S05: High Performance Computing with CUDA

Optimizing CUDA

Mark Harris
NVIDIA Developer Technology
CUDA is fast and efficient

CUDA enables efficient use of the massive parallelism of NVIDIA GPUs
- Direct execution of data-parallel programs
- Without the overhead of a graphics API

Using CUDA on Tesla GPUs can provide large speedups on data-parallel computations straight out of the box!

Even higher speedups are achievable by understanding and tuning for GPU architecture
- This presentation covers general performance, common pitfalls, and useful strategies
Outline

- General optimization guidance
  - Coalescing memory operations
  - Occupancy and latency hiding
  - Using shared memory
- Example 1: transpose
  - Coalescing and bank conflict avoidance
- Example 2: efficient parallel reductions
  - Using peak performance metrics to guide optimization
  - Avoiding SIMD divergence & bank conflicts
  - Loop unrolling
  - Using template parameters to write general-yet-optimized code
  - Algorithmic strategy: Cost efficiency
Quick terminology review

*Thread*: concurrent code and associated state executed on the CUDA device (in parallel with other threads)
- The unit of parallelism in CUDA
- Note difference from CPU threads: creation cost, resource usage, and switching cost of GPU threads is much smaller

*Warp*: a group of threads executed *physically* in parallel (SIMD)

*Thread Block*: a group of threads that are executed together and can share memory on a single multiprocessor

*Grid*: a group of thread blocks that execute a single CUDA program *logically* in parallel

*Device*: GPU  
*Host*: CPU  
*SM*: Multiprocessor
CUDA Optimization Strategies

- Optimize Algorithms for the GPU
- Optimize Memory Access Coherence
- Take Advantage of On-Chip Shared Memory
- Use Parallelism Efficiently
Optimize Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it’s better to recompute than to cache
  - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host
Optimize Memory Coherence

- Coalesced vs. Non-coalesced = order of magnitude
  - Global/Local device memory

- Optimize for spatial locality in cached texture memory

- In shared memory, avoid high-degree bank conflicts
Take Advantage of Shared Memory

- Hundreds of times faster than global memory
- Threads can cooperate via shared memory

- Use one / a few threads to load / compute data shared by all threads

- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
  - Matrix transpose example later
Use Parallelism Efficiently

Partition your computation to keep the GPU multiprocessors equally busy
- Many threads, many thread blocks

Keep resource usage low enough to support multiple active thread blocks per multiprocessor
- Registers, shared memory
Global Memory Reads/Writes

- Highest latency instructions: 400-600 clock cycles
- Likely to be performance bottleneck
- Optimizations can greatly increase performance
  - Coalescing: up to 10x speedup
Coalescing

A coordinated read by a warp
A contiguous region of global memory:
- 128 bytes - each thread reads a word: \texttt{int}, \texttt{float}, …
- 256 bytes - each thread reads a double-word: \texttt{int2}, \texttt{float2}, …
- 512 bytes – each thread reads a quad-word: \texttt{int4}, \texttt{float4}, …

Additional restrictions:
- Starting address for a region must be a multiple of region size
- The $k$\textsuperscript{th} thread in a warp must access the $k$\textsuperscript{th} element in a block being read

Exception: not all threads must be participating
- Predicated access, divergence within a warp
Coalesced Access: Reading floats

All threads participate

Some Threads Do Not Participate
Uncoalesced Access: Reading floats

Permuted Access by Threads

Misaligned Starting Address (not a multiple of 64)
Coalescing: Timing Results

Experiment:
- Kernel: read a float, increment, write back
- 3M floats (12MB)
- Times averaged over 10K runs

12K blocks x 256 threads:
- 356µs – coalesced
- 357µs – coalesced, some threads don’t participate
- 3,494µs – permuted/misaligned thread access
Uncoalesced float3 Code

```c
__global__ void accessFloat3(float3 *d_in, float3 d_out)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    float3 a = d_in[index];

    a.x += 2;
    a.y += 2;
    a.z += 2;

    d_out[index] = a;
}
```
Uncoalesced Access: float3 Case

- float3 is 12 bytes
- Each thread ends up executing 3 reads
  - sizeof(float3) ≠ 4, 8, or 12
  - Half-warp reads three 64B non-contiguous regions

The diagram shows the first read:
- t0: Access to float3
- t1: Access to float3
- t2: Access to float3
- t3: Access to float3
Coalescing float3 Access

Similarly, Step3 starting at offset 512
Coalesced Access: float3 Case

- Use shared memory to allow coalescing
  - Need `sizeof(float3)*(threads/block)` bytes of SMEM
  - Each thread reads 3 scalar floats:
    - Offsets: 0, (threads/block), 2*(threads/block)
    - These will likely be processed by other threads, so sync

- Processing
  - Each thread retrieves its float3 from SMEM array
    - Cast the SMEM pointer to (float3*)
    - Use thread ID as index
  - Rest of the compute code does not change!
Coalesced float3 Code

```c
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
    int index = 3 * blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float s_data[256*3];
    s_data[threadIdx.x] = g_in[index];
    s_data[threadIdx.x+256] = g_in[index+256];
    s_data[threadIdx.x+512] = g_in[index+512];
    __syncthreads();
    float3 a = ((float3*)s_data)[threadIdx.x];
    a.x += 2;
    a.y += 2;
    a.z += 2;
    ((float3*)s_data)[threadIdx.x] = a;
    __syncthreads();
    g_out[index] = s_data[threadIdx.x];
    g_out[index+256] = s_data[threadIdx.x+256];
    g_out[index+512] = s_data[threadIdx.x+512];
}
```

Read the input through SMEM

Compute code is not changed

Write the result through SMEM
Coalescing: Structures of Size ≠ 4, 8, or 16 Bytes

- Use a structure of arrays instead of AoS
- If SoA is not viable:
  - Force structure alignment: __align(X), where X = 4, 8, or 16
  - Use SMEM to achieve coalescing
Coalescing: Timing Results

Experiment:
- Kernel: read a **float**, increment, write back
- 3M floats (12MB)
- Times averaged over 10K runs

12K blocks x 256 threads:
- 356µs – coalesced
- 357µs – coalesced, some threads don’t participate
- 3,494µs – permuted/misaligned thread access

4K blocks x 256 threads:
- 3,302µs – **float3** uncoalesced
- 359µs – **float3** coalesced through shared memory
Coalescing: Summary

- Coalescing greatly improves throughput
- Critical to small or memory-bound kernels
- Reading structures of size other than 4, 8, or 16 bytes will break coalescing:
  - Prefer Structures of Arrays over AoS
  - If SoA is not viable, read/write through SMEM
- Futureproof code: coalesce over whole warps
- Additional resources:
  - Aligned Types CUDA SDK Sample
Data Transfers

- Device memory to host memory bandwidth much lower than device memory to device bandwidth
  - 4GB/s peak (PCI-e x16) vs. 80 GB/s peak (Quadro FX 5600)

- Minimize transfers
  - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to host memory

- Group transfers
  - One large transfer much better than many small ones
Page-Locked Memory Transfers

- `cudaMallocHost()` allows allocation of page-locked host memory
- Enables highest `cudaMemcpy` performance
  - 3.2 GB/s+ common on PCI-express x16
  - ~4 GB/s measured on nForce 680i motherboards (overclocked PCI-e)
- See the “bandwidthTest” CUDA SDK sample
- Use with caution
  - Allocating too much page-locked memory can reduce overall system performance
  - Test your systems and apps to learn their limits
Occupancy

Thread instructions executed sequentially, executing other warps is the only way to hide latencies and keep the hardware busy.

**Occupancy** = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently.

Minimize occupancy requirements by minimizing latency
Maximize occupancy by optimizing threads per multiprocessor.
Occupancy != Performance

Increasing occupancy does not necessarily increase performance

**BUT...**

Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels

(It all comes down to arithmetic intensity and available parallelism)
Grid/Block Size Heuristics

- # of blocks / # of multiprocessors > 1
  - So all multiprocessors have at least one block to execute
  - Per-block resources at most half of total available
    - Shared memory and registers
    - Multiple blocks can run concurrently in a multiprocessor
    - If multiple blocks coexist that aren’t all waiting at a __syncthreads(), machine can stay busy
- # of blocks / # of multiprocessors > 2
  - So multiple blocks run concurrently in a multiprocessor
- # of blocks > 100 to scale to future devices
  - Blocks stream through machine in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations
Parameterize Your Application

Parameterization helps adaptation to different GPUs
GPUs vary in many ways
- # of multiprocessors
- Memory bandwidth
- Shared memory size
- Register file size
- Threads per block

You can even make apps self-tuning (like FFTW and ATLAS)
- “Experiment” mode discovers and saves optimal configuration
Optimization Example 1: Matrix Transpose
Matrix Transpose

- SDK Sample ("transpose")
- Illustrates coalescing using shared memory
- Speedups for even small matrices
__global__ void transpose_naive(float *odata, float *idata, int width, int height)
{
1.  unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
2.  unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
3.  if (xIndex < width && yIndex < height)
4.      {
5.          unsinged int index_in  = xIndex + width * yIndex;
6.          unsigned int index_out = yIndex + height * xIndex;
7.          odata[index_out] = idata[index_in];
8.      }
9.}

Uncoalesced Transpose
Uncoalesced Transpose

Reads input from GMEM

<table>
<thead>
<tr>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th>0,15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,15</td>
</tr>
<tr>
<td>15,0</td>
<td>15,1</td>
<td>15,2</td>
<td>15,15</td>
</tr>
</tbody>
</table>

Write output to GMEM

<table>
<thead>
<tr>
<th>0,0</th>
<th>1,0</th>
<th>2,0</th>
<th>15,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>2,1</td>
<td>15,1</td>
</tr>
<tr>
<td>0,15</td>
<td>1,15</td>
<td>2,15</td>
<td>15,15</td>
</tr>
</tbody>
</table>

GMEM

Stride = 1, coalesced

GMEM

Stride = 16, uncoalesced
Coalesced Transpose

Assumption: matrix is partitioned into square tiles

Threadblock \((bx, by)\):
- Read the \((bx,by)\) input tile, store into SMEM
- Write the SMEM data to \((by,bx)\) output tile
- Transpose the indexing into SMEM

Thread \((tx,ty)\):
- Reads element \((tx,ty)\) from input tile
- Writes element \((tx,ty)\) into output tile

Coalescing is achieved if:
- Block/tile dimensions are multiples of 16
Coalesced Transpose

<table>
<thead>
<tr>
<th>Reads from GMEM</th>
<th>Writes to SMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0 0,1 0,2</td>
<td>0,15 1,15 1,15</td>
</tr>
<tr>
<td>1,0 1,1 1,2</td>
<td>1,0 1,1 1,2</td>
</tr>
<tr>
<td>15,0 15,1 15,2</td>
<td>15,0 15,1 15,2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reads from SMEM</th>
<th>Writes to GMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0 1,0 2,0</td>
<td>0,15 15,0</td>
</tr>
<tr>
<td>0,1 1,1 2,1</td>
<td>1,0 15,1</td>
</tr>
<tr>
<td>0,15 1,15 2,15</td>
<td>15,0 15,15</td>
</tr>
</tbody>
</table>

- Reads from GMEM to Writes to SMEM:
  - Coalesces access to SMEM from GMEM access.
- Reads from SMEM to Writes to GMEM:
  - Coalesces access to GMEM from SMEM access.
Coalesced Transpose

```c
__global__ void transpose(float *odata, float *idata, int width, int height)
{
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];

    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = 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 = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;
        block[index_block] = idata[index_in];
        index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }

    __syncthreads();

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

**Speedups with coalescing**
- 128x128: 0.011ms vs. 0.022ms  (**2.0X** speedup)
- 512x512: 0.07ms vs. 0.33ms  (**4.5X** speedup)
- 1024x1024: 0.30ms vs. 1.92ms  (**6.4X** speedup)
- 1024x2048: 0.79ms vs. 6.6ms  (**8.4X** speedup)

(Note: above times also include optimization for shared memory bank conflicts. Only accounts for ~10% of speedup – see transpose SDK example.)
Optimization Example 2: Parallel Reduction
Parallel Reduction

- Common and important data parallel primitive
- Easy to implement in CUDA
  - Harder to get it right
- Serves as a great optimization example
  - We’ll walk step by step through 7 different versions
  - Demonstrates several important optimization strategies
Parallel Reduction

Tree-based approach used within each thread block

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

But 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 \# multiprocessors * \# resident blocks / multiprocessor) to avoid deadlock, which 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
Solution: Kernel Decomposition

Avoid global sync by decomposing computation into multiple kernel invocations

In the case of reductions, code for all levels is the same
- Recursive kernel invocation
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

- Will use G80 GPU for this example
  - 384-bit memory interface, 900 MHz DDR
  - \(384 \times 1800 / 8 = 86.4 \text{ GB/s}\)
__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];
}
### Parallel Reduction: Interleaved Addressing

<table>
<thead>
<tr>
<th>Step</th>
<th>Stride</th>
<th>Threads</th>
<th>Values (shared memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0, 1</td>
<td>10, 1, 8, -1, 0, -2, 3, 5, -2, -3, 2, 7, 0, 11, 0, 2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0, 1</td>
<td>11, 1, 7, -1, -2, -2, 8, 5, -5, -3, 9, 7, 11, 11, 2, 2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0, 1</td>
<td>18, 1, 7, -1, 6, -2, 8, 5, 4, -3, 9, 7, 13, 11, 2, 2</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0, 1</td>
<td>24, 1, 7, -1, 6, -2, 8, 5, 17, -3, 9, 7, 13, 11, 2, 2</td>
</tr>
</tbody>
</table>
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 mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem: highly divergent branching results in very poor performance!
### Performance for 4M element reduction

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

Note: Block Size = 128 threads for all tests
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();
}
```

New Problem: Shared Memory Bank Conflicts
## Performance for 4M element reduction

### Kernel 1:
- Interleaved addressing with divergent branching
- **Time (2^22 ints):** 8.054 ms
- **Bandwidth:** 2.083 GB/s

### Kernel 2:
- Interleaved addressing with bank conflicts
- **Time (2^22 ints):** 3.456 ms
- **Bandwidth:** 4.854 GB/s
- **Step Speedup:** 2.33x
- **Cumulative Speedup:** 2.33x
Parallel Reduction: Sequential Addressing

Sequential addressing is 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>Addressing Pattern</th>
<th>Time (2^22 ints)</th>
<th>Bandwidth (GB/s)</th>
<th>Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1:</td>
<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>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>
</tbody>
</table>
Idle Threads

Problem:

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

Half of the threads are idle on first loop iteration!

This is wasteful…
Reduction #4: First Add During Load

Halve the number of blocks, and 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,
// 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();
```
<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>
</tbody>
</table>
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, # “active” threads decreases
  - When $s \leq 32$, we have only one warp left
- Instructions are SIMD synchronous within a warp
- That means when $s \leq 32$:
  - We don’t need to `__syncthreads()`
  - We don’t need “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

```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];
}
```

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
## 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</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addressing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with divergent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addressing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with bank</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>during global</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>warp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Complete Unrolling

If we knew the number of iterations at compile time, we could completely unroll the reduction.
- Luckily, the block size is limited by the GPU to 512 threads.
- 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?

Templates to the rescue!
- CUDA supports C++ template parameters on device and host functions.
Unrolling with Templates

Specify block size as a function template parameter:

```cpp
template <unsigned int blockSize>
__global__ void reduce5(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) {
    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];
}
```

Note: all code in RED will be evaluated at compile time.

Results in a very efficient inner loop!
Don’t we still need block size at compile time?

Nope, just a switch statement for 10 possible block sizes:

```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;
}
```
## 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>
<tr>
<td>Kernel 6:</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
<tr>
<td>completely unrolled</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parallel Reduction Complexity

- Log($N$) parallel steps, each step $S$ does $N/2^S$ independent ops
  - Step Complexity is $O(\log N)$

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

- With $P$ threads physically in parallel ($P$ processors),
  time complexity is $O(N/P + \log N)$
  - 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 × time complexity

- Allocate threads instead of processors: \( O(N) \) threads
- Within a block, 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 \( O(\log N) \) steps
- Cost = \( O((N/\log N) \times \log N) = O(N) \)

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
- In my experience, 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 perf with 64-256 blocks of 128 threads
  - 1024-4096 elements per thread
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();
```
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

<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 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 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>
<tr>
<td>Kernel 6:</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
<tr>
<td>completely unrolled</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 7:</td>
<td>0.268 ms</td>
<td>62.671 GB/s</td>
<td>1.42x</td>
<td>30.04x</td>
</tr>
<tr>
<td>multiple elements per thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Kernel 7 on 16M elements: 72 GB/s!
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];
    __syncthreads();
}

Final Optimized Kernel
Performance Comparison

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)
Types of optimization

Interesting observation:

Algorithmic optimizations
- Changes to addressing, algorithm cascading
  - 11.84x speedup, combined!

Code optimizations
- Loop unrolling
  - 2.54x speedup, combined
Conclusion

- Understand CUDA performance characteristics
  - Memory coalescing
  - Divergent branching
  - Bank conflicts
  - Latency hiding

- Use peak performance metrics to guide optimization

- Understand parallel algorithm complexity theory

- Know how to identify type of bottleneck
  - e.g. memory, core computation, or instruction overhead

- Optimize your algorithm, *then* unroll loops

- Use template parameters to generate optimal code

Questions: mharris@nvidia.com
S05: High Performance Computing with CUDA

Extra Slides
Parallel Memory Architecture

In a parallel machine, many threads access memory
- Therefore, memory is divided into banks
- Essential to achieve high bandwidth

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

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

No Bank Conflicts
  - Linear addressing
  - stride == 1

No Bank Conflicts
  - Random 1:1 Permutation
Bank Addressing Examples

2-way Bank Conflicts
- Linear addressing
- stride == 2

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 8
Thread 9
Thread 10
Thread 11

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 15

8-way Bank Conflicts
- Linear addressing
- stride == 8

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 15

Bank 0
Bank 1
Bank 2
Bank 7
Bank 8
Bank 9
Bank 15

Thread 15
How addresses map to banks on G80

- Bandwidth of each bank is 32 bits per 2 clock cycles
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
    - No bank conflicts possible between threads in first and second half of a warp

- Shared memory is as fast as registers if there are no bank conflicts
Shared memory bank conflicts

No conflicts:
- If all threads of a half-warp access different banks, there is no bank conflict
- If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)

Conflicts:
- Bank Conflict: multiple threads in the same half-warp access the same bank
- Must serialize the accesses
- Cost = max # of simultaneous accesses to a single bank
Optimizing threads per block

Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps

More threads per block == better memory latency hiding

But, more threads per block == fewer registers per thread
  - Kernel invocations can fail if too many registers are used

Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 128 to 256 threads a better choice
    - Usually still enough regs to compile and invoke successfully
  - This all depends on your computation!
    - Experiment!
Global memory access: 400-600 cycle latency
- Blocks dependent instructions in the same thread

Remedy:
- More threads!
- Instructions in other threads are not blocked
- Maximize occupancy

Same idea as pipelining:
- 4 sequential reads take at least $4 \times 400 = 1,600$ cycles
- 4 threads, one read each, take: $400 + 1 + 1 + 1 = 403$ cycles
Latency Hiding: Global Memory

- Multiprocessor can run up to 768 threads
  - Max threadblock size is 512 threads
- Configurations with 100% occupancy:
  - 2 blocks x 384 threads
  - 3 blocks x 256 threads
  - 4 blocks x 192 threads
  - 6 blocks x 128 threads
  - 8 blocks x 96 threads
- Minimal latency:
  - 50% or higher occupancy AND
  - 128 or more threads/block
Latency Hiding: Register Dependency

Read-after-write register dependency
- Instruction’s result can be read 11 cycles later
- Scenarios:
  - CUDA:
    - `x = y + 5;
    - `z = x + 3;
  - PTX:
    - `add.f32 $f3, $f1, $f2
    - `add.f32 $f5, $f3, $f4
    - `ld.shared.f32 $f3, [$r31+0]
    - `add.f32 $f3, $f3, $f4

To completely hide the latency:
- Run at least 192 threads (6 warps) per multiprocessor
- At least 25% occupancy
- Threads do not have to belong to the same thread block
Latency Hiding: Synchronization

- Thread synchronization (__syncthreads)
- More threads per block = higher latency
  - Waiting on threads in other warps to reach the sync point
- Smaller thread blocks will reduce latency
- BUT: usually not really a problem
Register Pressure

Solution to latency issues = more threads per SM

Limiting Factors:

- Number of registers per kernel
  - 8192 per SM, partitioned among concurrent threads

- Amount of shared memory
  - 16KB per SM, partitioned among concurrent threadblocks

Check .cubin file for # registers / kernel

Use \texttt{--maxrregcount=N} flag to NVCC

- \texttt{N} = desired maximum registers / kernel

At some point “spilling” into LMEM may occur

- Reduces performance – LMEM is slow
- Check .cubin file for LMEM usage
Determining resource usage

Compile the kernel code with the -cubin flag to determine register usage.

Open the .cubin file with a text editor and look for the “code” section.

```plaintext
architecture {sm_10}
abiversion {0}
modname {cubin}

code {
  name = BlackScholesGPU
  lmem = 0
  smem = 68
  reg = 20
  bar = 0
  bincode {
    0xa0004205 0x04200780 0x40024c09 0x00200780
    ...
  }
}
```

- per thread local memory
- per thread block shared memory
- per thread registers
CUDA Occupancy Calculator

CUDA GPU Occupancy Calculator

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

2. Enter your resource usage:
   - Threads Per Block
   - Registers Per Thread
   - Shared Memory Per Block (Bytes)

3. GPU Occupancy Data is displayed here and in the graphs:
   - Active Threads per Multiprocessor
   - Active Warps per Multiprocessor
   - Active Thread Blocks per Multiprocessor
   - Occupancy of each Multiprocessor
   - Maximum Simultaneous Blocks per GPU

(Note: This assumes there are at least this many blocks)

Physical Limits for GPU
- Multiprocessors (per GPU): 18
- Warps (per Multiprocessor): 32
- Threads (per Multiprocessor): 768
- Thread Blocks (per Multiprocessor): 9
- Total # of SRF registers per Multiprocessor: 1584
- Shared Memory / Multiprocessor (Bytes): 16 KB

Allocation Per Thread Block
- Warps: 6
- Registers: 3048
- Shared Memory: 64

These values used in computing the occupancy data below:

Maximum Thread Blocks Per Multiprocessor: Blocks
Limited by Max Warps / Multiprocessor: 48
Limited by Registers / Multiprocessor: 2
Limited by Shared Memory / Multiprocessor: 24
Thread Block Limit Per Multiprocessor: the minimum of these 3

CUDA Occupancy Calculator
- Occupancy: 1.1

Copyright and License

Read

start
**SMEM Optimization**

Reads from SMEM

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1,0</td>
<td>2,0</td>
<td>15,0</td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>2,1</td>
<td>15,1</td>
</tr>
<tr>
<td>0,15</td>
<td>1,15</td>
<td>2,15</td>
<td>15,15</td>
</tr>
</tbody>
</table>

- Threads read SMEM with stride = 16
  - Bank conflicts

**Solution**

- Allocate an “extra” column
- Read stride = 17
- Threads read from consecutive banks