ME759
High Performance Computing
for Engineering Applications

Optimizing CUDA Code

October 11, 2013

“Attitude is a little thing that makes a big difference.”
-- Winston Churchill
Before We Get Started…

- Last time
  - Atomic operations (part of “Synchronization for Data Communication”)
  - Profiling CUDA code

- Today: more of an hands-on lecture
  - Tiling as a programming pattern to speed up CUDA code
  - Simple example of finding bugs and improving performance: stencil operation
  - More complex example of optimizing code: vector reduction on the GPU (like your HW)

- Miscellaneous
  - Feedback has been uploaded on the course website
  - A webpage is available that reports the use of the HW extension (see my email of last night)
  - Email to describe rules of engagement for Midterm project coming your way soon
  - Exam: Th, November 7, 7:15-9:15 PM (no class on Friday, Nov. 8). Room: 1153ME
    - Review session on Wd, Nov. 6 @ 6 PM in this room (2121ME)
    - Exam will draw on material covered in class and information provided in the primer
    - It’ll be a pen and paper exam. Open book and open anything
Optimization Summary

[Looking Back at 1D Stencil Example…]

- Initial CUDA parallelization
  - Expeditious, kernel is almost word-for-word replica of sequential code
  - 2.2x speedup

- Optimize memory throughput
  - Expeditious, need to know about pinned memory
  - 4.3x speedup

- Overlap compute and data movement
  - More involved, need to know about the inner works of CUDA
  - Problem should be large enough to justify mem-transfer/execution
  - 7.2x speedup
Iterative Optimization

- Identify Optimization Opportunities
- Parallelize
- Optimize
Take Home Message...

- Regard CUDA as a way to accelerate the compute-intensive parts of your application

- Visual profiler (nvprof) helps in performance analysis and optimization
Revisit Stencil Example

- Problem setup
  - 1,000,000 elements
  - RADIUS is 3

- Purpose:
  - Show a typical bug and then one easy way to get some extra performance out of the code
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    float *cuda_out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    applyStencil1D_SEQ(RADIUS, N-RADIUS, weights, in, out);
    cudaMemcpy(cuda_out, d_out, size, cudaMemcpyDeviceToHost);
    int nDiffs = checkResults(cuda_out, out, N);
    nDiffs==0? std::cout<<"Looks good.\n": std::cout<<"Doesn't look good: "<< nDiffs << “ differences
”;
    //free resources
    free(weights); free(in); free(out); free(cuda_out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
    return 0;
}
Example: Debugging & Profiling
[1DStencil Code: Supporting Cast]

```c
void initializeArray(float* arr, int nElements) {
    const int myMinNumber = -5;
    const int myMaxNumber = 5;
    srand(time(NULL));
    for (int i=0; i<nElements; i++)
        arr[i] = (float)(rand() % (myMaxNumber - myMinNumber + 1) + myMinNumber);
}

void initializeWeights(float* weights, int rad) {
    // for now hardcoded for RADIUS=3
    weights[0] = 0.50f;
    weights[1] = 0.75f;
    weights[2] = 1.25f;
    weights[3] = 2.00f;
    weights[4] = 1.25f;
    weights[5] = 0.75f;
    weights[6] = 0.50f;
}

int checkResults(float* cudaRes, float* res, int nElements) {
    int nDiffs=0;
    const float smallVal = 0.000001f;
    for(int i=0; i<nElements; i++)
        if( fabs(cudaRes[i]-res[i])>smallVal )
            nDiffs++;
    return nDiffs;
}
```
**Example: Debugging & Profiling**

[1DStencil Code: the actual stencil function]
[negrut@euler CodeBits]$ qsub -I -l nodes=1:gpus=1:default -X
[negrut@euler01 CodeBits]$ nvcc -gencode arch=compute_20, code=sm_20 testV1.cu
[negrut@euler01 CodeBits]$ ./testV1

Doesn't look good: 57 differences

[negrut@euler01 CodeBits]$
```c
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    float *cuda_out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<(N+511)/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    applyStencil1D_SEQ(RADIUS, N-RADIUS, weights, in, out);
    cudaMemcpy(cuda_out, d_out, size, cudaMemcpyDeviceToHost);

    int nDiffs = checkResults(cuda_out, out, N);
    nDiffs==0? std::cout<<"Looks good.\n": std::cout<<"Doesn't look good: " << nDiffs << " differences\n";

    //free resources
    free(weights); free(in); free(out); free(cuda_out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

Example: Debugging & Profiling
[1DStencil Code]
Doesn't look good: 4 differences

- Reason: `checkResults` runs a loop over all 1,000,000 entries. It should exclude the first RADIUS and last RADIUS of them. Those entries are not computed, you pick up whatever was there when memory was allocated on the host and on the device. As such, it gives false positives.

- NOTE: this problem is not reproducible always (sometimes code runs ok, sometimes gives you a false positive)
int checkResults(float* cudaRes, float* res, int nElements) {
    int nDiffs=0;
    const float smallVal = 0.000001f;
    for(int i=0; i<nElements; i++)
        if(fabs(cudaRes[i]-res[i])>smallVal )
            nDiffs++;
    return nDiffs;
}

int checkResults(int startElem, int endElem, float* cudaRes, float* res) {
    int nDiffs=0;
    const float smallVal = 0.000001f;
    for(int i=startElem; i<endElem; i++)
        if(fabs(cudaRes[i]-res[i])>smallVal )
            nDiffs++;
    return nDiffs;
}
Third Version [V3]...

[negrut@euler01 CodeBits]$ nvcc -gencode arch=compute_20, code=sm_20 testV3.cu
[negrut@euler01 CodeBits]$ ./testV3
Looks good.
[negrut@euler01 CodeBits]$

● Things are good now...
Code Profiling…

- Code looks like running ok, no evident bugs

- Time to profile the code, we’ll use the Lazy Man’s approach

- Profile V3 version
  - Create base results, both for compute capability 1.0 (Tesla) and 2.0 (Fermi)
Lazy Man’s Solution…

```
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV3.cu -o testV3_20
>> ./testV3_20

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff6c689a59e98
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1.664 ] cputime=[ 9.000 ]
method=[ memcpyHtoD ] gputime=[ 995.584 ] cputime=[ 1193.000 ]
method=[ _Z14applyStencil1DiiPKfPfS1_ ] gputime=[ 189.856 ] cputime=[ 12.000 ] occupancy=[1.0]
method=[ memcpyDtoH ] gputime=[ 1977.728 ] cputime=[ 2525.000 ]
```

```
>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV3.cu -o testV3_10
>> ./testV3_10

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b1842064
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1787.232 ] cputime=[ 2760.139 ]
method=[ _Z14applyStencil1DiiPKfPfS1_ ] gputime=[ 68357.69 ] cputime=[ 8.85 ] occupancy=[0.667]
```
Improving Performance

Here’s what we’ll be focusing on:

```c
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out)
{
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

There are several opportunities for improvement to move from V3 to V4:

- Too many accesses to global memory (an issue if you don’t have L1 cache)
- You can unroll the 7-iteration loop (it’ll save you some pocket change)
- You can use shared memory (important if you don’t have L1 cache, i.e., in 1.0)
- You can use pinned host memory [you have to look into `main()` to this end]
Improving Performance [V4]

- Version V4: Take care of
  - Repeated access to global memory
  - Loop unrolling

```c
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out)
{
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if (i < eIdx ) {
        float result = 0.f;
        result += weights[0]*in[i-3];
        result += weights[1]*in[i-2];
        result += weights[2]*in[i-1];
        result += weights[3]*in[i];
        result += weights[4]*in[i+1];
        result += weights[5]*in[i+2];
        result += weights[6]*in[i+3];
        result /= 7.f;
        out[i] = result;
    }
}
```

- Even now there is room for improvement
  - You can have `weights` and `in` stored in shared memory
  - You can use pinned memory (mapped memory) on the host
Lazy Man’s Profiling: V4

```
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> ./testV4_20

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR ffff6c689a404a8

method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```

```
>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
>> ./testV4_10

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e

method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[ __Z14applyStencil1DiiPKfPfPS1__ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```
## Timing Results

[Two Different Approaches (V3, V4) & Two Different GPUs (sm_20, sm_10)]
[each executable was run 7 times; script available on the class website]

<table>
<thead>
<tr>
<th></th>
<th>V4_20</th>
<th>V3_20</th>
<th>V4_10</th>
<th>V3_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>166.752</td>
<td>190.560</td>
<td>47341.566</td>
<td>68611.008</td>
</tr>
<tr>
<td>2nd</td>
<td>166.912</td>
<td>190.016</td>
<td>47332.930</td>
<td>68531.875</td>
</tr>
<tr>
<td>3rd</td>
<td>166.976</td>
<td>190.208</td>
<td>47391.039</td>
<td>68674.109</td>
</tr>
<tr>
<td>4th</td>
<td>166.368</td>
<td>190.048</td>
<td>47252.734</td>
<td>68679.422</td>
</tr>
<tr>
<td>5th</td>
<td>166.848</td>
<td>189.696</td>
<td>47371.426</td>
<td>68357.695</td>
</tr>
<tr>
<td>6th</td>
<td>166.592</td>
<td>189.856</td>
<td>47250.465</td>
<td>68618.492</td>
</tr>
<tr>
<td>7th</td>
<td>166.944</td>
<td>190.240</td>
<td>47379.902</td>
<td>68687.266</td>
</tr>
</tbody>
</table>

### Averages

<table>
<thead>
<tr>
<th></th>
<th>V4_20</th>
<th>V3_20</th>
<th>V4_10</th>
<th>V3_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>166.7702857</td>
<td>190.0891429</td>
<td>47331.43743</td>
<td>68594.26671</td>
</tr>
</tbody>
</table>

### Standard Deviations

<table>
<thead>
<tr>
<th></th>
<th>V4_20</th>
<th>V3_20</th>
<th>V4_10</th>
<th>V3_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Deviation</td>
<td>0.132410266</td>
<td>0.147947777</td>
<td>0.123060609</td>
<td>0.171466201</td>
</tr>
</tbody>
</table>

### Slowdown, sm_20

- 13.98262109%

### Slowdown, sm_10

- 44.92326969%
This is how you should thing about code profiling and optimization:

- Would you ever send out your CV right after you completed writing it?
- Probably not, you always go back and spend a bit of time polishing it…
Putting Things in Perspective...

Here’s what we’ve covered so far:
- CUDA execution configuration (grids, blocks, threads)
- CUDA scheduling issues (warps, thread divergence, synchronization, etc.)
- CUDA Memory ecosystem (registers, shared mem, device mem, L1/L2 cache, etc.)
- Practical things: building, debugging, profiling CUDA code

Next: CUDA GPU Programming - Examples & Code Optimization Issues
- Tiling: a CUDA programming pattern
- Example: CUDA optimization exercise in relation to a vector reduction operation
- CUDA Execution Configuration Optimization Heuristics: Occupancy issues
- CUDA Optimization Rules of Thumb
Tiling [Blocking]:
A Fundamental CUDA Programming Pattern

- Partition data to operate in well-sized blocks
  - Small enough to be staged in shared memory
  - Assign each data partition to a block of threads
  - No different from cache blocking!
    - Except you now have full control over it

- Provides several significant performance benefits
  - Working in shared memory reduces memory latency dramatically
  - More likely to have address access patterns that coalesce well on load/store to shared memory
Fundamental CUDA Pattern: Tiling

Partition data into subsets that fit into __shared__ memory

This is your data: one big chunk, about to be broken into subsets suitable to be stored into shared memory.
Fundamental CUDA Pattern: Tiling

- Process each data subset with one thread block
Fundamental CUDA Pattern: Tiling

- Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
Fundamental CUDA Pattern: Tiling

- Perform the computation on the subset from shared memory
Fundamental CUDA Pattern: Tiling

- Copy the result from \textit{shared} memory back to global memory
A large number of CUDA kernels are built this way

However, tiling [blocking] may not be the only approach to solving a problem, sometimes it might not apply…

Two questions that can guide you in deciding if tiling is it:
- Does a thread require several loads from global memory to serve its purpose?
- Could data used by a thread be used by some other thread in the same block?
- If answer to both questions is “yes”, consider tiling as a design pattern

The answer to these two questions above is not always obvious
- Sometime it’s useful to craft an altogether new approach (algorithm) that is capable of using tiling: you force the answers to be “yes”
A CUDA Optimization Exercise
[A Demonstration Using the Parallel Reduction Application]
Exercise draws on material made available by Mark Harris of NVIDIA [acknowledgement at bottom of slides]

Parallel Reduction: Common and very important data parallel primitive
  - Example: Used to compute the norm of a large vector

Easy to implement in CUDA
  - Challenging to get it to run fast though

Serves as a good optimization example
  - Walk step by step through several different versions
  - Demonstrates several important optimization strategies
Parallel Reduction

- Basic Idea: tree-based approach used within each thread block

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

- Need to be able to use multiple thread blocks
  - Why? To process very large arrays
  - Why? To keep all multiprocessors on the GPU busy
  - How? Each thread block reduces a portion of the array to one single value

- Q: How do we communicate partial results between thread blocks?
Problem: Global Synchronization

- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively

- But CUDA has no global synchronization. Why?
  - Expensive to build in hardware for GPUs with high processor count
  - Would force programmer to run fewer blocks (no more than number of SMs times the number of resident blocks / SM) → this may reduce overall efficiency

- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead
Multiple Kernel Calls
[An Example, and how it all works out…]

- Imagine you launch a grid in which each block has 256 threads
- Assume that the number or elements in the array is \( N = 100,000 \)
  - Note that \( 100,000 = 390 \times 256 + 160 \), therefore \( \text{ceil}[N/256.0] = 391 \)
- For the first stage, you launch 391 blocks of 256 threads
  - At the end of this stage you still have to operate on 391 elements
- For the second stage, you launch two blocks of 256 threads
  - At the end of this stage you only have to operate on two elements
- For the third and last stage, you launch one block of 32 threads
  - Almost all threads idle…
- **NOTE:** after the first stage, each subsequent stage operates on a number of entries equal to the number of blocks in the previous stage
Vector Reduction: 30,000 Feet Perspective

- At the block level: Bring data in shared memory, then start adding in parallel
- Fewer and fewer threads of a block participate
- The process is memory bound, low arithmetic intensity…
What is Our Optimization Goal?

- We should strive to reach GPU peak performance

- Choose the right metric:
  - GFLOP/s: for compute-bound kernels
  - Bandwidth: for memory-bound kernels

- Reductions have very low arithmetic intensity
  - 1 flop per element loaded (bandwidth-optimal)

- Therefore we should strive for peak bandwidth

- This example uses results generated using a G80 GPU
  - Compute capability (CC) 1.0
  - 384-bit memory interface, 900 MHz DDR
  - \(384 \times 900 \times 2 / 8 = 86.4 \text{ GB/s}\)
  - Example carries over to other CCs, this algorithm will be memory bound
Parallel Reduction: Interleaved Addressing

Values (shared memory)

Step 1  Stride 1
Thread IDs
Values

Step 2  Stride 2
Thread IDs
Values

Step 3  Stride 4
Thread IDs
Values

Step 4  Stride 8
Thread IDs
Values

Note: in stage s, only threads divisible to $2^s$ get to work. Stride: $2^{(s-1)}$
Reduction #1: Interleaved Addressing

```c
__global__ void reduce1(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 memory
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem: highly divergent warps are very inefficient, and % operator is very slow

NVIDIA [M. Harris]→
Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1:</td>
<td></td>
</tr>
<tr>
<td>interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
</tr>
</tbody>
</table>

Note: Block Size = 128 threads for all tests
Parallel Reduction: Interleaved Addressing

Values (shared memory)

Step 1
Stride $2^0$

Thread IDs

Values

Step 2
Stride $2^1$

Thread IDs

Values

Step 3
Stride $2^2$

Thread IDs

Values

Step 4
Stride $2^3$

Thread IDs

Values

New Problem: Shared Memory Bank Conflicts
Reduction #2: Interleaved Addressing

Just replace divergent branch in inner loop...

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

...with strided index and non-divergent branch:

```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();
}
```
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th></th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel 1:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Kernel 2:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
</tbody>
</table>
Parallel Reduction: Sequential Addressing

Values (shared memory)

<table>
<thead>
<tr>
<th>Values</th>
<th>10</th>
<th>1</th>
<th>8</th>
<th>-1</th>
<th>0</th>
<th>-2</th>
<th>3</th>
<th>5</th>
<th>-2</th>
<th>-3</th>
<th>2</th>
<th>7</th>
<th>0</th>
<th>11</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>Thread</td>
<td>Stride 8</td>
<td>IDs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values</td>
<td>8</td>
<td>-2</td>
<td>10</td>
<td>6</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>7</td>
<td>-2</td>
<td>-3</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Step 2</td>
<td>Thread</td>
<td>Stride 4</td>
<td>IDs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values</td>
<td>8</td>
<td>7</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>7</td>
<td>-2</td>
<td>-3</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Step 3</td>
<td>Thread</td>
<td>Stride 2</td>
<td>IDs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values</td>
<td>21</td>
<td>20</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>7</td>
<td>-2</td>
<td>-3</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Step 4</td>
<td>Thread</td>
<td>Stride 1</td>
<td>IDs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values</td>
<td>41</td>
<td>20</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>7</td>
<td>-2</td>
<td>-3</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Sequential addressing is Shared Mem conflict free
Reduction #3: Sequential Addressing

Just replace strided indexing in inner loop...

```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();
}
```

...with reversed loop and threadID-based indexing:

```c
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1:</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interleaved addressing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with divergent branching</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2:</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>interleaved addressing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with bank conflicts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 3:</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>sequential addressing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Idle Threads...

Current solution:

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

Note that half of the threads are idle on first loop iteration! This is wasteful...
Reduction #4: First Add During Load

Replace single load:

```c
// 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();
```

...With two loads and first add of the reduction:

```c
// perform first level of reduction upon reading from
// global memory and 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();
```

One side effect: the number of blocks you need now is half of what it used to be...
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel 1:</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleaved addressing</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>with divergent branching</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Kernel 2:                     | 3.456 ms           | 4.854 GB/s| 2.33x        | 2.33x              |
| interleaved addressing       |                    |           |              |                    |
| with bank conflicts           |                    |           |              |                    |

| Kernel 3:                     | 1.722 ms           | 9.741 GB/s| 2.01x        | 4.68x              |
| sequential addressing        |                    |           |              |                    |

| Kernel 4:                     | 0.965 ms           | 17.377 GB/s| 1.78x        | 8.34x              |
| first add during global load  |                    |           |              |                    |
Instruction Bottleneck

- At 17 GB/s, we’re far from bandwidth bound
  - And we know reduction has low arithmetic intensity

- Therefore a likely bottleneck is instruction overhead
  - Ancillary instructions that are not loads, stores, or arithmetic for the core computation
  - In other words: address arithmetic and loop overhead

- Strategy: unroll loops
Unrolling the Last Warp

- As reduction proceeds, the number of “active” threads decreases
  - When \( s \leq 32 \), we have only one warp left

- Instructions are SIMD synchronous within a warp
  - All threads in a warp proceed in lockstep fashion

- That means when \( s \leq 32 \):
  - We don’t need to \texttt{__syncthreads()}\texttt{()}
  - We don’t need \texttt{if (tid < s)} because it doesn’t save any work

- Let’s unroll the last 6 iterations of the inner loop
Reduction #5: Unroll the Last Warp

void warpReduce(volatile int* sdata, int tid) {
    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];
}

// and use later like this...
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}

if (tid < 32) warpReduce(sdata, tid);

Note: This saves useless work in all warps, not just the last one!
Without unrolling, all warps execute every iteration of the for loop and if statement

IMPORTANT: For this to be correct, we must use the "volatile" keyword!
**Performance for 4M element reduction**

<table>
<thead>
<tr>
<th>Kernel 1:</th>
<th>Time (2(^{22}) ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleaved addressing</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>with divergent branching</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2:</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>interleaved addressing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with bank conflicts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 3:</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>sequential addressing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 4:</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>first add during global load</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 5:</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>unroll last warp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Complete Unrolling

- If we knew the number of iterations (or equivalently, of threads in a block) at compile time, we could completely unroll the reduction
  - Luckily, the block size on G80 is limited by the GPU to 512 threads
    - 1024 on newer Fermi GPUs
  - Also, we are sticking to power-of-2 block sizes

- So we can easily unroll for a fixed block size
  - But we need to be generic – how can we unroll for block sizes that we don’t know at compile time?

- Use of templates can solve this issue…
  - CUDA supports C++ template parameters on device and host functions
Unrolling with Templates

- Specify block size as a function template parameter
- The kernel is parameterized:

```c
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata)
```
Reduction #6: Completely Unrolled

```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) warpReduce<blockSize>(sdata, tid);
}
```

This is the key part of the kernel

```c
template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
    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];
}
```

This is a helper function (device only)

- All code in RED will be evaluated at compile time. Results in a very efficient inner loop.
- For Fermi, you’d have one more if statement that covers the case when blockSize >= 1024
- You can call the warpReduce function only when you got to one wrap. Reason: you don’t have to synchronize at that point.
Invoking Template Kernels

switch (threads) {
  case 512:
    reduce6<512><<<< dimGrid, dimBlock, smemSize >>(d_idata, d_odata); break;
  case 256:
    reduce6<256><<<< dimGrid, dimBlock, smemSize >>(d_idata, d_odata); break;
  case 128:
    reduce6<128><<<< dimGrid, dimBlock, smemSize >>(d_idata, d_odata); break;
  case 64:
    reduce6<64><<<< dimGrid, dimBlock, smemSize >>(d_idata, d_odata); break;
  case 32:
    reduce6<32><<<< dimGrid, dimBlock, smemSize >>(d_idata, d_odata); break;
  case 16:
    reduce6<16><<<< dimGrid, dimBlock, smemSize >>(d_idata, d_odata); break;
  case 8:
    reduce6<8><<<< dimGrid, dimBlock, smemSize >>(d_idata, d_odata); break;
  case 4:
    reduce6<4><<<< dimGrid, dimBlock, smemSize >>(d_idata, d_odata); break;
  case 2:
    reduce6<2><<<< dimGrid, dimBlock, smemSize >>(d_idata, d_odata); break;
  case 1:
    reduce6<1><<<< dimGrid, dimBlock, smemSize >>(d_idata, d_odata); break;
}
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1: interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2: interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Kernel 3: sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>Kernel 4: first add during global load</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>Kernel 5: unroll last warp</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>Kernel 6: completely unrolled</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
</tbody>
</table>

NVIDIA [M. Harris]→
Parallel Reduction Complexity

- Assume that the number of elements in array is of the form $N = 2^D$

- $\log(N)$ parallel stages, each stage $S$ requires $N/2^S$ independent ops
  - Stage Complexity is $O(\log N)$

- For $N = 2^D$, approach requires a total of $\sum_{S \in [1..D]} 2^{D-S} = N - 1$ operations
  - Work Complexity is $O(N)$ – It is work-efficient
  - That is, it does not perform more operations than a sequential algorithm

- Time complexity, for $P$ threads physically in parallel ($P$ processors): $O(N/P + \log N)$
  - Compare to $O(N)$ for sequential reduction
  - In a thread block, $N=P$, so $O(\log N)$
What About Cost?

- **Cost** of a parallel algorithm is processors $\times$ time complexity
  - Allocate threads instead of processors: $O(N)$ threads
  - Time complexity is $O(\log N)$, so cost is $O(N \log N)$: not cost efficient!

- Brent’s theorem suggests $O(N/\log N)$ threads
  - Each thread does $O(\log N)$ sequential work
  - Then all $O(N/\log N)$ threads cooperate for $O(\log N)$ stages
  - Cost = $O((N/\log N) \times \log N) = O(N) \rightarrow$ cost efficient

- Sometimes called *algorithm cascading*
  - Can lead to significant speedups in practice
Algorithm Cascading

- Combine sequential and parallel reduction
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory

- Brent’s theorem says each thread should sum $O(\log n)$ elements
  - i.e. 1024 or 2048 elements per block vs. 256

- Probably beneficial to push it even further
  - Possibly better latency hiding with more work per thread
  - More threads per block reduces levels in tree of recursive kernel invocations
  - High kernel launch overhead in last levels with few blocks

- On G80, best performance with 64-256 blocks of 128 threads
  - 1024-4096 elements per thread
Kernel 7, Comments

- For the first six kernels a large number of blocks was used to “tile” the array

- Kernel 7: reduce the number of blocks and have a thread do more work than just fetch something to shared memory

- Example [cooked up, not related to actual CUDA warp size, typical CUDA block dim, etc.]:
  - Say you have 1024 elements stored in an array; you need to reduce that array
  - You start with 32 blocks, each with 4 threads
  - Then, 128 threads total. It means that a thread, say in block 11, would have to add two numbers, then two numbers, then two numbers, then two more numbers.
  - At this point, everything is in the union of the shared memory associated with the 32 blocks. At this point proceed like before with kernel 6.
Reduction #7: Multiple Adds / Thread

Replace load and add of two elements:

```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();
```

With a while loop to add as many as necessary:

```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();
```

Note: gridSize loop stride to maintain coalescing!
Performance for 4M element reduction

Kernel 7 on 32M elements: 73 GB/s!

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Interleaved addressing with divergent branching</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth (GB/s)</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1</td>
<td></td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2</td>
<td>interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Kernel 3</td>
<td>sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</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 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>Kernel 5</td>
<td>unroll last warp</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>Kernel 6</td>
<td>completely unrolled</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
<tr>
<td>Kernel 7</td>
<td>multiple elements per thread</td>
<td>0.268 ms</td>
<td>62.671 GB/s</td>
<td>1.42x</td>
<td>30.04x</td>
</tr>
</tbody>
</table>
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    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];
}

template <unsigned int blockSize>
__global__ void reduce7(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;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; } __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) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}