CSC367 Parallel computing

Lecture 20: Parallel Reduction with CUDA

Thanks to Mark Harris from NVIDIA for this parallel reduction example, also thanks to Andreas Moshovos for some of the images.

Warm up!

- The below are used frequently in CUDA code!

```
int lane = threadIdx.x % warpSize;  //Thread ID inside a warp, OR
int lane = threadIdx.x & (warpSize-1);  //Thread ID inside a warp
int warp_id = threadIdx.x / warpSize;  //Warp ID inside a thread block
```

- To dynamically allocate shared memory use the below
  - From the host pass the shared memory size as an argument:

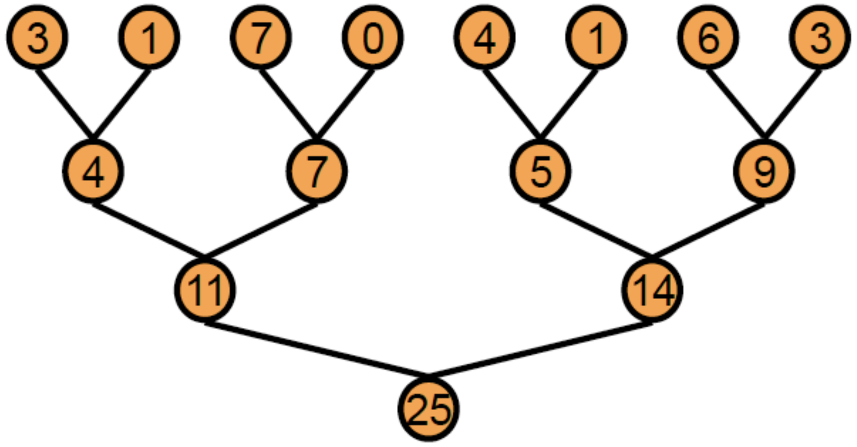
```
reduce1<<< dimGrid, dimBlock, shMemSize >>>(d_idata, d_odata);
```

  - From the device function use the unsized extern array syntax:

```
extern __shared__ int sdata[];
```

Parallel Reduce in CUDA

- A tree-based approach can be used to reduce in each thread block



- To process large arrays:
  - We need multiple thread blocks
  - Each thread block reduces a portion of the array
  - But how are the partial results communicated between thread blocks?

How to synchronize globally?

- If we could synchronize across thread blocks then we could use a global sync after each block produces its results and when all blocks reach sync just recursively continue.

How to synchronize globally?

- If we could synchronize across thread blocks then we could use a global sync after each block produces its results and when all blocks reach sync just recursively continue.

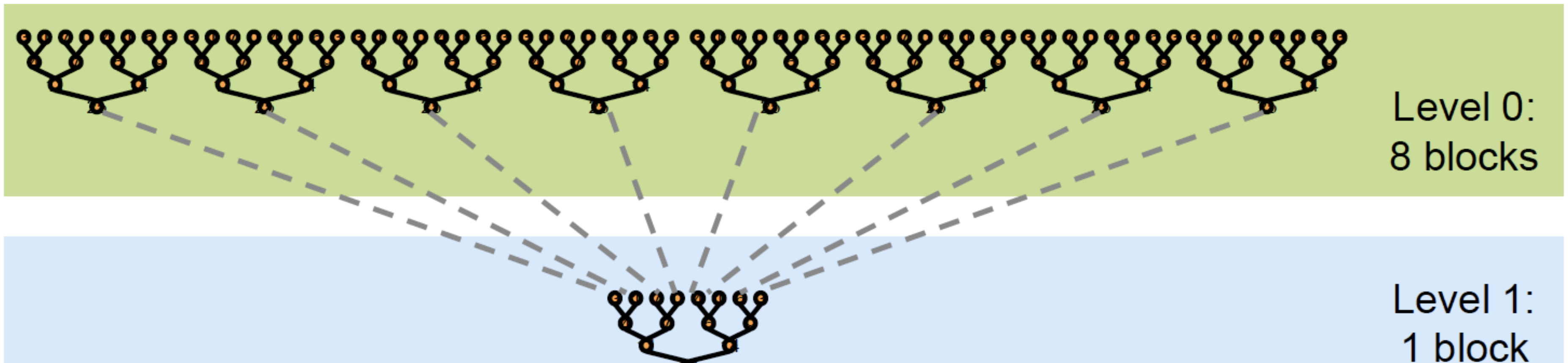
NOT POSSIBLE!!

- CUDA does not have global synchronization!
- Solution: decompose into multiple kernels where each kernel launch serves as a global synchronization.

Solution: Kernel Decomposition

- Decompose the computation into multiple kernel (level) invocations.
- We will see different implementations in the upcoming slides that strive to reach GPU peak performance.
  - Reductions are memory bound (only one operation per element loaded): strive for peak bandwidth!

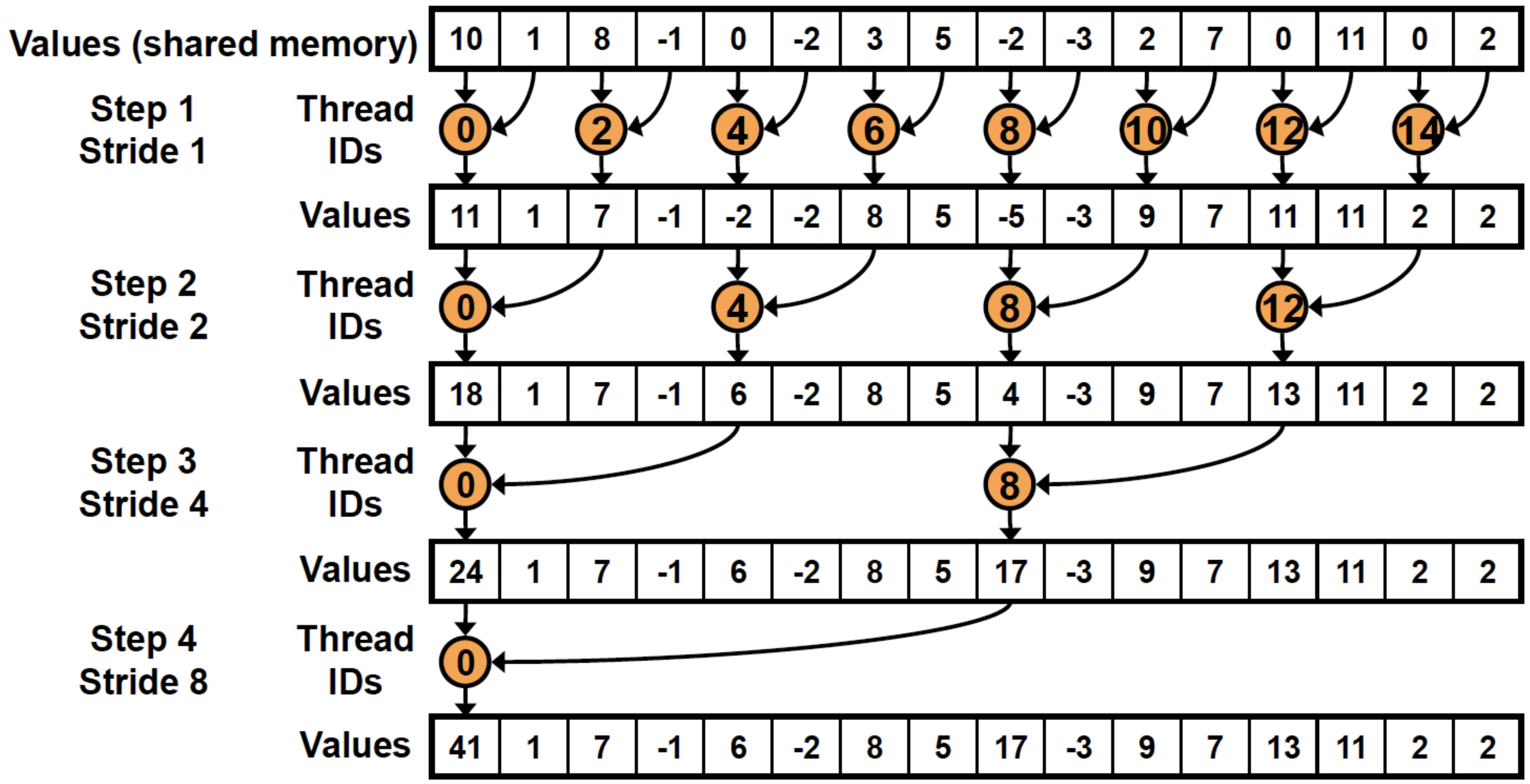
For the code see: /u/csc367h/winter/pub/labs/examples/GPUExamples.tgz



Implementation 1: Interleave Addressing

- Load data: Each thread loads one element from global memory to shared memory
- For reduce:
  - A thread reduces two elements: thread1 adds first 2 elements, thread2 address the next two and so on.
  - Half of the threads are deactivated (not used any more) after each step!
  - When only one thread left: terminate
- Write back to global memory

Implementation 1: Interleave Addressing



Implementation 1: Interleave Addressing

```
__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];
}
```

Implementation 1: Interleave Addressing

```
__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];
}
```

Problem: Divergent warps, also % is an expensive operation!

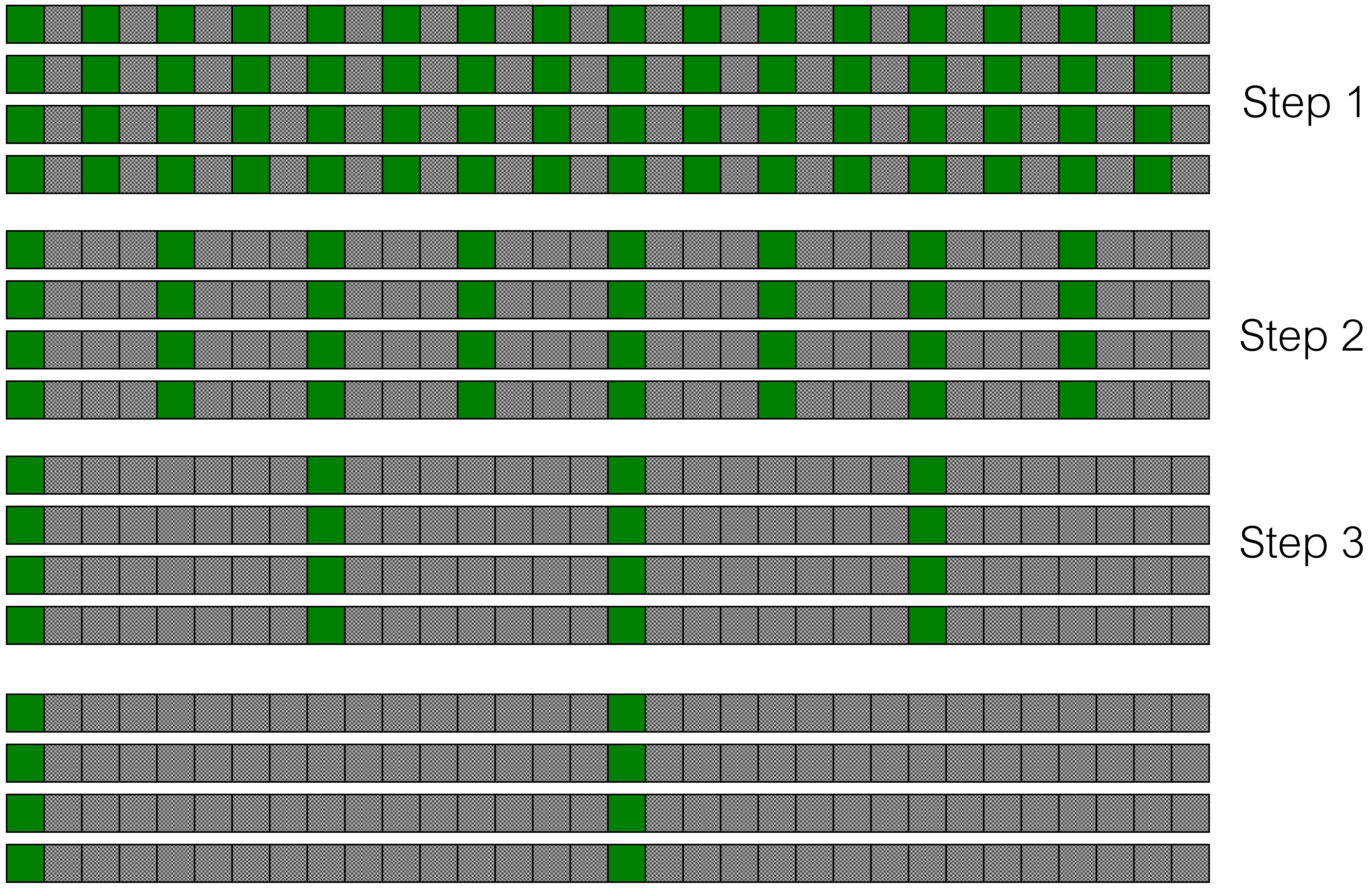
Performance for 4M element reduction

	Time	Bandwidth
Kernel 1: **interleaved addressing with divergent branching**	**8.054 ms**	**2.083 GB/s**

Note: Block size is 128 threads here

The times are reported on NVIDIA G80

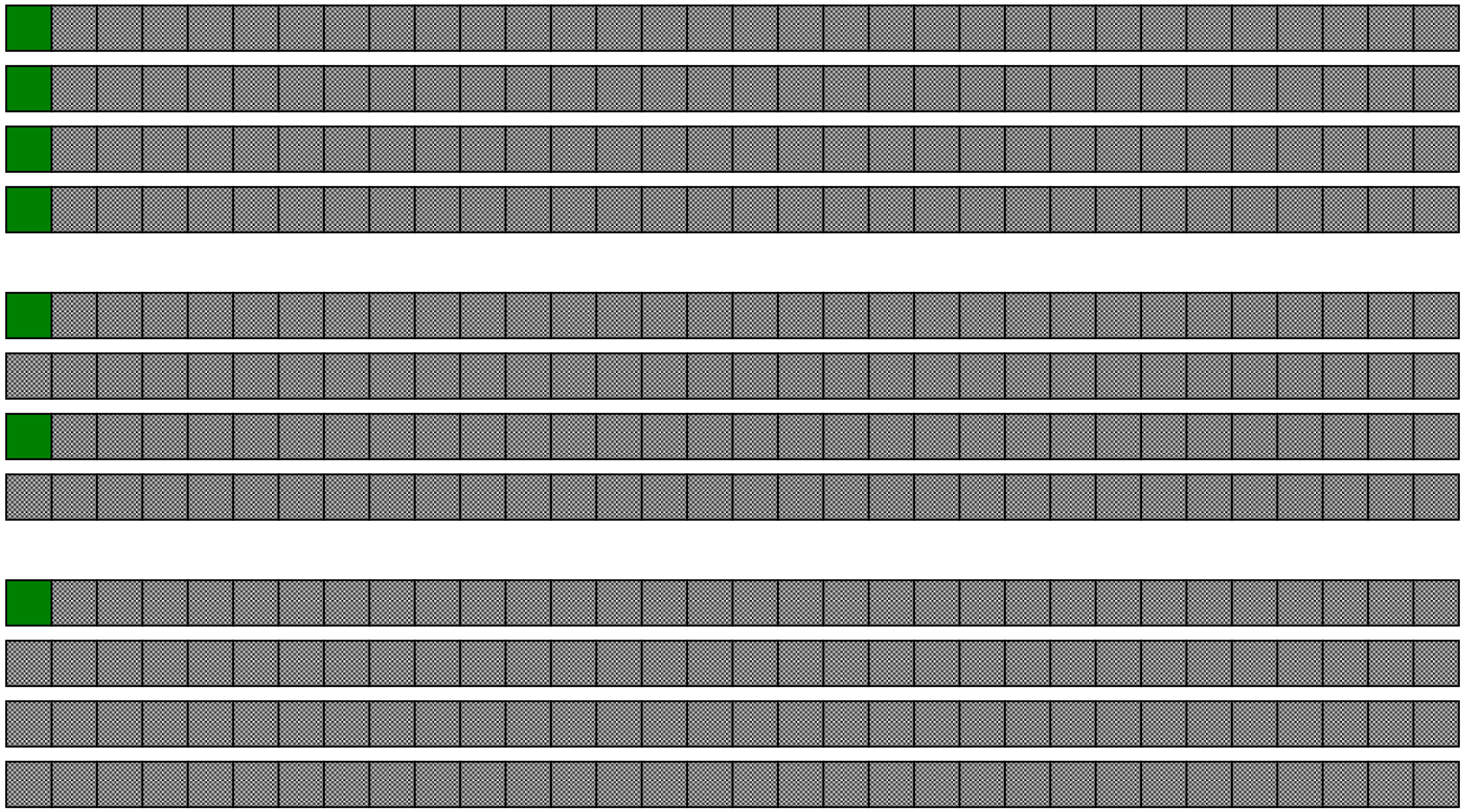
Divergent branches? See inside a block



4 warps/step: Remember that we had 128 threads per block

Slide source: Moshovos

Divergent branches? See inside a block



Slide source: Moshovos

Implementation 2: Interleave Addressing: no divergence

- Replace the divergent branching code:

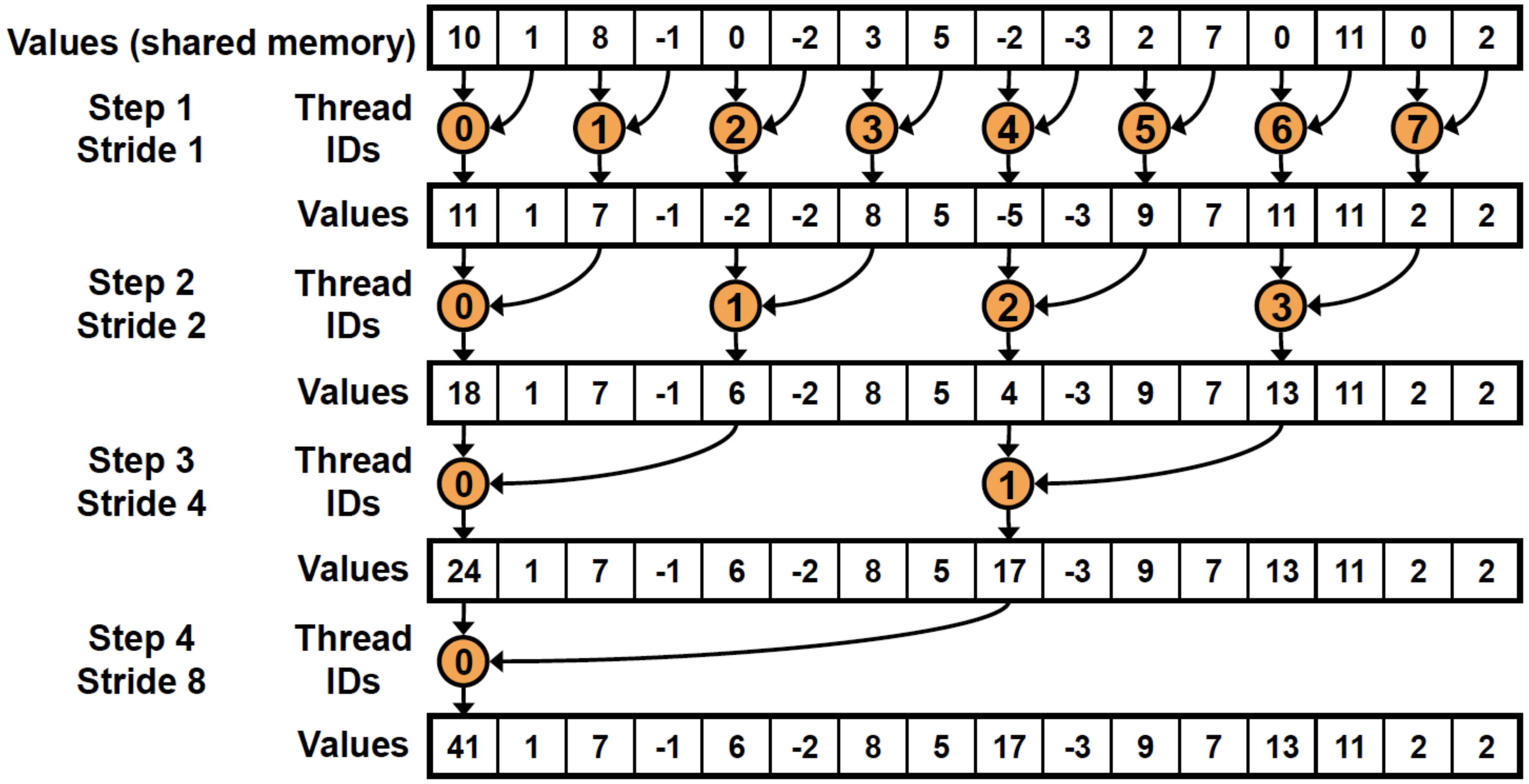
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

- With strided index and non-divergent branch

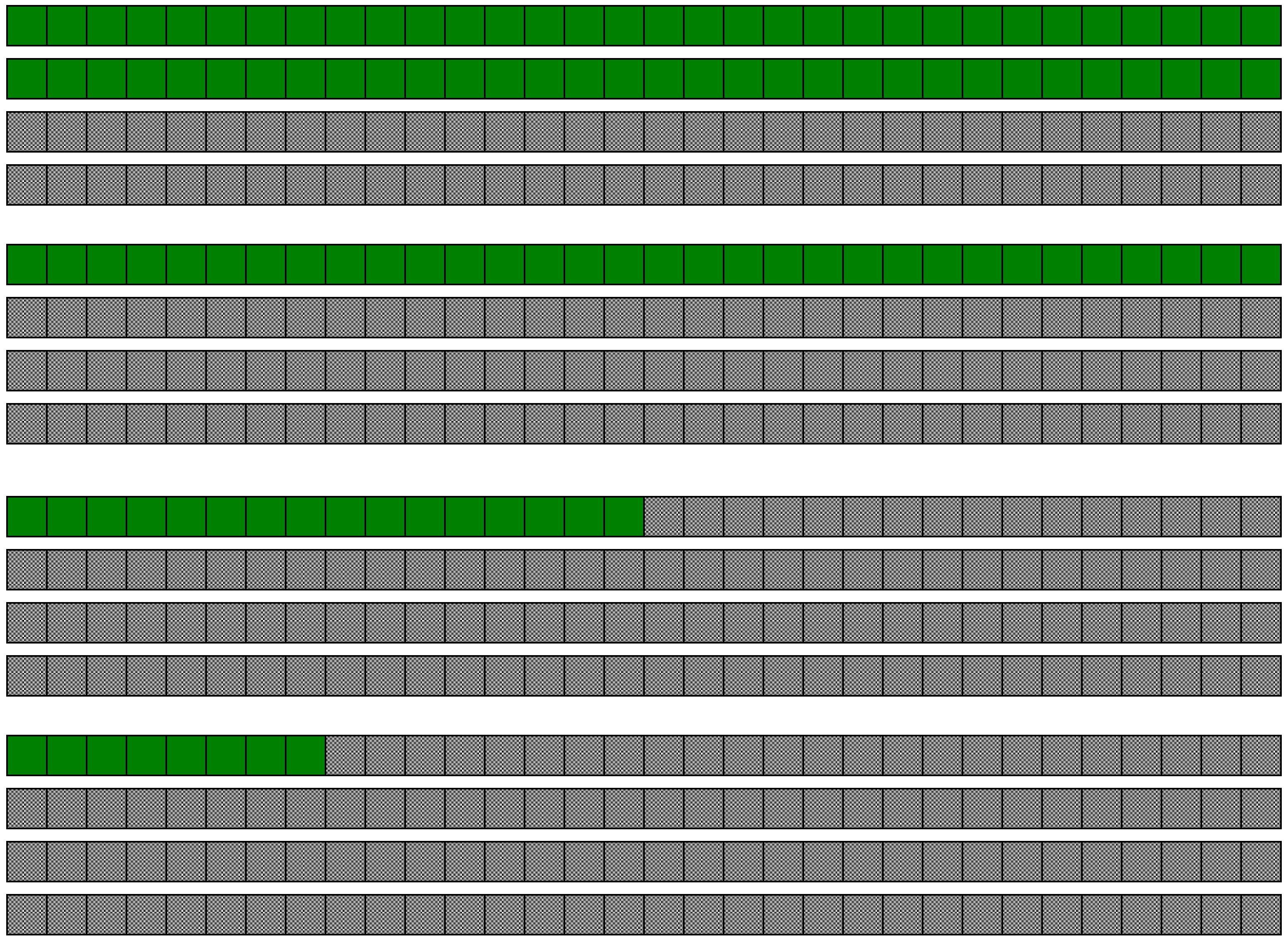
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

Implementation 2: Interleave Addressing: no divergence



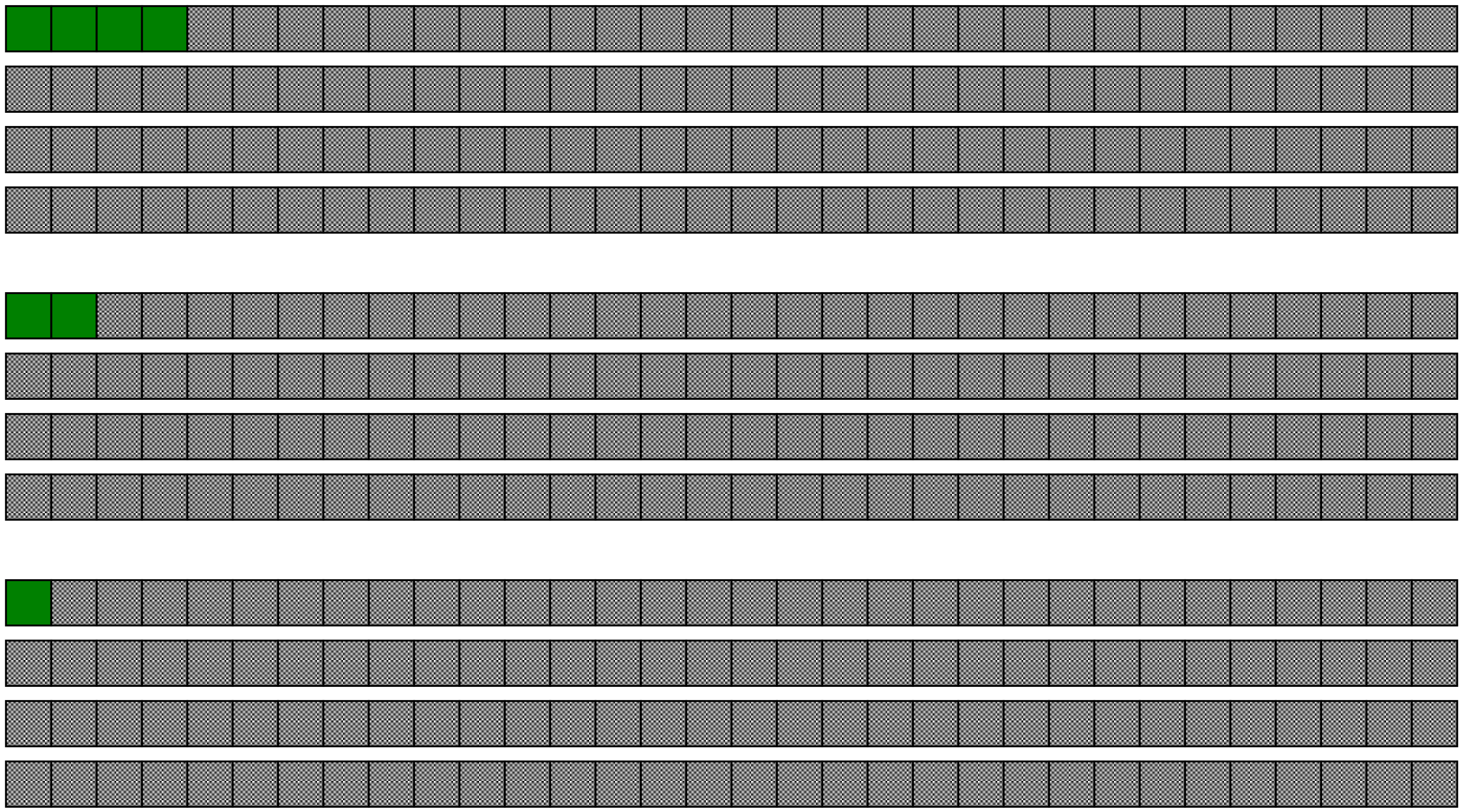
Implementation 2: Warp flow



4 warps/step

Slide source: Moshovos

Implementation 2: Warp flow

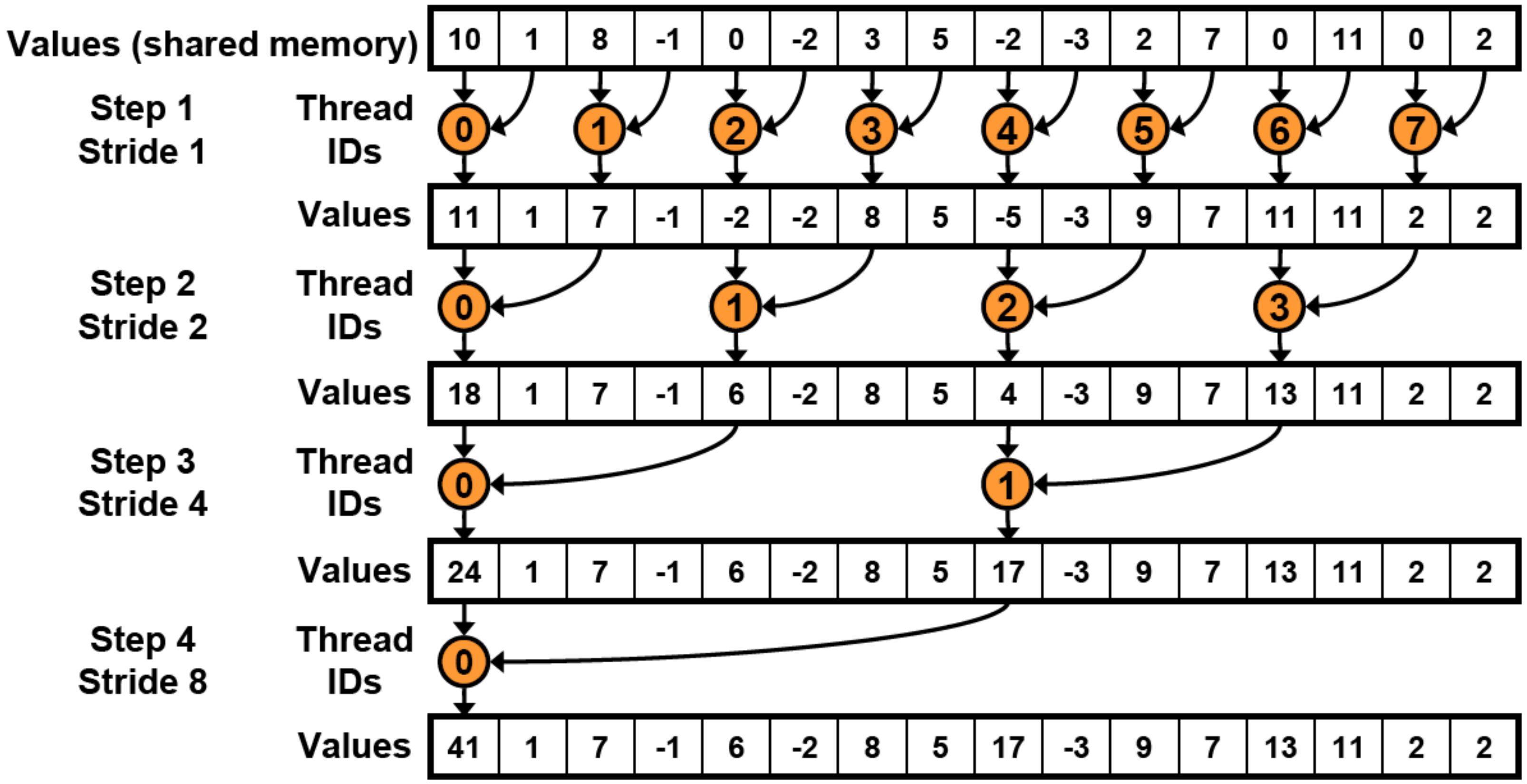


Slide source: Moshovos

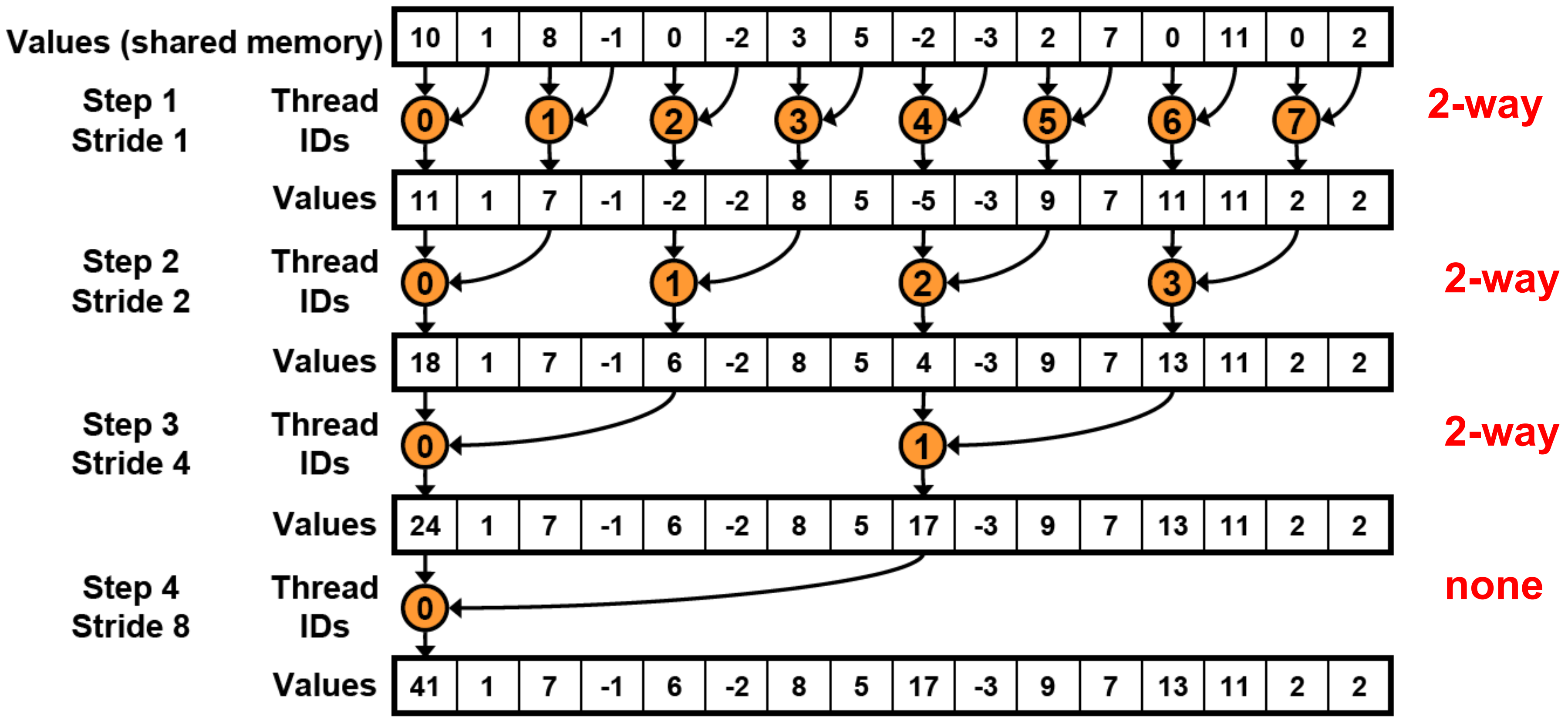
Performance for 4M element reduction

	Time	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**

Problem with implementation 2?

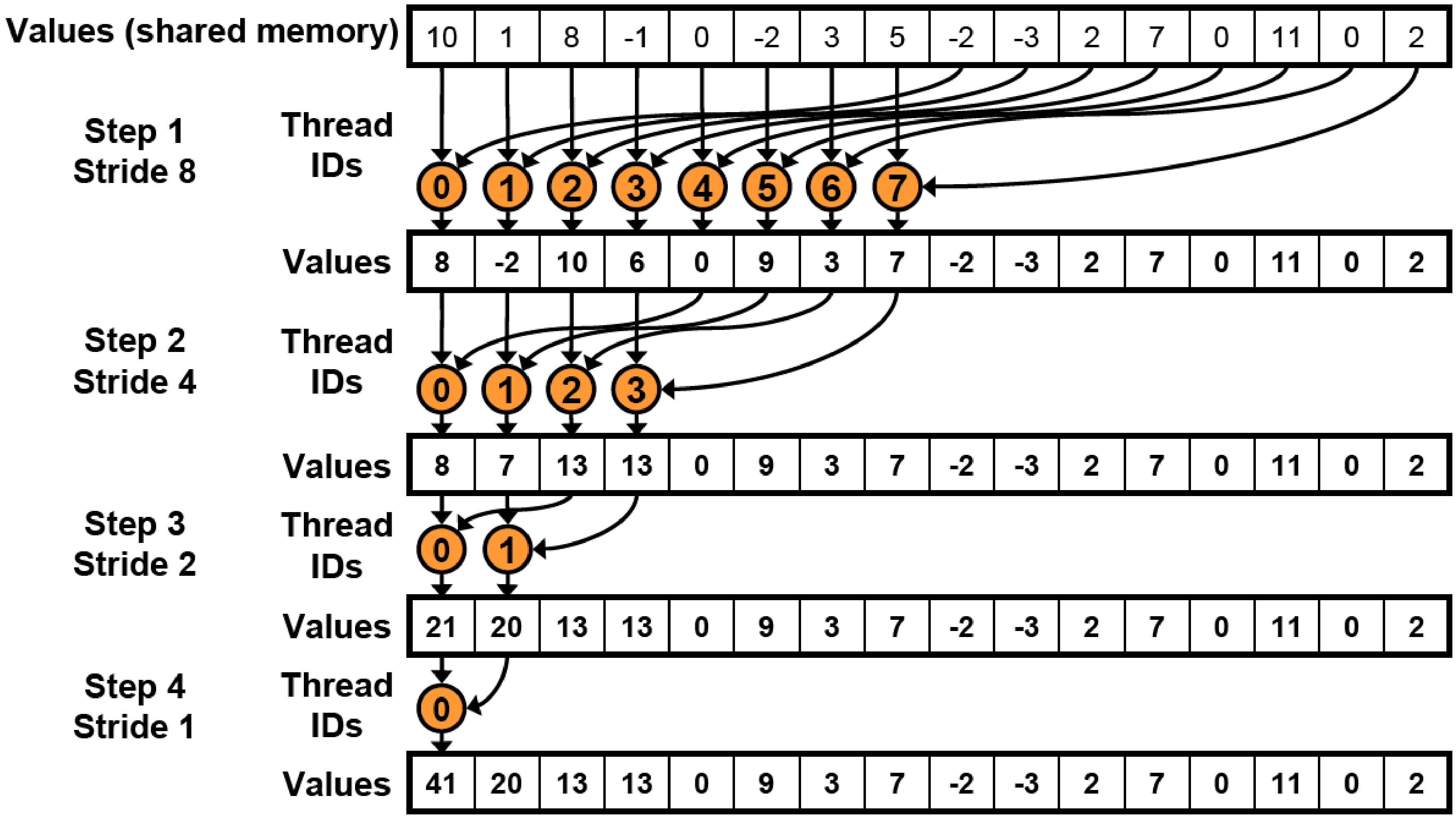


Problem with implementation 2: Bank conflicts



- 2-way bank conflicts at every step
- Remember that there are more than 16 threads!
- So expand the figure to 128 threads to realize why we have bank conflicts.

Implementation 3: Sequential accesses



- Eliminates bank conflicts

Implementation 3: Interleave Addressing: no divergence

- Replace the strided indexing 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();
}
```

- With reserved loop and threadID-based indexing

```
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

	Time	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

Implementation 3 problem: Wasteful!

- All threads read an element
- Half of the threads are idle in the first step, another half become idle in the next step
- This is wasteful

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

Implementation 4: Read two elements and add during load

- Original: Each thread reads one element

```
// 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();
```

- Instead, half the number of blocks: each thread reads two elements and adds during load

```
// 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();
```

Performance for 4M element reduction

	Time	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

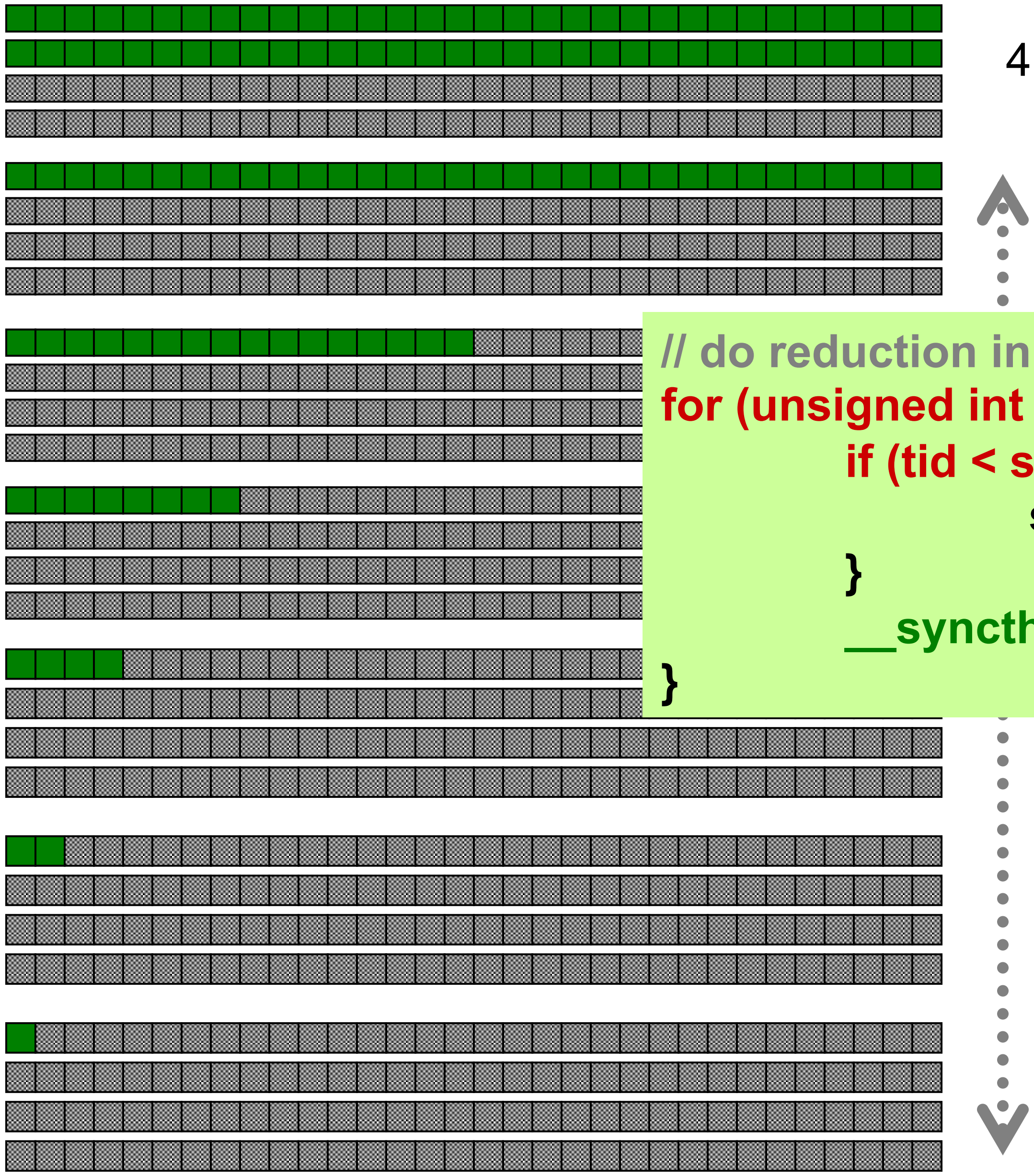
Implementation 4 problem: Instruction bottleneck!

- Memory bandwidth is still underutilized
  - We know that reductions have low arithmetic density (are memory bound).
- What is the potential bottleneck?
  - Ancillary instructions that are not loads, stores, or arithmetic for the core computation.
  - Address arithmetic and loop overhead
- Unroll loops to eliminate these “extra” instructions

Unroll the Last Warp

- At every step the number of active threads halves
  - When s <=32 there is only one warp left
- Instructions are synchronous within a warp
  - They all happen in lock step
  - No need to use __syncthreads()
  - We don’t need “if (tid < s)” since it does not save any work
    - All threads in a warp will “see” all instructions whether they execute them or not
- Unroll the last 6 iterations of the inner loop
  - s <= 32

Warp control flow of Implementation 2



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

Slide source: Moshovos

Implementation 5: Unroll the Last Warp

```
__device__ void warpReduce(volatile int* sdata, int tid) {
      sdata[tid] += sdata[tid + 32];
      sdata[tid] += sdata[tid + 16];
      sdata[tid] += sdata[tid +  8];
      sdata[tid] += sdata[tid +  4];
      sdata[tid] += sdata[tid +  2];
      sdata[tid] += sdata[tid +  1];
 }
```

IMPORTANT:
For this to be correct, we must use the "volatile" keyword!

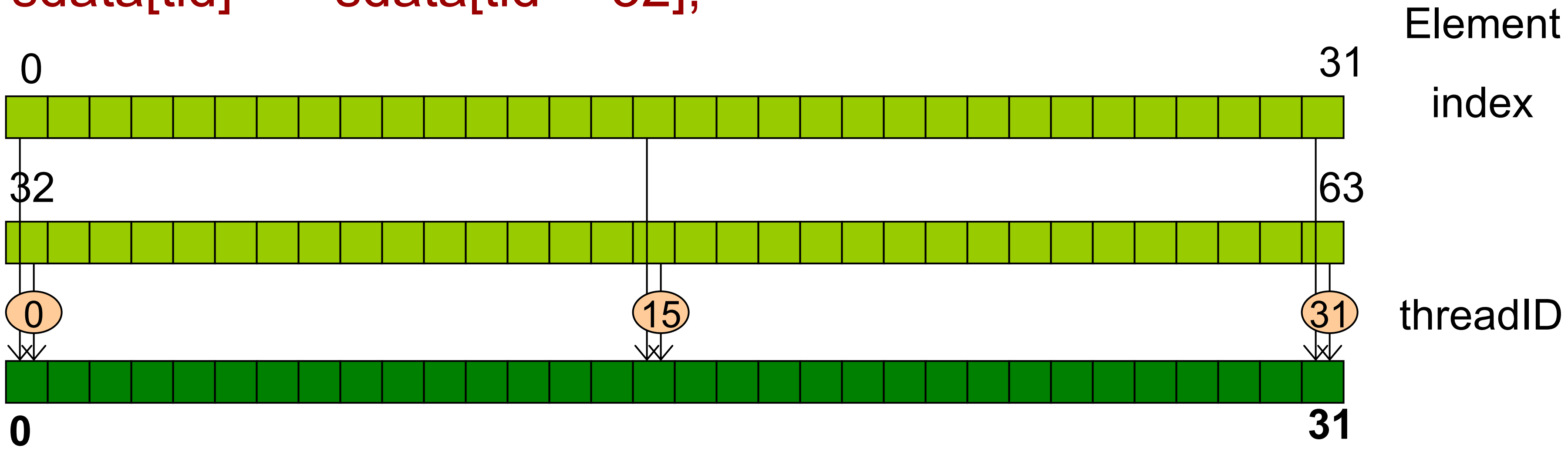
```
// later…
 for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
      if (tid < s)
         sdata[tid] += sdata[tid + s];
      __syncthreads();
 }

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

- This saves work in all warps not just the last one
  - Without unrolling all warps execute the for loop and if statement

Unroll the Last Warp: Inside the warp

- sdata[tid] += sdata[tid + 32];

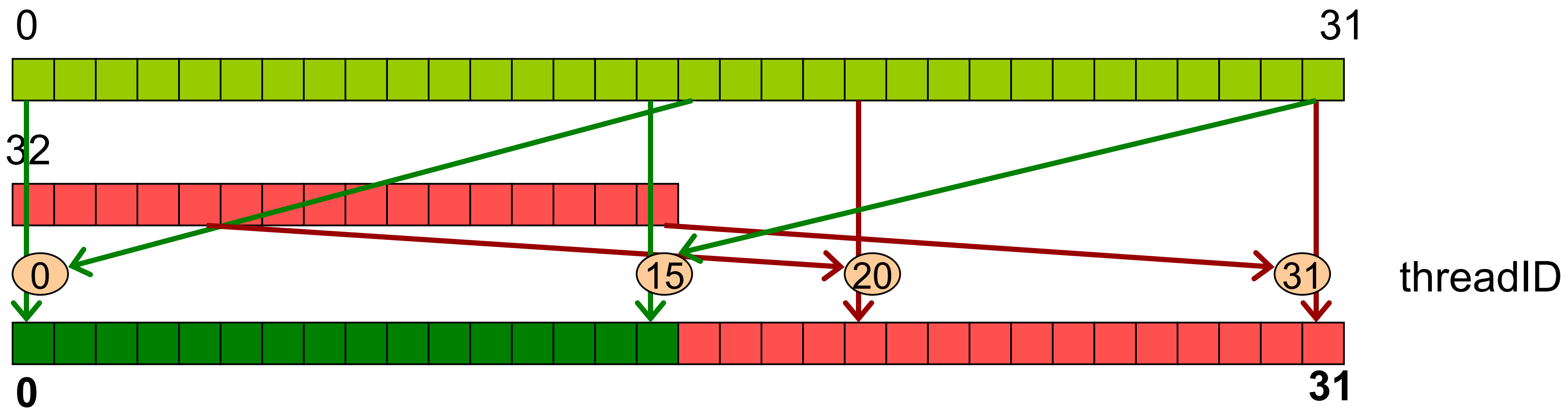


- All threads doing useful work

Slide source: Moshovos

Unroll the Last Warp: Inside the warp

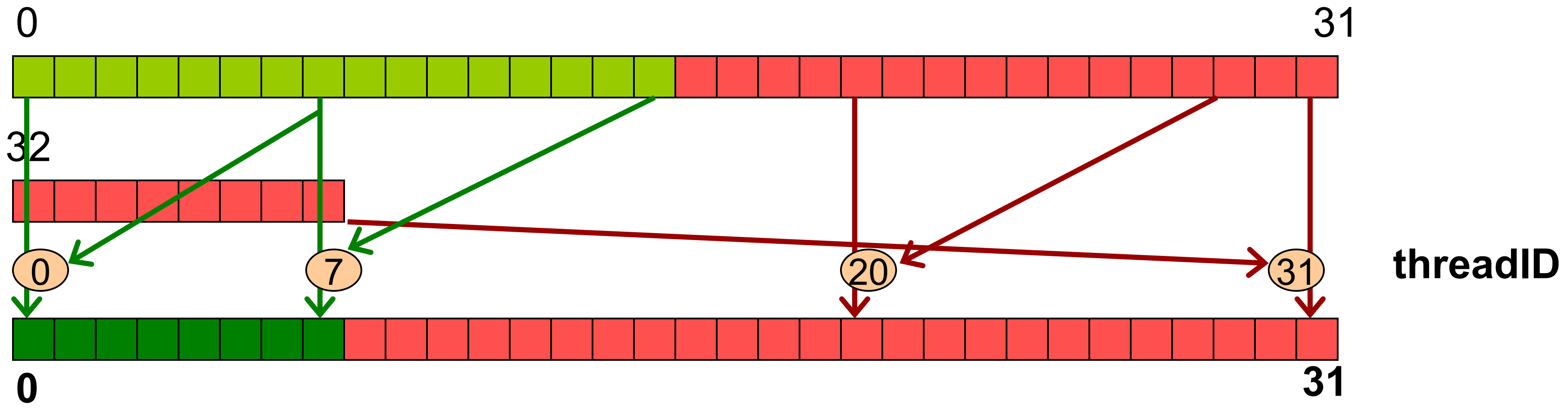
- sdata[tid] += sdata[tid + 16];

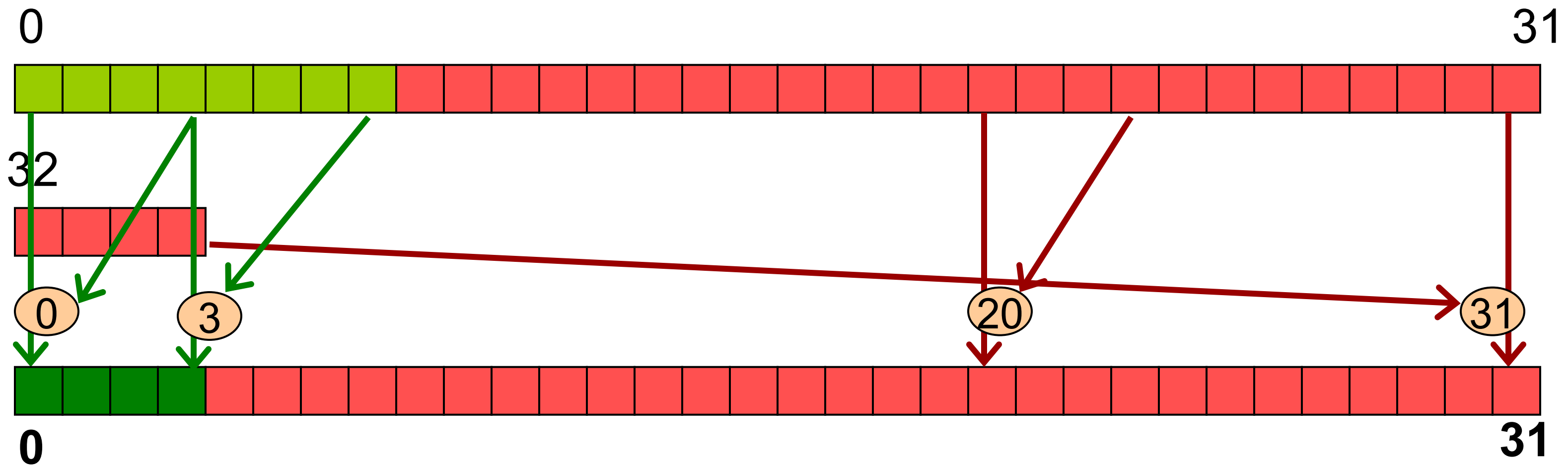


  – Half of the threads, 16-31, are doing useless work, but that's fine!

Slide source: Moshovos

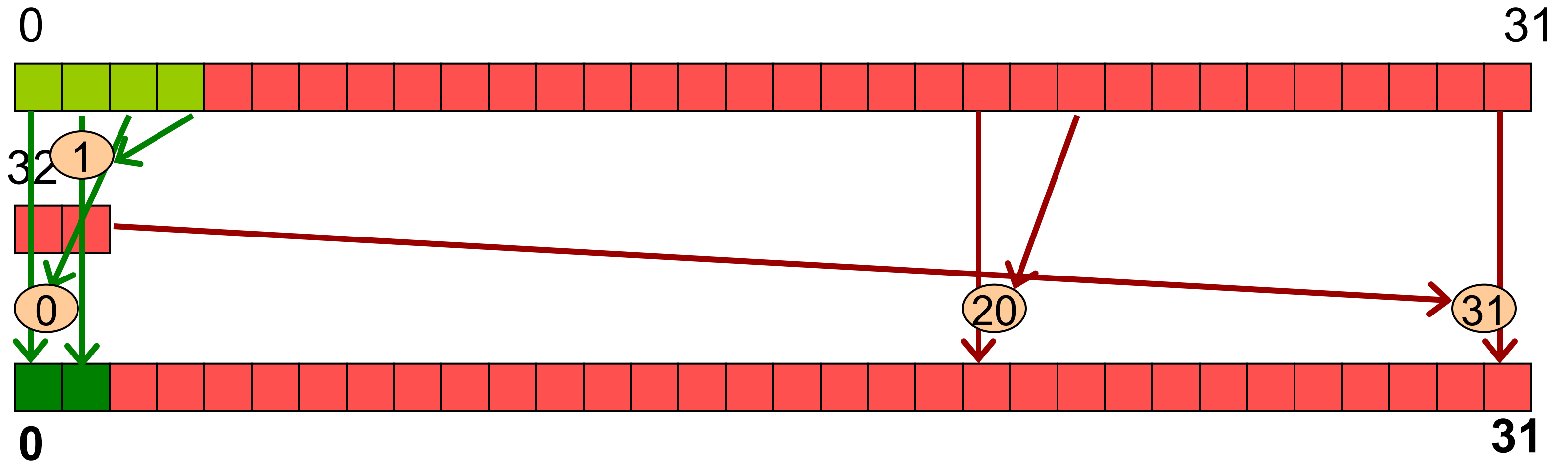
Unroll the Last Warp: Inside the warp

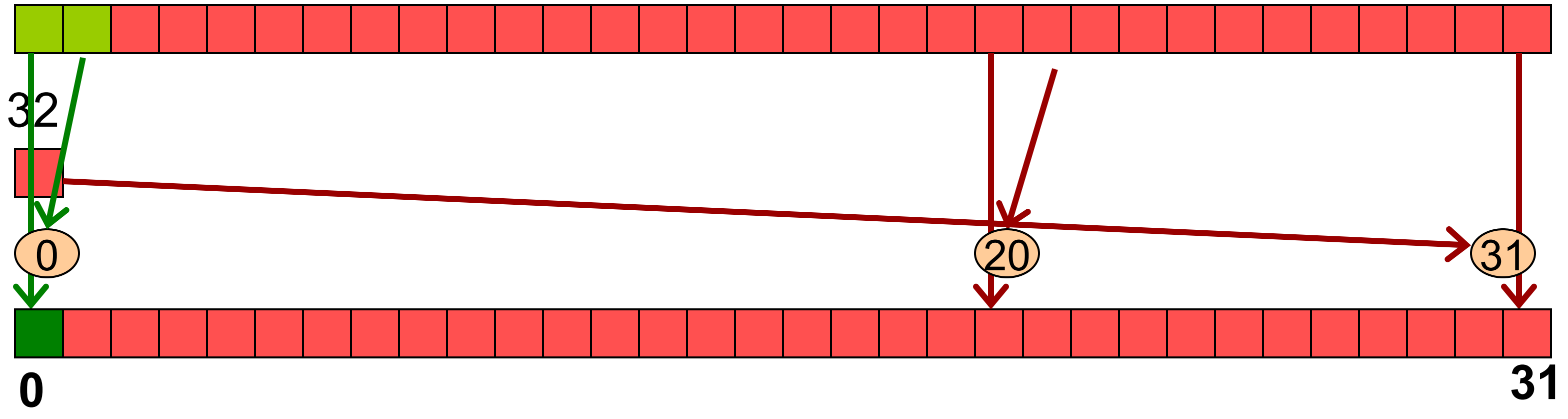




Slide source: Moshovos

Unroll the Last Warp: Inside the warp





Slide source: Moshovos

What is volatile

- The volatile keyword enforces warp synchronous execution!
- It is used to prevent the compiler from optimizing operations in shared memory, i.e. it wont allow registers to be used for the operation on sdata in the previous code.
- If registers were used, then the unrolled loop in the previous slide would not execute correctly.
- NVIDIA was not a big fan of developers using volatile so it introduced warp shuffle instructions (implementations 8 to 10 in your example code)! Warp shuffle only works on Kerpler onwards GPUs!

Performance for 4M element reduction

	Time	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

Implementation 6: Complete Unrolling

- If we knew the number of iterations at compile time, we could completely unroll the reduction
  - Lets assume block size is limited to 512
  - We can restrict our attention to powers-of-two block sizes
- We can easily unroll for a fixed block size
  - But we need to be generic
  - How can we unroll for block sizes we don’t know at compile time?
- C++ Templates
  - CUDA supports C++ templates on device and host functions

Unrolling with Templates

Specify block size as a function template parameter:

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

Implementation 6: Complete Unrolling

```
Template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
        if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
        if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
        if (blockSize >= 16) sdata[tid] += sdata[tid +  8];
        if (blockSize >=  8) sdata[tid] += sdata[tid +  4];
        if (blockSize >=  4) sdata[tid] += sdata[tid +  2];
        if (blockSize >=  2) sdata[tid] += sdata[tid +  1];
 }
```

```
    if (blockSize >= 512) {
        if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) {
        if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) {
        if (tid <  64)  { sdata[tid] += sdata[tid +   64]; } __syncthreads(); }

    if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

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

Results in a very efficient inner loop

Invoking Kernel Templates

- Don’t we need the block size at compile time:
- Nope! There are “only” 10 possibilities:

```
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

	Time	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

Implementation 7: Multiply Add / 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();
```

gridSize steps to achieve coalescing

How to determine gridDim in Implementation 7

- gridDim which is the number of blocks launched per kernel call determines the number of elements that each thread reduces in the while-loop.

- The best value for gridDim in implementation 7 is determined by mixing concepts from Brent's theorem (not discussed in this class) and tuning based on the GPU type.

- In the example code provided to you, see what we set maxBlocks to, this tuned!

Performance for 4M element reduction

	Time	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>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
   if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
   if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
   if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
   if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
   if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
   if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
}
template <unsigned int blockSize>
__global__ void 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;

   while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
   __syncthreads();

   if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
   if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
   if (blockSize >= 128) { if (tid <   64) { sdata[tid] += sdata[tid +   64]; } __syncthreads(); }

   if (tid < 32) warpReduce(sdata, tid);
   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Max Reductions Example Code

- We are providing you with all of the above implementations at: /u/csc367h/winter/pub/labs/examples/GPUExamples.tgz
- Have to read the entire code before the lab: TAKES TIME!
- *should_repeat* is a function that allows for multiple invocations of the reduction kernel until all the array is reduced to just one element; it is used in the code!
- Cases 1-7 implement everything that we discussed up to now for reduction.
- Case 11 also implements implementation 7 but it is for computing a max.

Implementations 8-10

- We also prove 3 additional implementations which only work on GPUs that support warp shuffle.
- Recall that NIVIDA introduced warp shuffle so developers don’t have to use volatile.
- You have to read the below document before the lab to understand those implementations:
- https://devblogs.nvidia.com/faster-parallel-reductions-kepler/
- Note that the shuffle instructions changed a bit after the Volta architecture (they use _sync at the end) but this does not apply to the lab GPUs!

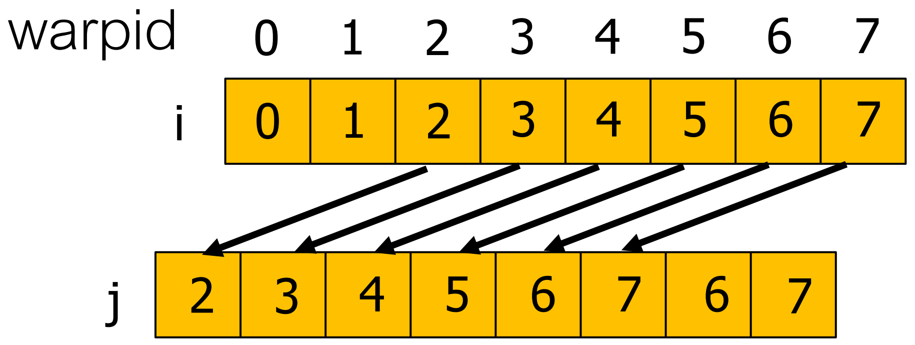
Warp Shuffle

- Shuffle allows a thread to directly read a register from another thread in the same warp.
- Shuffle can be used to replace a shared memory sequence with a single instruction.
- Shuffle does not use any shared memory so no need to use volatile anymore.
- Synchronization is within a warp and is implicit, so no need to synchronize the whole thread block with __syncthreads().

Shuffle Down

- __shfl_down() calculates a source lane ID by adding delta to the caller's lane ID. Recall that lane is the ID of the thread inside the warp.
- The value of var held by the resulting lane ID is returned: thus var is shifted down the warp by delta lanes.
- If the source lane ID is out of range, the calling thread's own var is returned.

```
int __shfl_down(int var, unsigned int delta, int width=warpSize);
```



```
int i = threadIdx.x % 32;
int j = __shfl_down(i, 2, 8);
```

Implementations 8: Use Warp Shuffle

```
__inline__ __device__ int warpReduceSum(int val)
{
    for (int offset = warpSize/2; offset > 0; offset /= 2)
        val += __shfl_down(val, offset);
    return val;
}
```

```
__inline__ __device__ int blockReduceSum(int val)
{
 ….
    val = warpReduceSum(val);      // Each warp performs partial reduction
    if (lane==0) shared[wid]=val; // Write reduced value to shared memory
    __syncthreads();              // Wait for all partial reductions

    //read from shared memory only if that warp existed
    val = (threadIdx.x < blockDim.x / warpSize) ? shared[lane] : 0;
    if (wid==0) val = warpReduceSum(val); //Final reduce within first warp

    return val;
}
```

Implementations 8: Use Warp Shuffle

```
__inline__ __device__ int warpReduceSum(int val)
{
    for (int offset = warpSize/2; offset > 0; offset /= 2)
        val += __shfl_down(val, offset);
    return val;
}
```

```
__inline__ __device__ int blockReduceSum(int val)
{
 ….
    val = warpReduceSum(val);      // Each warp performs partial reduction
    if (lane==0) shared[wid]=val; // Write reduced value to shared memory
    __syncthreads();              // Wait for all partial reductions

    //read from shared memory only if that warp existed
    val = (threadIdx.x < blockDim.x / warpSize) ? shared[lane] : 0;
    if (wid==0) val = warpReduceSum(val); //Final reduce within first warp

    return val;
}
```

For a small array (smaller than blockDim)

some warps do bogus work

Implementations 8: Reduce Large Arrays

```
_global__ void reduce8(int *in, int* out, unsigned int N)
{
    int sum = 0;
    //reduce multiple elements per thread
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
            sum += in[i];

    sum = blockReduceSum(sum);
    if (threadIdx.x==0)
        out[blockIdx.x]=sum;

}
```

```
  int threads = 512;
  int blocks = min((N + threads - 1) / threads, 512);

  reduce8<<<blocks, threads>>>(in, out, N);
```

The kernel is invoked many times until blocks=1

Implementations 9: Device Memory Atomic: Warp

```
__global__ void reduce9(int *in, int* out, unsigned int N)
{
    int sum = 0;
    for(int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
        sum += in[i];

    sum = warpReduceSum(sum);
    if ((threadIdx.x & (warpSize - 1)) == 0)
        atomicAdd(out, sum);
}
```

This code only needs to launch a single kernel and does not use shared memory, a temporary global array, or any __syncthreads().

The disadvantage is the use of lot of atomics.

Implementation 10: Device Memory Atomic: Block

```
__global__ void reduce10(int *in, int* out, unsigned int N)
{
    int sum = 0;
    for(int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
        sum += in[i];

    sum = blockReduceSum(sum);
    if (threadIdx.x == 0)
        atomicAdd(out, sum);
}
```

- Requires fewer atomics than the warp atomic approach, executes only a single kernel, and does not require temporary memory.
- It does however use __syncthreads() and shared memory (in the blockReduceSum function).
- Useful when you want to avoid two kernel calls or extra allocation and don't want the atomic pressure of using Implementation 9.