Lecture 9

Performance programming
Fast Reduction
Announcements

• A3 has been posted
  ◆ Develop code on Lilliput
• Try out your lincoln account
  ◆ See *Getting Started with Lincoln* for the details
  ◆ Compile and run the incrArr example
• Next time
  ◆ Be prepared to discuss Volkov and Demmel Paper
• Calendar
  ◆ Triton/SDSC workshop and visit (Lecture 13)
    ◆ Weds Nov 3rd: 3:30 to 5:00
  ◆ Midterm return and lecture 17 on Friday Nov 12 (4p to 5:20p)
  ◆ No class on Tuesday/Thursday Nov 16th/18th

©2010 Scott B. Baden /CSE 260/ Fall '10
Today’s lecture

- Performance Programming
  - Using shared memory
  - Memory coalescing
- Occupancy calculation
- Fast reduction
Streaming processor cluster

- GTX-280 GPU
  10 clusters @ 3 streaming multiprocessors or vector cores
- Each vector core
  - 8 scalar cores: fused multiply adder + multiplier (32 bits), truncate intermediate result
  - Shared memory (16KB) and registers (64 KB)
  - 64-bit fused multiply-adder + 2 super function units (2 fused multiply-adders)
  - 1 FMA + 1 multiply per cycle = 3 flops / cycle / core * 240 cores = 720 flops/cycle
Thread execution model

- Spawn virtualized, hierarchically organized threads
  - Grid ⊇ Block ⊇ Thread
  - Specify number and geometry of threads in a block and similarly for blocks
    - $\Pi: \mathbb{Z}^n \rightarrow \mathbb{Z}^2 \times \mathbb{Z}^3$
    - $\Theta(\Pi_I), \forall \Pi_I \in \Pi$
- Thread Blocks
  - Cooperate, synchronize, access to fast on-chip shared memory
  - Threads in different blocks communicate only through slow global memory
- Threads assigned to SM in units of blocks
- Compiler re-arranges loads to hide latencies
Constraints

- **SM**
  - Up to 8 resident blocks
  - Not more than 1024 threads
  - Up to 32 warps
- **Grid**: 1 or 2-dimensional (64k-1)
- **Block**: 1, 2, or 3-dimensional
  - $\leq$ 512 threads
  - Max dimensions: 512, 512, 64
  - Registers subdivided over threads
  - Synchronization among all threads in the block
- **Warp**
  - All threads warp execute the same instruction: all branches followed
  - Divergence, serialization
Running times of CUDA instructions

• To execute an instruction for all threads of a warp, the warp scheduler must therefore issue the instruction over
  - 4 clock cycles for an integer or single-precision floating-point arithmetic instruction [throughput = 8]
  - 32 cycles for double-precision [1]
  - 16 cycles for single-precision floating-point transcendental [2]
• Let’s change the increment function to the \text{sin}() function …
• Time taken on the device

• –use\_fast\_math
# Bandwidth test results

`./bandwidthTest`

**Device 0: Tesla C1060**

**Host to Device Bandwidth, 1 Device(s), Paged memory**

<table>
<thead>
<tr>
<th>Transfer Size (Bytes)</th>
<th>Bandwidth (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32M</td>
<td>3294.8</td>
</tr>
</tbody>
</table>

**Device to Host Bandwidth, 1 Device(s), Paged memory**

<table>
<thead>
<tr>
<th>Transfer Size (Bytes)</th>
<th>Bandwidth (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32M</td>
<td>2775.0</td>
</tr>
</tbody>
</table>

**Device to Device Bandwidth, 1 Device(s)**

<table>
<thead>
<tr>
<th>Transfer Size (Bytes)</th>
<th>Bandwidth (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32M</td>
<td>73285.0</td>
</tr>
</tbody>
</table>
A better matrix multiply

• Shared memory

• Avoid Thread divergence

• Memory Coalescing and Bank Conflicts
Data access pattern - the basic implementation

- Recall unblocked matrix multiply
- Each thread block computes a block of the result
  - multiplies two skinny matrices of size \( n \times 1 \) and \( 1 \times n \)
  - we load each value \( n \) times

\[
C[i,j] = A[i,:) \ast B[:,j]
\]
Improving locality on using shared memory

- Naïve algorithm
  - Each thread loads all the data it needs, independently loads a row and column of input
  - Each input element loaded multiple times
  - Each thread computes 1 MAD + 2 loads + 1 store

- Blocked algorithm
  - Threads cooperate to load a block of A&B into on-chip shared memory
  - Each thread performs the $ijk$ loop within shared memory
  - Each thread; $b$ multiply-adds + 1 load + 1 store
Memory coalescing (compute capability $\geq 1.2$)

- Memory banks: consecutive addresses can be read very quickly: $\text{shared}[\text{threadIdx.x}] = \text{globalg}[g_{tid}]$

- Certain non-sequential access patterns to global memory degrade performance: $K \mod 16 \neq 0; Q \neq 1$

  \[
  \text{tid} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} + K
  \]
  \[
  \text{tid} = (\text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}) \times Q
  \]

1 transaction - 64B segment

2 transactions - 64B and 32B segments
Bank conflicts in 2d arrays

• All warps in a block access consecutive elements within a row as they step through neighboring columns (matrix multiply)
  \[ I = \text{blockIdx.y} \times \text{by} + \text{ty}; \]
  \[ J = \text{blockIdx.x} \times \text{bx} + \text{tx}; \]
  \[ a[\text{ty}][\text{tx}] = A[I \times \text{N} + k \times \text{by} + \text{tx}]; \]
  \[ b[\text{ty}][\text{tx}] = B[J + \text{N} \times (k \times \text{bx} + \text{ty})]; \]

• Accesses by threads in a block along a column don’t coalesce
Coalesced memory accesses

```c
__shared__ float a[BLK][BLK], b[BLK][BLK];
if ((I < N) && (J < N)){
    float c = 0.0f;
    for (k=0; k < gy; k++){
        a[ty][tx] = A[I*N+k*by+tx];
        b[ty][tx] = B[J+N*(k*bx+ty)];
        __syncthreads();
        for (kk=0; kk < bx; kk++)
            c += = a[ty][kk]*b[kk][tx];
        __syncthreads();
    }
    C[I*N+J] = c;
}
```

Slow:
```c
I = blockIdx.y*by + ty;
J = blockIdx.x*bx + tx;
```
Processing rate and effective bandwidth

Effective bandwidth = \( \frac{N_{\text{read}} + N_{\text{write}}}{\text{time}} \)

- Running time for Matrix Multiplication, \( N=512, b\{x,y\}=(16,16) \): 0.02147 sec
- Effective bandwidth = \( 3*512^3/0.02097 = 18.7 \) GBytes/sec
- Peak bandwidth for C1060
  102 GBytes / sec
Hiding latency

• Peak throughput: 4 cycles per instruction
• How many warps are needed to hide latency?
• How might ILP lead to a smaller value?
• Compiler re-arranges instructions to hide latency, but software pipelining can help

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unit</th>
<th>GTX280</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a = a + b, a = a \times b)</td>
<td>SP</td>
<td>24</td>
</tr>
<tr>
<td>same (w/ b) is in smem</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>(a = a \times b + c)</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>same (w/ b) is in smem</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>(a = \log_2(</td>
<td>a</td>
<td>), a = \sqrt{a})</td>
</tr>
<tr>
<td>(a = a + b, a = a \times b)</td>
<td>DP</td>
<td>48</td>
</tr>
<tr>
<td>(a = a \times b + c)</td>
<td></td>
<td>52</td>
</tr>
</tbody>
</table>
Occupancy Calculator

Dynamic behavior – resource utilization

- Dynamic partitioning → underutilized resources
- Matrix multiply uses 10 reg/thread
- Blocked algorithm (single precision)
  - 16×16 blocks:
    - 2560 registers → 6 blocks (15360/16k)
    - 1024 thread limit → 4 blocks (125 GF)
    - 2076+16 bytes smem ((user + compiler)
  - 8×8 blocks:
    - 640 registers → 25 blocks
    - 8 block limit → 512 threads (53 GF)
    - 540+16 bytes smem

NVIDIA
1.) Select Compute Capability (click): 1.3

2.) Enter your resource usage:
- Threads Per Block: 256
- Registers Per Thread: 8
- Shared Memory Per Block (bytes): 2048

(Don’t edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:
- Active Threads per Multiprocessor: 1024
- Active Warps per Multiprocessor: 32
- Active Thread Blocks per Multiprocessor: 4
- Occupancy of each Multiprocessor: 100%

Physical Limits for GPU:
- Threads / Warp: 32
- Warps / Multiprocessor: 32
- Threads / Multiprocessor: 1024
- Thread Blocks / Multiprocessor: 8
- Total # of 32-bit registers / Multiprocessor: 16384
- Register allocation unit size: 512
- Shared Memory / Multiprocessor (bytes): 16384
- Warp allocation granularity (for register allocation): 2

Allocation Per Thread Block
- Warps: 8
- Registers: 2048
- Shared Memory: 2048

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor

<table>
<thead>
<tr>
<th>Limited by</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Warps</td>
<td>4</td>
</tr>
<tr>
<td>Registers</td>
<td>8</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>8</td>
</tr>
</tbody>
</table>

Thread Block Limit Per Multiprocessor highlighted RED

CUDA Occupancy Calculator
Version: ©2010 Scott B. Baden / CSE 260 / Fall 10 1.5
**Occupyancy calculation with 16 x 16 threads**

\[ \text{Occupancy} = \frac{\# \text{ active warps per SM}}{\text{Maximum possible } \# \text{ active warps}} \]

---

**CUDA GPU Occupancy Calculator**

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click): 1.3

2.) Enter your resource usage:
   - Threads Per Block: 256
   - Registers Per Thread: 10
   - Shared Memory Per Block (bytes): 2092

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:
   - Active Threads per Multiprocessor: 1024
   - Active Warps per Multiprocessor: 32
   - Active Thread Blocks per Multiprocessor: 4
   - Occupancy of each Multiprocessor: 100%

---

**Physical Limits for GPU:**

<table>
<thead>
<tr>
<th>Limit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads / Warp</td>
<td>32</td>
</tr>
<tr>
<td>Warps / Multiprocessor</td>
<td>32</td>
</tr>
<tr>
<td>Threads / Multiprocessor</td>
<td>1024</td>
</tr>
<tr>
<td>Thread Blocks / Multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>Total # of 32-bit registers / Multiprocessor</td>
<td>16384</td>
</tr>
<tr>
<td>Register allocation unit size</td>
<td>512</td>
</tr>
<tr>
<td>Shared Memory / Multiprocessor (bytes)</td>
<td>16384</td>
</tr>
<tr>
<td>Warp allocation granularity (for register allocation)</td>
<td>2</td>
</tr>
</tbody>
</table>

**Allocation Per Thread Block**

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warps</td>
<td>8</td>
</tr>
<tr>
<td>Registers</td>
<td>2560</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>2560</td>
</tr>
</tbody>
</table>

These data are used in computing the occupancy data in blue

**Maximum Thread Blocks Per Multiprocessor**

<table>
<thead>
<tr>
<th>Limit</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited by Max Warps / Multiprocessor</td>
<td>4</td>
</tr>
<tr>
<td>Limited by Registers / Multiprocessor</td>
<td>6</td>
</tr>
<tr>
<td>Limited by Shared Memory / Multiprocessor</td>
<td>6</td>
</tr>
</tbody>
</table>

Thread Block Limit Per Multiprocessor highlighted **RED**

---

©2010 Scott B. Baden / CSE 260/ Fall '10
Full occupancy
8 x 8 thread blocks

2.) Enter your resource usage:

<table>
<thead>
<tr>
<th>Resource Usage</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads Per Block</td>
<td>64</td>
</tr>
<tr>
<td>Registers Per Thread</td>
<td>10</td>
</tr>
<tr>
<td>Shared Memory Per Block (bytes)</td>
<td>2092</td>
</tr>
</tbody>
</table>

Maximum Thread Blocks Per Multiprocessor

<table>
<thead>
<tr>
<th>Limited by</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Warps / Multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>Registers / Multiprocessor</td>
<td>16</td>
</tr>
<tr>
<td>Shared Memory / Multiprocessor</td>
<td>6</td>
</tr>
</tbody>
</table>

Thread Block Limit Per Multiprocessor highlighted RED

©2010 Scott B. Baden /CSE 260/ Fall '10
<table>
<thead>
<tr>
<th>Threads</th>
<th>Warps/Multiprocessor</th>
<th>Registers</th>
<th>Warps/Multiprocessor</th>
<th>Shared Mem Warps/Multiprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>32</td>
<td>8</td>
<td>32</td>
<td>2048</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>8</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
<td>2</td>
<td>32</td>
<td>512</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>3</td>
<td>32</td>
<td>1024</td>
</tr>
<tr>
<td>96</td>
<td>24</td>
<td>4</td>
<td>32</td>
<td>1536</td>
</tr>
<tr>
<td>112</td>
<td>32</td>
<td>5</td>
<td>32</td>
<td>2048</td>
</tr>
<tr>
<td>128</td>
<td>32</td>
<td>6</td>
<td>32</td>
<td>2560</td>
</tr>
<tr>
<td>144</td>
<td>30</td>
<td>7</td>
<td>32</td>
<td>3072</td>
</tr>
<tr>
<td>160</td>
<td>30</td>
<td>8</td>
<td>32</td>
<td>3584</td>
</tr>
<tr>
<td>176</td>
<td>30</td>
<td>9</td>
<td>32</td>
<td>4096</td>
</tr>
<tr>
<td>192</td>
<td>30</td>
<td>10</td>
<td>32</td>
<td>4608</td>
</tr>
<tr>
<td>208</td>
<td>28</td>
<td>11</td>
<td>32</td>
<td>5120</td>
</tr>
<tr>
<td>224</td>
<td>28</td>
<td>12</td>
<td>32</td>
<td>5632</td>
</tr>
<tr>
<td>240</td>
<td>32</td>
<td>13</td>
<td>32</td>
<td>6144</td>
</tr>
<tr>
<td>256</td>
<td>32</td>
<td>14</td>
<td>32</td>
<td>6656</td>
</tr>
<tr>
<td>272</td>
<td>27</td>
<td>15</td>
<td>32</td>
<td>7168</td>
</tr>
<tr>
<td>288</td>
<td>27</td>
<td>16</td>
<td>32</td>
<td>7680</td>
</tr>
<tr>
<td>304</td>
<td>30</td>
<td>17</td>
<td>24</td>
<td>8192</td>
</tr>
<tr>
<td>320</td>
<td>30</td>
<td>18</td>
<td>24</td>
<td>8704</td>
</tr>
<tr>
<td>336</td>
<td>22</td>
<td>19</td>
<td>24</td>
<td>9216</td>
</tr>
<tr>
<td>352</td>
<td>22</td>
<td>20</td>
<td>24</td>
<td>9728</td>
</tr>
<tr>
<td>368</td>
<td>24</td>
<td>21</td>
<td>16</td>
<td>10240</td>
</tr>
<tr>
<td>384</td>
<td>24</td>
<td>22</td>
<td>16</td>
<td>10752</td>
</tr>
<tr>
<td>400</td>
<td>26</td>
<td>23</td>
<td>16</td>
<td>11264</td>
</tr>
<tr>
<td>416</td>
<td>26</td>
<td>24</td>
<td>16</td>
<td>11776</td>
</tr>
<tr>
<td>432</td>
<td>28</td>
<td>25</td>
<td>16</td>
<td>12288</td>
</tr>
<tr>
<td>448</td>
<td>28</td>
<td>26</td>
<td>16</td>
<td>12800</td>
</tr>
<tr>
<td>464</td>
<td>30</td>
<td>27</td>
<td>16</td>
<td>13312</td>
</tr>
<tr>
<td>480</td>
<td>30</td>
<td>28</td>
<td>16</td>
<td>13824</td>
</tr>
<tr>
<td>496</td>
<td>32</td>
<td>29</td>
<td>16</td>
<td>14336</td>
</tr>
<tr>
<td>512</td>
<td>32</td>
<td>30</td>
<td>16</td>
<td>14848</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31</td>
<td>16</td>
<td>15360</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32</td>
<td>16</td>
<td>15872</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16384</td>
</tr>
</tbody>
</table>

©2010 Scott B. Baden /CSE 260/ Fall '10
Reduction
Thread Divergence

• All the threads in a warp execute the same instruction
• Different control paths are serialized
• Divergence when predicates are a function of the threadId
  
  \[
  \text{if (threadId < 2) \{ \}}
  \]

• We can avoid divergence, everyone executes the same path
  
  \[
  \text{if (threadId / WARP\_SIZE < 2) \{ \}}
  \]

• Consider reduction, e.g. summation \( \sum_i x_i \)
A naïve reduction

Thread 0  Thread 2  Thread 4  Thread 6  Thread 8  Thread 10

0  1  2  3  4  5  6  7  8  9  10  11

0+1  2+3  4+5  6+7  8+9  10+11

0...3  4..7  8..11

0..7  8..15

©2010 Scott B. Baden /CSE 260/ Fall '10
The naïve code

```c
__global__ void reduce(int *input, unsigned int N, int *total)
{
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ int x[BSIZE];
    x[tid] = (i<N) ? input[i] : 0;
    __syncthreads();

    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        __syncthreads();
        if (tid % (2*stride) == 0)
            x[tid] += x[tid + stride];
    }

    if (tid == 0) atomicAdd(total, x[tid]);
}
```

©2010 Scott B. Baden /CSE 260/ Fall '10
Reducing divergence and avoiding bank conflicts

Thread 0
The Code

```c
unsigned int tid = threadIdx.x;
__shared__ int x[BSIZE];

...

for (unsigned int s = blockDim.x/2; s>1; s /= 2) {
    __syncthreads();
    if (tid < s)
        x[tid] += x[tid + s];
}
```