Outline of lecture

- Recap of Lecture 4
- Further optimization techniques
- Instruction optimization
- Memory as a limiting factor
- Thread and block heuristics
Recap

- Control flow
  - Hardware serializes divergent thread execution within same warp (*if* *else* instructions)
  - All threads of warp have to complete execution before warp ends
    - Loop divergence may drag run time of the whole warp
- Recommendations
  - Try to make all threads of same warp do the same thing
  - If you have branching code, cut your branch between warps
Recap

» Memory coalescing

- Memory is accessed in chunks of 32, 64 or 128 bytes

- Protocol
  » Find memory segment that contains address requested by the lowest numbered active thread
  » Find other active threads whose requested address lies in same segment, and reduce transaction size if possible
  » Do transaction. Mark serviced threads as inactive.
  » Repeat until all threads are serviced
Recap

- Memory coalescing - Recommendations
  - Have contiguous data storage patterns so to waste the least amount of bandwidth possible
  - Try to use a SoA programming model
  - Take advantage of cudaMalloc that guarantees contiguous data in memory
  - If your problem is compute bound don’t worry to much!

- Latency hiding
  - Do as much work per memory transaction as possible
Further optimization techniques

- Data prefetching
  
  - Can be used with tiling to hide latency

  Loop
  {
  
  Load tile to shared memory
  
  __syncthreads();
  
  Compute tile;
  
  __syncthreads;

  }

  Load first tile to registers from global memory

  Loop
  {

  Load tile to shared memory from registers
  
  __syncthreads();

  Compute tile;

  __syncthreads;

  Load next tile to registers from global memory

  Compute tile;

  __syncthreads;

  }

  With prefetching

  No prefetching
Further optimization techniques

- Loop unrolling
  - Loops involve branching code that slows things down!

```c
for (int k=0; k<BLOCKSIZE; k++)
    Pvalue += Ms[ty][k]*Ns[k][tx];
```

2 floating point arithmetic instructions
1 loop branch instruction
2 fetch instructions
1 loop counter instruction

Compiler can help with

```c
#pragma unroll 5
for (int k=0; k<BLOCKSIZE; k++)
    Pvalue += Ms[ty][k]*Ns[k][tx];
```
Further optimization techniques

- Instruction optimization
  - There are two types of runtime math operations
    
    \_sinf(x), \_expf(x)
    
    \texttt{sinf}(x), \texttt{expf}(x)
  
  - One is faster, but less accuracy
    
    \- One order of magnitude throughput
  
    - \texttt{-use\_fast\_math} flag changes all function() to \_function()
Thread and block heuristics

› Recommendations

- More blocks than SMs, so each SM has at least one block

- Check your device details
  
  › If maximum number of threads is 768, using 512 threads per block limits to 66% occupancy, but 256 yields 100% occupancy

- Rules of thumb
  
  › Threads per block multiple of warp size

  › 128-256 threads per block is a good initial guess to experiment

  › Several (3-4) small blocks is better than one large block
Example - Optimizing a parallel reduction

- Slides from Mark Harris - NVIDIA
- Optimizing parallel reduction within a block - Tests over 4M entries
  - No global synchronization in CUDA
  - Recursive kernel invocation necessary to complete
- Reductions are a memory bound problem
  - Will measure optimization using bandwidth
- G80 GPU used in this example
  - 384 bit memory interface, 900MHz DDR
  - \( \frac{384}{8} \times 900 \times 2 / 10^3 = 86.4 \text{GB/s} \)
Example - Optimizing a parallel reduction

- Interleaved addressing

```c
__global__ void reduce0(int* g_idata, int* g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s = 1; s < blockDim.x; s *= 2) {
        if(tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if(tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```
Example - Optimizing a parallel reduction

- Interleaved addressing
Example - Optimizing a parallel reduction

| Kernel 1: interleaved addressing with divergent branching | 8.054 ms | 2.083 GB/s |
Example - Optimizing a parallel reduction

Avoid divergent branching

Change

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

by

```c
for(unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

No divergence, but bank conflicts arise
## Example - Optimizing a parallel reduction

<table>
<thead>
<tr>
<th>Kernel 1: interleaved addressing with divergent branching</th>
<th>8.054 ms</th>
<th>2.083 GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 2: interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
</tr>
</tbody>
</table>
Example - Optimizing a parallel reduction

- Sequential addressing
Example - Optimizing a parallel reduction

- Sequential addressing

Change

```c
for(unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

by

```c
for(unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```
Example - Optimizing a parallel reduction

<table>
<thead>
<tr>
<th>Kernel 1:</th>
<th>8.054 ms</th>
<th>2.083 GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleaved addressing with divergent branching</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2:</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
</tr>
<tr>
<td>interleaved addressing with bank conflicts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 3:</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
</tr>
<tr>
<td>sequential addressing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example - Optimizing a parallel reduction

- First add during load

Change

```c
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

by

```c
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```
Example - Optimizing a parallel reduction

<table>
<thead>
<tr>
<th>Kernel 1: interleaved addressing with divergent branching</th>
<th>8.054 ms</th>
<th>2.083 GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 2: interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
</tr>
<tr>
<td>Kernel 3: sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
</tr>
<tr>
<td>Kernel 4: first add during global load</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
</tr>
</tbody>
</table>

- Still far from peak bandwidth (86GB/s)
  - We can’t be bounded by bandwidth
- Look into instruction overhead (address arithmetic and loops)
- Let’s unroll loops
Example - Optimizing a parallel reduction

- Unroll last warp
  - As instruction proceeds, number of active threads decreases
  - When we have less than 32 threads
    - We don’t need __syncthreads()
    - Don’t need if statement “if (tid < s)”
Example - Optimizing a parallel reduction

- Unroll last warp

```c
for(unsigned int s=blockDim.x/2; s>32; s>>=1)
{
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}

if (tid < 32)
{
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}
```
**Example - Optimizing a parallel reduction**

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (ms)</th>
<th>Bandwidth (GB/s)</th>
<th>Speedup</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1: interleaved addressing with divergent branching</td>
<td>8.054</td>
<td>2.083</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2: interleaved addressing with bank conflicts</td>
<td>3.456</td>
<td>4.854</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Kernel 3: sequential addressing</td>
<td>1.722</td>
<td>9.741</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>Kernel 4: first add during global load</td>
<td>0.965</td>
<td>17.377</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>Kernel 5: unroll last warp</td>
<td>0.536</td>
<td>31.289</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
</tbody>
</table>
Example - Optimizing a parallel reduction

- Complete unrolling
  - If we knew the number of iterations at compile time we could completely unroll reduction
    - Luckily for G80 we’re limited to 512 threads
- Use templates!
  - Some of the branching will be evaluated at compile time
  - CUDA supports C++ templating on device and host functions
- Block size will be our template parameter

```c
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```
Example - Optimizing a parallel reduction

- Complete unrolling

```c
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; }__syncthreads();
}
if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
}
if (blockSize >= 128) {
    if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
}
if (tid < 32){
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
```
Example - Optimizing a parallel reduction

- We don’t need to know the block size at compile time

```c
switch (threads) {
  case 512:
    reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 256:
    reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 128:
    reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 64:
    reduce5<64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 32:
    reduce5<32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 16:
    reduce5<16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 8:
    reduce5<8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 4:
    reduce5<4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 2:
    reduce5<2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
  case 1:
    reduce5<1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
}
```
Example - Optimizing a parallel reduction

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Description</th>
<th>Time (ms)</th>
<th>Bandwidth (GB/s)</th>
<th>Speedup</th>
<th>Speedup Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1:</td>
<td>interleaved addressing with divergent branching</td>
<td>8.054</td>
<td>2.083</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2:</td>
<td>interleaved addressing with bank conflicts</td>
<td>3.456</td>
<td>4.854</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Kernel 3:</td>
<td>sequential addressing</td>
<td>1.722</td>
<td>9.741</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>Kernel 4:</td>
<td>first add during global load</td>
<td>0.965</td>
<td>17.377</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>Kernel 5:</td>
<td>unroll last warp</td>
<td>0.536</td>
<td>31.289</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>Kernel 6:</td>
<td>completely unrolled</td>
<td>0.381</td>
<td>43.996</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
</tbody>
</table>
Example - Optimizing a parallel reduction

- Mix parallel and sequential execution to find optimal point
  - Instead of doing the first add when loading to shared memory, do as many as necessary

Change

```c
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

by

```c
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;
while (i < n){
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```
Example - Optimizing a parallel reduction

<table>
<thead>
<tr>
<th>Kernel 1:</th>
<th>8.054 ms</th>
<th>2.083 GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleaved addressing with divergent branching</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2:</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
</tr>
<tr>
<td>interleaved addressing with bank conflicts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 3:</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
</tr>
<tr>
<td>sequential addressing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 4:</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
</tr>
<tr>
<td>first add during global load</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 5:</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
</tr>
<tr>
<td>unroll last warp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 6:</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
</tr>
<tr>
<td>completely unrolled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 7:</td>
<td>0.268 ms</td>
<td>62.671 GB/s</td>
</tr>
<tr>
<td>multiple elements per thread</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Kernel 7 on 16M elements: 72 GB/s!
Example - Optimizing a parallel reduction

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;
    do{sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; }
    while (i < n);
__syncthreads();
if (blockSize >= 512) {if(tid<256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
if (blockSize >= 256) {if(tid<128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
if (blockSize >= 128) {if(tid<64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }
if (tid < 32){
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```
Example - Optimizing a parallel reduction

[Graph showing performance variations with number of elements for different addressing modes and optimizations.]

1: Interleaved Addressing: Divergent Branches
2: Interleaved Addressing: Bank Conflicts
3: Sequential Addressing
4: First add during global load
5: Unroll last warp
6: Completely unroll
7: Multiple elements per thread (max 64 blocks)
Optimizations - Wrapping up

- Nice summary in CUDA Best Practices Guide Appendix A

- Basic strategies
  - Maximize parallel execution
  - Optimize memory usage to achieve maximum bandwidth
  - Optimize instruction usage to achieve maximum throughput