Outline

Motivation for optimizing in CUDA
Demo performance increases
Tesla 10-series architecture details
Optimization case studies
  Particle Simulation
  Finite Difference
Summary
Motivation for Optimization

- 20-50X performance over CPU-based code
- Tesla 10-series chip has 1 TeraFLOPs compute
- A Tesla workstation can outperform a CPU cluster

Demos
- Particle Simulation
- Finite Difference
- Molecular Dynamics

Need to optimize code to get performance
Not too hard – 3 main rules
Tesla 10-series Architecture
Tesla 10-Series Architecture

- Massively parallel general computing architecture
- 30 Streaming multiprocessors @ 1.45 GHz with 4.0 GB of RAM
  - 1 TFLOPS single precision (IEEE 754 floating point)
  - 87 GFLOPS double precision
10-Series Streaming Multiprocessor

- **8 SP Thread Processors**
  - IEEE 754 32-bit floating point
  - 32-bit float and 64-bit integer
  - 16K 32-bit registers
- **2 SFU Special Function Units**
- **1 Double Precision Unit (DP)**
  - IEEE 754 64-bit floating point
  - Fused multiply-add
- **Scalar register-based ISA**
- **Multithreaded Instruction Unit**
  - 1024 threads, hardware multithreaded
  - Independent thread execution
  - Hardware thread scheduling
- **16KB Shared Memory**
  - Concurrent threads share data
  - Low latency load/store
10-series DP 64-bit IEEE floating point

- IEEE 754 64-bit results for all DP instructions
  - DADD, DMUL, DFMA, DtoF, FtoD, Dtol, ItoD, DMAX, DMIN
  - Rounding, denorms, NaNs, +/- Infinity
- Fused multiply-add (DFMA)
  - \( D = A \times B + C \); with no loss of precision in the add
  - DDIV and DSQRT software use FMA-based convergence
- IEEE 754 rounding: nearest even, zero, +inf, -inf
- Full-speed denormalized operands and results
- No exception flags
- Peak DP (DFMA) performance 87 GFLOPS at 1.45 GHz
- Applications will almost always be bandwidth limited before limited by double precision compute performance?
Optimizing CUDA Applications For 10-series Architecture
General Rules for Optimization

- Optimize memory transfers
  - Minimize memory transfers from host to device
  - Use shared memory as a cache to device memory
  - Take advantage of coalesced memory access

- Maximize processor occupancy
  - Optimize execution configuration

- Maximize arithmetic intensity
  - More computation per memory access
  - Re-compute instead of loading data
Data Movement in a CUDA Program

Host Memory
Device Memory
[Shared Memory]
COMPUTATION
[Shared Memory]
Device Memory
Host Memory
Particle Simulation Example

Newtonian mechanics on point masses:

```c
struct particleStruct{
    float3 pos;
    float3 vel;
    float3 force;
};

pos = pos + vel*dt
vel = vel + force/mass*dt
```
Particle Simulation Applications

- Film Special Effects
- Game Effects
- Monte-Carlo Transport Simulation
- Fluid Dynamics
- Plasma Simulations
1 million non-interacting particles
Radial (inward) and Vortex (tangent) force per particle
Expected Performance

- **1 Million Particles**
  - Pos, Vel = 36 bytes per particle = 36MB total

- **Host to device transfer (PCI-e Gen2)**
  - $2 \times 36\text{MB} / 5.2\text{GB/s} \rightarrow 13.8\text{ms}$

- **Device memory access**
  - $2 \times 36\text{MB} / 80\text{GB/s} \rightarrow 0.9\text{ms}$

- **1 TFLOPS / 1 million particles**
  - Compute Euler Integration $\rightarrow 0.02\text{ms}$
Visual Profiler

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Method</th>
<th>GPU Time</th>
<th>CPU Time</th>
<th>Occupancy</th>
<th>gld_incoherent</th>
<th>gld_coherent</th>
<th>gsu_incoherent</th>
<th>gsu_coherent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>memcopy</td>
<td>3.236</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>memcopy</td>
<td>2.752</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>memcopy</td>
<td>2.88</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.00126</td>
<td>0.00126</td>
<td>0.00126</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>memcopy</td>
<td>2.88</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.00126</td>
<td>0.00126</td>
<td>0.00126</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>memcopy</td>
<td>2.752</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.00126</td>
<td>0.00126</td>
<td>0.00126</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>memcopy</td>
<td>2.752</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.00126</td>
<td>0.00126</td>
<td>0.00126</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>ComplexPoint</td>
<td>2.816</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>memcopy</td>
<td>2.752</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0.00126</td>
<td>0.00126</td>
<td>0.00126</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>memcopy</td>
<td>2.752</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Columns plot

- gld_incoherent
- gld_coherent
- gsu_incoherent
- gsu_coherent

ComplexPointwiseMulixedScale: 11

© NVIDIACorporation 2008
Measured Performance

- Host to device transfer (PCI-e Gen2)
  - 15.3 ms (one-way)
- Integration Kernel (including device memory access)
  - 1.32 ms
Host to Device Memory Transfer

Host Memory
Device Memory
Shared Memory
COMPUTATION
Shared Memory
Device Memory
Host Memory
Host to Device Memory Transfer

cudaMemcpy(dst, src, nBytes, direction)
  Can only go as fast as the PCI-e bus

Use page-locked host memory
  Instead of malloc(…), use cudaMallocHost(…)
  Prevents OS from paging host memory
  Allows PCI-e DMA to run at full speed

Use asynchronous data transfers
  Requires page-locked host memory

Copy all data to device memory only once
  Do all computation locally on T10 card
Asynchronous Data Transfers

- Use asynchronous data transfers
- Requires page-locked host memory

```c
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync(dst1, src1, size, dir, stream1);
kernel<<<grid, block, 0, stream1>>>(...);

cudaMemcpyAsync(dst2, src2, size, dir, stream2);
kernel<<<grid, block, 0, stream2>>>(...);
```
OpenGL Interoperability
Rendering directly from device memory

- OpenGL buffer objects can be mapped into the CUDA address space and then used as global memory
  - Vertex buffer objects
  - Pixel buffer objects
- Allows direct visualization of data from computation
  - No device to host transfer with Quadro or GeForce
  - Data stays in device memory – very fast compute / viz
  - Automatic DMA from Tesla to Quadro (via host for now)
- Data can be accessed from the kernel like any other global data (in device memory)
Register a buffer object with CUDA
- `cudaGLRegisterBufferObject(GLuint buffObj);`
- OpenGL can use a registered buffer only as a source
- Unregister the buffer prior to rendering to it by OpenGL

Map the buffer object to CUDA memory
- `cudaGLMapBufferObject(void **devPtr, GLuint buffObj);`
- Returns an address in global memory
- Buffer must be registered prior to mapping

Launch a CUDA kernel to process the buffer

Unmap the buffer object prior to use by OpenGL
- `cudaGLUnmapBufferObject(GLuint buffObj);`

Unregister the buffer object
- `cudaGLUnregisterBufferObject(GLuint buffObj);`
- Optional: needed if the buffer is a render target

Use the buffer object in OpenGL code
Moving Data to/from Device Memory

Host Memory
Device Memory
Shared Memory
COMPUTATION
Shared Memory
Device Memory
Host Memory
Device and Shared Memory Access

- SM’s can access device memory at 80 GB/s
- But, with hundreds of cycles of latency!
- Pipelined execution hides latency
- Each SM has 16KB of shared memory
  - Essentially a user managed cache
  - Latency comparable to registers
- Reduces load/stores to device memory
- Threads cooperatively use shared memory
- Best case – multiple memory access per thread, maximum use of shared memory
Parallel Memory Sharing

- **Registers**: per-thread
  - Private per thread
  - Auto variables, register spill
- **Shared Memory**: per-block
  - Shared by threads of block
  - Inter-thread communication
- **Device Memory**: per-application
  - Shared by all threads
  - Inter-Grid communication

Sequential Grids in Time

© NVIDIA Corporation 2008
P[idx].pos = P[idx].pos + P[idx].vel * dt;
P[idx].vel = P[idx].vel + P[idx].force / mass;

- Data is accessed directly from device memory in this usage case
- .vel is accessed twice (6 float accesses)
- Hundreds of cycles of latency each time
- Make use of shared memory?
Shared memory as a cache

__shared__ float3 s_pos[N_THREADS];
__shared__ float3 s_vel[N_THREADS];
__shared__ float3 s_force[N_THREADS];

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

s_pos[tx] = P[idx].pos;
s_vel[tx] = P[idx].vel;
s_force[tx] = P[idx].force;

s_pos[tx] = s_pos[tx] + s_vel[tx] * dt;
s_vel[tx] = s_vel[tx] + s_force[tx]/mass * dt ;

P[idx].pos = s_pos[tx];
P[idx].vel = s_vel[tx];
Coalesced Device Memory Access on 10-series architecture

- When half warp (16 threads) accesses contiguous region of device memory
- 16 data elements loaded in one instruction
  - int, float: 64 bytes (fastest)
  - int2, float2: 128 bytes
  - int4, float4: 256 bytes (2 transactions)
- Regions aligned to multiple of size
- If un-coalesced, issues 16 sequential loads
Particle Simulation Example
Worst Case for Coalescing!

```c
struct particleStruct{
    float3 pos;
    float3 vel;
    float3 force;
};
```

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load pos.x</td>
<td>0</td>
<td>36</td>
<td>72</td>
<td>108</td>
<td>...540</td>
</tr>
<tr>
<td>Load pos.y</td>
<td>4</td>
<td>40</td>
<td>76</td>
<td>112</td>
<td>...544</td>
</tr>
<tr>
<td>Load pos.z</td>
<td>8</td>
<td>44</td>
<td>80</td>
<td>118</td>
<td>...548</td>
</tr>
</tbody>
</table>
Coalesced Memory Access

- Use structure of arrays instead
  - float3 pos[nParticles]
  - float3 vel[nParticles]
  - float3 force[nParticles]
- Accesses coalesced within a few segments

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load pos[idx].x</td>
<td>0</td>
<td>12</td>
<td>24</td>
<td>36</td>
<td>...180</td>
</tr>
<tr>
<td>Load pos[idx].y</td>
<td>4</td>
<td>16</td>
<td>28</td>
<td>40</td>
<td>...184</td>
</tr>
<tr>
<td>Load pos[idx].z</td>
<td>8</td>
<td>20</td>
<td>32</td>
<td>44</td>
<td>...188</td>
</tr>
</tbody>
</table>

- Only using 1/3 bandwidth - Not ideal
Better Coalesced Access
Option 1 – Structure of Arrays

- Have separate arrays for pos.x, pos.y,…
  
  float posx[nParticles];
  float posy[nParticles];
  float posz[nParticles];

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>…15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load posx[idx]</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>…60</td>
</tr>
<tr>
<td>Load posy[idx]</td>
<td>64</td>
<td>68</td>
<td>72</td>
<td>76</td>
<td>…124</td>
</tr>
<tr>
<td>Load posz[idx]</td>
<td>128</td>
<td>132</td>
<td>136</td>
<td>140</td>
<td>…188</td>
</tr>
</tbody>
</table>

All threads of warp within 64byte region – 2x
Better Coalesced Access Option 2 - Typecasting

Load as array of floats (3x size), then typecast to array of float3 for convenience

float fdata[16*3]

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load fdata[i+0]</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>...60</td>
</tr>
<tr>
<td>Load fdata[i+16]</td>
<td>64</td>
<td>68</td>
<td>72</td>
<td>76</td>
<td>...124</td>
</tr>
<tr>
<td>Load fdata[i+32]</td>
<td>128</td>
<td>132</td>
<td>136</td>
<td>140</td>
<td>...188</td>
</tr>
</tbody>
</table>

float3* pos = (float3*)&fdata
Shared Memory and Computation

Host Memory
Device Memory
Shared Memory
COMPUTATION
Shared Memory
Device Memory
Host Memory
Details of Shared Memory

Many threads accessing memory
- Therefore, memory is divided into banks
- Essential to achieve high bandwidth

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

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

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

- **Random 1:1 Permutation**

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

© NVIDIA Corporation 2008
Bank Addressing Examples

2-way Bank Conflicts
  Linear addressing stride == 2

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
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
Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 8
Bank 9
Bank 15

© NVIDIA Corporation 2008
Shared memory bank conflicts

Shared memory access is comparable to registers if there are no bank conflicts.

Use the visual profiler to check for conflicts.
- `warp_serialize` signal can usually be used to check for 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 identical address, 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.
Arrays of float3 in shared memory

float3 s_pos[N_THREADS]

Do any threads of a half-warp access same bank?

<table>
<thead>
<tr>
<th>Thread</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>s_pos.x</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>30</td>
<td>33</td>
<td>36</td>
<td>39</td>
<td>42</td>
<td>45</td>
</tr>
<tr>
<td>bank</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>14</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>13</td>
</tr>
</tbody>
</table>

No bank conflicts 😊

Always true when stride is a prime of 16
Optimizing Computation

- Execution Model Details
- SIMT Multithread Execution
- Register and Shared Memory Usage
- Optimizing for Execution Model
- 10-series Architecture Details
- Single and Double Precision Floating Point
- Optimizing Instruction Throughput
Thread:
- Runs a kernel program and performs the computation for 1 data item.
- Thread Index is a built-in variable
- Has a set of registers containing it’s program context
NVIDIA multi-tier data parallel model

**Warp:**
- 32 Threads executed together
- Processed in SIMT on SM
- All threads execute all branches

**Half Warp:**
- 16 Threads
- Coordinated memory access
- Can coalesce load/stores in batches of 16 elements
NVIDIA multi-tier data parallel model

Block:
- 1 or more warps running on the same SM
- Different warps can take different branches
- Can synchronize all warps within a block
- Have common shared memory for extremely fast data sharing

Thread

Warp = 32 Threads

Block of Threads

© NVIDIA Corporation 2008
SIMT: Single-Instruction Multi-Thread

Warp: the set of 32 parallel threads that execute a SIMT instruction

Hardware implements zero-overhead warp and thread scheduling

Deeply pipelined to hide memory and instruction latency

SIMT warp diverges and converges when threads branch independently

Best efficiency and performance when threads of a warp execute together
Register and Shared Memory Usage

**Registers**
- Each block has access to a set of registers on the SM
- 8-series has 8192 32-bit registers
- 10-series has 16384 32-bit registers
- Registers are partitioned among threads
- Total threads * registers/thread should be < number registers

**Shared Memory**
- 16KB of shared memory on SM
- If blocks use <8KB, multiple blocks may run on one SM
- Warps from multiple blocks
Optimizing Execution Configuration

Use maximum number of threads per block
- Should be multiple of warp size (32)
- More warps per block, deeper pipeline
- Hides latency, gives better processor occupancy
- Limited by available registers

Maximize concurrent blocks on SM
- Use less than 8KB shared memory per block
- Allows more than one block to run on an SM
- Can be a tradeoff for shared memory usage
Maximize Arithmetic Intensity

Particle simulation is still memory bound
How much more computation can we do?
Answer is almost unbelievable – 100x!
DEMO: 500+ GFLOPS!

Can use a higher-order integrator?
- More complex computationally
- Can take much larger time-steps
- Computation vs memory access is worth it!
1M particles x 100 fields
Executes in 8ms on GTX280
1M particles x 100 collision spheres executes in 20ms on GTX280
Particle Simulation Optimization Summary

- Page-lock host memory
- Asynchronous host-device transfer
- Data stays in device memory
- Using shared memory vs. registers
- Coalesced data access
- Optimize execution configuration
- Higher arithmetic intensity
Finite Differences Example

Solving Poisson equation in 2D on fixed grid

\[ \Delta u = f \]

\[ u = u(x,y) \]

\[ f = f(x,y) \]

Gauss-Seidel relaxation

5 - point stencil
Usual Method

Solve sparse matrix problem:

\[ A^*u = -f \]  \hspace{1cm} \text{(use } -f \text{ so } A \text{ is pos-def)}

\[
\begin{vmatrix}
4 & -1 & 0 & -1 & \cdots & 0 & 0 & 0 \\
-1 & 4 & -1 & 0 & -1 & \cdots & 0 & 0 \\
0 & -1 & 4 & -1 & 0 & -1 & \cdots & 0 \\
0 & 0 & 0 & -1 & \cdots & 0 & -1 & 4 \\
\end{vmatrix}
\]

\[ |u| = |-f| \]
Matrix is $N \times N$, where $N = N_x \times N_y$
Even a sparse representation is $N \times M$
u and f are of size $N$
Memory throughput = $N \times (M + 2)$ per frame
For a 1024x1024 grid, $N = 1$ million
For a 2nd order stencil, $M = 5$
For double precision: $1M \times 8 \times (5+2) = 56$MB
Host to device memory transfer takes $10.7$ms
Device memory load/store time $0.7$ms?
Improving Performance

- Transfer data host to device once at start
- 56MB easily fits on a 10-series card
- Iterate to convergence in device memory
- Use shared memory to buffer $u$
  - 4x duplicated accesses per block
- Use constant memory for stencil? (no matrix)
- Use texture memory for $\rho$? (read-only)
Using Shared Memory
Finite Difference Example

- Load sub-blocks into shared memory
  - 16x16 = 256 threads
  - 16x16x8 = 2048 KB shared memory
  - Each thread loads one double

- Need to synchronize block boundaries
  - Only compute stencil on 14x14 center of cell
  - Load ghost cells on edges
  - Overlap onto neighbor blocks
  - Only 2/3 of threads computing?
512x512 grid, Gauss-Seidel
Executes in 0.23ms on GTX280
Constant Memory

- Special section of device memory
  - Read only
  - Cached
- Whole warp, same address - one load
- Additional load for each different address
- Constant memory declared at file scope
- Set by cudaMemcpyToSymbol(…)
Using Constant Memory Finite Difference Example

Declare the stencil as constant memory

```c
__constant__ double stencil[5] = {4, -1, -1, -1, -1};
```

```c
-1

+4

-1

-1

-1
```
Texture Memory

- Special section of device memory
  - Read only
  - Cached by spatial location (1D, 2D, 3D)
- Best performance
  - All threads of a warp hit same cache locale
  - High spatial coherency in algorithm
- Useful when coalescing methods are impractical
Using Texture Memory
Finite Difference Example

- Declare a texture ref
  - `texture<float, 1, ...> fTex;`

- Bind f to texture ref via an array
  - `cudaMallocArray(fArray, ...)`
  - `cudaMemcpy2DToArray(fArray, f, ...);`
  - `cudaBindTextureToArray(fTex, fArray ...);`

- Access with array texture functions
  - `f[x,y] = tex2D(fTex, x,y);`
Finite Difference Performance Improvement

- Maximize execution configuration
  - 256 threads, each loads one double
  - 16 registers * 256 threads = 4096 registers
  - Ok for both 10-series, 8-series 😊

- Maximize arithmetic intensity for 3D
  - 27-point, 4th order stencil
  - Same memory bandwidth
  - More compute
  - Can use fewer grid points
  - Faster convergence
General Rules for Optimization Recap

- Optimize memory transfers
  - Minimize memory transfers from host to device
  - Use shared memory as a cache to device memory
  - Take advantage of coalesced memory access

- Maximize processor occupancy
  - Use appropriate numbers of threads and blocks

- Maximize arithmetic intensity
  - More computation per memory access
  - Re-compute instead of loading data