Parallel Computing with CUDA
Agenda

- Introduction & Motivation
- The Myth of GPU Computing
- CUDA Programming Model
- Parallel Algorithms in CUDA
- CUDA Tools and Resources
- Sample Applications
Introduction & Motivation
Future Science & Engineering Breakthroughs Hinge on Computing

- Computational Geoscience
- Computational Modeling
- Computational Medicine
- Computational Physics
- Computational Chemistry
- Computational Biology
- Computational Finance
- Image Processing
Faster is not “Just faster”
2-3x  “Just faster”

Do a little more, wait a little less

 Doesn’t change how you work
5-10x

“Significant”

Worth upgrading

Worth rewriting (parts of) your application
“Fundamentally Different”

Worth considering a new platform

Worth re-architecting your application

Makes new applications possible

Drives down “time to discovery”

Creates fundamental changes in science
CPU has hit a wall

- Even 40% increasing on performance is considered an achievement!
- Clock speed is stuck at 3GHz for years
- No matter how the cache is designed, it will never be large enough
- Adding more cores makes programming more difficult than ever. The magical compiler for multi-threading only exists on paper
Why GPU?

- GPUs have evolved into highly parallel machines
  - Increasing performance much faster rate than CPU
  - Already provide 100s of cores
  - 2x more core means 2x more performance
  - Fully programmable in C
    - NO graphics programming knowledge needed
  - Lots of compute power and memory bandwidth
  - Widely available
    - Over 100M CUDA-capable GPUs shipped as of 2008

- GPUs enable parallel computing for the masses!
Parallel Computing on GPU

GPUs are massively multithreaded manycore chips
- NVIDIA GPU products have up to 240 scalar processors
- Over 23,000 concurrent threads in flight
- 1 TFLOP of performance (Tesla)

Enabling new science and engineering
- By drastically reducing time to discovery
- Engineering design cycles: from days to minutes, weeks to days

Enabling new computer science
- By reinvigorating research in parallel algorithms, programming models, architecture, compilers, and languages
The complexity of the problem is hidden by the simplicity of the solution.
CUDA Use Cases

146X
Volumetric white matter visualization

36X
Ionic placement for molecular dynamics

18X
Transcoding HD H.264 video stream

17X
Isotropic turbulence simulation in Matlab

100X
Astrophysics n-body simulation

149X
Financial simulation of LIBOR Model

47X
M-script API for Linear Algebra

20X
Ultrasound imaging for cancer

24X
Molecular dynamics

30X
String matching for gene sequences
The Myth of GPU Computing
Myth of GPU Computing

- GPUs layer normal programs on top of graphics

- GPU architectures are
  - Very wide (1000s) SIMD machines
  - ...on which branching is impossible or prohibitive
  - ...with 4-wide vector registers

- GPUs are power-inefficient

- GPUs don’t do real floating point
Myth of GPU Computing

- GPUs layer normal programs on top of graphics
  No: CUDA compiles directly into the hardware

- GPU architectures are
  - Very wide (1000s) SIMD machines
  - ...on which branching is impossible or prohibitive
  - ...with 4-wide vector registers

- GPUs are power-inefficient

- GPUs don’t do real floating point
Myth of GPU Computing

- GPUs layer normal programs on top of graphics

- GPU architectures are
  - Very wide (1000s) SIMD machines
  - ...on which branching is impossible or prohibitive
  - ...with 4-wide vector registers

- GPUs are power-inefficient

- GPUs don’t do real floating point
Myth of GPU Computing

- GPUs layer normal programs on top of graphics

- GPU architectures are
  - Very wide (1000s) SIMD machines
  - ...on which branching is impossible or prohibitive   Nope
  - ...with 4-wide vector registers

- GPUs are power-inefficient

- GPUs don’t do real floating point
Myth of GPU Computing

- **GPUs layer normal programs on top of graphics**
- **GPU architectures are**
  - Very wide (1000s) SIMD machines
  - ...on which branching is impossible or prohibitive
  - ...with 4-wide vector registers **No. All are scalars.**
- **GPUs are power-inefficient**
- **GPUs don’t do real floating point**
Myth of GPU Computing

- GPUs layer normal programs on top of graphics

- GPU architectures are
  - Very wide (1000s) SIMD machines
  - ...on which branching is impossible or prohibitive
  - ...with 4-wide vector registers

- CPUs are power-inefficient
  - No. 4x ~ 10x perf/watt advantage

- GPUs don’t do real floating point
# Double Precision Floating Point

<table>
<thead>
<tr>
<th></th>
<th>NVIDIA Tesla T10</th>
<th>SSE2</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>
</tr>
<tr>
<td><strong>Rounding modes for FADD and FMUL</strong></td>
<td>All 4 IEEE, round to nearest, zero, Inf, -Inf</td>
<td>All 4 IEEE, round to nearest, zero, Inf, -Inf</td>
<td>All 4 IEEE, round to nearest, zero, Inf, -Inf</td>
</tr>
<tr>
<td><strong>Denormal handling</strong></td>
<td>Full speed</td>
<td>Supported, costs 1000's of cycles</td>
<td>Supported only for results, not input operands (input denormals flushed-to-zero)</td>
</tr>
<tr>
<td><strong>NaN support</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Overflow and Infinity support</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>FMA</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Square root</strong></td>
<td>Software with low-latency FMA-based convergence</td>
<td>Hardware</td>
<td>Software only</td>
</tr>
<tr>
<td><strong>Division</strong></td>
<td>Software with low-latency FMA-based convergence</td>
<td>Hardware</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 + step</td>
</tr>
<tr>
<td><strong>Reciprocal sqrt estimate accuracy</strong></td>
<td>23 bit</td>
<td>12 bit</td>
<td>12 bit + step</td>
</tr>
<tr>
<td><strong>log2(x) and 2^x estimates accuracy</strong></td>
<td>23 bit</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
## GPU Single Floating Point Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>G80</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>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, only clamps to max norm</td>
<td>Yes</td>
<td>Yes</td>
<td>No, infinity</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>
Do GPUs do real IEEE 754 FP?

- G8x/GT200 GPU FP is IEEE 752
  - Comparable to other processors
  - More precise / usable in some ways
  - Less precise in other ways

- GPU FP getting better every generation
  - Double precision supported in GT200
  - Goal: best of class in by 2009
Myth of GPU Computing

- GPUs layer normal programs on top of graphics
- GPU architectures are
  - Very wide (1000s) SIMD machines
  - ...on which branching is impossible or prohibitive
  - ...with 4-wide vector registers
- GPUs are power inefficient
- GPUs don’t do real floating point
CUDA Programming Model
CUDA Goal: Easy to Program

**Strategies:**

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

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

- **Simple parallel abstractions**
  - Simple barrier synchronization
  - Shared memory semantics
  - *Hardware-managed* hierarchy of threads
Goal: Performance per millimeter

For GPUs, performance == throughput

Strategy: hide latency with computation not cache
  - Heavy multithreading

Implication: need many threads to hide latency
  - Occupancy – typically need 128 threads/SM minimum
  - Multiple thread blocks/SM good to minimize effect of barriers

Strategy: Single Instruction Multiple Thread (SIMT)
  - Balances performance with ease of programming
CUDA Programming Model

- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
  - Synchronize their execution
  - Communicate via shared memory
- Parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables
- CUDA threads vs CPU threads
  - CUDA thread switching is free
  - CUDA uses many threads per core
IDs and Dimensions

- **Threads:**
  - 3D IDs, unique within a block

- **Blocks:**
  - 2D IDs, unique within a grid

- **Dimensions set at launch time**
  - Can be unique for each section

- **Built-in variables:**
  - threadIdx, blockIdx
  - blockDim, gridDim
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 has its own program counter, variables (registers), processor state, etc.
  - Threads work different data based on its threadID

```
threadID = 0 1 2 3 4 5 6 7

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

Hierarchy of Concurrent Threads

- **Thread block** = virtualized multiprocessor
- **Kernel** = grid of thread blocks

Threads in the same block can be **synchronized with barriers**

```cpp
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
Example: Increment Array Elements

Increment N-element vector \( \mathbf{a} \) by scalar \( b \)

Let's assume \( N=16 \), \( \text{blockDim}=4 \)  \( \rightarrow \) 4 blocks

\[
\text{int idx} = \text{blockDim}.x \times \text{blockId}.x + \text{threadId}.x;
\]

- \( \text{blockId}.x=0 \)
- \( \text{blockDim}.x=4 \)
- \( \text{threadId}.x=0,1,2,3 \)
- \( \text{idx}=0,1,2,3 \)

- \( \text{blockId}.x=1 \)
- \( \text{blockDim}.x=4 \)
- \( \text{threadId}.x=0,1,2,3 \)
- \( \text{idx}=4,5,6,7 \)

- \( \text{blockId}.x=2 \)
- \( \text{blockDim}.x=4 \)
- \( \text{threadId}.x=0,1,2,3 \)
- \( \text{idx}=8,9,10,11 \)

- \( \text{blockId}.x=3 \)
- \( \text{blockDim}.x=4 \)
- \( \text{threadId}.x=0,1,2,3 \)
- \( \text{idx}=12,13,14,15 \)
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()
{

}
```

CUDA program

```c
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    a[idx] = a[idx] + b;
}

void main()
{

}
```
Scalability: Make Blocks Independent

- Thread blocks can run in any order
  - Concurrently or sequentially
  - Hardware does not synchronize between blocks
- This independence gives scalability:
  - A kernel scales across any number of parallel cores
Memory Model of Threads and Thread Blocks

Thread

Per-thread Local Memory

Block

Per-block Shared Memory
Memory Model with Each GPU

Kernel 0

Per-device Global Memory

Kernel 1

Sequential Kernels
Memory Model of Host with Multiple GPUs

Host memory

`cudaMemcpy()`

Device 0 memory

Device 1 memory
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, N);

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

// free device memory
cudaFree(d_A);
free(h_A);
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`
Parallel Algorithms in CUDA
Common Situations in Parallel Computation

- Many parallel threads need to generate a single result value
  - Reduction

- Many parallel threads that need to partition data
  - Split

- Many parallel threads and variable output per thread
  - Compact / Allocate
Parallel Reductions

Common Data Parallel Operation
- **Reduce** vector to a single value
- Operator: +, *, min/max, AND/OR
- Tree-based implementation

```
          3  1  7  0  4  1  6  3
           /   /   /   /   /   /   /
          4   7   5   9   11  14  25
```
Split Operation

- Given an array of true and false elements (and payloads)
  - Payload: [3, 1, 7, 0, 4, 1, 6, 3]

- Return an array with all true elements at the beginning
  - Payload: [3, 0, 6, 1, 7, 4, 1, 3]
Variable Output Per Thread (1): Compact

Remove null elements

3 0 7 0 4 1 0 3

3 7 4 1 3
Variable Output Per Thread (2): Allocation

Allocate Variable Storage Per Thread

\[
\begin{array}{cccc}
2 & 1 & 0 & 3 & 2 \\
\end{array}
\]

\[
\begin{array}{ccc}
A & C \\
B \\
\end{array} \begin{array}{ccc}
D & G \\
E & H \\
F \\
\end{array}
\]
Parallel Prefix Sum (Scan)

Given an array \( A = [a_0, a_1, \ldots, a_{n-1}] \)
and a binary associative operator \( \oplus \) with identity \( I \),

\[
\text{scan}(A) = [I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]
\]

Example: if \( \oplus \) is addition, then scan on the set

\[
[3 1 7 0 4 1 6 3]
\]

returns the set

\[
[0 3 4 11 11 15 16 22]
\]
Build the Sum Tree

| 3 | 1 | 7 | 0 | 4 | 1 | 6 | 3 |

Assume array is already in shared memory
Build the Sum Tree

Each $\oplus$ corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value $stride$ elements away to its own value
Build the Sum Tree

Stride 1

Stride 2

Iteration 2, $n/4$ threads

Each $+$ corresponds to a single thread.

Iterate log(n) times. Each thread adds value stride elements away to its own value.
Iterate log(n) times. Each thread adds value \textit{stride} elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering.
We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.
Build Scan From Partial Sums

| 3 | 4 | 7 | 11 | 4 | 5 | 6 | 0 |
Build Scan From Partial Sums

Iterate $\log(n)$ times. Each thread adds value $stride$ elements away to its own value, and sets the value $stride$ elements away to its own previous value.
Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

Each thread corresponds to a single thread.
Iterate log(n) times. Each thread adds value \textit{stride} elements away to its own value, and sets the value \textit{stride} elements away to its own \textit{previous} value.

Each \ding{162} corresponds to a single thread.
Done! We now have a completed scan that we can write out to device memory.

Total steps: $2 \cdot \log(n)$.
Total work: $2 \cdot (n-1)$ adds = $O(n)$ Work Efficient!
CUDA Scan Performance

GPU vs. CPU
20x
CUDA vs. OpenGL
7x

GeForce 8800 GTX, Intel Core2 Duo Extreme 2.93 GHz
Application: Stream Compaction

1M elements: ~0.6-1.3ms
16M elements: ~8-20ms

Perf depends on # elements retained

Harris, M., S. Sengupta, and J.D. Owens. “Parallel Prefix Sum (Scan) in CUDA”. GPU Gems 3
Sort 4M 32-bit integers: 165ms

Can also sort each block and merge. Slower due to cost of merge.

Perform split operation on each bit using scan.

Split based on least significant bit.

e = Set a “1” in each “0” input bits.

\[ t = \text{Scan input} \]
\[ f = \text{Scan the Is} \]
\[ \text{totalFalsees} = e[\max] + f[\max] \]

\[ d = b?t:f \]
CUDA Tools and Resources
CUDA Programming Resources

**CUDA Toolkit**
- Compiler and libraries
- Free download for Windows, Linux, and Mac OSX

**CUDA SDK**
- Code samples
- Whitepapers

**Instruction materials**
- Slides and audio
- Parallel programming course at University of Illinois UC
- Tutorials

**Development tools**

**Libraries**
GPU Tools

**Profiler**
- Available now for all supported OSs
- Command-line or GUI
- Sampling signals on GPU for:
  - Memory access parameters
  - Execution (serialization, divergence)

**Debugger**
- Demo shown at SC07
- Runs on the GPU

**Emulation mode**
- Compile and execute in emulation on CPU
- Allows CPU-style debugging in GPU source
New Features

CUDA 2.02 Beta
- Beta available on the NVIDIA website
- Support for GeForce GTX 260 & 280:
  - Double precision
  - Integer atomic operations in shared memory
- New features:
  - 3D textures
  - Video Decoding Interface with Compute 1.1+ GPU
  - Improved and extended Direct3D interoperability

CUDA implementation on multi-core CPUs
- Beta coming soon
Double Precision

- NVIDIA GPUs (G8x and G9x) are single precision
  - IEEE 32-bit floating-point precision ("FP32")
  - You can use double, but it gets demoted to float

- NVIDIA GPUs (GT200) have double precision
  - IEEE 64-bit floating-point precision ("FP64")
  - Double precision will be slower (more register pressure and more cycles)

- Be explicit about float and double!
  - Use double only where needed
Sample Applications
CUDA N-Body Simulation

10B interactions/s
16K bodies

44 FPS
x 20 FLOPS / interaction
x $16K^2$ interactions / frame

= 240 GFLOP/s on G80
DXT Compression

- Offline
- Runtime
- Real-Time

256x256 RGB = 256 kB
128x128 RGB = 64 kB
256x256 DXT1 = 32 kB
Histogram

- Representation of the distribution of colors in an image

- Applications:
  - Image Analysis
  - HDR Tone Mapping

Reinhard HDR Tonemapping operator

HDR in Valve’s source engine
CUDA Histogram is 300x faster than previous GPGPU approaches

<table>
<thead>
<tr>
<th></th>
<th>64 bins</th>
<th>256 bins</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA¹</td>
<td>6500 MB/s</td>
<td>3676 MB/s</td>
</tr>
<tr>
<td>R2VB²</td>
<td>22.8 MB/s</td>
<td>42.6 MB/s</td>
</tr>
<tr>
<td>CPU³</td>
<td>826 MB/s</td>
<td>1096 MB/s</td>
</tr>
</tbody>
</table>

¹ http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#histogram64
² Efficient Histogram Generation Using Scattering on GPUs, T. Sheuermann, AMD Inc, I3D 2007
³ Intel Core 2 @ 2.9 GHz
Game AI Breakdown

- **3 Main AI Computations to Accelerate**
  - Spatial reasoning
  - Decision making
  - Path finding

- **Computation breakdown:**
  - For “bot” simulation:
    - Spatial reasoning: 35%
    - Path finding: 65%
  - For crowd simulation:
    - Spatial reasoning: 5%
    - Path finding: 95%

- Decision making always negligible

- Biggest opportunity for GPU acceleration is path finding
A* Algorithm

- Commonly used path finding algorithm
  - A* itself is not very parallel
  - Parallelism comes from computing many separate A* paths in parallel
    - Many units moving simultaneously
    - Massive worlds with thousands of characters
CUDA Particle-Based Fluid

- Grid-based fluids have limitations for use in games
  - Expensive, constrained to a box
- Smoothed Particle Hydrodynamics (SPH) simulates fluid as a collection of interacting particles
  - Localized collisions and pressure distribution
- CUDA enables **dynamic** construction of a uniform grid **data structure** to accelerate neighborhood computations
- 32K particles at 60fps
  - GeForce 8800 GTX
GPU Fluid Simulation
CUDA and Physics: PhysX implemented in Cuda

**Typical physics core simulation features**
- Rigid Body Dynamics
- Universal Collision-Detection
- Joints, Springs and Motors
- Advanced ragdoll and vehicle constraints

**High performance realism (beyond the basics)**
- **Volumetric Fluids (SPH)**
  - CCD with rigid bodies and static geometry
  - One- or two-way interaction with rigid bodies
- Cloth and **Soft Bodies**
  - Attachment and CCD with rigid bodies and static geometry
  - Cloth self collision
  - Tearing
  - Derivatives: Sheet metal and vegetation
Where to go from here


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

Questions?