

CS529 Lecture 08:

Topics on CUDA Optimization

Dimitrios S. Nikolopoulos

University of Crete and FORTH-ICS

May 24, 2011

Sources of material

- ▶ NVIDIA CUDA Supercomputing'2007 Tutorial
 - ▶ <http://gpgpu.org/sc2007>
 - ▶ In particular, Section 2 (Language), Section 5 (Optimization)

CUDA at a glance

- ▶ CUDA enables efficient use of the massive parallelism of NVIDIA GPUs
 - ▶ Direct execution of data-parallel programs
 - ▶ Without the overhead of a graphics API
- ▶ Even higher speedups are achievable by understanding and tuning for GPU architecture
 - ▶ This class covers performance aspects and useful optimizations

Terminology

- ▶ **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 across all “multi-processors” of the GPU (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 (operations/byte)
- ▶ Sometimes is 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 data back and forth from/to host

Optimize Memory Coherence

- ▶ Coalesced vs. non-coalesced accesses to device memory and shared cache can provide an order of magnitude higher performance
- ▶ Optimize for spatial locality in cached texture memory (if used)
- ▶ In shared memory, avoid high-degree bank conflicts

Using 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-coalescable addressing
 - Matrix transpose example later

Efficiency in Parallelism

- ▶ 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

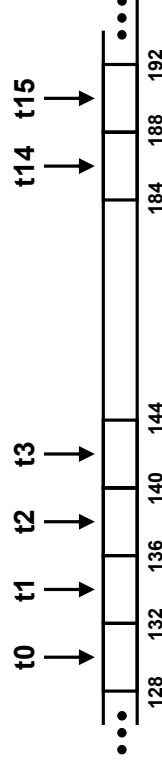
Global Memory Reads/Writes

- ▶ Highest latency instructions: 400–600 clock cycles
- ▶ Likely to be performance bottleneck
- ▶ Optimizations can greatly increase performance
 - ▶ Coalescing: up to $10\times$ speedup

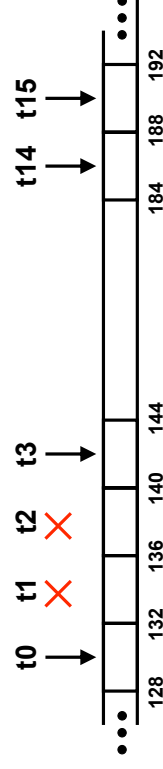
Coalescing

- ▶ A **coordinated** read by a warp
- ▶ A **contiguous** region of global memory:
 - ▶ **128 bytes** - each thread reads a word: int, float, ...
 - ▶ **256 bytes** - each thread reads a double-word: int2, float2, ...
 - ▶ **512 bytes** - each thread reads a quad-word: int4, float4, ...
- ▶ **Additional restrictions:**
 - ▶ Starting address for a region must be a multiple of region size
 - ▶ The k^{th} thread in a warp must access the k^{th} element in a block being read
- ▶ **Exception:** not all threads must be participating
 - ▶ Predicated access, divergence within a warp

Coalesced reading of floats

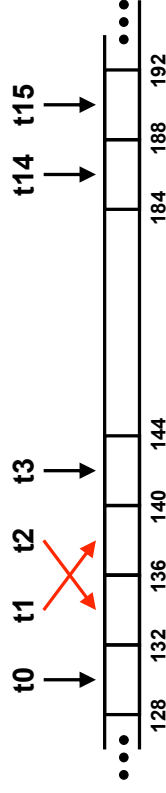


All threads participate

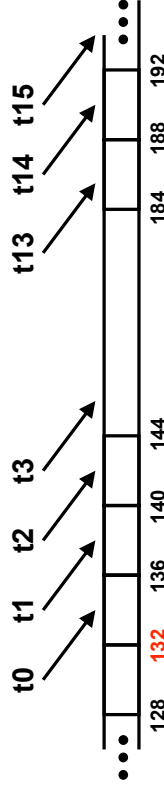


Some Threads Do Not Participate

Uncoalesced reading of floats



Permuted Access by Threads



Misaligned Starting Address (not a multiple of 64)

Coalescing performance

- ▶ Experiment:
 - ▶ Kernel: read a float, increment, write back
 - ▶ 3M floats (12MB)
 - ▶ Times averaged over 10K runs
- ▶ 12K blocks \times 256 threads:
 - ▶ 356 μ s – coalesced
 - ▶ 357 μ s – coalesced, some threads don't participate
 - ▶ 3,494 μ s – permuted/misaligned thread access

Uncoalesced accesses to float3's

```

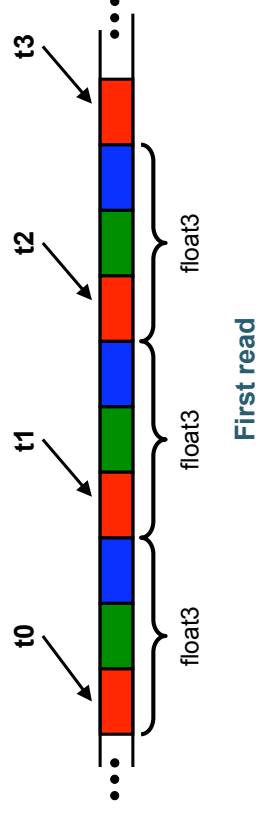
__global__ void accessFloat3(float3* d_in, float3 d_out)
{
    int index = blockIdx.x*blockDim.x+threadIdx.x;
    float3 a=d_in[index];

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

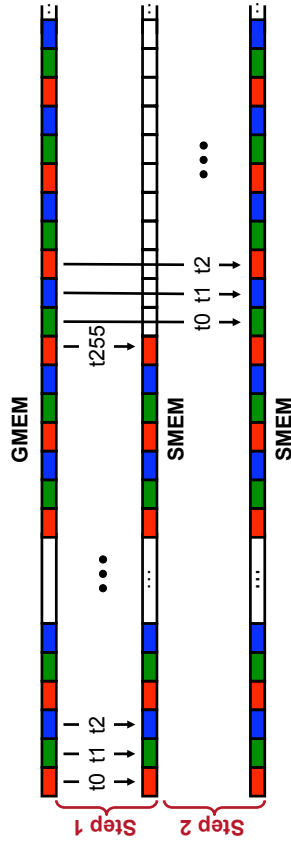
    d_out[index]=a;
}
    
```

float3 example

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



Coalescing float3's



Similarly, Step3 starting at offset 512

Coalesced accesses of float3's

- ▶ Use shared memory to allow coalescing
 - ▶ Need $\text{sizeof(float3)} * (\text{threads/block})$ bytes of SMEM
 - ▶ Each thread reads 3 scalar floats:
 - ▶ Offsets: 0, (threads/block), $2 * (\text{threads/block})$
 - ▶ 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!

Improved coalescing of float3's

```

__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 of data types of size $\neq 4, 8, \text{ or } 16$ 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 shared memory to achieve coalescing

Performance of coalescing

- ▶ Experiment:
 - ▶ Kernel: read a float, increment, write back
 - ▶ 3M floats (12MB)
 - ▶ Times averaged over 10K runs
- ▶ 12K blocks \times 256 threads:
 - ▶ 356 μ s - coalesced
 - ▶ 357 μ s - coalesced, some threads don't participate
 - ▶ 3,494 μ s - permuted/misaligned thread access
- ▶ 4K blocks \times 256 threads:
 - ▶ 3,302 \times s - float3 uncoalesced
 - ▶ 359 \times s - 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 Arrays of Structures
 - ▶ If SoA is not viable, read/write through Shared Memory
- ▶ Future-proof 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 (QuadroFX 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
- ▶ Problem may become obsolete in new processors integrating general-purpose cores with GPUs (e.g. ARM-NVIDIA partnership).

Page-locked Memory Transfers

- ▶ cudaMallocHost() allows allocation of page-locked host memory
- ▶ Enables highest cudaMemcopy performance
 - ▶ 3.2 GB/s+ common on PCI-express x16
 - ▶ 4 GB/s measured on nForce680i motherboards (overclocked PCI-e)
- ▶ See the “bandwidthTest” CUDA SDK sample
- ▶ Use with caution
 - ▶ Allocating too much page-locked memory may reduce overall system performance

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 Trade-Off

- ▶ Increasing occupancy does not necessarily increase performance BUT ...
- ▶ Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels
 - ▶ It comes down to arithmetic intensity and available parallelism

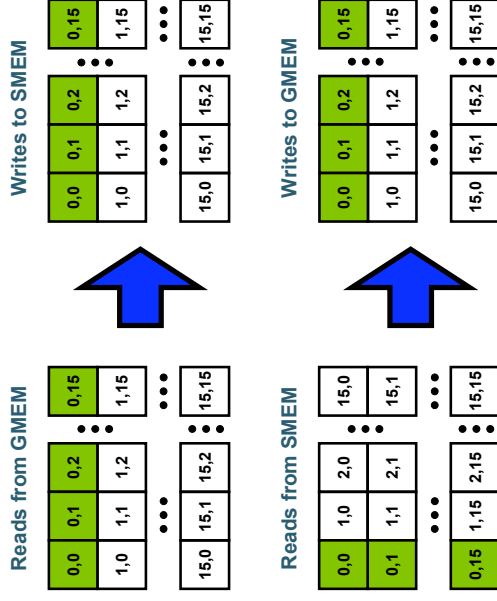
Grid/Block Size Heuristics

- ▶ # 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 are not 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

Parameterization

- ▶ Parameterization helps adaptation to different GPUs
- ▶ GPUs vary in many ways
 - ▶ # of multiprocessors
 - ▶ Memory bandwidth
 - ▶ Shared memory size
 - ▶ Register file size
 - ▶ Threads per block
- ▶ Self-tuning, auto-tuning approaches (like FFTW and ATLAS)
 - ▶ “Experiment” mode discovers and saves optimal configuration

Coalesced Transpose



```

__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];
    }
}

```

Coalesced Transpose

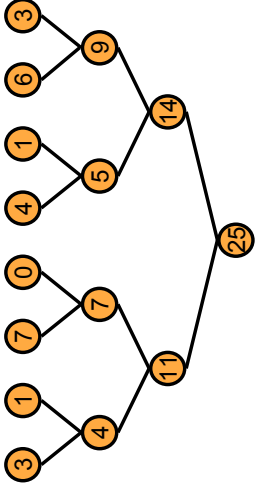
Coalesced Transpose Performance

- ▶ Speedups with coalescing
 - ▶ 128 × 128: 0.011ms vs. 0.022ms (2.0 × speedup)
 - ▶ 512 × 512: 0.07ms vs. 0.33ms (4.5 × speedup)
 - ▶ 1024 × 1024: 0.30ms vs. 1.92ms (6.4 × speedup)
 - ▶ 1024 × 2048: 0.79ms vs. 6.6ms (8.4 × speedup)

Parallel Reduction

- ▶ Common and important data parallel primitive
- ▶ Easy to implement in CUDA
 - ▶ Harder to get it right
- ▶ Serves as a great optimization example
 - ▶ Step by step through 7 different versions
 - ▶ Demonstrates several important optimization strategies

Parallel Reduction

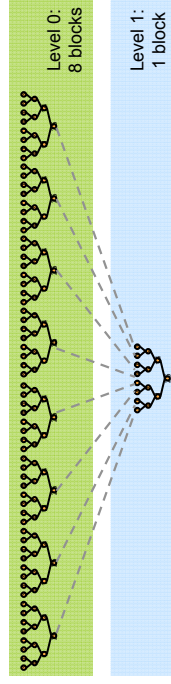
- ▶ Tree-based approach used within each thread block
- 
- ▶ Need to be able to use multiple thread blocks
 - ▶ To process very large arrays
 - ▶ To keep all multiprocessors on the GPU busy
 - ▶ Each thread block reduces a portion of the array
 - ▶ But how do we communicate partial results between thread blocks?

Global synchronization

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

Global Synchronization through 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

Optimization Goal for Reductions

- ▶ 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
- ▶ G80 GPU for this example
 - ▶ 384-bit memory interface, 900 MHz DDR
 - ▶ $384 * 1800 / 8 = 86.4$ GB/s

Reduction #1: Interleaved Accesses

```

__global__ void reduce1(int* g_idata, int* g_odata) {
    extern __shared__ int sdata[];

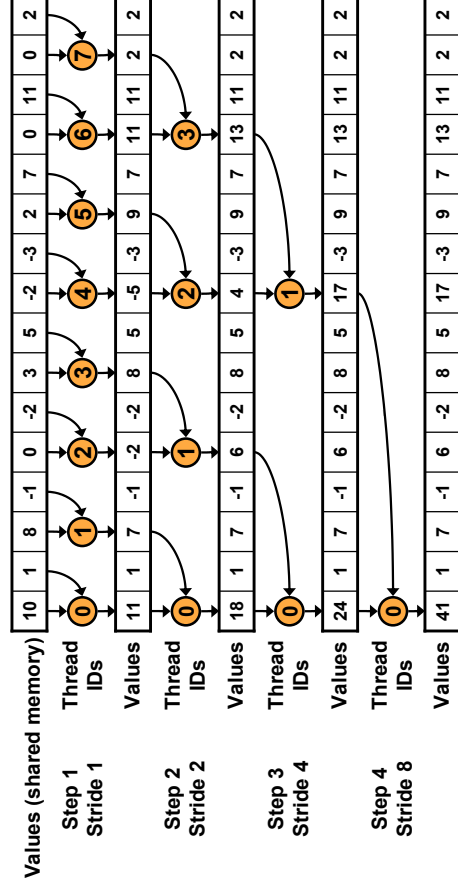
    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // 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];
}

```

Reduction #1: Interleaved Accesses



Reduction #1: Interleaved Accesses

```

__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];
}

```

Performance of 4M Element Reduction

	Time (2 ²² ints)	Bandwidth
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s

Note: Block Size = 128 threads for all tests

- ▶ Problem: highly divergent branching results in very poor performance

Reduction #2: Interleaved Addressing

- ▶ Replace divergent branch in inner loop

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

- ▶ Use strided index and non-divergent branch

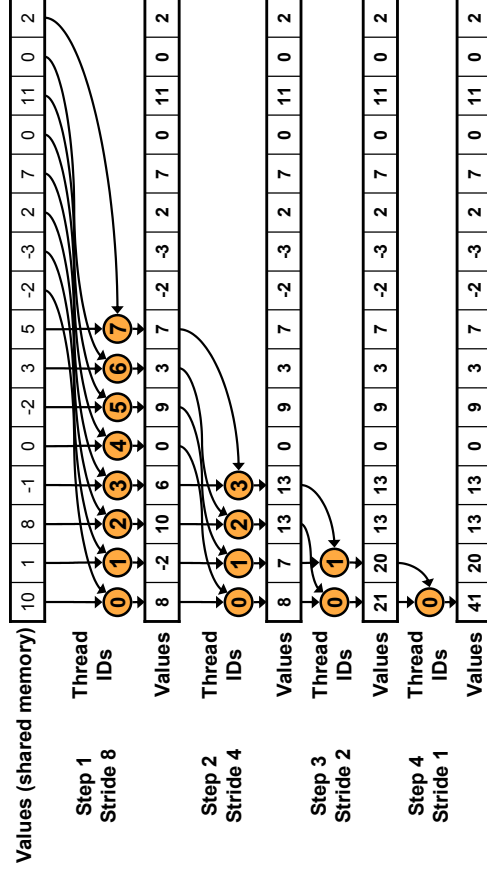
```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

- ▶ New problem: shared memory bank conflicts

Performance of 4M Element Reduction

	Time (2 ²² ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x

Reduction with Sequential Addressing



Reduction #3: Sequential Addressing

- ▶ Replace strided addressing in inner loop

```
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();
}
```

- ▶ Use reversed loop and thread ID-based indexing

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

Performance of 4M Element Reduction

	Time (2 ²² ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

- ▶ Problem: Half of the threads are idle on first loop iteration!

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

Idle Threads

Reduction #4: First Add During Load

- ▶ Halve the number of blocks, and replace single load:

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockDim.x*blockDim.x+ threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

- ▶ With two loads and first add of the reduction:

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

Performance of 4M Element Reduction

	Time (2 ²² ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x

Instruction Execution Overhead

- ▶ At 17 GB/s, reduction is 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

Reduction #5: Unrolling Last Warp

- ▶ Save work in all warps by unrolling last 32 iterations – avoid loop overhead and if statement

```
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];
}
```

Unrolling 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
- ▶ Unroll the last 6 iterations of the inner loop

Performance of 4M Element Reduction

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

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

Reduction #6: Completely Unrolled

- ▶ All conditionals on **blockSize** evaluated at compile time

```
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];  
}
```

Unrolling with Templates

- ▶ Specify block size as a function template parameter:

```
template <unsigned int blockSize>  
__global__ void reduce5(int *g_idata, int *g_odata)
```

Invoking Template Kernels

- ▶ Replace compile-time block size constant with a case statement

```
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;  
}
```

Reduction #6: Completely Unrolled

- ▶ All conditionals on **blockSize** evaluated at compile time

```
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];  
}
```

Reduction #6: Completely Unrolled

- ▶ All conditionals on **blockSize** evaluated at compile time

```
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];  
}
```

Reduction #6: Completely Unrolled

- ▶ All conditionals on **blockSize** evaluated at compile time

```
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];  
}
```

Reduction #6: Completely Unrolled

- ▶ All conditionals on **blockSize** evaluated at compile time

```
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];  
}
```

Parallel Reduction Complexity

- ▶ $\text{Log}(N)$ parallel steps, each step S does $N/2^S$ independent operations
Step complexity is $O(\log N)$
- ▶ For $N = 2^D$ reduction performs $\sum_{s \in [1 \dots D]} 2^{D-s} = N - 1$ operations
 - ▶ **Work Complexity** is $O(N)$ therefore reduction is **work-efficient**
 - ▶ Reduction does not perform more operations than 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 work is $O(\log N)$

Algorithm Cascading

- ▶ Combine sequential and parallel reduction
 - ▶ Each thread loads and sums multiple elements into shared memory
- ▶ Brent's theorem says each thread should sum $O(\log N)$ elements
 - ▶ i.e. 1024 or 2048 elements per block vs. 256
- ▶ 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 and 1024–4096 elements per thread

Cost Efficiency

- ▶ Cost of a parallel algorithm is processors \times time complexity
 - ▶ Allocate threads instead of processors: $O(N)$ threads
 - ▶ Within a block, time complexity is $O(\log N)$, so cost is $O(N/\log N)$, therefore parallel reduction is 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 = $O((N/\log N) * \log N) = O(N)$
- ▶ Sometimes called algorithm cascading
 - ▶ Can lead to significant speedups in practice

Reduction # 7: Multiple Adds/Thread

- ▶ Replace load and add of two elements:

```
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:

```
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();
```

Performance of 4M Element Reduction

	Time (2 ²² ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Final Optimized Kernel

```
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];
}
```