3 \\
\text{idx} &= 8, 9, 10, 11
\end{align*}
\]

\[
\begin{align*}
\text{blockIdx.x} &= 3 \\
\text{blockDim.x} &= 4 \\
\text{threadIdx.x} &= 0, 1, 2, 3 \\
\text{idx} &= 12, 13, 14, 15
\end{align*}
\]

\[
\text{int idx} = \text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x};
\] will map from local index \( \text{threadIdx} \) to global index

\[
\text{Common Pattern!}
\]

NB: \( \text{blockDim} \) should be \( \geq 32 \) 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
ingcrement_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

© NVIDIA Corporation 2008
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**
  - `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.
Any source file containing CUDA language extensions must be compiled with **NVCC**
- NVCC separates code running on the host from code running on the device

**Two-stage compilation:**
1. Virtual ISA
   - Parallel Thread eXecution
2. Device-specific binary object
Debugging Using the Device Emulation Mode

An executable compiled in **device emulation mode** (nvcc -deviceemu) runs completely on the host using the CUDA runtime
- No need of any device and CUDA driver
- Each device thread is emulated with a host thread

When running in device emulation mode, one can:
- Use host native debug support (breakpoints, inspection, etc.)
- Access any device-specific data from host code and vice-versa
- Call any host function from device code (e.g. printf) and vice-versa
- Detect deadlock situations caused by improper usage of __syncthreads
Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads potentially produce different results.

- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode.

- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs
  - Different instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host.
Reduction Example

Reduce N values to a single one:
- Sum($v_0, v_1, \ldots, v_{N-2}, v_{N-1}$)
- Min($v_0, v_1, \ldots, v_{N-2}, v_{N-1}$)
- Max($v_0, v_1, \ldots, v_{N-2}, v_{N-1}$)

Common primitive in parallel programming
Easy to implement in CUDA
Less so to get it right

Divided into 5 exercises throughout the day
Each exercise illustrates one particular optimization strategy
Reduction Exercise

At the end of each exercise, the result of the reduction computed on the device is checked for correctness.

- “Test PASSED” or “Test FAILED” is printed out to the console.

The goal is to replace the “TODO” words in the code by the right piece of code to get “test PASSED”.
Reduction Exercise 1

Open up `reduce\src\reduce1.sln`

Code walkthrough:

- **main.cpp**
  - Allocate host and device memory
  - Call `reduce()` defined in `reduce1.cu`
  - Profile and verify result

- **reduce1.cu**
  - CUDA code compiled with `nvcc`
  - Contains TODOs

Device emulation compilation configurations: `Emu*`
Reduce 1: Blocking the Data

Split the work among the N multiprocessors (16 on G80) by launching $numBlocks=N$ thread blocks

Array of the $numValues$ values to be reduced
Reduce 1: Blocking the Data

Within a block, split the work among the threads
- A block can have at most 512 threads
- We choose \( \text{numThreadsPerBlock} = 512 \) threads

\[
\begin{array}{ccccccc}
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & \cdots & t-1 & 0 & \cdots & t-1 & \cdots & \cdots & b-1 \\
\end{array}
\]

- \( t = \text{numThreadsPerBlock} \)
- \( b = \text{numBlocks} \)
Reduce 1: Multi-Pass Reduction

Blocks cannot synchronize so reduce_kernel is called multiple times:

- First call reduces from numValues to numThreads
- Each subsequent call reduces by half
- Ping pong between input and output buffers (d_Result[2])
Goal: Replace the TODOs in reduce1.cu to get “test PASSED”

\[ t = \text{numThreadsPerBlock} \]
\[ b = \text{numBlocks} \]
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-warp 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
Hardware Implementation: Execution Model

- 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 *physically* in parallel
- Warps and blocks are executed *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
Device Management

**CPU can query and select GPU devices**
- `cudaGetDeviceCount( int *count )`
- `cudaSetDevice( int device )`
- `cudaGetDevice( int *current_device )`
- `cudaGetDeviceProperties( cudaDeviceProp* prop, int device )`
- `cudaChooseDevice( int *device, cudaDeviceProp* prop )`

**Multi-GPU setup:**
- device 0 is used by default
- one CPU thread can control only one GPU
  - multiple CPU threads can control the same GPU
    - calls are serialized by the driver
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
Expose Parallelism:
CPU/GPU Parallelism

Take advantage of asynchronous kernel launches by overlapping CPU computations with kernel execution

Take advantage of asynchronous CPU ↔ GPU memory transfers (cudaMemcpyAsync()) that overlap with kernel execution (only available for G84 and up)

- Overlap implemented by using a CUDA stream
- CUDA Stream = Sequence of CUDA operations that execute in order

Example:

```c
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst, src, size, stream1);
kernel<<<grid, block, 0, stream2>>>(...);
cudaMemcpyAsync(dst2, src2, size, stream2);
cudaStreamQuery(stream2);
```

overlapped
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\textsuperscript{th} element is accessed by the N\textsuperscript{th} 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
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
Avoiding Non-Coalesced Accesses

For irregular read patterns, texture fetches can be a better alternative to global memory reads.

If all threads read the same location, use constant memory.

For sequential access patterns, but a structure of size ≠ 4, 8, or 16 bytes:

- Use a Structure of Arrays (SoA) instead of Array of Structures (AoS)
- Or force structure alignment
  - Using __align(X), where X = 4, 8, or 16
- Or use shared memory to achieve coalescing
  - More on this later
CUDA Visual Profiler

- Helps measure and find potential performance problem
  - GPU and CPU timing for all kernel invocations and memcpys
  - Time stamps

- Access to hardware performance counters
Profiler Signals

Events are tracked with hardware counters on signals in the chip:

- timestamp
- gld_incoherent
- gld_coherent
- gst_incoherent
- gst_coherent
- local_load
- local_store
- branch
- divergent_branch

- Global memory loads/stores are coalesced (coherent) or non-coalesced (incoherent)
- Local loads/stores
- Total branches and divergent branches taken by threads
- instructions – instruction count
- warp_serialize – thread warps that serialize on address conflicts to shared or constant memory
- cta_launched – executed thread blocks
Interpreting profiler counters

- Values represent events within a thread warp

- Only targets one multiprocessor
  - Values will not correspond to the total number of warps launched for a particular kernel.
  - Launch enough thread blocks to ensure that the target multiprocessor is given a consistent percentage of the total work.

- Values are best used to identify relative performance differences between unoptimized and optimized code
  - In other words, try to reduce the magnitudes of gld/gst_incoherent, divergent_branch, and warp_serialize
Back to Reduce Exercise: Profile with the Visual Profiler
Back to Reduce Exercise: Problem with Reduce 1

Non-coalesced memory reads!

- Thread IDs
  - 0 … t-1

- Block IDs
  - 0 … b-1

- Elements read by a warp in one memory access

- \( t = \text{numThreadsPerBlock} \)
- \( b = \text{numBlocks} \)

© NVIDIA Corporation 2008
Reduce 2

Distribute threads differently to achieve coalesced memory reads

Bonus: No need to ping pong anymore

- Thread IDs: 0, ..., t-1
- Block IDs: 0, ..., b-1
- $t = \text{numThreadsPerBlock}$
- $b = \text{numBlocks}$

Elements read by a warp in one memory access
Open up `reduce\src\reduce2.sln`

Goal: Replace the TODOs in `reduce2.cu` to get “test PASSED”

<table>
<thead>
<tr>
<th>Thread IDs</th>
<th>0</th>
<th>…</th>
<th>t-1</th>
<th>0</th>
<th>…</th>
<th>t-10</th>
<th>…</th>
<th>t-1</th>
<th>0</th>
<th>…</th>
<th>t-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block IDs</td>
<td>0</td>
<td>…</td>
<td>b-1</td>
<td>0</td>
<td>…</td>
<td>b-1</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td></td>
</tr>
</tbody>
</table>

\( t = \text{numThreadsPerBlock} \)
\( b = \text{numBlocks} \)
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() (if needed)
    - 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

\[ C = A \cdot B \text{ of size } N \times N \]

With blocking:
- One **thread block** handles one \( M \times M \) sub-matrix \( C_{\text{sub}} \) of \( C \)
- \( A \) and \( B \) are only loaded \( (N / M) \) times from global memory

- Much less bandwidth
- Much better balance of work to bandwidth
Example: Avoiding Non-Coalesced float3 Memory Accesses

```c
__global__ void accessFloat3(float3 *d_in, float3 d_out) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    float3 a = d_in[index];
    a.x += 2;
    a.y += 2;
    a.z += 2;
    d_out[index] = a;
}
```
Example: Avoiding Non-Coalesced float3 Memory Accesses

- `float3` is 12 bytes
- Each thread ends up executing 3 reads
  - `sizeof(float3) \neq 4, 8, or 16`
  - Half-warp reads three 64B **non-contiguous** regions
Example: Avoiding Non-Coalesced float3 Memory Accesses

Similarly, Step3 starting at offset 512
Example: Avoiding Non-Coalesced float3 Memory Accesses

- Use shared memory to allow coalescing
  - Need sizeof(float3)*(threads/block) bytes of SMEM
  - Each thread reads 3 scalar floats:
    - Offsets: 0, (threads/block), 2*(threads/block)
    - These will likely be processed by other threads, so sync

Processing
- Each thread retrieves its float3 from SMEM array
  - Cast the SMEM pointer to (float3*)
  - Use thread ID as index
- Rest of the compute code does not change!
Example: Avoiding Non-Coalesced float3 Memory Accesses

```c
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float s_data[256*3];
    s_data[threadIdx.x] = g_in[index];
    s_data[threadIdx.x+256] = g_in[index+256];
    s_data[threadIdx.x+512] = g_in[index+512];
    __syncthreads();
    float3 a = ((float3*)s_data)[threadIdx.x];
    a.x += 2;
    a.y += 2;
    a.z += 2;
    ((float3*)s_data)[threadIdx.x] = a;
    __syncthreads();
    g_out[index] = s_data[threadIdx.x];
    g_out[index+256] = s_data[threadIdx.x+256];
    g_out[index+512] = s_data[threadIdx.x+512];
}
```

Read the input through SMEM

Compute code is not changed

Write the result through SMEM
Example: Avoiding Non-Coalesced float3 Memory Accesses

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 uncoalesced
- 359µs – float3 coalesced through shared memory
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
Determining Resource Usage

Compile the kernel code with the -cubin flag to determine register usage.

Open the .cubin file with a text editor and look for the “code” section.

```plaintext
architecture {sm_10}
abiversion {0}
modname {cubin}

code {
  name = BlackScholesGPU
  lmem = 0
  smem = 68
  reg = 20
  bar = 0
  bincode {
    0xa0004205 0x04200780 0x40024c09 0x00200780
    ...
```
Execution Configuration: Heuristics

- (# of threads per block) = multiple of warp size
  - To avoid wasting computation on under-populated warps
- (# of blocks) / (# of multiprocessors) > 1
  - So all multiprocessors have at least a block to execute
- Per-block resources (shared memory and registers) at most half of total available
- And: (# of blocks) / (# of multiprocessors) > 2
  - To get more than 1 active block per multiprocessor
  - With multiple active blocks that aren’t all waiting at a __syncthreads(), the multiprocessor can stay busy
- (# of blocks) > 100 to scale to future devices
  - Blocks stream through machine in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations
- Very application-dependent: experiment!
To help you: the CUDA occupancy calculator
Back to Reduce Exercise:
Problem with Reduce 2

- Reduce 2 does not take advantage of shared memory!

- Reduce 3 fixes this by implementing parallel reduction in shared memory

Runtime shared memory allocation:

```c
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

The optional `SharedMemBytes` bytes are:
- Allocated in addition to the compiler allocated shared memory
- Mapped to any variable declared as:
  ```c
  extern __shared__ float DynamicSharedMem[];
  ```
Reduce 3:
Parallel Reduction Implementation

Indices
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Values
10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2

Step 1
Stride 1
Thread IDs
0 1 2 3 4 5 6 7

Values
11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2

Step 2
Stride 2
Thread IDs
0 1 2 3

Values
18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2

Step 3
Stride 4
Thread IDs
0 1

Values
24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

Step 4
Stride 8
Thread IDs
0

Values
41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

© NVIDIA Corporation 2008
Parallel Reduction Complexity

- Takes $\log(N)$ steps and each step $S$ performs $N/2^S$ independent operations
  - Step complexity is $O(\log(N))$

- For $N=2^D$, performs $\sum_{S \in [1..D]} 2^{D-S} = N-1$ operations
  - Work complexity is $O(N)$
  - Is work-efficient (i.e. does not perform more operations than a sequential reduction)

- With $P$ threads physically in parallel ($P$ processors), performs $\sum_{S \in [1..D]} \lceil 2^{D-S}/P \rceil$ operations
  - $\sum_{S \in [1..D]} \lceil 2^{D-S}/P \rceil < \sum_{S \in [1..D]} (\lfloor 2^{D-S}/P \rfloor + 1) < N/P + \log(N)$
  - Time complexity is $O(N/P + \log(N))$
  - Compare to $O(N)$ for sequential reduction
Each thread stores its result in an array of `numThreadsPerBlock` elements in shared memory.

Each block performs a parallel reduction on this array.

`reduce_kernel` is called only 2 times:
- First call reduces from `numValues` to `numBlocks`.
- Second call performs final reduction using one thread block.
Open up `reduce/src/reduce3.sln`

Goal: Replace the TODOs in `reduce3.cu` to get “test PASSED”

<table>
<thead>
<tr>
<th>Indices</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>-1</td>
<td>0</td>
<td>-2</td>
<td>3</td>
<td>5</td>
<td>-2</td>
<td>-3</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

**Step 1**  
Stride 1  
Thread IDs: 0, 1, 2, 3, 4, 5, 6, 7

| Values  | 11| 1 | 7 |-1| -2| -2| 8 | 5 |-5| -3| 9 | 7 | 11| 11| 2 | 2 |

**Step 2**  
Stride 2  
Thread IDs: 0, 1, 2, 3

| Values  | 18| 1 | 7 |-1| 6 |-2| 8 | 5 |4 |-3| 9 | 7 | 13| 11| 2 | 2 |

**Step 3**  
Stride 4  
Thread IDs: 0, 1

© NVIDIA Corporation 2008
Optimize Instruction Usage: Basic Strategies

- Minimize use of low-throughput instructions
- Use high precision only where necessary
- Minimize divergent warps
**Arithmetic Instruction Throughput**

- **float** add/mul/mad, **int** add, shift, min, max: 4 cycles per warp
  - **int** multiply (*) is by default 32-bit
    - Requires multiple cycles per warp
  - Use __[u]mul24() intrinsics for 4-cycle 24-bit **int** multiply

**Integer divide and modulo are more expensive**
- Compiler will convert literal power-of-2 divides to shifts
  - But we have seen it miss some cases
- Be explicit in cases where compiler can’t tell that divisor is a power of 2!
- **Useful trick:** foo%n == foo&(n-1) if n is a power of 2
Reciprocal, reciprocal square root, sin/cos, log, exp: 16 cycles per warp

These are the versions prefixed with “__”

Examples: __rcp(), __sin(), __exp()

Other functions are combinations of the above:

y/x = rcp(x) * y takes 20 cycles per warp

sqrt(x) = rcp(rsqrt(x)) takes 32 cycles per warp
There are two types of runtime math operations:

- `__func()`: direct mapping to hardware ISA
  - Fast but lower accuracy (see prog. guide for details)
  - Examples: `__sin(x)`, `__exp(x)`, `__pow(x,y)`

- `func()`: compile to multiple instructions
  - Slower but higher accuracy (5 ulp or less)
  - Examples: `sin(x)`, `exp(x)`, `pow(x,y)`

The `-use_fast_math` compiler option forces every `func()` to compile to `__func()`
Double Precision Is Coming…

- Current NVIDIA GPUs support single precision only
  - IEEE 32-bit floating-point precision ("FP32")

- Upcoming NVIDIA GPUs will support double precision
  - IEEE 64-bit floating-point precision ("FP64")
What You Need To Know

- FP64 instructions will be slower than FP32
  - It takes more than just wider data paths to implement double precision

- For best performance, use FP64 judiciously
  - Analyze your computations
  - Use FP64 only for precision/range-sensitive computations
  - Use FP32 for computations that are accurate and robust with 32-bit floating point

- CUDA compiler supports mixed usage of float and double
  - Supported since CUDA 1.0
Float “Safety”

Don’t accidentally use FP64 where you intend FP32:
- Standard math library and floating-point literals default to double precision
  
  ```
  float f = 2.0 * sin(3.14); // warning: double precision!
  // fp64 multiply, several fp64 insts. for sin(), and fp32 cast
  ```

  ```
  float f = 2.0f * sinf(3.14f); // warning: double precision!
  // fp32 multiply and several fp32 instructions for sin()
  ```

- On FP64-capable NVIDIA GPUs, the green code will be much faster than the orange code
Abstract: By using a combination of 32-bit and 64-bit floating point arithmetic, the performance of many dense and sparse linear algebra algorithms can be significantly enhanced while maintaining the 64-bit accuracy of the resulting solution. The approach presented here can apply not only to conventional processors but also to exotic technologies such as Field Programmable Gate Arrays (FPGA), Graphical Processing Units (GPU), and the Cell BE processor. Results on modern processor architectures and the Cell BE are presented.
Single Precision IEEE Floating Point

- Addition and multiplication are IEEE compliant
  - Maximum 0.5 ulp error
- However, often combined into multiply-add (FMAD)
  - Intermediate result is truncated
- Division is non-compliant (2 ulp)
- Not all rounding modes are supported
- Denormalized numbers are not supported
- No mechanism to detect floating-point exceptions
# Single Precision Floating Point

<table>
<thead>
<tr>
<th></th>
<th>8-Series GPU</th>
<th>SSE</th>
<th>IBM Altivec</th>
<th>Cell SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precision</strong></td>
<td>IEEE 754</td>
<td>IEEE 754</td>
<td>IEEE 754</td>
<td>near IEEE 754</td>
</tr>
<tr>
<td><strong>Rounding modes for FADD and FMUL</strong></td>
<td>Round to nearest and round to zero</td>
<td>All 4 IEEE, round to nearest, zero, inf, -inf</td>
<td>Round to nearest only</td>
<td>Round to zero/truncate only</td>
</tr>
<tr>
<td><strong>Denormal handling</strong></td>
<td>Flush to zero</td>
<td>Supported, 1000’s of cycles</td>
<td>Supported, 1000’s of cycles</td>
<td>Flush to zero</td>
</tr>
<tr>
<td><strong>NaN support</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Overflow and Infinity support</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No infinity, only clamps to max norm</td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Some</td>
</tr>
<tr>
<td><strong>Square root</strong></td>
<td>Software only</td>
<td>Hardware</td>
<td>Software only</td>
<td>Software only</td>
</tr>
<tr>
<td><strong>Division</strong></td>
<td>Software only</td>
<td>Hardware</td>
<td>Software only</td>
<td>Software only</td>
</tr>
<tr>
<td><strong>Reciprocal estimate accuracy</strong></td>
<td>24 bit</td>
<td>12 bit</td>
<td>12 bit</td>
<td>12 bit</td>
</tr>
<tr>
<td><strong>Reciprocal sqrt estimate accuracy</strong></td>
<td>23 bit</td>
<td>12 bit</td>
<td>12 bit</td>
<td>12 bit</td>
</tr>
<tr>
<td><strong>log2(x) and 2^x estimates accuracy</strong></td>
<td>23 bit</td>
<td>No</td>
<td>12 bit</td>
<td>No</td>
</tr>
</tbody>
</table>
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:
  - `If (threadIdx.x > 2) { }`
  - Branch granularity < warp size
- Example without divergence:
  - `If (threadIdx.x / WARP_SIZE > 2) { }`
  - Branch granularity is a whole multiple of warp size
Instruction Predication

- Comparison instructions set condition codes (CC)
- Instructions can be predicated to write results only when CC meets criterion (CC != 0, CC >= 0, etc.)
- Compiler tries to predict if a branch condition is likely to produce many divergent warps
  - If guaranteed not to diverge: only predicates if < 4 instructions
  - If not guaranteed: only predicates if < 7 instructions
- May replace branches with instruction predication
- ALL predicated instructions take execution cycles
  - Those with false conditions don’t write their output
    - Or invoke memory loads and stores
  - Saves branch instructions, so can be cheaper than serializing divergent paths
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
Bank Addressing Examples

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

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

© NVIDIA Corporation 2008
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
Back to Reduce Exercise:
Problem with Reduce 3

Reduce 3 has **shared memory bank conflicts**!

Reduce 4 fixes this by modifying the mapping between threads and data during parallel reduction.
## Reduce 3: Bank Conflicts

- **Showed for step 1 below**
- **First simultaneous memory access**
  
  $sresult[2 \times stride \times threadID]$

### Diagram

- Threads 0 and 8 access the same bank
- Threads 1 and 9 access the same bank
- Threads 2 and 10 access the same bank, etc.

### Table

<table>
<thead>
<tr>
<th>Banks</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indices</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>Values</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>-1</td>
<td>0</td>
<td>-2</td>
<td>3</td>
<td>5</td>
<td>-2</td>
<td>-3</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>2</td>
<td>-1</td>
<td>4</td>
<td>11</td>
<td>-5</td>
<td>0</td>
<td>12</td>
<td>...</td>
</tr>
<tr>
<td>Stride 1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread IDs</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© NVIDIA Corporation 2008
Reduce 3: Bank Conflicts

- Showed for step 1 below
- Second simultaneous memory access

\[ \text{sresult}[2 \times \text{stride} \times \text{threadID} + \text{stride}] \]

Threads 0 and 8 access the same bank
Threads 1 and 9 access the same bank
Threads 2 and 10 access the same bank, etc.
Reduce 4: Parallel Reduction Implementation

### Step 1
**Stride 8**

<table>
<thead>
<tr>
<th>Indices</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>-1</td>
<td>0</td>
<td>-2</td>
<td>3</td>
<td>5</td>
<td>-2</td>
<td>-3</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

### Step 2
**Stride 4**

<table>
<thead>
<tr>
<th>Thread IDs</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>8</td>
<td>-2</td>
<td>10</td>
<td>6</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

### Step 3
**Stride 2**

<table>
<thead>
<tr>
<th>Thread IDs</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>8</td>
<td>7</td>
<td>13</td>
</tr>
</tbody>
</table>

### Step 4
**Stride 1**

<table>
<thead>
<tr>
<th>Thread IDs</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>21</td>
</tr>
</tbody>
</table>

© NVIDIA Corporation 2008
Reduce 4: Go Ahead!

Open up `reduce\src\reduce4.sln`

Goal: Replace the TODOs in `reduce4.cu` to get “test PASSED”
Reduce 5:
More Optimizations through Unrolling

- **Parallel reduction inner loop:**
  ```c
  for (int stride = numThreadsPerBlock / 2; 
      stride > 0; stride /= 2)
  {
      __syncthreads();
      if (threadID < stride)
          sresult[threadID] += sresult[threadID + stride];
  }
  ```

- There are only so many values for `numThreadsPerBlock`:
  - Multiple of 32, less or equal to 512
- So, **templatize on numThreadsPerBlock**:
  ```c
  template <uint numThreadsPerBlock>
  __global__ void reduce_kernel(const float* valuesIn,
                                 uint numValues,
                                 float* valuesOut)
  ```

- And **unroll**: 
Reduce 5: Unrolled Loop

if (numThreadsPerBlock >= 512)
{ __syncthreads(); if (threadID < 256) sresult[threadID] += sresult[threadID + 256]; }
if (numThreadsPerBlock >= 256)
{ __syncthreads(); if (threadID < 128) sresult[threadID] += sresult[threadID + 128]; }
if (numThreadsPerBlock >= 128)
{ __syncthreads(); if (threadID < 64) sresult[threadID] += sresult[threadID + 64]; }
if (numThreadsPerBlock >= 64)
{ __syncthreads(); if (threadID < 32) sresult[threadID] += sresult[threadID + 32]; }
if (numThreadsPerBlock >= 32)
{ __syncthreads(); if (threadID < 16) sresult[threadID] += sresult[threadID + 16]; }
if (numThreadsPerBlock >= 16)
{ __syncthreads(); if (threadID < 8) sresult[threadID] += sresult[threadID + 8]; }
if (numThreadsPerBlock >= 8)
{ __syncthreads(); if (threadID < 4) sresult[threadID] += sresult[threadID + 4]; }
if (numThreadsPerBlock >= 4)
{ __syncthreads(); if (threadID < 2) sresult[threadID] += sresult[threadID + 2]; }
if (numThreadsPerBlock >= 2)
{ __syncthreads(); if (threadID < 1) sresult[threadID] += sresult[threadID + 1]; }

All code in blue will be evaluated at compile time!
Reduce 5: Last Warp Optimization

- As reduction proceeds, the number of “active” threads decreases
- When \(\text{stride} \leq 32\), we have only one warp left
- Instructions are synchronous within a warp
- That means when \(\text{stride} \leq 32\):
  - We don’t need to \(__\text{syncthreads}\)__()
  - We don’t need “if (threadID < stride)” because it doesn’t save any work

So, final version of unrolled loop is:
Reduce 5: Final Unrolled Loop

if (numThreadsPerBlock >= 512)
{ __syncthreads(); if (threadID < 256) sresult[threadID] += sresult[threadID + 256]; }

if (numThreadsPerBlock >= 256)
{ __syncthreads(); if (threadID < 128) sresult[threadID] += sresult[threadID + 128]; }

if (numThreadsPerBlock >= 128)
{ __syncthreads(); if (threadID < 64) sresult[threadID] += sresult[threadID + 64]; }

__syncthreads();

if (threadID < 32) {
    if (numThreadsPerBlock >= 64) sresult[threadID] += sresult[threadID + 32];
    if (numThreadsPerBlock >= 32) sresult[threadID] += sresult[threadID + 16];
    if (numThreadsPerBlock >= 16) sresult[threadID] += sresult[threadID + 8];
    if (numThreadsPerBlock >= 8) sresult[threadID] += sresult[threadID + 4];
    if (numThreadsPerBlock >= 4) sresult[threadID] += sresult[threadID + 2];
    if (numThreadsPerBlock >= 2) sresult[threadID] += sresult[threadID + 1];
}

All code in blue will be evaluated at compile time!
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
Coming Up Soon

CUDA 2.0
- Public beta this week
- Support for upcoming new GPU:
  - Double precision
  - Integer atomic operations in shared memory
- New features:
  - 3D textures
  - Improved and extended Direct3D interoperability

CUDA implementation on multicore CPU
- Beta in a few weeks
Where to go from here

  - CUDA works on all NVIDIA 8-Series GPUs (and later)
    - GeForce, Quadro, and Tesla

- Talk about CUDA http://forums.nvidia.com
## Tesla Architecture Family

<table>
<thead>
<tr>
<th>Model</th>
<th>Number of Multiprocessors</th>
<th>Compute Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce 8800 Ultra, 8800 GTX</td>
<td>16</td>
<td>1.0</td>
</tr>
<tr>
<td>GeForce 8800 GT</td>
<td>14</td>
<td>1.1</td>
</tr>
<tr>
<td>GeForce 8800M GTX</td>
<td>12</td>
<td>1.1</td>
</tr>
<tr>
<td>GeForce 8800 GTS</td>
<td>12</td>
<td>1.0</td>
</tr>
<tr>
<td>GeForce 8800M GTS</td>
<td>8</td>
<td>1.1</td>
</tr>
<tr>
<td>GeForce 8600 GT, 8600 GTX, 8700M GT, 8600M GT, 8600M GS</td>
<td>4</td>
<td>1.1</td>
</tr>
<tr>
<td>GeForce 8500 GT, 8400 GS, 8400M GT, 8400M GS</td>
<td>2</td>
<td>1.1</td>
</tr>
<tr>
<td>GeForce 8400M G</td>
<td>1</td>
<td>1.1</td>
</tr>
<tr>
<td>Tesla S870</td>
<td>4x16</td>
<td>1.0</td>
</tr>
<tr>
<td>Tesla D870</td>
<td>2x16</td>
<td>1.0</td>
</tr>
<tr>
<td>Tesla C870</td>
<td>16</td>
<td>1.0</td>
</tr>
<tr>
<td>Quadro Plex 1000 Model S4</td>
<td>4x16</td>
<td>1.0</td>
</tr>
<tr>
<td>Quadro Plex 1000 Model IV</td>
<td>2x16</td>
<td>1.0</td>
</tr>
<tr>
<td>Quadro FX 5600</td>
<td>16</td>
<td>1.0</td>
</tr>
<tr>
<td>Quadro FX 4600</td>
<td>12</td>
<td>1.0</td>
</tr>
<tr>
<td>Quadro FX 1700, FX 570, NVS 320M, FX 1600M, FX 570M</td>
<td>4</td>
<td>1.1</td>
</tr>
<tr>
<td>Quadro FX 370, NVS 290, NVS 140M, NVS 135M, FX 360M</td>
<td>2</td>
<td>1.1</td>
</tr>
<tr>
<td>Quadro NVS 130M</td>
<td>1</td>
<td>1.1</td>
</tr>
</tbody>
</table>
New Applications

Real-time options implied volatility engine

Ultrasound imaging

Swaption volatility cube calculator

HOOMD Molecular Dynamics

Manifold 8 GIS

SDK: Mandelbrot, computer vision

Also...

Image rotation/classification

Graphics processing toolbox

Microarray data analysis

Data parallel primitives

Astrophysics simulations

Seismic migration
Acceleware

GPU Electromagnetic Field simulation
- Cell phone irradiation
- MRI Design / Modeling
- Printed Circuit Boards
- Radar Cross Section (Military)

Seismic Migration
- 8X Faster than Quad Core alone

Performance
- CPU 3.2 GHz Core 2 Duo
- 1X
- 11X 1 GPU
- 22X 2 GPUs
- 45X 4 GPUs

Pacemaker with Transmit Antenna
Three GeForce 8800GTX cards outrun ~300 CPUs

Parallel GPUs with Multithreading:
705 GFLOPS /w 3 GPUs

- One host thread is created for each CUDA GPU
- Threads are spawned and attach to their GPU based on their host thread ID
  - First CUDA call binds that thread’s CUDA context to that GPU for life
  - Handling error conditions within child threads is dependent on the thread library and, makes dealing with any CUDA errors somewhat tricky, left as an exercise to the reader.
- Map slices are computed cyclically by the GPUs
- Want to avoid false sharing on the host memory system
  - Map slices are usually much bigger than the host memory page size, so this is usually not a problem for this application
- Performance of 3 GPUs is stunning!
- Power: 3 GPU test box consumes 700 watts running flat out

http://www.ks.uiuc.edu/Research/vmd/projects/ece498/lecture/
EvolvedMachines

- 130X Speed up
- Brain circuit simulation
- Sensory computing: vision, olfactory
Matlab: Language of Science

17X with MATLAB CPU+GPU


Pseudo-spectral simulation of 2D Isotropic turbulence

http://www.amath.washington.edu/courses/571-winter-2006/matlab/FS_2Dturb.m
nbody Astrophysics

Astrophysics research
1 GF on standard PC
300+ GF on GeForce 8800GTX
Faster than GRAPE-6Af custom simulation computer

http://progrape.jp/cs/

Video demo
© NVIDIA Corporation 2008
CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
  - Thread can access any memory location
- Unlimited access to memory
  - Thread can read/write as many locations as needed
- Shared memory (per block) and thread synchronization
  - Threads can cooperatively load data into shared memory
  - Any thread can then access any shared memory location
- Low learning curve
  - Just a few extensions to C
  - No knowledge of graphics is required
- No graphics API overhead
A quick review

- **device** = GPU = set of multiprocessors
- **Multiprocessor** = set of processors & shared memory
- **Kernel** = GPU program
- **Grid** = array of thread blocks that execute a kernel
- **Thread block** = group of SIMD threads that execute a kernel and can communicate via shared memory

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Who</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A</td>
<td>Read/write</td>
<td>All threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
</tbody>
</table>
The API is an extension to the C programming language. It consists of:

- **Language extensions**
  - To target portions of the code for execution on the device

- **A runtime library split into:**
  - A **common component** providing built-in vector types and a subset of the C runtime library supported in both host and device codes
  - A **host component** to control and access one or more devices from the host
  - A **device component** providing device-specific functions
### Language Extensions: Function Type Qualifiers

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

- **__global__** defines a kernel function
- **__device__** functions cannot have their address taken
- **__device__** and **__host__** can be used together
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments
## Language Extensions: Variable Type Qualifiers

<table>
<thead>
<tr>
<th><strong>device</strong></th>
<th><strong>shared</strong></th>
<th>int SharedVar;</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong></td>
<td></td>
<td>int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td><strong>constant</strong></td>
<td>int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __shared__ or __constant__
- **Automatic variables** without any qualifier reside in a **register**
  - Except for large structures or arrays that reside in local memory
- **Pointers** can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:
    __global__ void KernelFunc(float* ptr)
  - Obtained as the address of a global variable:
    float* ptr = &GlobalVar;
Language Extensions: Execution Configuration

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 >>>(...);
```

The optional SharedMemBytes bytes are:
- Allocated in addition to the compiler allocated shared memory
- Mapped to any variable declared as:
  ```c
  extern __shared__ float DynamicSharedMem[];
  ```

Any call to a kernel function is asynchronous
- Control returns to CPU immediately
Language Extensions: Built-in Variables

- `dim3 gridDim;`  Dimensions of the grid in blocks (`gridDim.z` unused)
- `dim3 blockDim;`  Dimensions of the block in threads
- `dim3 blockIdx;`  Block index within the grid
- `dim3 threadIdx;`  Thread index within the block
Common Runtime Component

Provides:
- Built-in vector types
- A subset of the C runtime library supported in both host and device codes
Common Runtime Component: Built-in Vector Types

- [u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4]
- Structures accessed with x, y, z, w fields:
  ```
  uint4 param;
  int y = param.y;
  ```

- dim3
  - Based on uint3
  - Used to specify dimensions
Common Runtime Component: Mathematical Functions

- powf, sqrtf, cbrtf, hypotf
- expf, exp2f, expm1f
- logf, log2f, log10f, log1pf
- sinf, cosf, tanf
- asinf, acosf, atanf, atan2f
- sinh, coshf, tanhf
- asinhf, acoshf, atanhf
- ceil, floor, trunc, round
- Etc.

When executed in host code, a given function uses the C runtime implementation if available. These functions are only supported for scalar types, not vector types.
Texture memory is accessed through `texture references`:

```cpp
texture<float, 2> myTexRef; // 2D texture of float values
myTexRef.addressMode[0] = cudaAddressModeWrap;
myTexRef.addressMode[1] = cudaAddressModeWrap;
myTexRef.filterMode = cudaFilterModeLinear;
```

Texture fetching in device code:

```cpp
float4 value = tex2D(myTexRef, u, v);
```
Host Runtime Component

Provides functions to deal with:

- **Device** management (including multi-device systems)
- **Memory** management
- **Texture** management
- **Interoperability** with OpenGL and Direct3D9
- **Error** handling

Initializes the first time a runtime function is called

A host thread can execute device code on only one device

- Multiple host threads required to run on multiple devices
Host Runtime Component: Device Management

- **Device enumeration**
  - `cudaGetDeviceCount()`, `cudaGetDeviceProperties()`

- **Device selection**
  - `cudaChooseDevice()`, `cudaSetDevice()`
Host Runtime Component: Memory Management

- Two kinds of memory:
  - **Linear memory**: accessed through 32-bit pointers
  - **CUDA arrays**: opaque layouts with dimensionality, only readable through texture fetching

- **Device memory allocation**
  - `cudaMalloc()`, `cudaMallocPitch()`, `cudaFree()`, `cudaMallocArray()`, `cudaFreeArray()`

- **Memory copy from host to device, device to host, device to device**
  - `cudaMemcpy()`, `cudaMemcpy2D()`, `cudaMemcpyToArray()`, `cudaMemcpyFromArray()`, etc.
  - `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`

- **Memory addressing**
  - `cudaGetSymbolAddress()`
Host Runtime Component: Texture Management

Texture references can be bound to:

- CUDA arrays
- Linear memory
  - 1D texture only, no filtering, integer texture coordinate
- `cudaBindTexture()`, `cudaUnbindTexture()`
Host Runtime Component:
Interoperability with Graphics APIs

- **OpenGL buffer objects** and **Direct3D9 vertex buffers** can be mapped into the address space of CUDA:
  - To read data written by OpenGL
  - To write data for consumption by OpenGL
  - `cudaGLMapBufferObject()`, `cudaGLUnmapBufferObject()`
  - `cudaD3D9MapResources()`, `cudaD3D9UnmapResources()`
Host Runtime Component: Events

Events are inserted (recorded) into CUDA call streams

**Usage scenarios:**

- measure elapsed time for CUDA calls (clock cycle precision)
- query the status of an asynchronous CUDA call
- block CPU until CUDA calls prior to the event are completed
- asyncAPI sample in CUDA SDK

```c
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel<<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float et;
cudaEventElapsedTime(&et, start, stop);
cudaEventDestroy(start); cudaEventDestroy(stop);
```
Host Runtime Component: Error Handling

- All CUDA calls return error code:
  - except for kernel launches
  - cudaError_t type

- cudaError_t cudaGetLastError(void)
  - returns the code for the last error (no error has a code)

- char* cudaGetErrorString(cudaError_t code)
  - returns a null-terminated character string describing the error

```c
printf("\%s\n", cudaGetErrorString( cudaGetLastError() ) );
```
Device Runtime Component

Provides **device-specific functions**
Some mathematical functions (e.g. \( \sin(x) \)) have a less accurate, but faster device-only version (e.g. \( \_\_\sin(x) \))

\( \_\_\text{pow} \)
\( \_\_\log, \_\_\log2, \_\_\log10 \)
\( \_\_\exp \)
\( \_\_\sin, \_\_\cos, \_\_\tan \)
Device Runtime Component: GPU Atomic Integer Operations

Atomic operations on integers in global memory:
- Associative operations on signed/unsigned ints
- add, sub, min, max, ...
- and, or, xor
- Increment, decrement
- Exchange, compare and swap

Requires hardware with compute capability 1.1 or higher
Device Runtime Component: Texture Functions

- For texture references bound to CUDA arrays:
  ```c
  float u, v;
  float4 value = tex2D(myTexRef, u, v);
  ```

- For texture references bound to linear memory:
  ```c
  int i;
  float4 value = tex2D(myTexRef, i);
  ```
Device Runtime Component: Synchronization Function

```c
void __syncthreads();
```

- Synchronizes all threads in a block
- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory
- Allowed in conditional code only if the conditional is uniform across the entire thread block
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 can output:
  - Either C code (CPU Code)
    - That must then be compiled with the rest of the application using another tool
  - Or PTX object code directly
- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (**cudart**)
  - The CUDA core library (**cuda**)
NVCC & PTX Virtual Machine

C/C++ CUDA Application

EDG

Open64

CPU Code

PTX Code

EDG
- Separate GPU vs. CPU code

Open64
- Generates GPU PTX assembly

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

ld.global.v4.f32 {$f1,$f3,$f5,$f7}, [$r9+0];
me.x += me.y * me.z;

mad.f32 $f1, $f5, $f3, $f1;
Role of Open64

Open64 compiler gives us

- A complete C/C++ compiler framework. Forward looking. We do not need to add infrastructure framework as our hardware arch advances over time.

- A good collection of high level architecture independent optimizations. All GPU code is in the inner loop.

- Compiler infrastructure that interacts well with other related standardized tools.
## GeForce 8800 Series and Quadro FX 5600/4600 Technical Specifications

<table>
<thead>
<tr>
<th>Model</th>
<th>Number of multiprocessors</th>
<th>Clock frequency (GHz)</th>
<th>Amount of device memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce 8800 GTX</td>
<td>16</td>
<td>1.35</td>
<td>768</td>
</tr>
<tr>
<td>GeForce 8800 GTS</td>
<td>12</td>
<td>1.2</td>
<td>640</td>
</tr>
<tr>
<td>Quadro FX 5600</td>
<td>16</td>
<td>1.35</td>
<td>1500</td>
</tr>
<tr>
<td>Quadro FX 4600</td>
<td>12</td>
<td>1.2</td>
<td>768</td>
</tr>
</tbody>
</table>

- Maximum number of threads per block: 512
- Maximum size of each dimension of a grid: 65535
- Warp size: 32 threads
- Number of registers per multiprocessor: 8192
- Shared memory per multiprocessor: 16 KB divided in 16 banks
- Constant memory: 64 KB

© NVIDIA Corporation 2008
CUDA Libraries

**CUBLAS**
- CUDA “Basic Linear Algebra Subprograms”
- Implementation of BLAS standard on CUDA
- For details see cublas_library.pdf and cublas.h

**CUFFT**
- CUDA Fast Fourier Transform (FFT)
- FFT one of the most important and widely used numerical algorithms
- For details see cufft_library.pdf and cufft.h
CUBLAS Library

- Self-contained at API level
  - Application needs no direct interaction with CUDA driver
- Currently only a subset of CUBLAS core functions are implemented

- Simple to use:
  - Create matrix and vector objects in GPU memory
  - Fill them with data
  - Call sequence of CUBLAS functions
  - Upload results back from GPU to host
- Column-major storage and 1-based indexing
  - For maximum compatibility with existing Fortran apps
CUFFT Library

Efficient implementation of FFT on CUDA

Features

- 1D, 2D, and 3D FFTs of complex-valued signal data
- Batch execution for multiple 1D transforms in parallel
- Transform sizes (in any dimension) in the range [2, 16384]