Programming on GPUs
(CUDA and OpenCL)

Gordon Erlebacher
Nov. 30, 2012
<table>
<thead>
<tr>
<th>Product</th>
<th>Tesla C870</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form Factor</td>
<td>ATX, 4.38&quot; x 12.28&quot;</td>
</tr>
<tr>
<td># of Tesla GPUs</td>
<td>1</td>
</tr>
<tr>
<td>Total Dedicated Memory</td>
<td>1.5 GB GDDR3</td>
</tr>
<tr>
<td>Peak Flops</td>
<td>Over 500 gigaflops</td>
</tr>
<tr>
<td>Floating Point Precision</td>
<td>IEEE 754 single-precision floating point</td>
</tr>
<tr>
<td>Memory Interface</td>
<td>384-bit</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>76.8 GB/sec.</td>
</tr>
<tr>
<td>Max Power Consumption</td>
<td>170W</td>
</tr>
<tr>
<td>System Interface</td>
<td>PCI Express x16</td>
</tr>
<tr>
<td>Auxiliary Power Connectors</td>
<td>Yes (2)</td>
</tr>
<tr>
<td>Number of Slots</td>
<td>2</td>
</tr>
<tr>
<td>Thermal Solution</td>
<td>Active Fansink</td>
</tr>
</tbody>
</table>
### Tesla 1060: Nov. 2008

<table>
<thead>
<tr>
<th><strong>Form Factor</strong></th>
<th>10.5” x 4.376”, Dual Slot</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong># of Tesla GPUs</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong># of Streaming Processor Cores</strong></td>
<td>240</td>
</tr>
<tr>
<td><strong>Frequency of processor cores</strong></td>
<td>1.3GHz</td>
</tr>
<tr>
<td><strong>Single Precision floating point performance (peak)</strong></td>
<td>933</td>
</tr>
<tr>
<td><strong>Double Precision floating point performance (peak)</strong></td>
<td>78</td>
</tr>
<tr>
<td><strong>Floating Point Precision</strong></td>
<td>IEEE 754 single &amp; double</td>
</tr>
<tr>
<td><strong>Total Dedicated Memory</strong></td>
<td>4GB GDDR3</td>
</tr>
<tr>
<td><strong>Memory Speed</strong></td>
<td>800MHz</td>
</tr>
<tr>
<td><strong>Memory Interface</strong></td>
<td>512-bit</td>
</tr>
<tr>
<td><strong>Memory Bandwidth</strong></td>
<td>102GB/sec</td>
</tr>
<tr>
<td><strong>Max Power Consumption</strong></td>
<td>200 W peak, 160 W typical</td>
</tr>
<tr>
<td><strong>System Interface</strong></td>
<td>PCIe x16</td>
</tr>
<tr>
<td><strong>Auxiliary Power Connectors</strong></td>
<td>6-pin &amp; 8-pin</td>
</tr>
<tr>
<td><strong>Thermal Solution</strong></td>
<td>Active fan sink</td>
</tr>
<tr>
<td><strong>Programming environment</strong></td>
<td>CUDA</td>
</tr>
</tbody>
</table>
# GTX 480 (graphics Fermi)

## GPU Engine Specs:

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Cores</td>
<td>480</td>
</tr>
<tr>
<td>Graphics Clock (MHz)</td>
<td>700 MHz</td>
</tr>
<tr>
<td>Processor Clock (MHz)</td>
<td>1401 MHz</td>
</tr>
<tr>
<td>Texture Fill Rate (billion/sec)</td>
<td>42</td>
</tr>
</tbody>
</table>

## Memory Specs:

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Clock (MHz)</td>
<td>1848</td>
</tr>
<tr>
<td>Standard Memory Config</td>
<td>1536 MB GDDR5</td>
</tr>
<tr>
<td>Memory Interface Width</td>
<td>384-bit</td>
</tr>
<tr>
<td>Memory Bandwidth (GB/sec)</td>
<td>177.4</td>
</tr>
</tbody>
</table>

## Feature Support:
<table>
<thead>
<tr>
<th>Feature</th>
<th>GTX 680</th>
<th>GTX 580</th>
<th>GTX 580 Ti</th>
<th>GTX 480</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream Processors</td>
<td>1536</td>
<td>512</td>
<td>384</td>
<td>480</td>
</tr>
<tr>
<td>Texture Units</td>
<td>128</td>
<td>64</td>
<td>64</td>
<td>60</td>
</tr>
<tr>
<td>ROPs</td>
<td>32</td>
<td>48</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>Core Clock</td>
<td>1006MHz</td>
<td>772MHz</td>
<td>822MHz</td>
<td>700MHz</td>
</tr>
<tr>
<td>Shader Clock</td>
<td>N/A</td>
<td>1544MHz</td>
<td>1644MHz</td>
<td>1401MHz</td>
</tr>
<tr>
<td>Boost Clock</td>
<td>1058MHz</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>6.008GHz GDDR5</td>
<td>4.008GHz GDDR5</td>
<td>4.008GHz GDDR5</td>
<td>3.696GHz GDDR5</td>
</tr>
<tr>
<td>Memory Bus Width</td>
<td>256-bit</td>
<td>384-bit</td>
<td>256-bit</td>
<td>384-bit</td>
</tr>
<tr>
<td>Frame Buffer</td>
<td>2GB</td>
<td>1.5GB</td>
<td>1GB</td>
<td>1.5GB</td>
</tr>
<tr>
<td>FP64</td>
<td>1/24 FP32</td>
<td>1/8 FP32</td>
<td>1/12 FP32</td>
<td>1/12 FP32</td>
</tr>
<tr>
<td>TDP</td>
<td>195W</td>
<td>244W</td>
<td>170W</td>
<td>250W</td>
</tr>
<tr>
<td>Transistor Count</td>
<td>3.5B</td>
<td>3B</td>
<td>1.95B</td>
<td>3B</td>
</tr>
<tr>
<td>Manufacturing Process</td>
<td>TSMC 28nm</td>
<td>TSMC 40nm</td>
<td>TSMC 40nm</td>
<td>TSMC 40nm</td>
</tr>
<tr>
<td>Launch Price</td>
<td>$499</td>
<td>$499</td>
<td>$249</td>
<td>$499</td>
</tr>
</tbody>
</table>
Post GTX480 (code name: Kepler)

• Nivida has concentrated on power reduction rather than increase in computational speed
• Latest cards allow for GPU to GPU communication, bypassing the CPU (not checked out)
• Multi-GPU programming allows for unified memory layout
Prehistory
Low level Programming

#Phong lighting
#compute half angle vector
ADD spec.rgb, view, lVec;
DP3 spec.a, spec, spec;
RSQ spec.a, spec.a;
MUL spec.rgb, spec, spec.a;

#compute specular intensity
DP3_SAT spec.a, spec, tmp;
LG2 spec.a, spec.a;
MUL spec.a, spec.a, const.w;
EX2 spec.a, spec.a;

#compute diffuse illum
DP3_SAT dif, tmp, lVec;
ADD_SAT dif.rgb, dif, const;
GLSL/HLSL/Cg

void main() {
    vec4 alpha = texture2DRect(maskBoundariesCond, gl_TexCoord[0].xy);

    // Read the neighboring texture values
    vec4 u = texture2DRect(initialsValues, gl_TexCoord[0].xy);
    vec4 u_right = texture2DRect(initialsValues, gl_TexCoord[0].xy + vec2(h,0.0));
    vec4 u_left = texture2DRect(initialsValues, gl_TexCoord[0].xy - vec2(h,0.0));
    vec4 u_up = texture2DRect(initialsValues, gl_TexCoord[0].xy + vec2(0.0,h));
    vec4 u_bottom = texture2DRect(initialsValues, gl_TexCoord[0].xy - vec2(0.0,h));
    vec4 result = u + dt*((u_right + u_left + u_up + u_bottom) - 4.*u) - f_xyt(u);
    float normx = 0.5*(u_right - u_left).x;
    float normy = 0.5*(u_up - u_bottom).x;
    vec3 norm = vec3(normx, normy, 1./float(tex_size));
    gl_FragColor = vec4(result.x, norm);  // return |grad(u)| in vec[1], vec[2]
}
void Transpose() {
    float a[10][10], b[10][10];

    for (int p = 0; p < 10; p++) {
        for (int q = 0; q < 10; q++) {
            float tmp = a[p][q];
            b[q][p] = a[p][q];
            a[p][q] = tmp;
        }
    }
}
CUDA: Transpose Naïve
C-like programming

// This naive transpose kernel suffers from completely non-coalesced writes.
// It can be up to 10x slower than the kernel on following page for large matrices.
__global__ void transpose_naive(float *odata, float* idata, int width, int height)
{
    unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

    if (xIndex < width && yIndex < height)
    {
        unsigned int index_in  = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
CUDA: Transpose (more efficient)

```c
__global__ void transpose(float *odata, float *idata, int width, int height) {
  __shared__ float block[(BLOCK_DIM+1)*BLOCK_DIM];
  unsigned int xBlock = __mul24(blockDim.x, blockIdx.x);
  unsigned int yBlock = __mul24(blockDim.y, blockIdx.y);
  unsigned int xIndex = xBlock + threadIdx.x;
  unsigned int yIndex = yBlock + threadIdx.y;
  unsigned int index_out, index_transpose;
  if (xIndex < width && yIndex < height) {
    unsigned int index_in = __mul24(width, yIndex) + xIndex;
    unsigned int index_block = __mul24(threadIdx.y, BLOCK_DIM+1) + threadIdx.x;
    block[index_block] = idata[index_in];
    index_transpose = __mul24(threadIdx.x, BLOCK_DIM+1) + threadIdx.y;
    index_out = __mul24(height, xBlock + threadIdx.x) + yBlock + threadIdx.y;
  }
  __syncthreads();

  if (xIndex < width && yIndex < height) {
    odata[index_out] = block[index_transpose];
  }
}
```

Use of shared memory - Decrease memory transfers

__mul24 : hand-tuned functions
Parallelism

Task parallelism
Data parallelism
SIMD
MIMD
Stream (the basis of GPU architectures)
Multi-GPU
Task Parallelism

- Time $t$
- Navier-Stokes Equations
- Elasticity Equations
- $t ightarrow t + dt$
Data Parallelism

Add 2 matrices
\[ C(i,j) = A(i,j) + B(i,j) \]

\[ C(1,1) = A(1,1) + B(1,1) \]
\[ C(2,1) = A(2,1) + B(2,1) \]
\[ C(3,1) = A(3,1) + B(3,1) \]

\[ \ldots \]

\[ C(n,m) = A(n,m) + B(n,m) \]
Single Instruction Multiple Data

Every processor executes the identical instruction at the same time
Multiple Instruction
Multiple Data

Different processors execute different tasks
Stream Processing

Stream Processor

Input stream

Output stream
Stream Processing
block of threads

Input stream

Output stream
Stream Processing
block of threads

Input stream

Output stream
Multi-GPU

- GPU 1
- CPU 2
- Infiniband

- GPU 2
- CPU 2

...
Computer Unified Data Architecture

Based on the GTX480 (2010)

Acronym is no longer used
CUDA

- Compute Unified Device Architecture
- Stream Processor
- Remove graphics aspect of GPUs
- Geared towards scientific programming
- Can do graphics programming by moving data from an array to a OpenGL framebuffer and then using standard GPU programming
OpenCL

• A GPU-based language that
  – Leverages OpenGL (Graphics language)
  – Is portable across multiple vendors
  – Works on various graphics devices
    • Nvidia, ATI, multi-core chips, etc.

• Many similarities with CUDA
  – Less general
  – Less powerful
  – More portable
  – In early stages (was true in 2010, is still true today)
Memory Bandwidth

PCle2 Express-16

CPU

3 Gbytes/sec

GPU

70 Gbytes/sec (2007)

177 Gbytes/sec for GTX480 (2010)

PCle3 (2010), 5 Gbyte/sec

Increased disparity between speeds
Several Streaming Multiprocessors per GPU (280GTX has 24 MP) 
480GTX has 15 MP
Streaming Multiprocessor
Multiple stream processors
GTX480: 32 cores per SM
Grid of Blocks

Block of Threads = a grid of blocks

Single Stream Processor

Input to GPU
Anatomy of Unified Architecture

GEForce 8x

Multiprocessor 1

Thread

Registers

Shared Memory

Constant Cache

Texture Cache

Local Memory

Multiprocessor N

Thread

Registers

Shared Memory

Constant Cache

Texture Cache

Local Memory

Global Device Memory

O(1)

O(10)

O(100)
Fermi’s 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).
<table>
<thead>
<tr>
<th></th>
<th>Compute Capability</th>
<th>Number of Multiprocessors</th>
<th>Number of CUDA Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT 325M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GeForce 9700M GT, GT 240M, GT 230M</td>
<td>1.1</td>
<td>6</td>
<td>48</td>
</tr>
<tr>
<td>GeForce GT 120, 9500 GT, 8600 GTS, 8600 GT, 9700M GT, 9650M GS, 9600M GT, 9600M GS, 9500M GS, 8700M GT, 8600M GT, 8600M GS</td>
<td>1.1</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>GeForce 210, 310M, 305M</td>
<td>1.2</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>GeForce G100, 8500 GT, 8400 GS, 8400M GT, 9500M G, 9300M G, 8400M GS, 9400 mGPU, 9300 mGPU, 8300 mGPU, 8200 mGPU, 8100 mGPU, G210M, G110M</td>
<td>1.1</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>GeForce 9300M GS, 9200M GS, 9100M G, 8400M G, G105M</td>
<td>1.1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td><strong>Tesla C2050</strong></td>
<td><strong>2.0</strong></td>
<td><strong>14</strong></td>
<td><strong>448</strong></td>
</tr>
<tr>
<td><strong>Tesla S1070</strong></td>
<td><strong>1.3</strong></td>
<td><strong>4x30</strong></td>
<td><strong>4x240</strong></td>
</tr>
<tr>
<td><strong>Tesla C1060</strong></td>
<td><strong>1.3</strong></td>
<td><strong>30</strong></td>
<td><strong>240</strong></td>
</tr>
<tr>
<td><strong>Tesla S870</strong></td>
<td><strong>1.0</strong></td>
<td><strong>4x16</strong></td>
<td><strong>4x128</strong></td>
</tr>
<tr>
<td><strong>Tesla D870</strong></td>
<td><strong>1.0</strong></td>
<td><strong>2x16</strong></td>
<td><strong>2x128</strong></td>
</tr>
<tr>
<td><strong>Tesla C870</strong></td>
<td><strong>1.0</strong></td>
<td><strong>16</strong></td>
<td><strong>128</strong></td>
</tr>
<tr>
<td>Model</td>
<td>Compute Capability</td>
<td>Number of Multiprocessors</td>
<td>Number of CUDA Cores</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------</td>
<td>---------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>GeForce GTX 480</td>
<td>2.0</td>
<td>15</td>
<td>480</td>
</tr>
<tr>
<td>GeForce GTX 470</td>
<td>2.0</td>
<td>14</td>
<td>448</td>
</tr>
<tr>
<td>GeForce GTX 295</td>
<td>1.3</td>
<td>2x30</td>
<td>2x240</td>
</tr>
<tr>
<td>GeForce GTX 285, GTX 280,</td>
<td>1.3</td>
<td>30</td>
<td>240</td>
</tr>
<tr>
<td>GTX 275</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GeForce GTX 260</td>
<td>1.3</td>
<td>24</td>
<td>192</td>
</tr>
<tr>
<td>GeForce 9800 GX2</td>
<td>1.1</td>
<td>2x16</td>
<td>2x128</td>
</tr>
<tr>
<td>GeForce GTS 250, GTS 150,</td>
<td>1.1</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>9800 GTX, 9800 GTX+, 8800</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GTS 512, GTX 285M, GTX 280M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GeForce 8800 Ultra, 8800 GTX</td>
<td>1.0</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>GeForce 9800 GT, 8800 GT,</td>
<td>1.1</td>
<td>14</td>
<td>112</td>
</tr>
<tr>
<td>GTX 260M, 9800M GTX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GeForce GT 240, GTS 360M,</td>
<td>1.2</td>
<td>12</td>
<td>96</td>
</tr>
<tr>
<td>GTS 350M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GeForce GT 130, 9600 GSO,</td>
<td>1.1</td>
<td>12</td>
<td>96</td>
</tr>
<tr>
<td>8800 GS, 8800M GTX, GTS 260M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GTS 250M, 9800M GT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GeForce 8800 GTS</td>
<td>1.0</td>
<td>12</td>
<td>96</td>
</tr>
<tr>
<td>GeForce GT 335M</td>
<td>1.2</td>
<td>9</td>
<td>72</td>
</tr>
<tr>
<td>GeForce 9600 GT, 8800M GTS,</td>
<td>1.1</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>Technical Specifications</td>
<td>1.0</td>
<td>1.1</td>
<td>1.2</td>
</tr>
<tr>
<td>----------------------------------------------------------------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a grid of thread blocks</td>
<td></td>
<td></td>
<td>65535</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td></td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>512</td>
<td></td>
<td>1024</td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
<td>16 K</td>
<td>32 K</td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
<td></td>
<td>48 KB</td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
<td></td>
<td>512 KB</td>
</tr>
<tr>
<td>Constant memory size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Compute Capability</td>
<td>Number of Multiprocessors</td>
<td>Number of CUDA Cores</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------</td>
<td>-------------------------------------</td>
<td>---------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Cache working set per multiprocessor for texture memory</td>
<td>Device dependent, between 6 KB and 8 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width for a 1D texture or surface reference bound to a CUDA array</td>
<td>8192</td>
<td></td>
<td>32768</td>
</tr>
<tr>
<td>Maximum width for a 1D texture reference bound to linear memory</td>
<td>$2^{27}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width and height for a 2D texture reference bound to linear memory or for a 2D texture or surface reference bound to a CUDA array</td>
<td>$65536 \times 32768$</td>
<td>$65536 \times 65536$</td>
<td></td>
</tr>
<tr>
<td>Maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array</td>
<td>$2048 \times 2048 \times 2048$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of textures that can be bound to a kernel</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of surfaces that can be bound to a kernel</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of instructions per kernel</td>
<td>2 million</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CUDA on GEForce 8800

- # Multiprocessors: depends on card
- Each Multiprocessor: 8 stream processors
  - (8 concurrent blocks)
- Warp size: 32= 2 half-warps
- Max # threads per block: 512, # warps per block: 16
- Registers per MP: 8192 (faster access)
- Shared memory: 16,000 bytes per MP (fast access)
- Constant memory: 64,000 bytes + 8 kB cache per multiprocessor (fast access)
- Max concurrent blocks that can run concurrently on MP: 8
- Max # warps that can run concurrently on MP: 24
- Max # threads that can run concurrently on MP: 768
- Max kernel size: 2 million instructions
CUDA on GTX480

• # Multiprocessors: 15
• Each Multiprocessor: 32 cores
• Warp size: 32
• Max # threads per block: 512, # warps per block: 16
• Registers per MP: 8192 (faster access)
• Shared memory: 48,000 bytes per MP (fast access)
• Constant memory: 64,000 bytes + 8 kB cache per multiprocessor (fast access)
• Max concurrent blocks that can run concurrently on MP: ??
• Max # warps that can run concurrently on MP: ??
• Max # threads that can run concurrently on MP: 1500
• Max kernel size: 2 million instructions
Programming CUDA

• Easy to program, but there are multiple objectives (2007):
  – Maximize number of concurrent running blocks (> 2-3 per MP)
  – Maximize number of concurrent running threads
  – 768 threads/MP => 96 threads/block if all processors are running
  – Optimum block size: 32x16 = 512
  – Shared memory is shared among the threads of a single block
  – The more blocks are running concurrently on a single MP, the less shared memory per block
  – # threads/block should be multiple of warp size
  – Keep enough registers per thread

• Lots of room for code optimization
  – More recent ideas:
    – Maximize register usage
    – Can achieve close to peak performance with subset of threads
Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD)

At any given clock cycle, each processor of each multiprocessor executes the same instruction, but operates on different data.
Treatment of input blocks
Flow of data on GPU

Efficient Programming
Efficient CUDA Programming

- Keep everything on the GPU
- Minimize communication GPU $\leftrightarrow$ CPU
- Minimize transfers
  - Device Memory $\leftrightarrow$ GPU registers
- Minimize number registers to use all threads
- Minimize incoherent reads from Device Memory
- Minimize bank conflicts from in shared memory
- Maximize use of shared memory
- Maximize number of blocks running concurrently
- **Conflicting requirements!**
Efficient CUDA Programming

• Alternatives to maximizing number of threads
  – Maximize use of a single thread
  – Overlay I/O and arithmetic

• Need to study architecture of single Stream Multiprocessor in more detail
  – How are warps handled
  – Warp scheduling
  – How to achieve high performance on the Fermi?
Arithmetic Throughput

GTX480: Peak Flops

• Each cycle, single core
  – Initiate 16 flops (floating point operations)
  – Single flop = multiple, add, or multiply/add (MAD)
    • MAD: \( a*x+b \)
  – 32 cores: \( 32 \times 16 = 512 \) flops/cycle
  – Clock: 1.4 Ghz ➔ 700 Gflop/sec
  – Theoretical max: MAD = 2 ops ➔ 1.4 Tflop/sec
GTX480: peak memory throughput

- Each cycle, single SM
  - 16 x 32-bit Load/Store instructions
- Each cycle, entire GPU
  - 16*15=240 Load/Store instructions
- Each second, entire GPU
  - 240*1.4 10^9 = 384 Giga Load/Store
Two Fundamental Rules

• A thread *never* executes alone
  – It is the warp that executes, i.e., 32 threads
  – On older GPUs, half-warps, i.e., 16 threads

• Warps execute instructions, then wait
Latency

- **Computing Latency**
  - 2 cycles to issue an add, multiply, or MAD for up to two warps
  - 24 cycles to wait for the result

- **Memory Latency**
  - 2 cycles to issue a store/load for a warp
  - 400 cycles to retrieve from DRAM (global memory)

- **Must do many arithmetic operations to cover I/O latency and slow throughput (compared to FP)**

- **So need many warps available, waiting for execution**
Efficiency

- More art than science
- Need many warps. Options:
  - Single large block, many warps
  - Several blocks, fewer warps per block

- Barriers: synchronize all threads in a block
  - If no other blocks, time is wasted
  - Often good idea to have between 2 and 4 blocks per multi-processor
Programming Paradigm

• Minimize access to global memory
• Replace multiple access to global memory by
  – Single transfer to shared memory
  – Multiple access to shared memory
• Only 48 Kbytes of shared memory per SM
  – Under control of the user
  – Be frugal
Memory transfers

- Global to shared memory
  - Watch for coalescing problems
    - Each thread of warp have strict address restrictions for maximum throughput (pre-fermi)
    - On Fermi: there is L2/L1 caches, so coalescing much less of a problem
  - Lack of coalescing leads to less efficient transfer (partial serialization)

- On shared memory side: banking
  - Different threads of warps must be in different banks for throughput to occur in a single cycle per float (once latency is covered)
  - If this is not the case, transfer occur
Banking/Coalescing

- Global Memory
- Shared Memory
- Registers

Coalesced transfers for efficiency

Divided into banks. Successive elements in a warp must lie in different banks for efficiency
Shared memory: banking

- Successive 32-bit words are in successive banks
- Each address of a half-warp must be in a separate bank
- If this is not the case, transfer occurs in multiple requests ➔ slowdown
Deficiencies of OpenCL

- For the most part, OpenCL maps nicely to CUDA and vice-versa. However,
- No templates, namespaces
  - Difficult to create general frameworks
- Limitations on function arguments
- Constant parameters are hard to manage
- Hard to manage pointers
- No structures of structures on GPU
- Cannot access addresses on GPU
  - This is possible with CUDA
- On average, slower than CUDA
- Single code runs on all platforms
  - Efficiency varies a lot on different GPUs
- More recent, less developed
  - More opportunity
OpenCL vs. CUDA

• [http://wiki.tiker.net/CudaVsOpenCL#Code_Portability](http://wiki.tiker.net/CudaVsOpenCL#Code_Portability)

• Bottom line from the above link:

  “Overall, I have done OpenCL for 2 months, and CUDA for 2 days, and I have had more success with CUDA.”
Resources

- GPU Programming Guide 3.1
- Better Performance at Lower Occupancy
  - By Vasily Volvov, GTC2010
  - [www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf](http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf)
- OpenCL Specification (Khronos)
  - [http://www.khronos.org/registry/cl/](http://www.khronos.org/registry/cl/)
- Applications ported to CUDA
- Resources
  - [http://people.sc.fsu.edu/~gerlebacher/gpus/](http://people.sc.fsu.edu/~gerlebacher/gpus/)
Implementations on the GPU

• Voronoi Mesh generators (Evan Bollig)
• Smoothed Particle Hydrodynamics (within Blender) (Ian Johnson) (http://enja.org/2011/03/31/particles-in-bge-improved-code-collisions-and-hose/)
• Two-way interaction between body and fluid using SPH (Andrew Young) (http://andrewfsu.blogspot.com/)
• Radial Basis Functions (Evan Bollig) (http://www.sciencedirect.com/science/article/pii/S0021999112003452)
• Spectral-Element Code (Komatitsch, Erlebacher, Michea, Goedekke) (http://komatitsch.free.fr/published_papers/GPGPU_JPDC_2009.pdf)
CUDA

Let us return to the Spectral Finite-Element code

Wednesday, 12:30 pm
CUDA Implementation

• Each element: $5^3=125$ points
• Each CUDA block = 128 points (waste 3)
• For block max. running concurrently
CPU

GPU device memory

Shared memory
Texture memory
Constant memory
Registers
Arithmetic

Flow of data on GPU

Efficient Programming
tx = threadIdx.x; bx = blockIdx.x;

offset = tx + bx*512

offset < nglob?

no operation

YES

Each thread updates one global point in global arrays:

d_displ_x[offset] += Δt*d_veloc_x[offset] + Δt^2/2*d_accel_x[offset]
d_displ_y[offset] += Δt*d_veloc_y[offset] + Δt^2/2*d_accel_y[offset]
d_displ_z[offset] += Δt*d_veloc_z[offset] + Δt^2/2*d_accel_z[offset]

d_veloc_x[offset] += Δt/2*d_accel_x[offset]
d_veloc_y[offset] += Δt/2*d_accel_y[offset]
d_veloc_z[offset] += Δt/2*d_accel_z[offset]
Kernel 2: top half

Kernel 2

tx = threadIdx.x; bx = blockIdx.x;

Compute the coordinate of the local point of the thread in the spectral element:
K = (tx/25); j = ((tx-K*25)/5); l = (tx-K*25-j*5);

tx<125?

NO

YES

Each thread loads the value of displacement for its local point in shared memory:
iglob = d_ibool[bx*128+tx];
s_dummyx_loc[tx] = d_displ_x[iglob];
s_dummyy_loc[tx] = d_displ_y[iglob];
s_dummyz_loc[tx] = d_displ_z[iglob];

__syncthreads()
Kernel 2: bottom half

Each thread makes numerous local calculations using s_dummy*_loc and local device arrays and stores the results in shared memory.

---

Each thread computes the contribution of a point of each element and adds it to acceleration since there are no common points between spectral elements of the same color.

---

Return
Coloring

Six Elements

Two elements of given color do not share the same color
## Porting SPECFEM3D on CUDA: results

<table>
<thead>
<tr>
<th>Mesh size</th>
<th></th>
<th>GTX 280</th>
<th></th>
<th>8800 GTX</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Version 1</td>
<td>Time / element</td>
<td>Speedup</td>
<td>Time / element</td>
<td>Speedup</td>
</tr>
<tr>
<td>65 MB</td>
<td>Version 1</td>
<td>0.94 µs</td>
<td>21.5</td>
<td>1.5 µs</td>
<td>13.5</td>
</tr>
<tr>
<td>405 MB</td>
<td>Version 2</td>
<td>0.79 µs</td>
<td>24.8</td>
<td>1.3 µs</td>
<td>15</td>
</tr>
<tr>
<td>633 MB</td>
<td>Version 2</td>
<td>0.77 µs</td>
<td>25.3</td>
<td>1.3 µs</td>
<td>15</td>
</tr>
</tbody>
</table>

- **Speedup**

- **Performance evolution**
Efficiency vs packet size

![Graph showing the relationship between computation time per element (µs) and the number of elements in each packet. The graph illustrates a decreasing trend as the packet size increases, indicating improved efficiency. The x-axis represents the number of elements in each packet, ranging from 16 to 8192, while the y-axis shows the computation time per element, ranging from 7 to 0.]
CPU vs GPU versions

![Graph showing CPU vs GPU versions](image-url)
Current work: CUDA + MPI

Previous (classical) communication scheme (blocking MPI)

Communications cost on CPU version \( \sim 5\%\),

On the GPU version, with a speedup of 25,
communication cost \( \sim 58\%\)
"acm" : The Math (FSU) GPU Cluster

InfiniBand (10 GBit/s)

Math Cluster 16 Tflops

x4 Nodes

x17 Nodes

2x Quad Core CPU (~19 GFLOPS)
4GB RAM

~1.3 GB/s (in)
~0.9 GB/s (out)

Tesla C1060 (24x Multiprocessors; ~1 Tflop)
4GB Global RAM

~77 GB/s*

1.5GB Global RAM

Tesla C870 (16x Multiprocessors; ~518 GFLOPS*)
4GB RAM

~1.3 GB/s (in)
~0.9 GB/s (out)

~100 GB/s*

16 Tflops
Conclusion

• GPUs have higher rate of performance increase over time than CPUs
  – always appealing as “research for the future”
• In certain applications GPUs are 15 to 60 times faster than CPUs for low precision
• For certain floating point applications GPU’s and CPU’s performance is comparable
  – can be used as coprocessor
• GPUs are often constrained in memory, but
• It is feasible to use GPUs for numerical simulations
  – Languages: CUDA, RapidMind (cell processors and GPUs)
• Calculations in double precisions should be avoided
  – use mixed precision calculations
Resources

• SC Visualization Lab
  – SC 4 HPs with GeForce 7900, 8800, 280GTX cards
  – 1 HP 9100 with Quadro 5600 for GPU computation and Stereo viewing

• Math resource:
  – 20 processors with 20 Tesla cards.
Coloring

Six Elements

Two elements of given color do not share the same color
Coloring

Six Elements

Two elements of given color do not share the same color