CPSC/ECE 4780/6780

General-Purpose Computation on Graphical Processing Units (GPGPU)

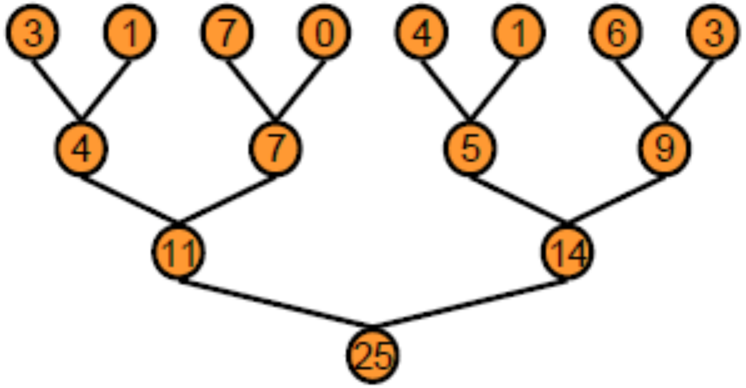
Lecture 7: Parallel Reduction

Recaps from Last Lecture

- Julia Set example: CPU-based sequential code => GPU-based parallel code using global memory
- Bitmap example: Shared memory (synchronization)
- Ray tracing example: Constant memory (faster execution)
- Heat transfer simulation example: Texture memory (“spatial locality”, 1D or 2D)

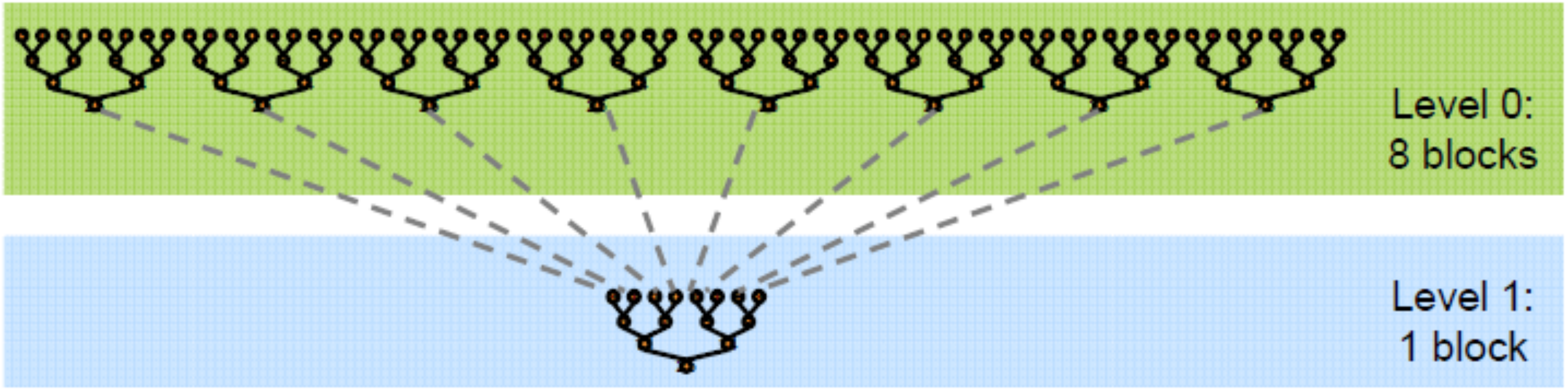
Reduction

- Reduction: a process of taking an input array and performing some computations that produce a smaller array of results
  - Sum, min, max, average…
- Sequential reduction (CPU)
  - for (int i = 0; i < n; i++) …
- Parallel reduction (GPU)
  - 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



How to Communicate Partial Results Between Thread Block?

- Problem: partial results need to be shared
- Ideally: global synchronization, but not feasible
- In reality: CUDA synchronization is only allowed at block level
- Solution: decompose into multiple kernel invocations
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible hardware overhead, low software overhead



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

Optimizing Parallel Reduction in CUDA

- Metrics of performance
  - GFLOP/s: for compute-bound kernels
  - Bandwidth: for memory-bound kernels
- Reductions are a memory bound problem
  - Measure optimization using bandwidth
- Example
  - Calculate the sum of an array of integers with N elements

Serial Reduction

```
// Reduction via serial iteration
float sum(float *data, int n) {
    float result = 0;
    for(int i = 0; i < n; ++i) {
        result += data[i];
    }
    return result;
}
```

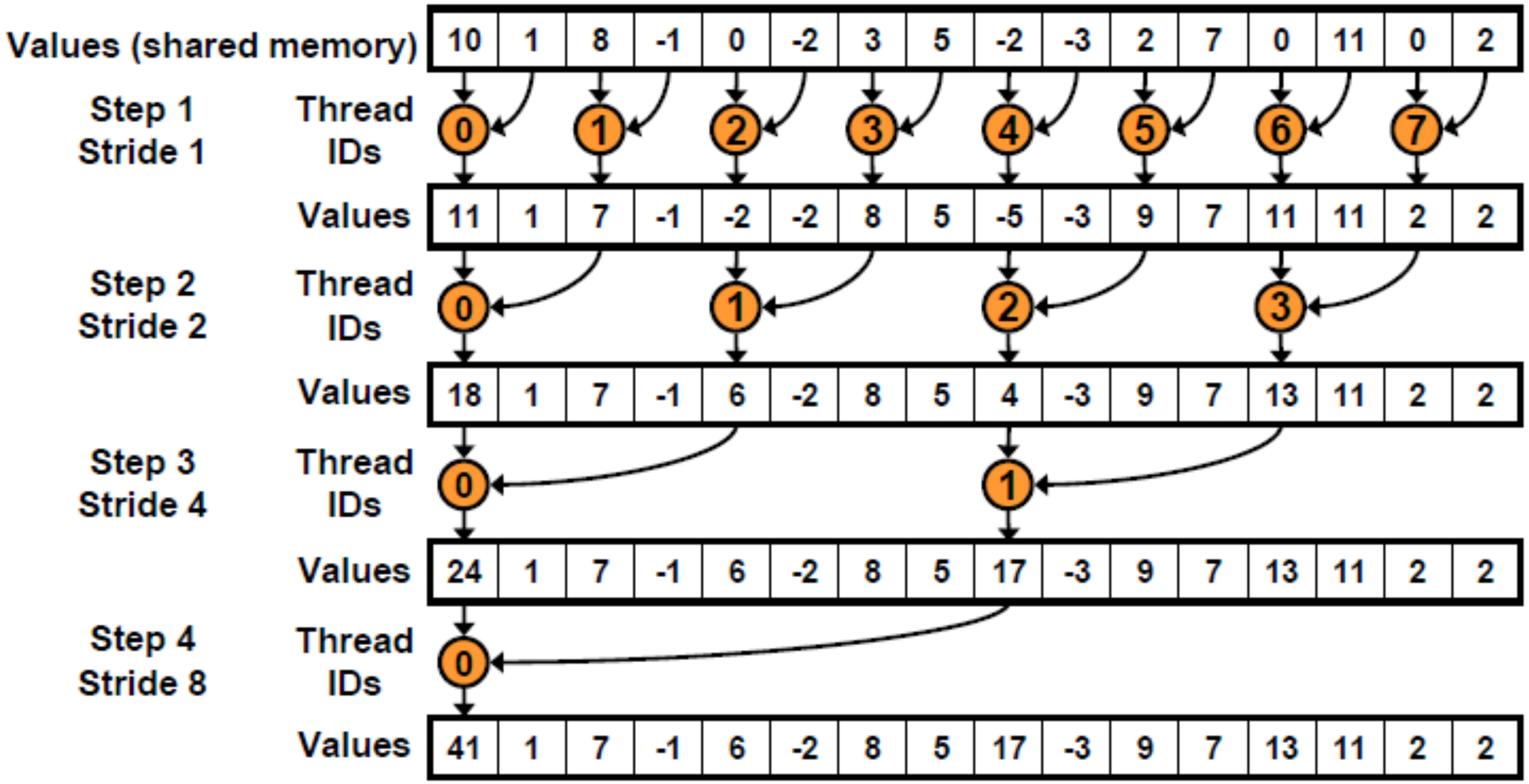
- Convert to parallel addition by
  - Partition the input vector into smaller chunks
  - Have a thread calculate the partial sum for each chunk
  - Add the partial results from each chunk into a final sum

Iterative Pairwise Implementation

- A chunk contains only a pair of elements
- A thread sums those two elements to produce one partial result
- The partial results are stored in-place in the original input vector
- The new values are used as the input to be summed in the next iteration
- A final sum is available when the length of the output vector reaches one

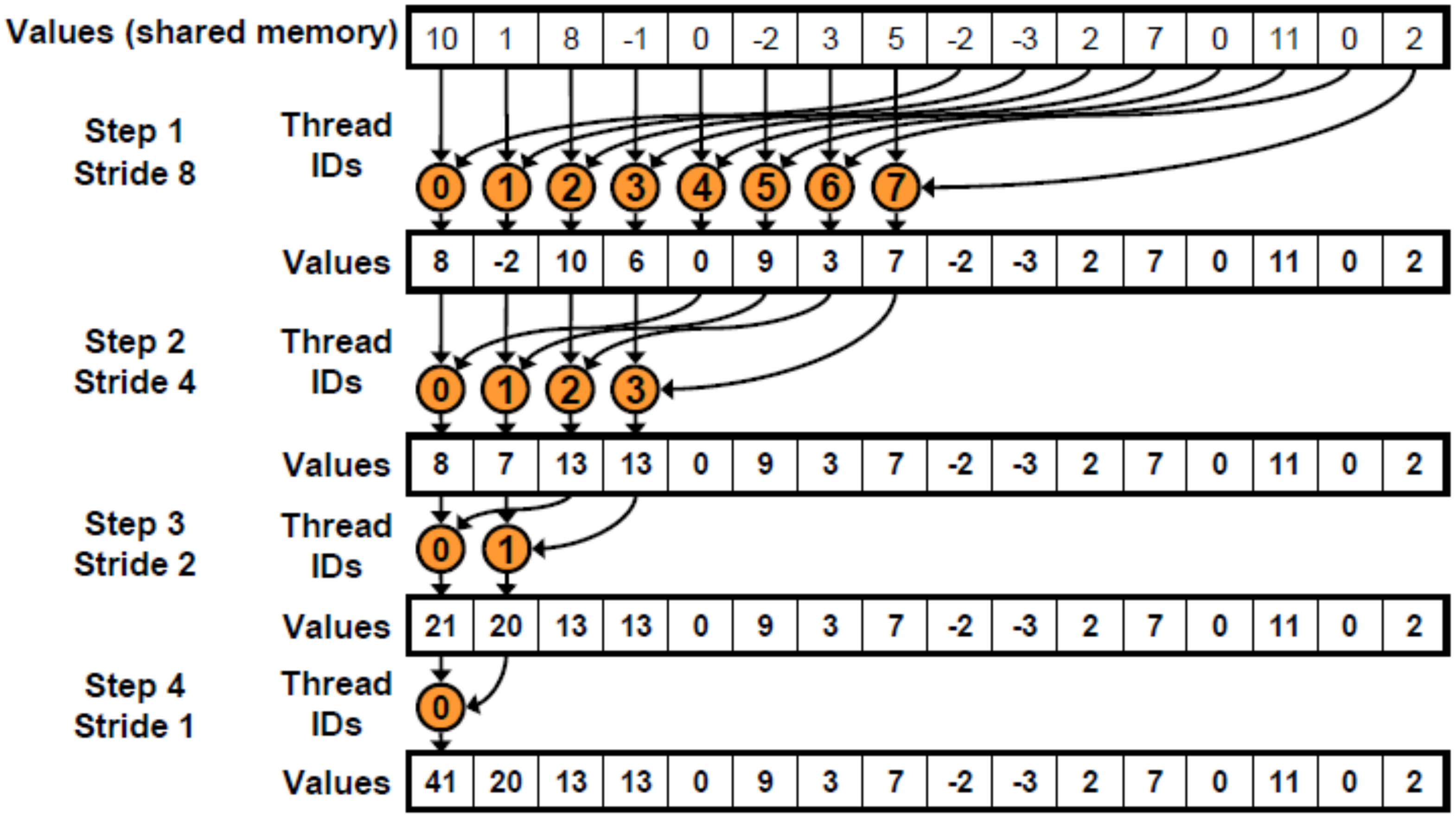
Parallel Reduction – Neighbored pair

- Elements are paired with their immediate neighbor
- N – 1 sums
- log_2^N Steps



Parallel Reduction – Interleaved pair

- Paired elements are separated by a given stride (N/2)
- N – 1 sums
- log_2^N Steps

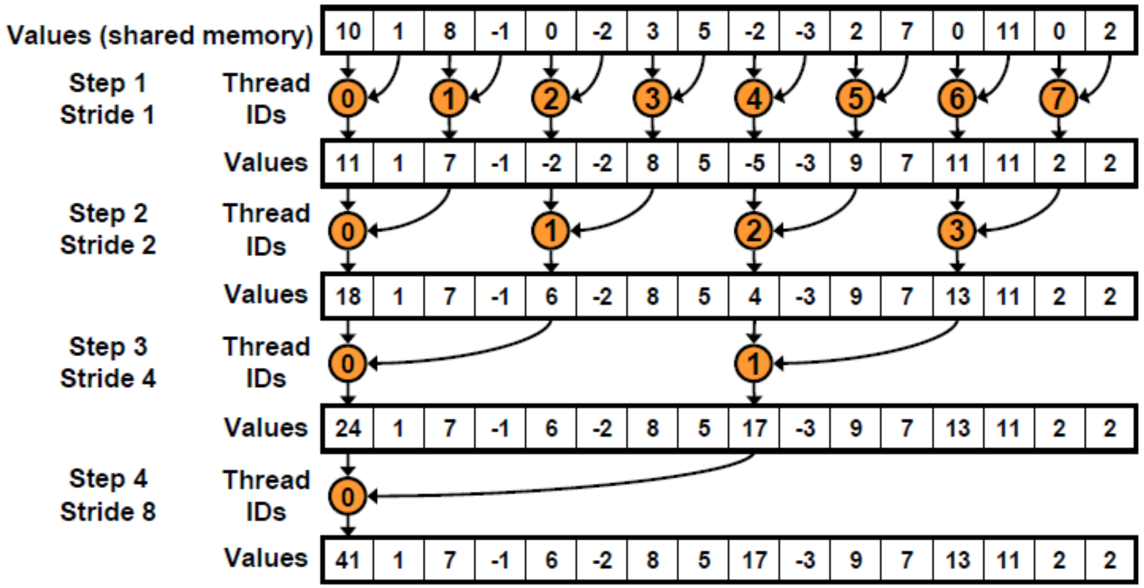


Interleaved Reduction on CPU

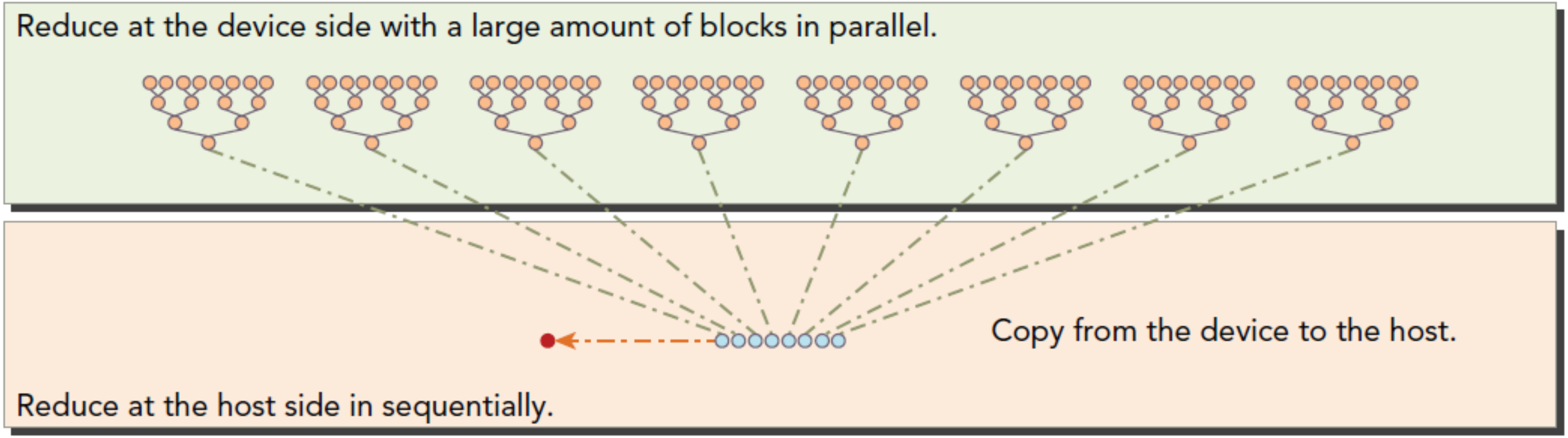
```
// Recursive Implementation of Interleaved Pair Approach
int recursiveReduce(int *data, int const size)
{
    if (size == 1) return data[0];

    int const stride = size / 2;

    for (int i = 0; i < stride; i++)
        data[i] += data[i + stride];

    return recursiveReduce(data, stride);
}
```

Parallel Reduction #1: Neighbored Pair Implementation on Global Memory

```
__global__ void reduceNeighboredGmem_1(int *g_idata, int *g_odata,
                                        unsigned int  n)
{
    // set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x * blockDim.x;

    // boundary check
    if (idx >= n) return;

    // in-place reduction in global memory
    for (int stride = 1; stride < blockDim.x; stride *= 2) {
        if ((tid % (2 * stride)) == 0) {
            idata[tid] += idata[tid + stride];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = idata[0];
}
```



Parallel Reduction #1




```
reduceNeighboredGmem_1<<<grid, block>>>(d_idata, d_odata, size);
CHECK(cudaMemcpy(h_odata, d_odata, grid.x * sizeof(int), cudaMemcpyDeviceToHost));
gpu_sum = 0;
for (int i = 0; i < grid.x; i++) gpu_sum += h_odata[i];
```

Parallel Reduction Kernel Performance

Kernel	Time (ms)	Step Speedup	Cumulative Speedup	Load Efficiency	Store Efficiency
reduceNeighboredGmem_1	4.1238			25.02%	25%

nvprof --metrics gld_efficiency,gst_efficiency ./a.out

Parallel Reduction #2: Neighbored Pair Implementation on Shared Memory

```
__global__ void reduceNeighboredGmem_1(int *g_idata, int *g_odata,
                                       unsigned int  n)
{
    // set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x * blockDim.x;

    // boundary check
    if (idx >= n) return;

    // in-place reduction in global memory
    for (int stride = 1; stride < blockDim.x; stride *= 2) {
        if ((tid % (2 * stride)) == 0) {
            idata[tid] += idata[tid + stride];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = idata[0];
}
```

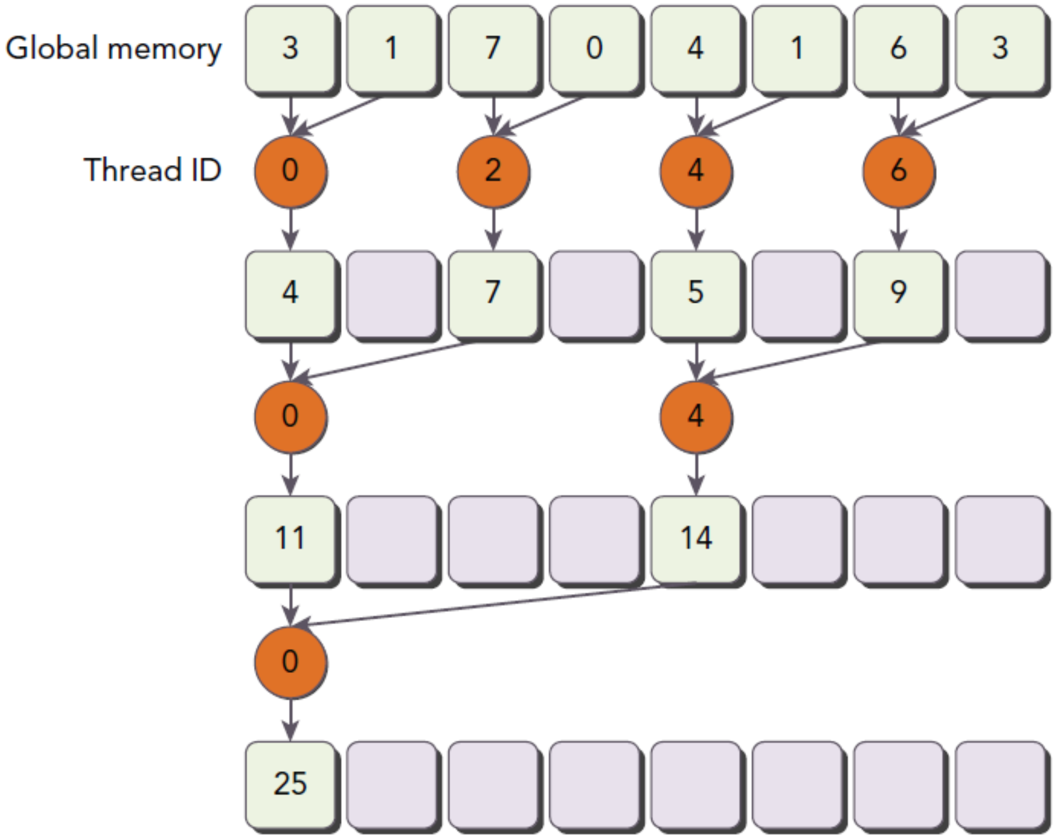
```
__global__ void reduceNeighboredSmem_2(int *g_idata, int *g_odata,
                                       unsigned int  n)
{
    __shared__ int smem[DIM];

    // set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x * blockDim.x;

    // boundary check
    if (idx >= n) return;

    smem[tid] = idata[tid];
    __syncthreads();

    // in-place reduction in global memory
    for (int stride = 1; stride < blockDim.x; stride *= 2) {
        if ((tid % (2 * stride)) == 0){
            smem[tid] += smem[tid + stride];
        }
        // synchronize within threadblock
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = smem[0];
}
```

Parallel Reduction Kernel Performance

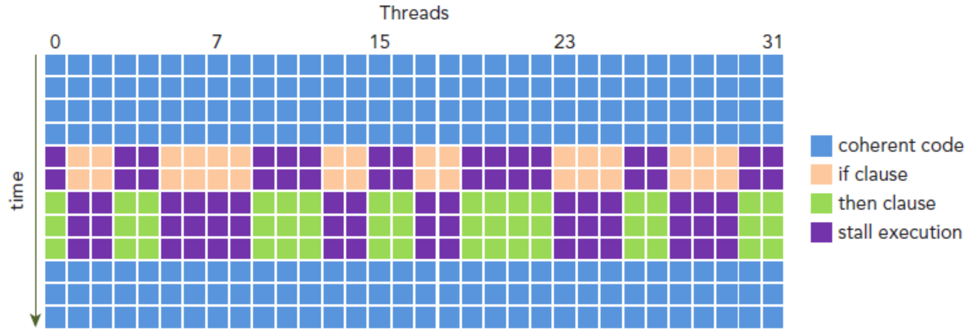
Kernel	Time (ms)	Step Speedup	Cumulative Speedup	Load Efficiency	Store Efficiency
reduceNeighboredGmem_1	4.1238			25.02%	25%
reduceNeighboredSmem_2	3.4246	1.20	1.20	100%	12.5%

Problem with Parallel Reduction #2: Warp Divergent

```
__global__ void reduceNeighboredSmem_2(int *g_idata, int *g_odata,
                                       unsigned int  n)
{
    __shared__ int smem[DIM];

    // set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x * blockDim.x;

    // boundary check
    if (idx >= n) return;

    smem[tid] = idata[tid];
    __syncthreads();

    // in-place reduction in global memory
    for (int stride = 1; stride < blockDim.x; stride *= 2) {
        if ((tid % (2 * stride)) == 0){
            smem[tid] += smem[tid + stride];
        }
        // synchronize within threadblock
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = smem[0];
}
```



Highly divergent warps
Slow % operation



Parallel Reduction #3: Reducing Warp Divergence for Parallel Reduction #2

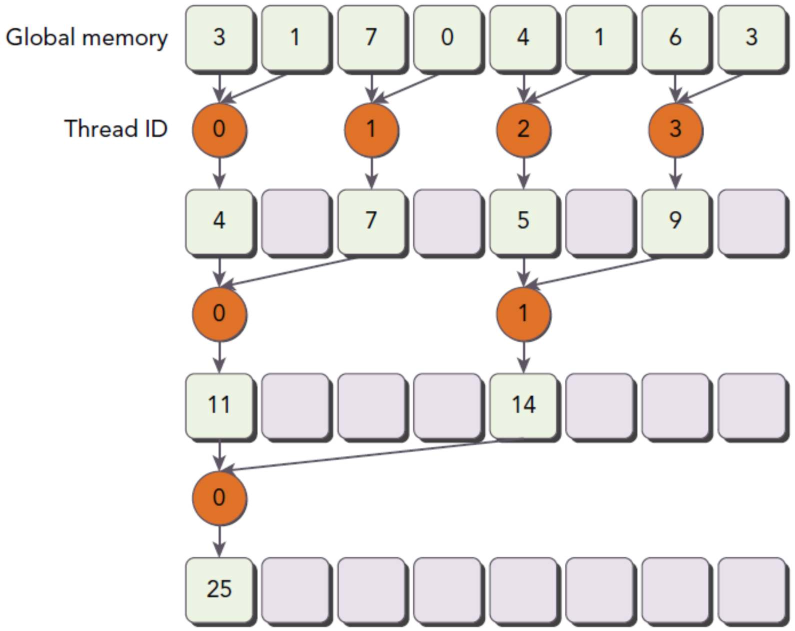
- Reduce warp divergence by rearranging the array index of each thread to force neighboring threads to perform the addition

Replace divergent branch in inner loop:

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

With strided index and non-divergent branch:

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



Parallel Reduction Kernel Performance

Kernel	Time (ms)	Step Speedup	Cumulative Speedup	Load Efficiency	Store Efficiency
reduceNeighboredGmem_1	4.1238			25.02%	25%
reduceNeighboredSmem_2	3.4246	1.20	1.20	100%	12.5%
reduceNeighboredSmemNoDivergence_3	2.5892	1.32	1.59	100%	12.5%

Parallel Reduction #4: Interleaved Pair Implementation

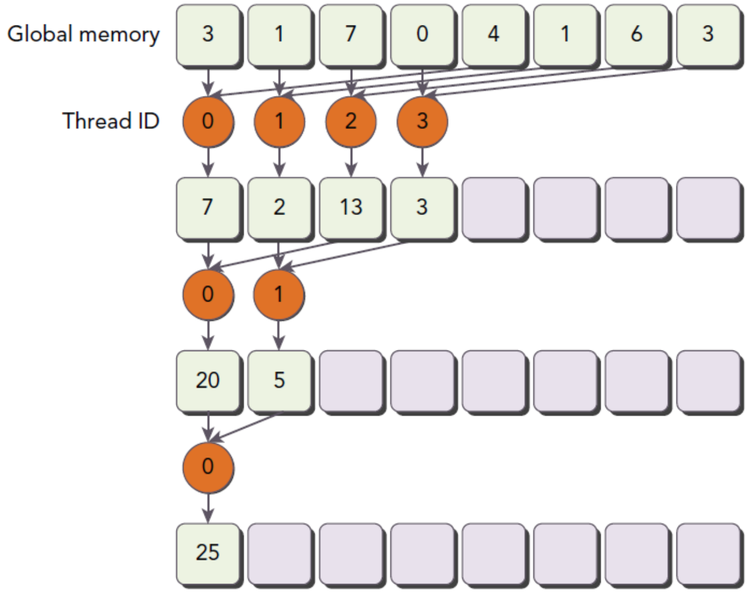
- The stride is started at half of the thread block size and then reduced by half on each iteration
- Each thread adds two elements separated by the current stride to produce a partial sum at each round

Replace strided indexing in inner loop:

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

With reversed loop and threadID-based indexing:

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



Parallel Reduction Kernel Performance

Kernel	Time (ms)	Step Speedup	Cumulative Speedup	Load Efficiency	Store Efficiency
reduceNeighboredGmem_1	4.1238			25.02%	25%
reduceNeighboredSmem_2	3.4246	1.20	1.20	100%	12.5%
reduceNeighboredSmemNoDivergence_3	2.5892	1.32	1.59	100%	12.5%
reduceInterleavedSmem_4	1.8775	1.38	2.20	100%	12.5%

Problem with Parallel Reduction #4: Idle Threads

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

Half of the threads are idle on first loop iteration!

Unrolling Loops

- **Loop unrolling** is a technique that attempts to optimize loop execution by reducing the frequency of branches and loop maintenance instructions
  - The body of a loop is written in code multiple times to reduce or remove iterations
  - **Loop unrolling factor**: the number of copies made of the loop body
  - The number of iterations in the enclosing loop is divided by the loop unrolling factor
  - Effective at improving performance for sequential array processing loops
- Performance gains come from low-level instruction improvements and optimizations that the compiler performs to the unrolled loop

A Simple Example of Loop Unrolling

- Consider the code fragment below:

```
for (int i = 0; i < 100; i++) {
  a[i] = b[i] + c[i];
}
```

- Replicate the body of the loop once:

```
for (int i = 0; i < 100; i += 2) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
}
```

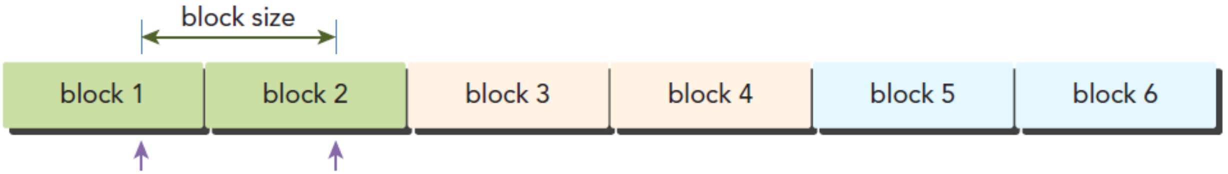
Less condition check
Simultaneous memory operation

Goal of loop unrolling:

- Improving performance by reducing instruction overheads
- Creating more independent instructions to schedule

Parallel Reduction #5: Interleaved Pair Implementation with Unrolling 2 Data Blocks

- Each thread works on more than one data block and processes a single element from each data block



The i^{th} thread first adds two elements before the reduction.

Halve the number of blocks, and replace single load:

```
// set thread ID
unsigned int tid = threadIdx.x;
unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

// convert global data pointer to the local pointer of this block
int *idata = g_idata + blockIdx.x * blockDim.x;
```

With two loads and first add of the reduction:

```
// set thread ID
unsigned int tid = threadIdx.x;
unsigned int idx = blockIdx.x * blockDim.x * 2 + threadIdx.x;

// convert global data pointer to the local pointer of this block
int *idata = g_idata + blockIdx.x * blockDim.x * 2;

// unrolling 2 data blocks
if (idx + blockDim.x < n) g_idata[idx] += g_idata[idx + blockDim.x];
```

<<<grid.x/2, block>>>

Parallel Reduction Kernel Performance

Kernel	Time (ms)	Step Speedup	Cumulative Speedup	Load Efficiency	Store Efficiency
reduceNeighboredGmem_1	4.1238			25.02%	25%
reduceNeighboredSmem_2	3.4246	1.20	1.20	100%	12.5%
reduceNeighboredSmemNoDivergence_3	2.5892	1.32	1.59	100%	12.5%
reduceInterleavedSmem_4	1.8775	1.38	2.20	100%	12.5%
reduceUnrolling2_5	0.9805	1.91	4.34	100%	98.65%

Unrolling 8 Data Blocks

```
__shared__ int smem[DIM];

// set thread ID
unsigned int tid = threadIdx.x;
unsigned int idx = blockIdx.x * blockDim.x * 8 + threadIdx.x;

// convert global data pointer to the local pointer of this block
int *idata = g_idata + blockIdx.x * blockDim.x * 8;

// unrolling 8
if (idx + 7 * blockDim.x < n)
{
    int a1 = g_idata[idx];
    int a2 = g_idata[idx + blockDim.x];
    int a3 = g_idata[idx + 2 * blockDim.x];
    int a4 = g_idata[idx + 3 * blockDim.x];
    int b1 = g_idata[idx + 4 * blockDim.x];
    int b2 = g_idata[idx + 5 * blockDim.x];
    int b3 = g_idata[idx + 6 * blockDim.x];
    int b4 = g_idata[idx + 7 * blockDim.x];
    g_idata[idx] = a1 + a2 + a3 + a4 + b1 + b2 + b3 + b4;
}
smem[tid] = idata[tid];
__syncthreads();
```

`<<<grid.x/8, block>>>`

Unrolling the Last Warp

- As reduction proceeds, # of “active” threads decreases
  - When stride <= 32, we have only one warp left
- Instructions are SIMD synchronous within a warp
- That means when stride <= 32:
  - We don’t need to __syncthreads()
  - We don’t need “if (tid < stride)” because it doesn’t save any work
- Unroll the last 6 iterations of the inner loop to avoid executing loop control and thread synchronization logic

Parallel Reduction #6: Unroll the Last Warp

```
// in-place reduction in global memory
for (int stride = blockDim.x / 2; stride > 32; stride >>= 1) {
    if (tid < stride) {
        smem[tid] += smem[tid + stride];
    }
    // synchronize within threadblock
    __syncthreads();
}

// unrolling warp
if (tid < 32)
{
    volatile int *vsmem = smem;
    vsmem[tid] += vsmem[tid + 32];
    vsmem[tid] += vsmem[tid + 16];
    vsmem[tid] += vsmem[tid +  8];
    vsmem[tid] += vsmem[tid +  4];
    vsmem[tid] += vsmem[tid +  2];
    vsmem[tid] += vsmem[tid +  1];
}
```

Volatile qualifier: tells the compiler that it must store vsmem[tid] back to global memory with every assignment

Parallel Reduction Kernel Performance

Kernel	Time (ms)	Step Speedup	Cumulative Speedup	Load Efficiency	Store Efficiency
reduceNeighboredGmem_1	4.1238			25.02%	25%
reduceNeighboredSmem_2	3.4246	1.20	1.20	100%	12.5%
reduceNeighboredSmemNoDivergence_3	2.5892	1.32	1.59	100%	12.5%
reduceInterleavedSmem_4	1.8775	1.38	2.20	100%	12.5%
reduceUnrolling2_5	0.9805	1.91	4.34	100%	98.65%
reduceUnrollingWarp8_6	0.3447	2.84	11.96	100%	98.65%

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

Parallel Reduction #7: Complete Unrolling

Replace iteration loop:

```
// in-place reduction in global memory
for (int stride = blockDim.x / 2; stride > 32; stride >>= 1) {
    if (tid < stride) {
        smem[tid] += smem[tid + stride];
    }
    // synchronize within threadblock
    __syncthreads();
}
```

With complete unrolling manually:

```
// in-place reduction in global memory
if (blockDim.x >= 1024 && tid < 512) smem[tid] += smem[tid + 512];
__syncthreads();

if (blockDim.x >= 512 && tid < 256) smem[tid] += smem[tid + 256];
__syncthreads();

if (blockDim.x >= 256 && tid < 128) smem[tid] += smem[tid + 128];
__syncthreads();

if (blockDim.x >= 128 && tid < 64) smem[tid] += smem[tid + 64];
__syncthreads();
```

Parallel Reduction Kernel Performance

Kernel	Time (ms)	Step Speedup	Cumulative Speedup	Load Efficiency	Store Efficiency
reduceNeighboredGmem_1	4.1238			25.02%	25%
reduceNeighboredSmem_2	3.4246	1.20	1.20	100%	12.5%
reduceNeighboredSmemNoDivergence_3	2.5892	1.32	1.59	100%	12.5%
reduceInterleavedSmem_4	1.8775	1.38	2.20	100%	12.5%
reduceUnrolling2_5	0.9805	1.91	4.34	100%	98.65%
reduceUnrollingWarp8_6	0.3447	2.84	11.96	100%	98.65%
reduceCompleteUnrolling8_7	0.3382	1.02	12.19	100%	98.65%

Parallel Reduction #8: Complete Unrolling with Template Functions

- Using template functions can help to further reduce branch overhead
- Specify block size as a function template parameter

```
Template <unsigned int iBlockSize>
__global__ void reductionKernel(int *g_idata, int *g_odata, unsigned int n)
```

Parallel Reduction #8

Replace blockDim.x:

```
// in-place reduction in global memory
if (blockDim.x >= 1024 && tid < 512) smem[tid] += smem[tid + 512];
__syncthreads();

if (blockDim.x >= 512 && tid < 256) smem[tid] += smem[tid + 256];
__syncthreads();

if (blockDim.x >= 256 && tid < 128) smem[tid] += smem[tid + 128];
__syncthreads();

if (blockDim.x >= 128 && tid < 64) smem[tid] += smem[tid + 64];
__syncthreads();
```

With iBlockSize to be evaluated at compile time:

```
// in-place reduction and complete unroll
if (iBlockSize >= 1024 && tid < 512) smem[tid] += smem[tid + 512];
__syncthreads();

if (iBlockSize >= 512 && tid < 256) smem[tid] += smem[tid + 256];
__syncthreads();

if (iBlockSize >= 256 && tid < 128) smem[tid] += smem[tid + 128];
__syncthreads();

if (iBlockSize >= 128 && tid < 64) smem[tid] += smem[tid + 64];
__syncthreads();
```

Parallel Reduction #8

The kernel must be called with the switch-case structure:

```
switch (blocksize) {
  case 1024:
    reduceCompleteUnrolling8Template_8<1024><<<grid.x/8, block>>>(d_idata, d_odata, size);
    break;
  case 512:
    reduceCompleteUnrolling8Template_8<512><<<grid.x/8, block>>>(d_idata, d_odata, size);
    break;
  case 256:
    reduceCompleteUnrolling8Template_8<256><<<grid.x/8, block>>>(d_idata, d_odata, size);
    break;
  case 128:
    reduceCompleteUnrolling8Template_8<128><<<grid.x/8, block>>>(d_idata, d_odata, size);
    break;
  case 64:
    reduceCompleteUnrolling8Template_8<64><<<grid.x/8, block>>>(d_idata, d_odata, size);
    break;
}
```

Parallel Reduction Kernel Performance

Kernel	Time (ms)	Step Speedup	Cumulative Speedup	Load Efficiency	Store Efficiency
reduceNeighboredGmem_1	4.1238			25.02%	25%
reduceNeighboredSmem_2	3.4246	1.20	1.20	100%	12.5%
reduceNeighboredSmemNoDivergence_3	2.5892	1.32	1.59	100%	12.5%
reduceInterleavedSmem_4	1.8775	1.38	2.20	100%	12.5%
reduceUnrolling2_5	0.9805	1.91	4.34	100%	98.65%
reduceUnrollingWarp8_6	0.3447	2.84	11.96	100%	98.65%
reduceCompleteUnrolling8_7	0.3382	1.02	12.19	100%	98.65%
reduceCompleteUnrolling8Template_8	0.3334	1.01	12.37	100%	98.65%

Results

```
[[jin6@node0263 Reduction]$ nvcc reduction.cu
[[jin6@node0263 Reduction]$ nvprof ./a.out
==34113== NVPROF is profiling process 34113, command: ./a.out
./a.out starting reduction at device 0: Tesla K20m     with array size 16777216  grid 32768 block 512
cpu_sum time: 0.04750 sec  cpu_sum: 2139353471
reduceNeighboredGmem_1 time: 0.00734 sec  gpu_sum: 2139353471 <<<grid 32768 block 512>>>
reduceNeighboredSmem_2 time: 0.00488 sec  gpu_sum: 2139353471 <<<grid 32768 block 512>>>
reduceNeighboredSmemNoDivergence_3 time: 0.00395 sec  gpu_sum: 2139353471 <<<grid 32768 block 512>>>
reduceInterleavedSmem_4 time: 0.00277 sec  gpu_sum: 2139353471 <<<grid 32768 block 512>>>
reduceUnrolling2_5 time: 0.00155 sec  gpu_sum: 2139353471 <<<grid 16384 block 512>>>
reduceUnrollingWarp8_6 time: 0.00062 sec  gpu_sum: 2139353471 <<<grid 4096 block 512>>>
reduceCompleteUnrolling8_7 time: 0.00060 sec  gpu_sum: 2139353471 <<<grid 4096 block 512>>>
reduceCompleteUnrolling8Template_8 time: 0.00060 sec  gpu_sum: 2139353471 <<<grid 4096 block 512>>>
==34113== Profiling application: ./a.out
==34113== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   88.52%  167.92ms         8  20.990ms  18.013ms  41.180ms  [CUDA memcpy HtoD]
                    3.74%  7.0948ms         1  7.0948ms  7.0948ms  7.0948ms  reduceNeighboredGmem_1(int*, int*, unsigned int)
                    2.54%  4.8262ms         1  4.8262ms  4.8262ms  4.8262ms  reduceNeighboredSmem_2(int*, int*, unsigned int)
                    2.05%  3.8882ms         1  3.8882ms  3.8882ms  3.8882ms  reduceNeighboredSmemNoDivergence_3(int*, int*, unsigned int)
                    1.43%  2.7086ms         1  2.7086ms  2.7086ms  2.7086ms  reduceInterleavedSmem_4(int*, int*, unsigned int)
                    0.78%  1.4807ms         1  1.4807ms  1.4807ms  1.4807ms  reduceUnrolling2_5(int*, int*, unsigned int)
                    0.29%  542.76us         1  542.76us  542.76us  542.76us  reduceUnrollingWarp8_6(int*, int*, unsigned int)
                    0.28%  537.06us         1  537.06us  537.06us  537.06us  reduceCompleteUnrolling8_7(int*, int*, unsigned int)
                    0.28%  534.85us         1  534.85us  534.85us  534.85us  void reduceCompleteUnrolling8Template_8<unsigned int=512>(int*, int*, unsigned int)
                    0.09%  161.38us         8  20.172us  19.872us  20.544us  [CUDA memcpy DtoH]
      API calls:   57.08%  341.79ms         2  170.90ms  191.34us  341.60ms  cudaMalloc
                   28.47%  170.48ms        16  10.655ms  86.436us  41.823ms  cudaMemcpy
                    8.96%  53.649ms         1  53.649ms  53.649ms  53.649ms  cudaDeviceReset
                    3.82%  22.902ms        16  1.4314ms  157.39us  7.1067ms  cudaDeviceSynchronize
                    1.20%  7.1856ms         2  3.5928ms  356.30us  6.8293ms  cudaFree
                    0.19%  1.1488ms       188  6.1100us     148ns  236.30us  cuDeviceGetAttribute
                    0.10%  625.16us         8  78.145us  48.510us  221.24us  cudaLaunch
                    0.09%  568.18us         1  568.18us  568.18us  568.18us  cudaGetDeviceProperties
                    0.04%  227.62us         2  113.81us  108.35us  119.28us  cuDeviceTotalMem
                    0.03%  178.36us         2  89.182us  66.189us  112.18us  cuDeviceGetName
                    0.00%  12.695us        24     528ns     134ns  1.3040us  cudaSetupArgument
                    0.00%  11.088us         8  1.3860us  1.0350us  3.0590us  cudaConfigureCall
                    0.00%  11.063us         1  11.063us  11.063us  11.063us  cudaSetDevice
                    0.00%  3.9190us         3  1.3060us     185ns  2.0930us  cuDeviceGetCount
                    0.00%  2.0400us         4     510ns     233ns     766ns  cuDeviceGet
```

Performance Comparison