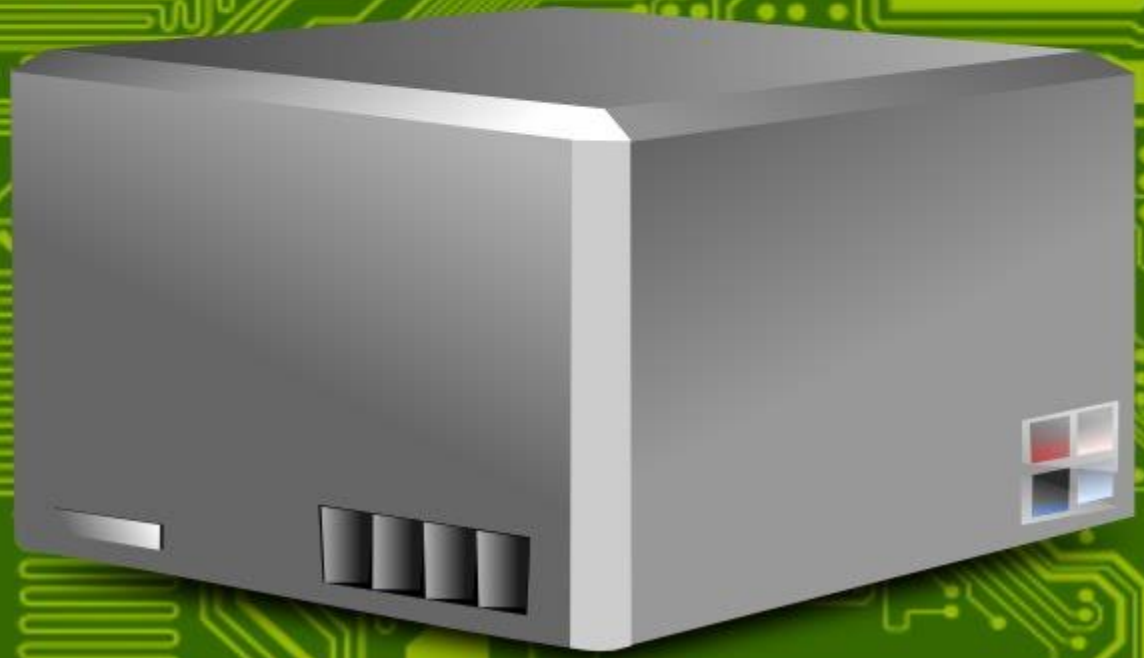
GPU Computing

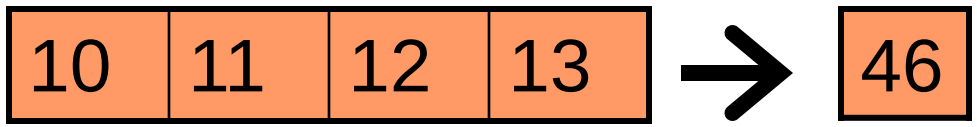


规约算法

Hui Liu
Email: hui.sc.liu@gmail.com

Reduction Operations

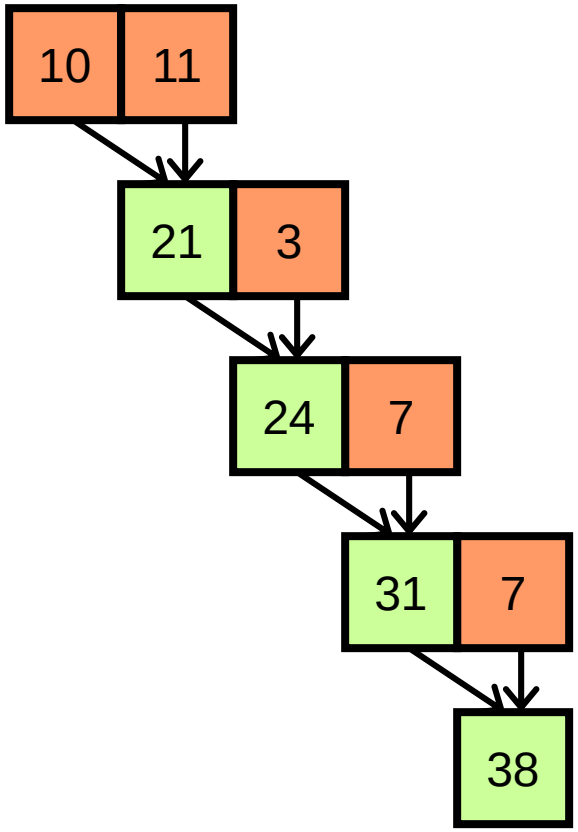
- Multiple values are reduced into a single value
  - ADD, MUL, AND, OR, ….



- Useful primitive

- Easy enough to allow us to focus on optimization techniques

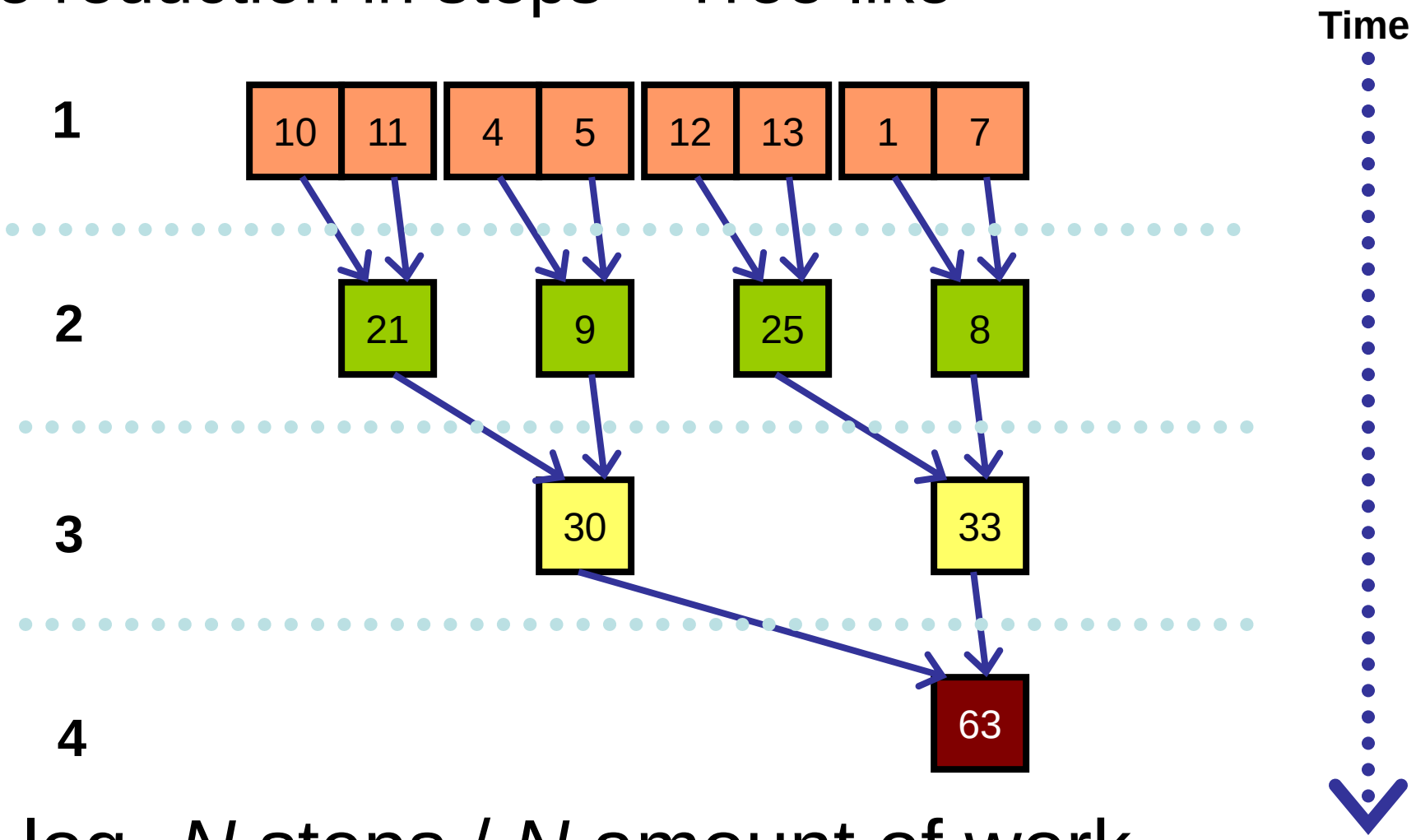
Sequential Reduction



- Start with the first two elements --> partial result
- Process the next element
- O(*N*)

Parallel Reduction

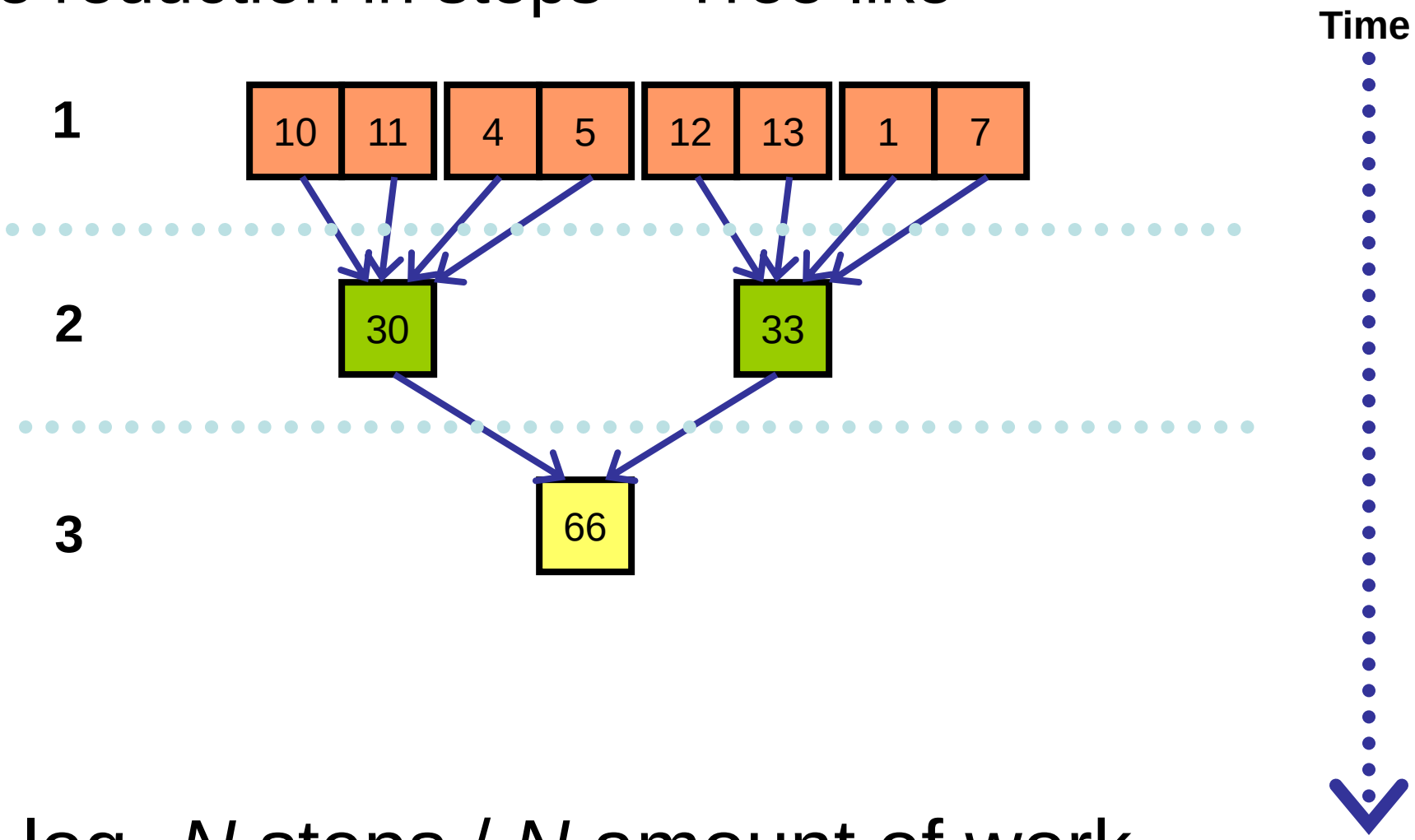
- Pair-wise reduction in steps – Tree-like



- $\log_2 N$ steps / N amount of work

Parallel Reduction – Different Degree-Trees possible

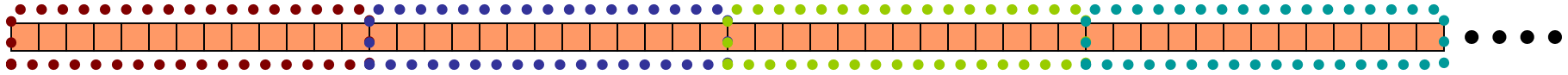
- Pair-wise reduction in steps – Tree-like



- $\log_4 N$ steps / N amount of work

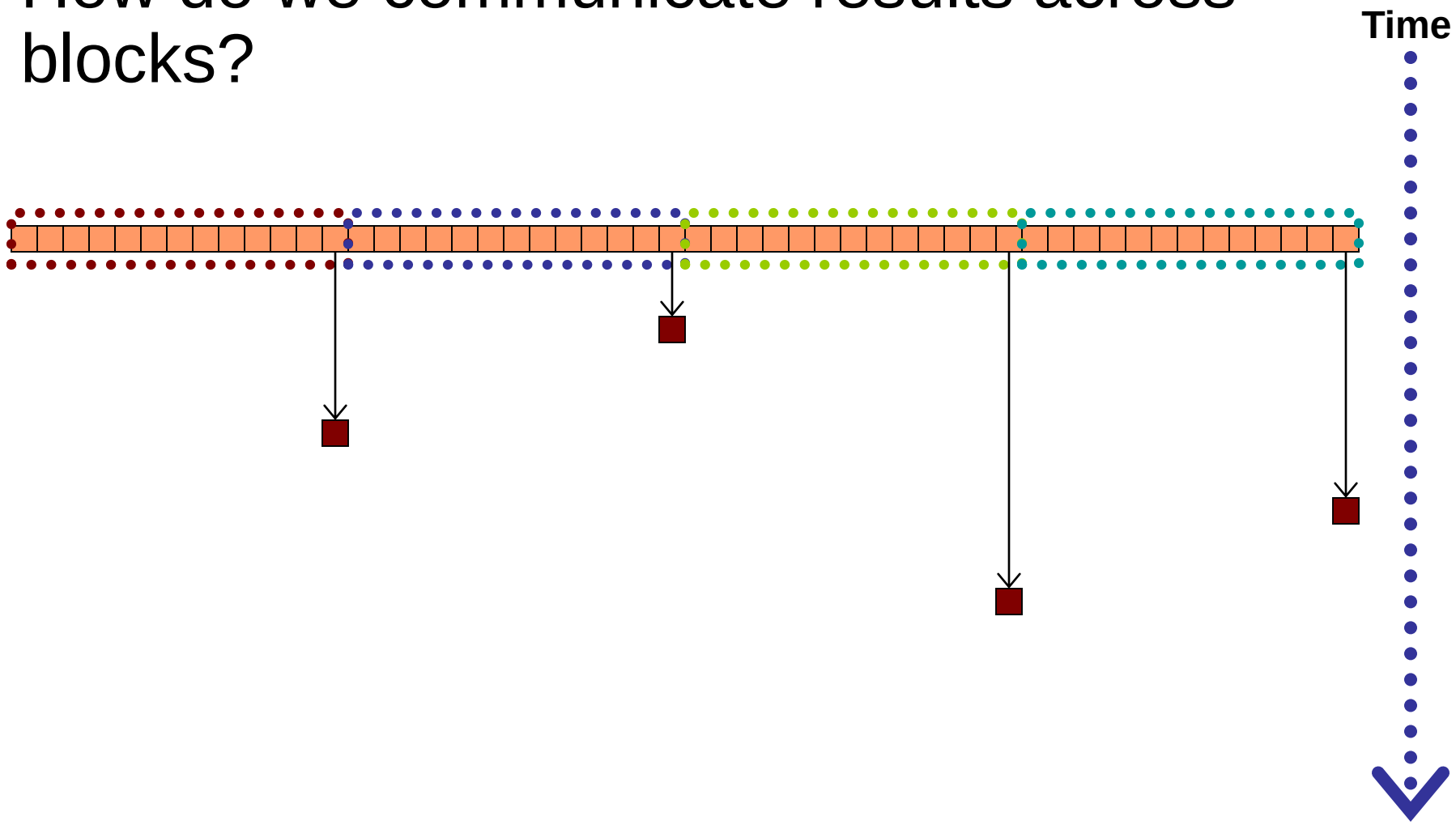
CUDA Strategy

- Single Block:
  - Use Tree-Like approach
- Multiple Blocks?
  - Not a necessity
    - one thread can always process many elements
    - But, will suffer from low utilization
  - Utilize GPU resources
  - Useful for large arrays
- Each block processes a portion of the input



How about multiple blocks

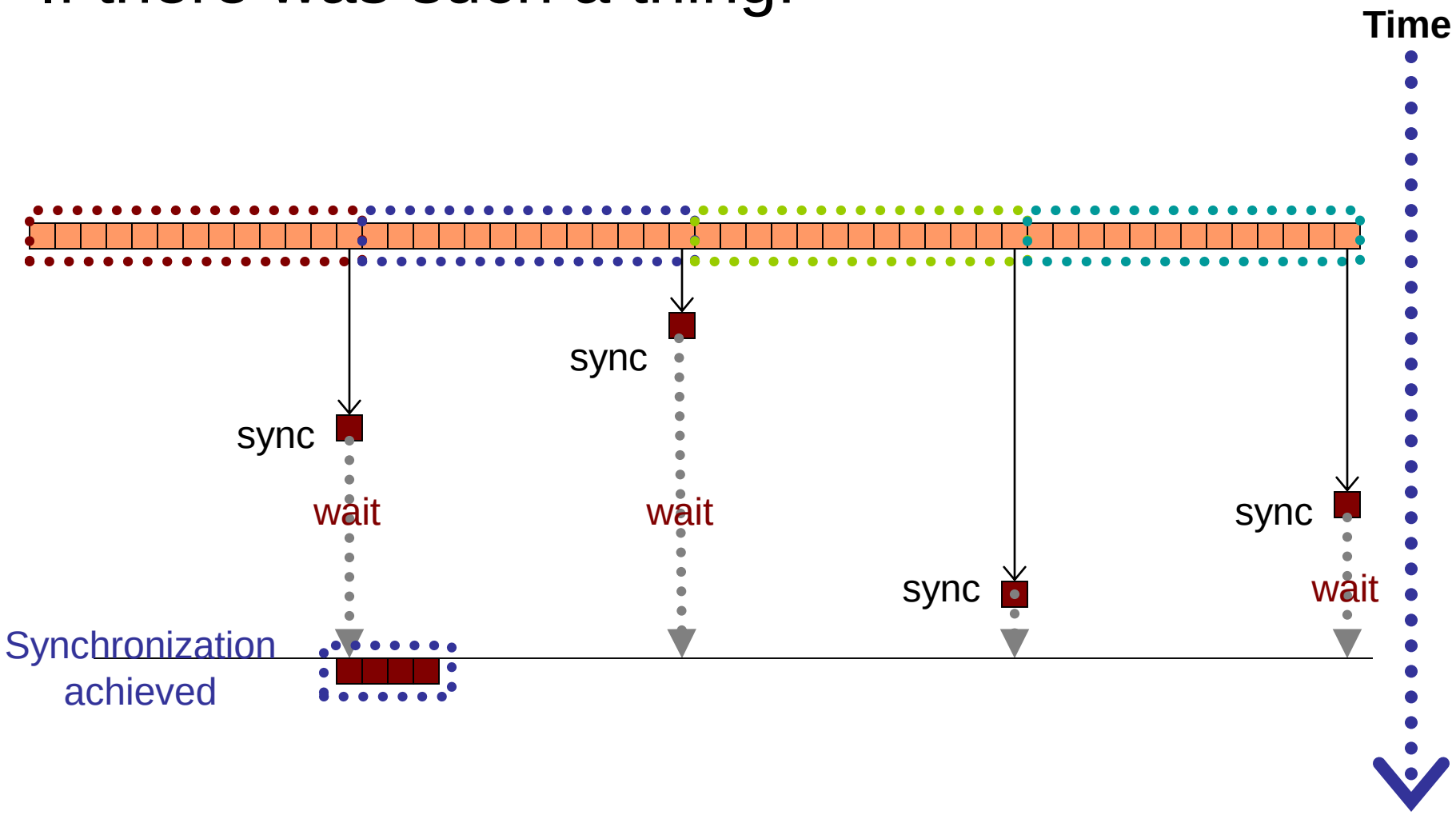
- How do we communicate results across blocks?



- The key problem is synchronization:
  - How do we know that each block has finished?

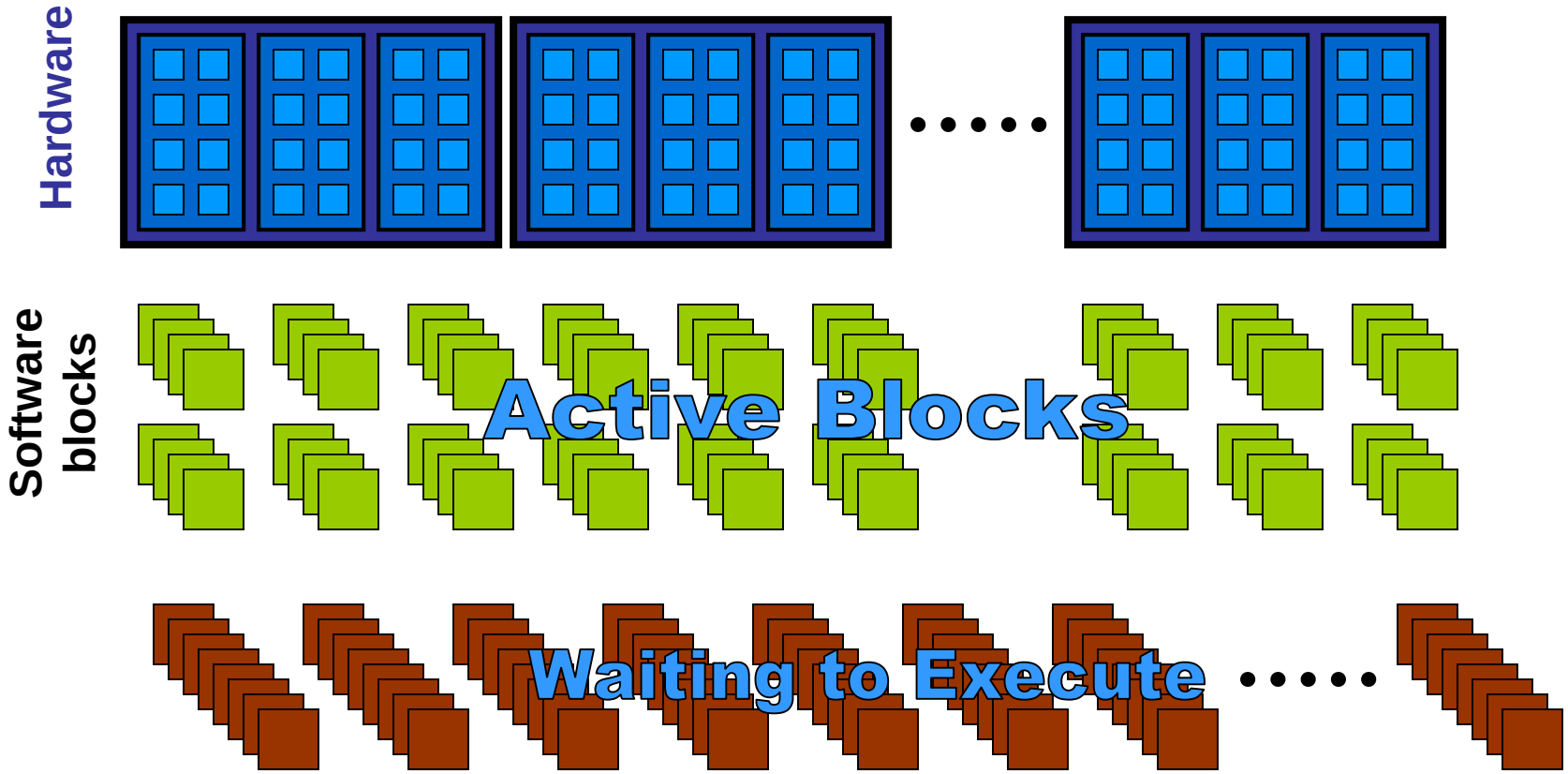
Global Synchronization

- If there was such a thing:

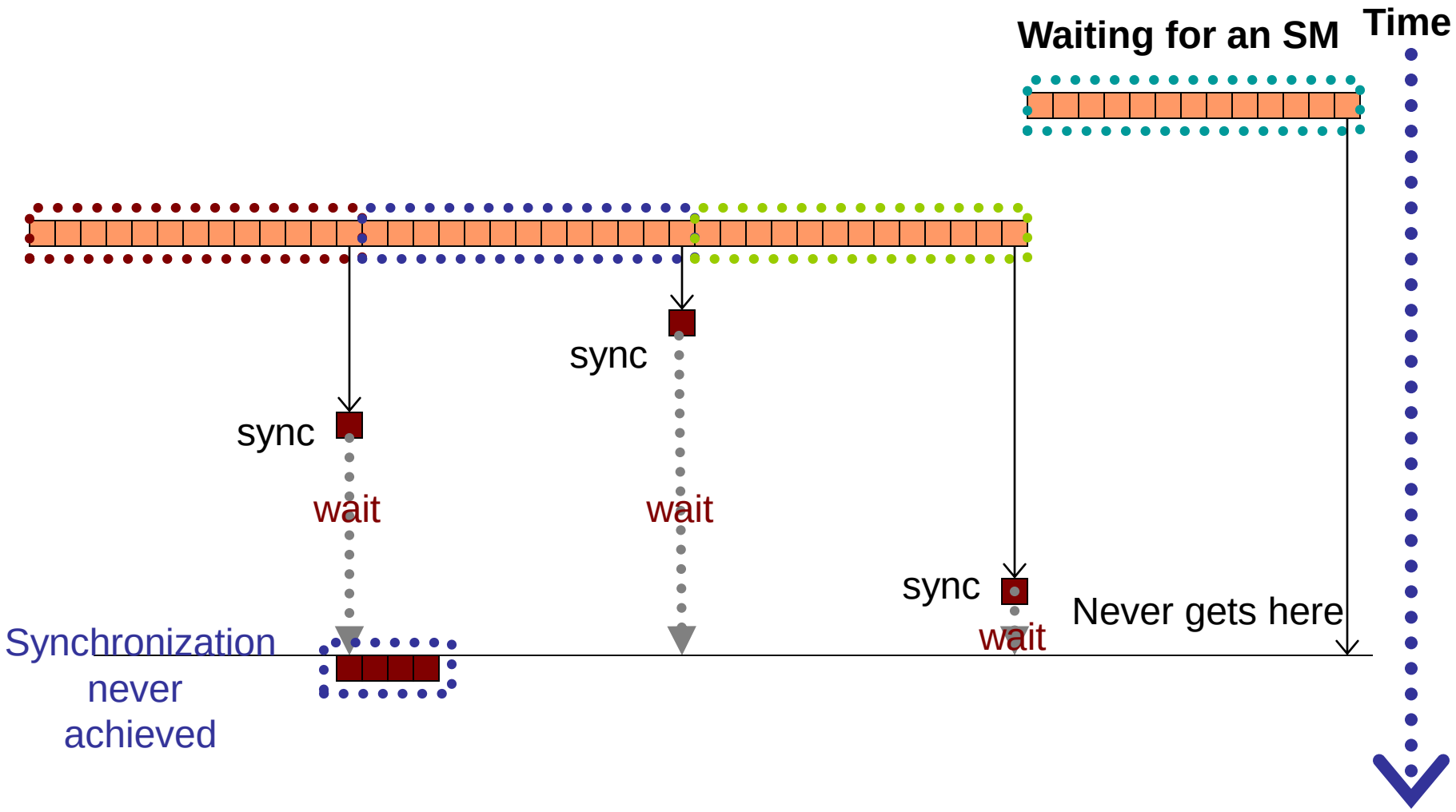


The Problem with Global Synchronization

- CUDA does not support it
  - One reason:
    - it's expensive to implement
  - Another reason:
    - "choose" between **limited blocks** or **deadlock**



Deadlock

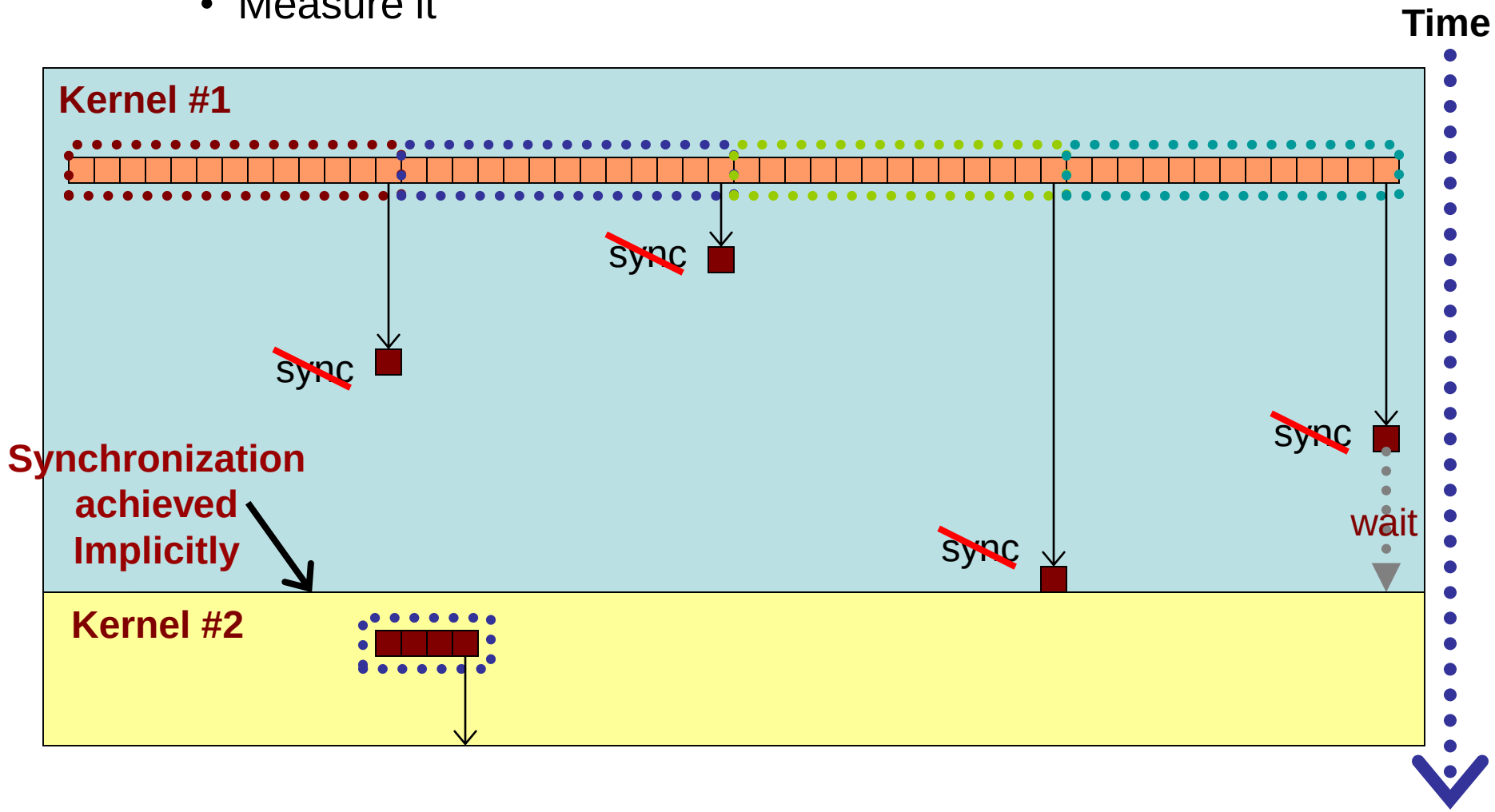

Waiting for an SM
Time
sync
sync
sync
wait
wait
wait
Never gets here
Synchronization
never
achieved

The Problem with Global Synchronization / Summary

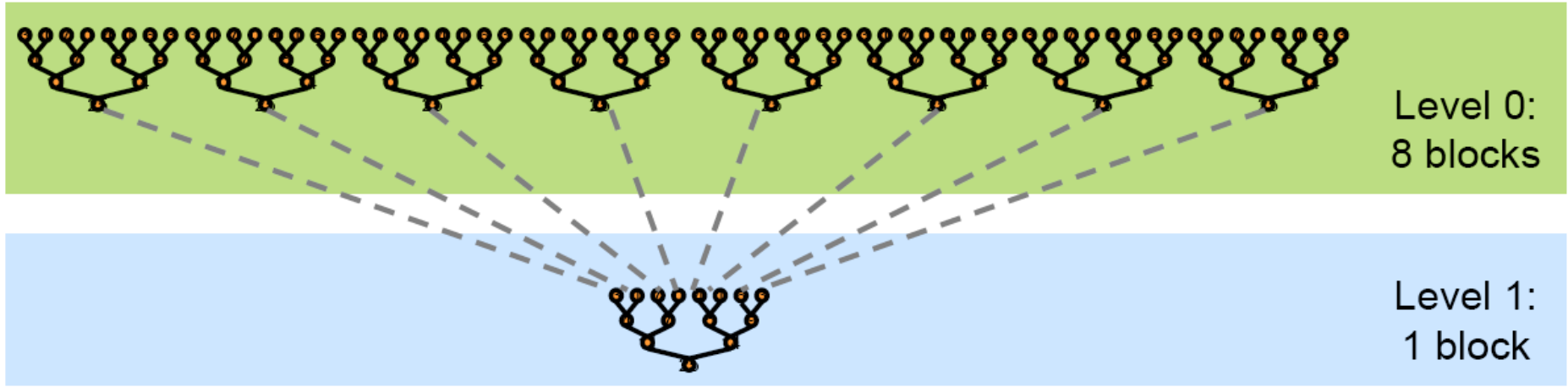
- If there was global sync
  - Global sync after each block
  - Once all blocks are done, continue recursively
- CUDA does not have global sync:
  - Expensive to support
  - Would limit the number of blocks
  - Otherwise deadlock will occur
    - Once a block gets assigned to an SM it stays there
    - Each SM can take only 8 blocks
    - So, at most #SMs x 8 blocks could be active at any given point of time
- Solution: Decompose into multiple kernels

Decomposing into Multiple Kernels

- Implicit Synchronization between kernel invocations
  - Overhead of launching a new kernel non-negligible
    - Don't know how much
    - Measure it



Reduction: Big Picture



- **The code for all levels is the same**
- **The same kernel code can be called multiple times**

Optimization Goal

- Get maximum GPU performance

- Two components:
  - Compute Bandwidth: GFLOPs
  - Memory Bandwidth: GB/s

Optimization Goals Cont.

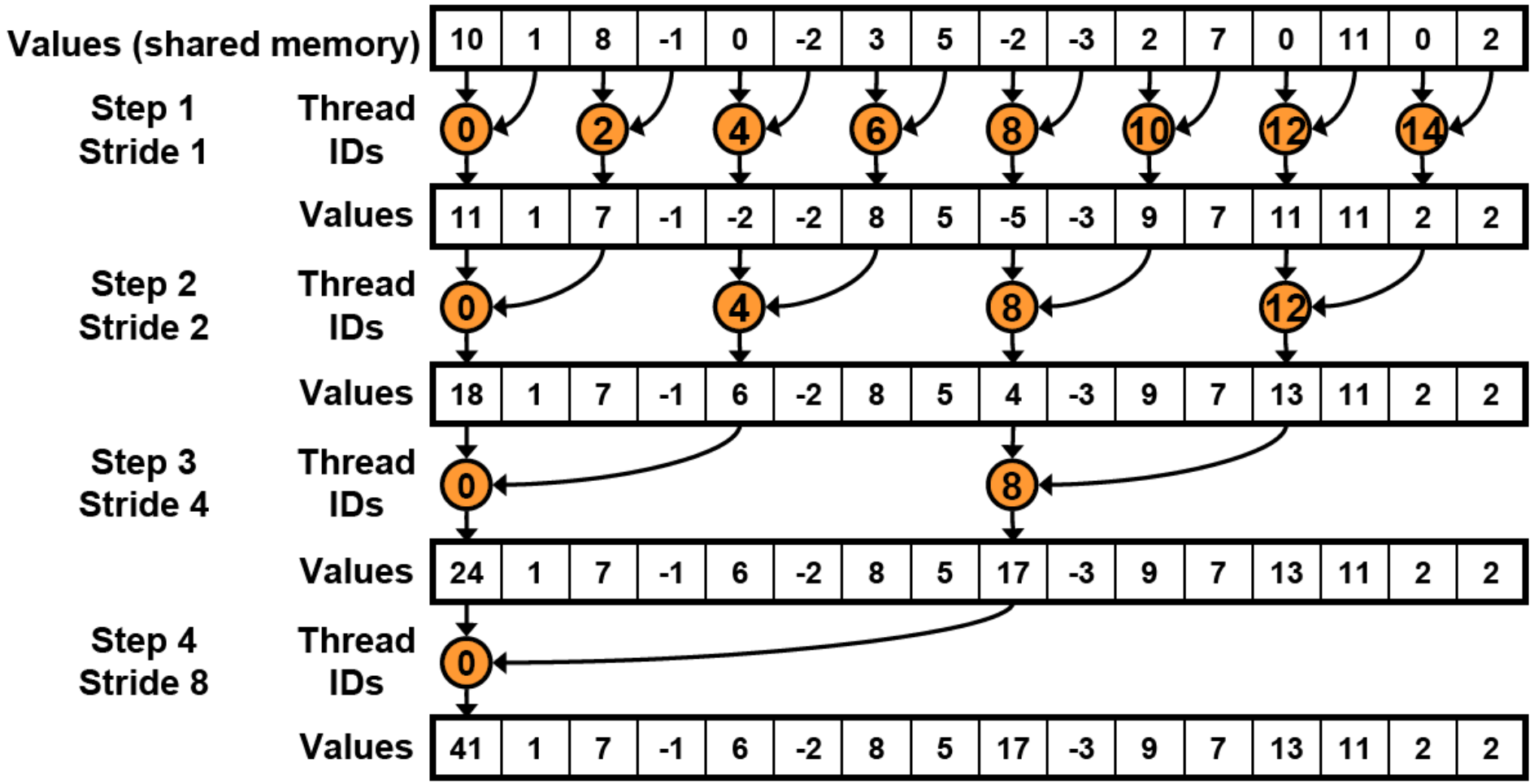
- Reductions typically have low arithmetic intensity
  - FLOPs/element loaded from memory
- So, bandwidth will be the limiter

- For the **ASUS ENGTX280 OC**
  - 512-bit interface, 1.14GHz DDR3
  - 512 / 8 x 1.14 x 2 = **145.92 GB/s**

Reduction #1: Strategy

- Load data:
  - Each thread loads one element from global memory to shared memory
- Actual Reduction: Proceed in log*N* steps
  - A thread reduces two elements
    - The first two elements by the first thread
    - The next two by the next thread
    - And so, on
  - At the end of each step:
    - Deactivate half of the threads
  - Terminate: when one thread left
- Write back to global memory

Reduction Steps



Reduction #1 Code: Interleaved Accesses

```
__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) { // step = s x 2
        if (tid % (2*s) == 0) { // only threadIDs divisible by the step participate
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

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

Performance for kernel #1

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

Note: Block size = 128 for all experiments

Caveat:

results are for a G80 processor

Bandwidth calculation:

Each block processes 128 elements and does:

128 reads & 1 write

We only care about global memory

At each kernel/step:

N (element reads) + N / 128 (element writes)

Every kernel/step reduces input size by 128x

next set N = N / 128

So for:

N = 4194304

Accesses = 4227394

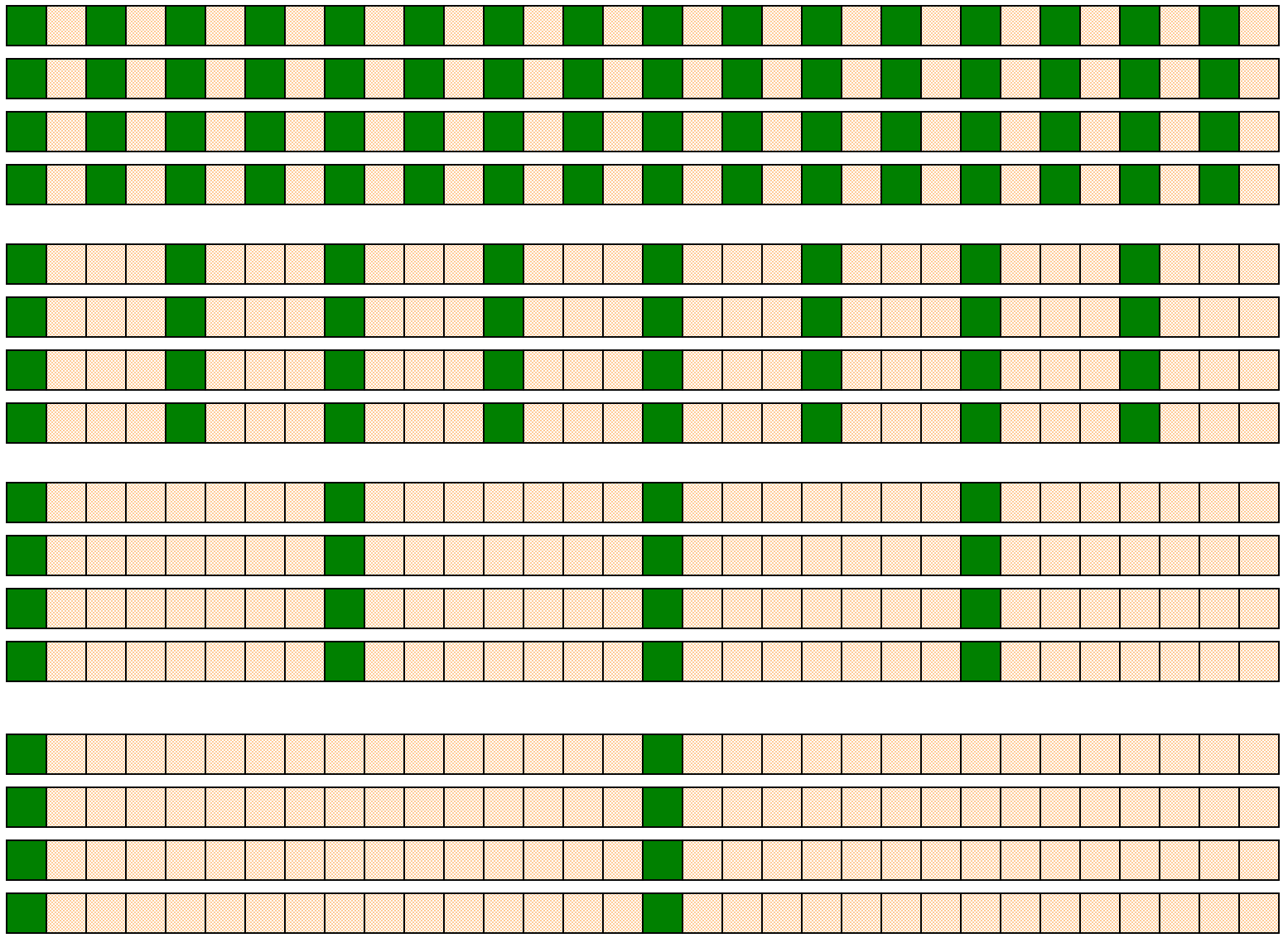
Each access is four bytes

Reduction #1 code: Interleaved Accesses

```
__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];
}
```

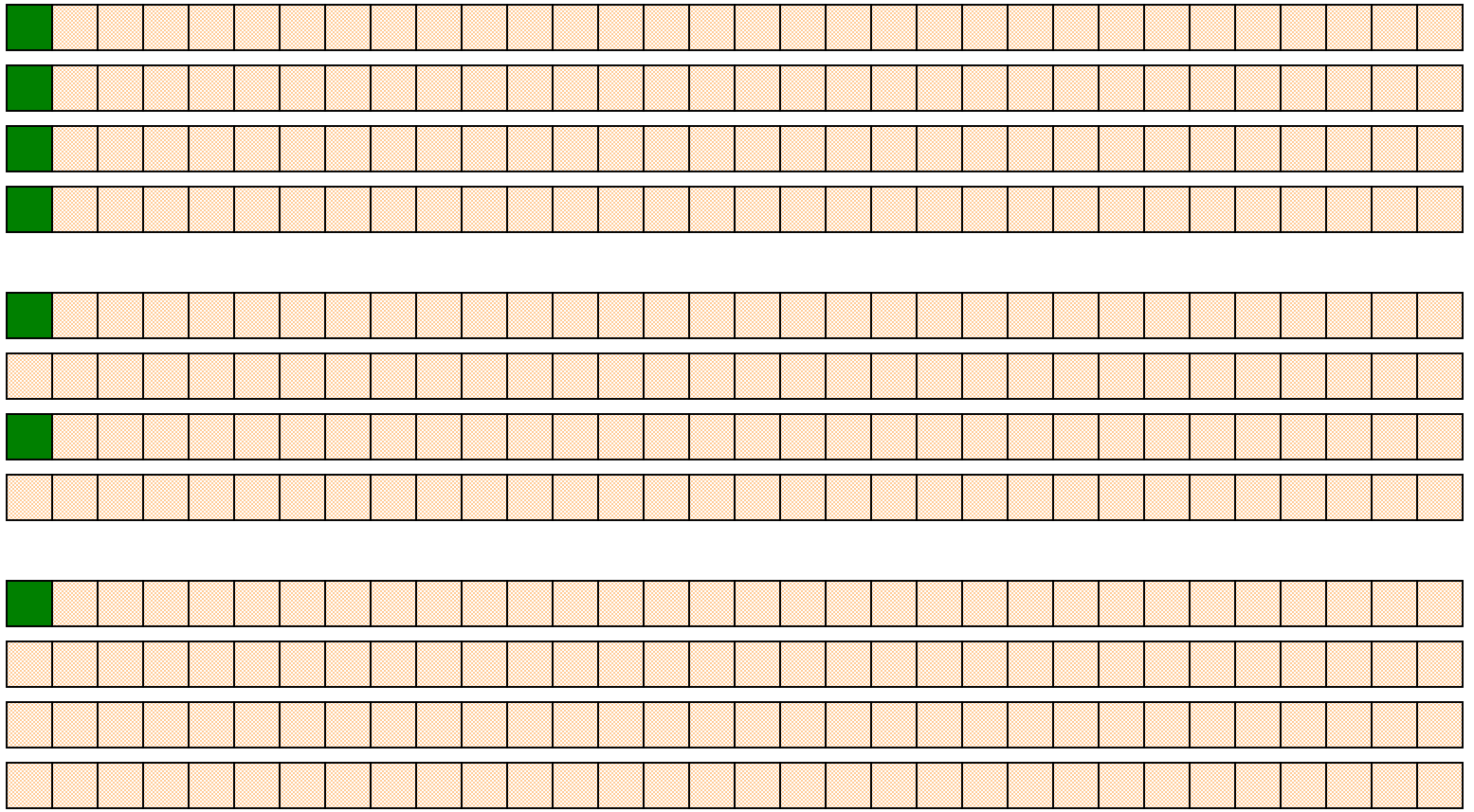
Highly divergent code leads to very poor performance

Divergent Branching: Warp Control Flow



4 warps/step

Divergent Branching: Warp Control Flow



Reduction #2: Interleaved Addr./non-divergent branching

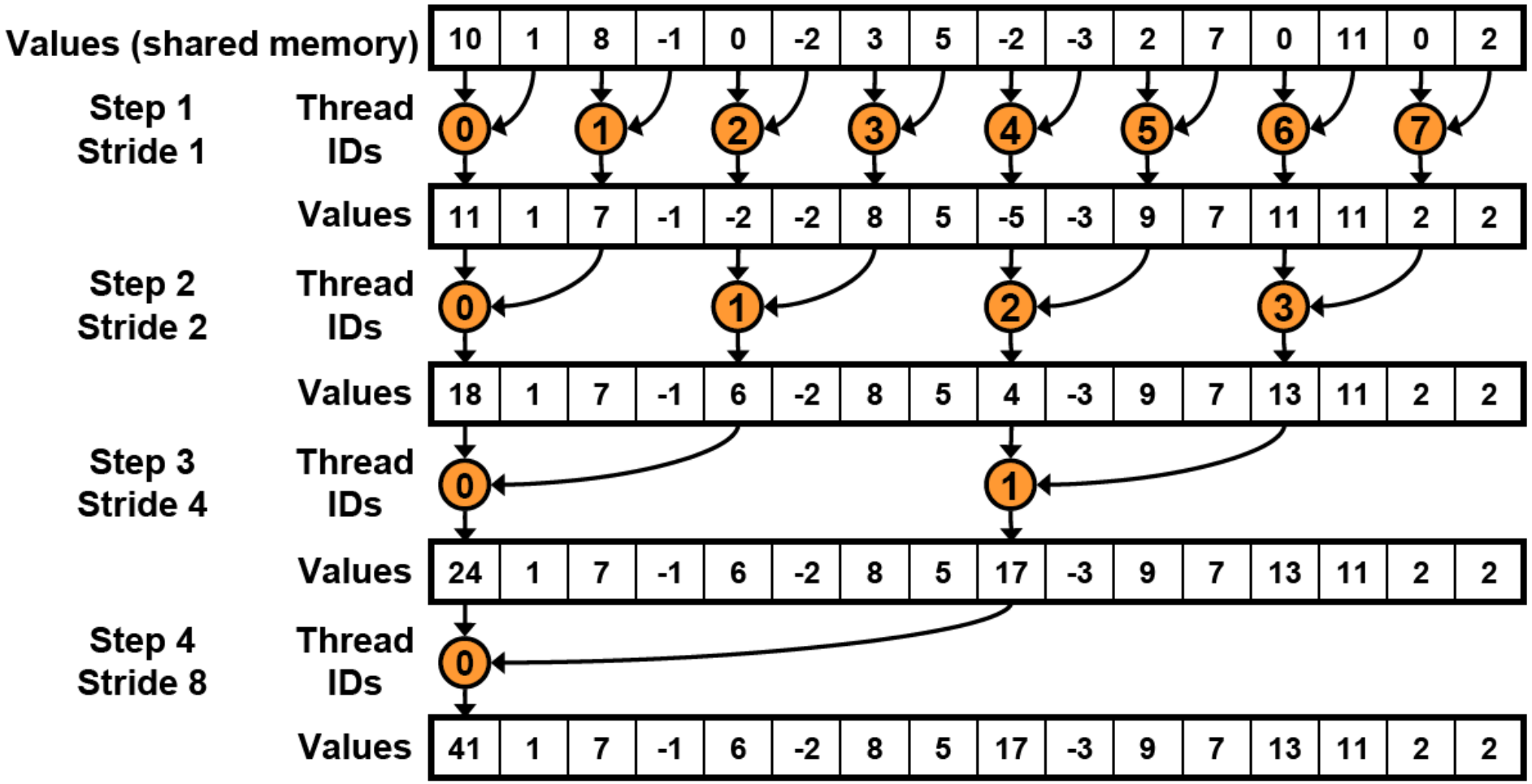
- Replace the divergent branching code:

```
// 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();
}
```

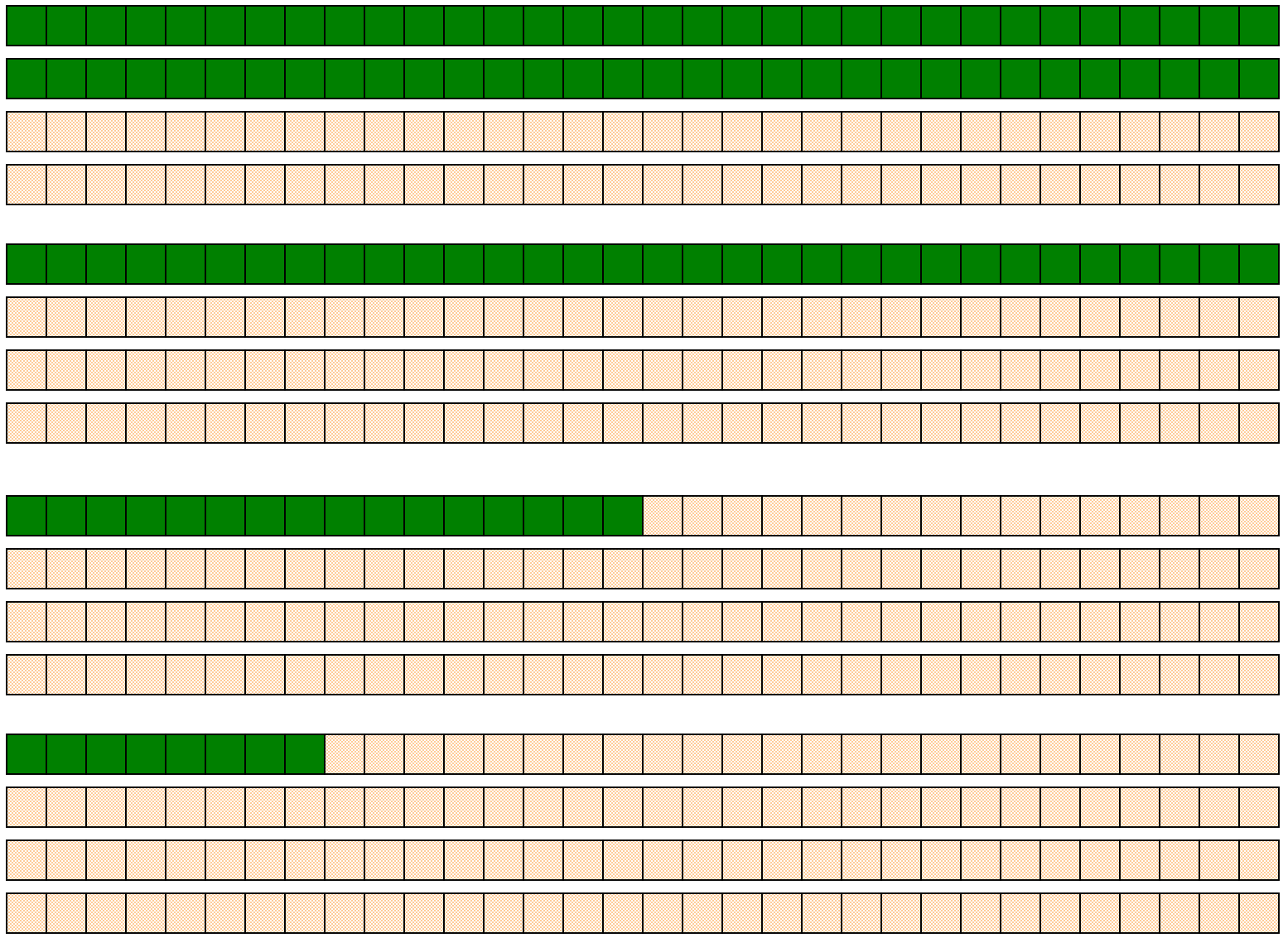
- With strided index and non-divergent branch

```
// do reduction in shared mem
for (unsigned int s=1; s < blockDim.x; s *= 2) {
        int index  = 2 * s * tid;

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

Reduction #2: Access Pattern

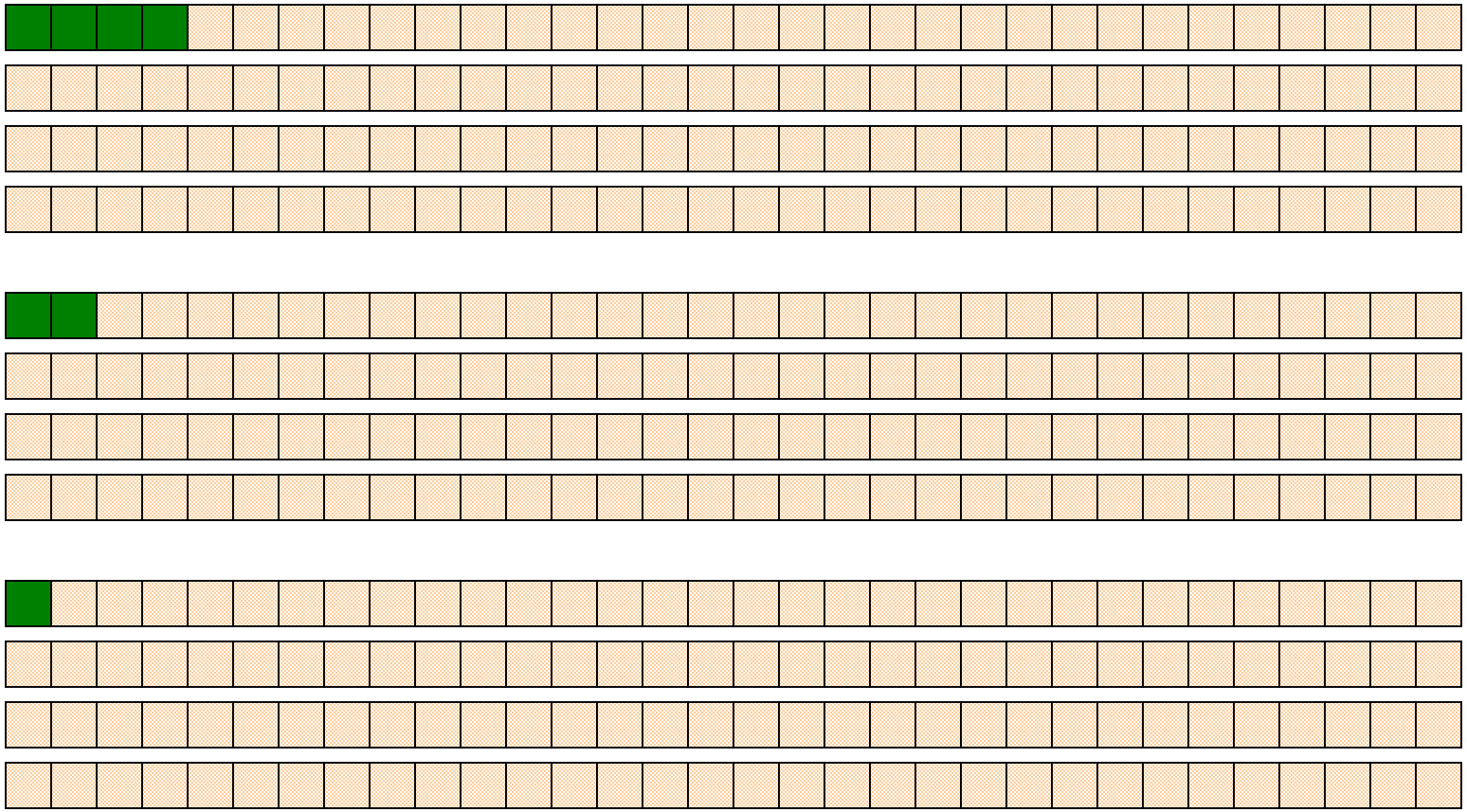


Reduction #2: Warp control flow



4 warps/step

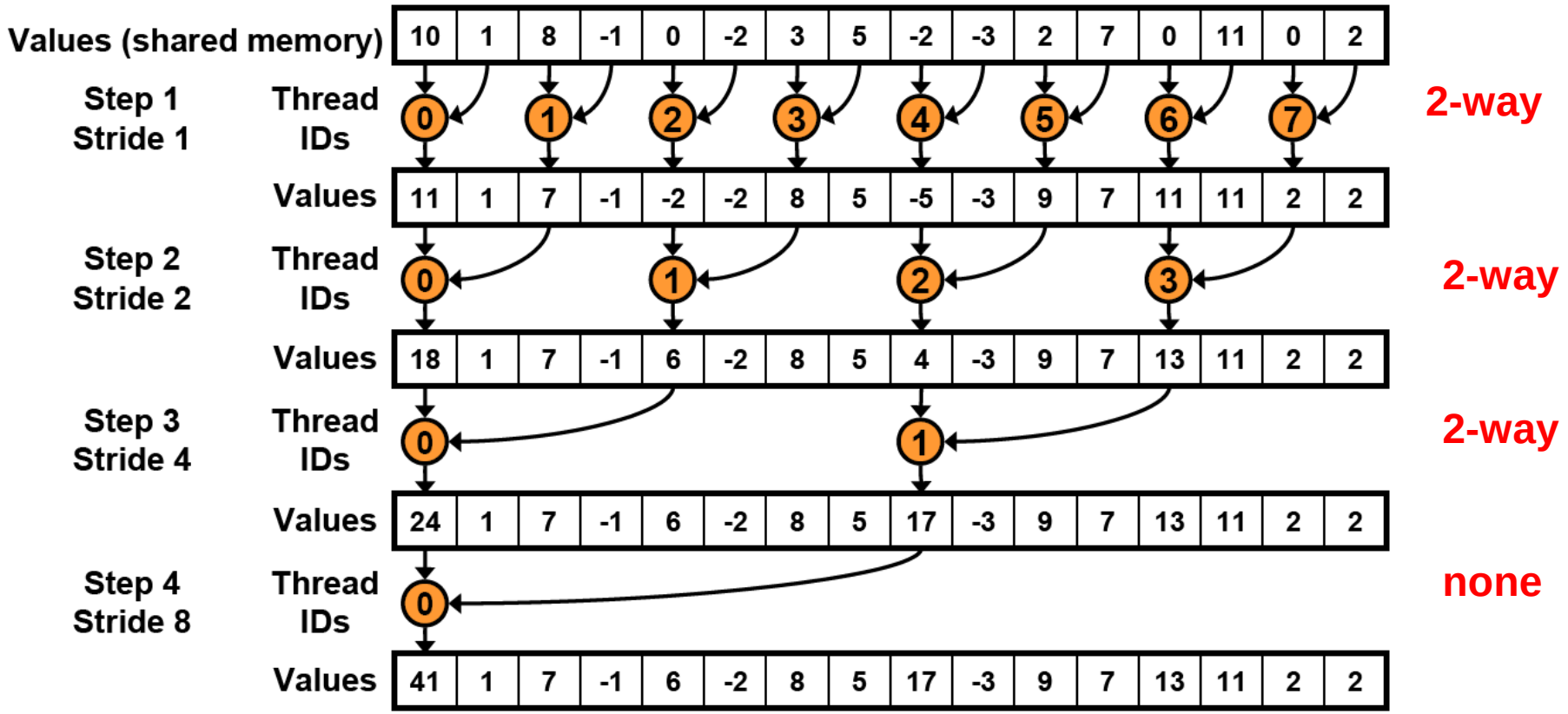
Reduction #2: Warp control flow



Performance for 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 non-divergent branching	**3.456 ms**	**4.854 GB/s**	**2.33x**	**2.33x**

Reduction #2: Problem

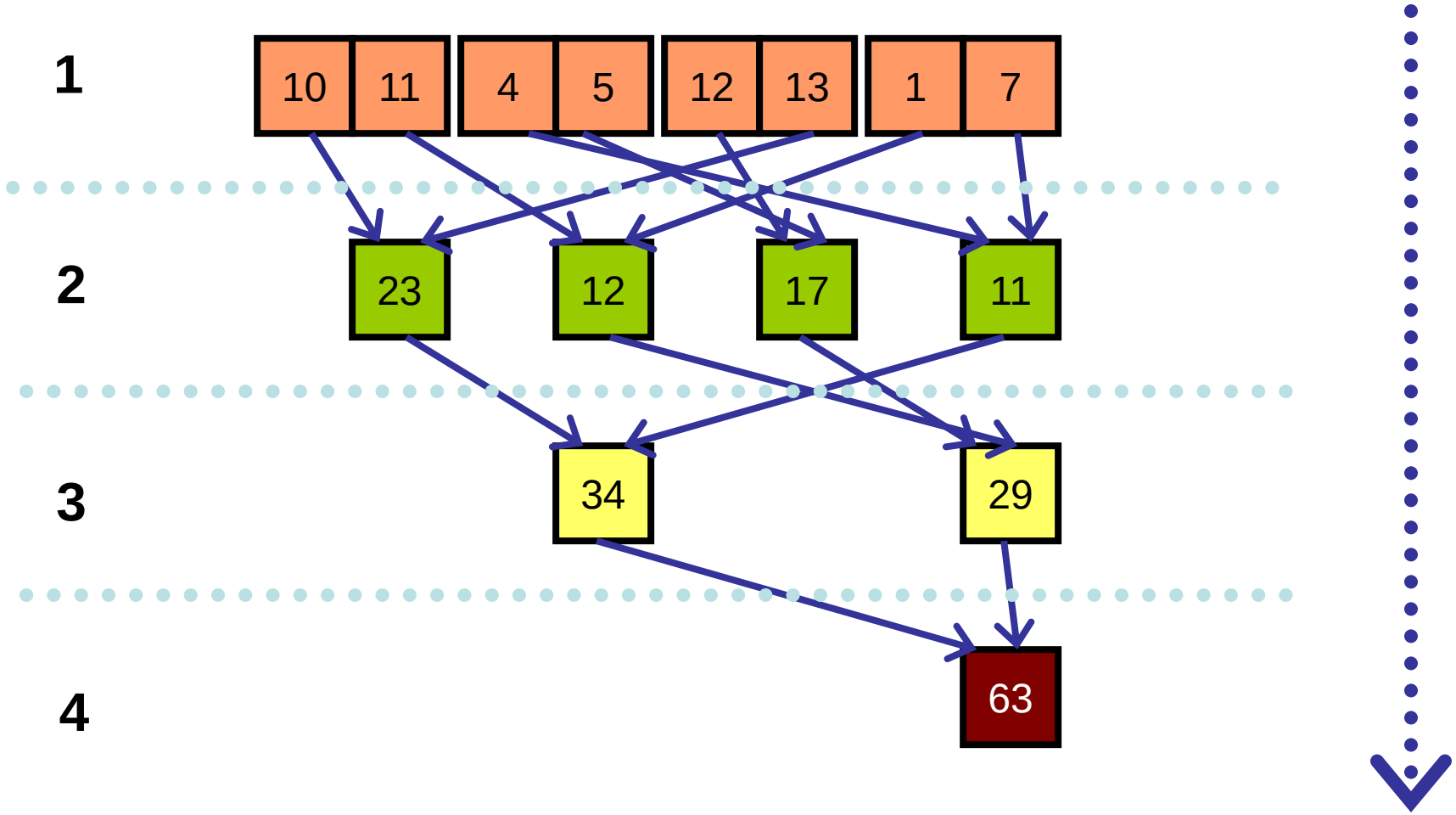


2-way bank conflicts at every step

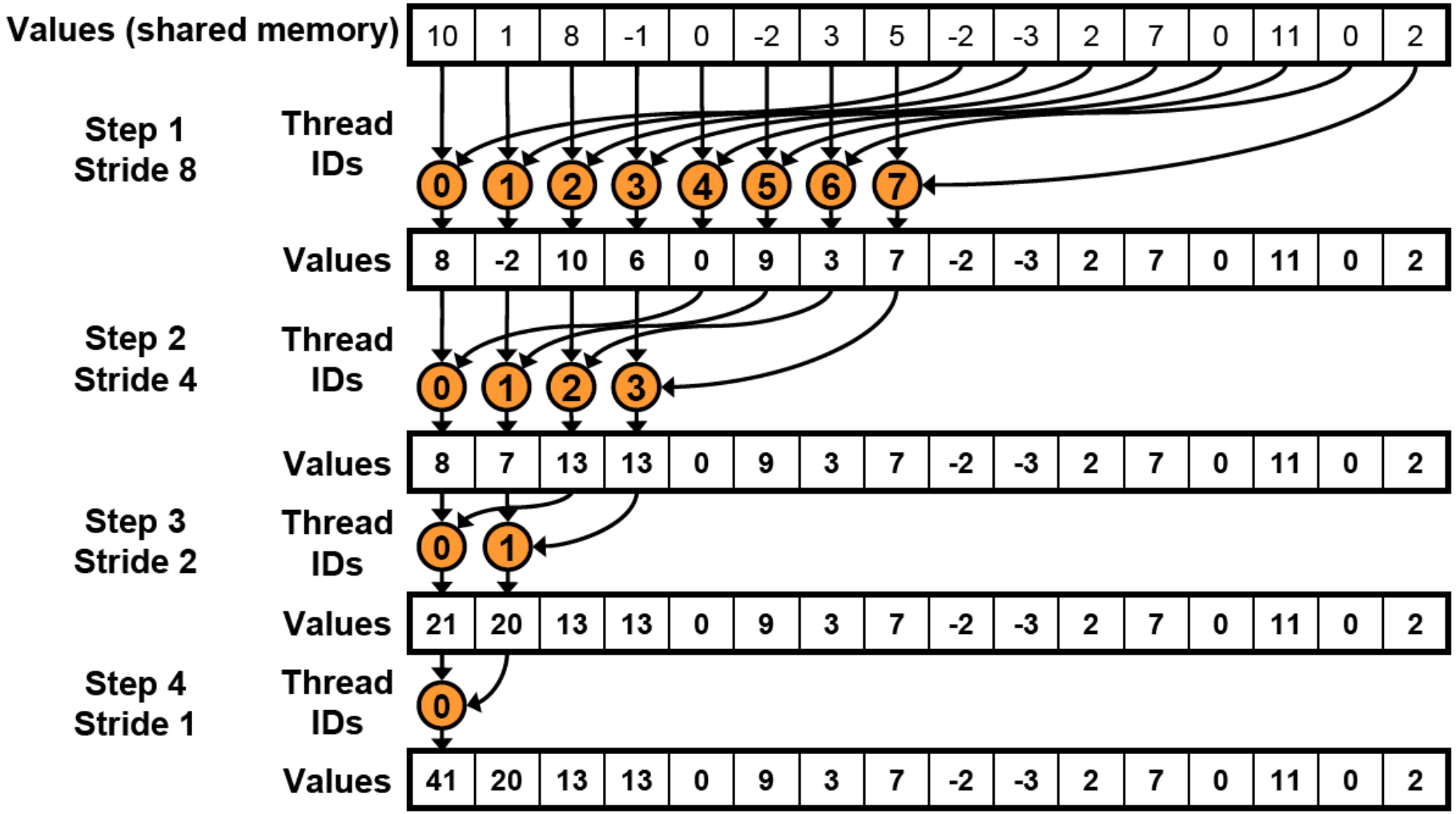
Recall there are more than 16 threads

To see the conflicts see what happens with 128 threads

Observe: Arbitrary Unique Pairs OK



Reduction #3: Sequential Accesses



- Eliminates bank conflicts

Reduction #3: Code Changes

- Replace stride indexing in the inner loop:

```
// do reduction in shared mem
for (unsigned int s=1; s < blockDim.x; s *= 2) {
        int index  = 2 * s * tid;

        if (index < blockDim.x == 0) {
                sdata[index] += sdata[index + s];
        }
        __syncthreads();
}
```

- With reversed loop and threadID-based indexing:

```
// 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();
}
```

Performance for 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 non-divergent branching	**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**

Reduction #3: Bad resource utilization

- All threads read one element
- First step: half of the threads are idle
- Next step: another half becomes idle

```
// 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();
}
```

Reduction #4: Read two elements and do the first step

- 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();
```

- Read two and do the first reduction step:

```
// each thread loads two elements from global to shared mem
// end performs the first step of the reduction
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 (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 non-divergent branching	**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 step during global load	**0.965 ms**	**17.377 GB/s**	**1.78x**	**8.34x**

Reduction #4: Still way off

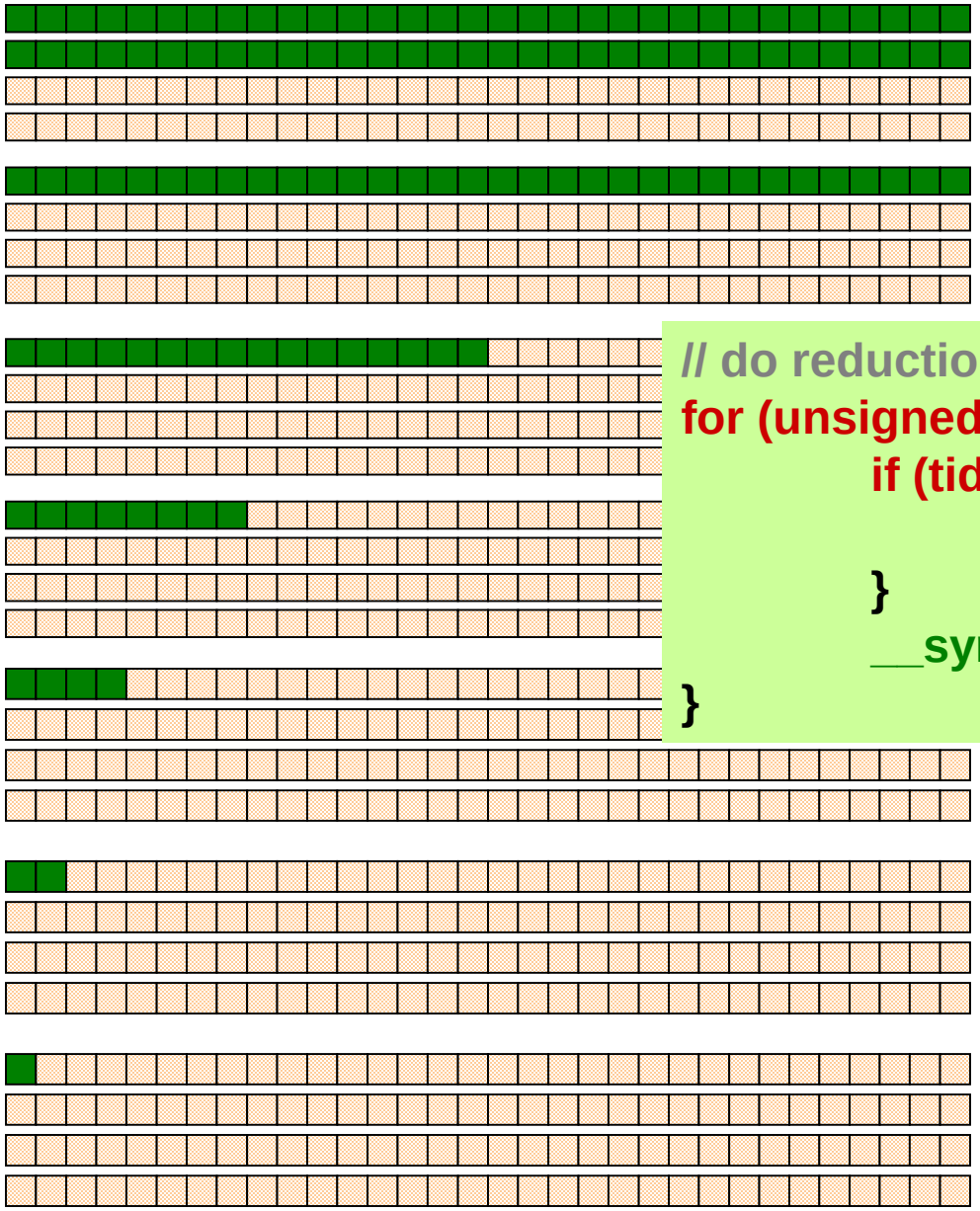
- Memory bandwidth is still underutilized
  - We know that reductions have low arithmetic density
- What is the potential bottleneck?
  - Ancillary instructions that are not loads, stores, or arithmetic for the core computation
  - Address arithmetic and loop overhead

- Unroll loops to eliminate these “extra” instructions

Unrolling the last warp

- At every step the number of active threads halves
  - When s <=32 there is only one warp left
- Instructions are SIMD-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

Reduction #2: Warp control flow



4 warps/step

```
// 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();
}
```

Reduction #5: Unrolling the last 6 iterations

```
// do reduction in shared mem
for (unsigned int s = blockDim.x/2; s > 32; s /= 2) {

        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];
}
```

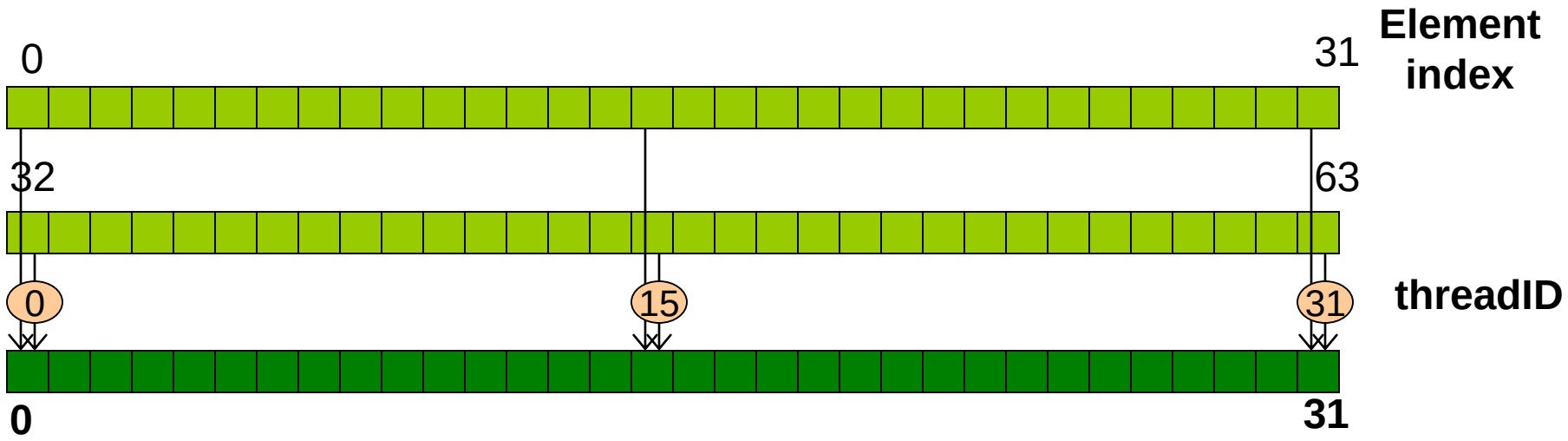
- This saves work in all warps not just the last one
  - Without unrolling all warps execute the for loop and if statement

Unrolling the Last Warp: A closer look

- Keep in mind:
- #1: Warp execution proceeds in lock step for all threads
  - All threads execute the same instruction
  - So:
    - **sdata[tid] += sdata[tid + 32];**
  - Becomes:
    - Read into a register: sdata[tid]
    - Read into a register: sdate[tid+32]
    - Add the two
    - Write: sdata[tid]
- #2: Shared memory can provide up to 16 words per cycle
  - If we don’t use this capability it just gets wasted

Unrolling the Last Warp: A Closer Look

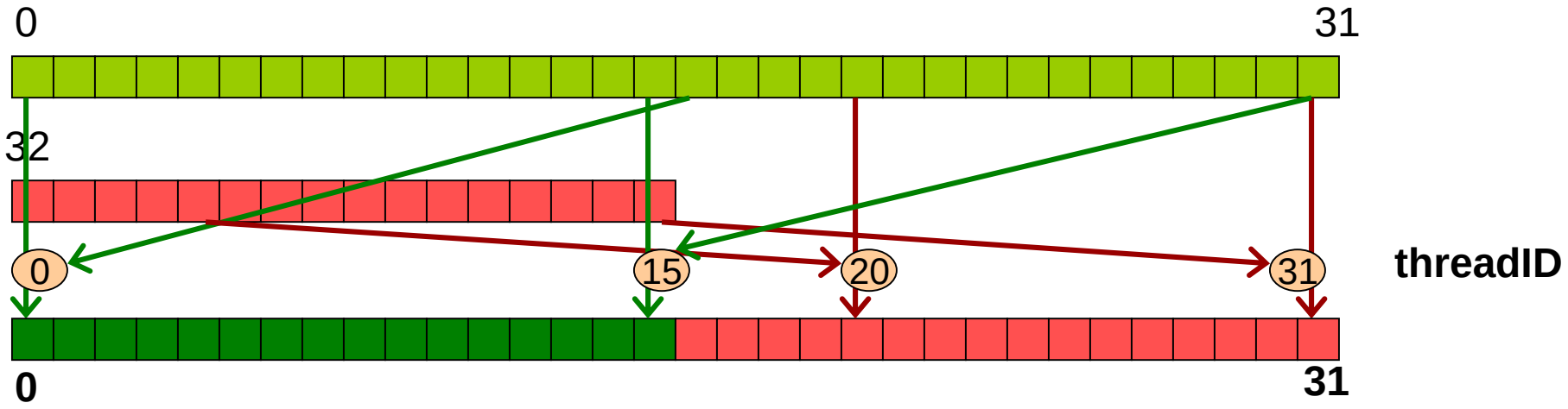
- sdata[tid] += sdata[tid + 32];



- All threads doing useful work

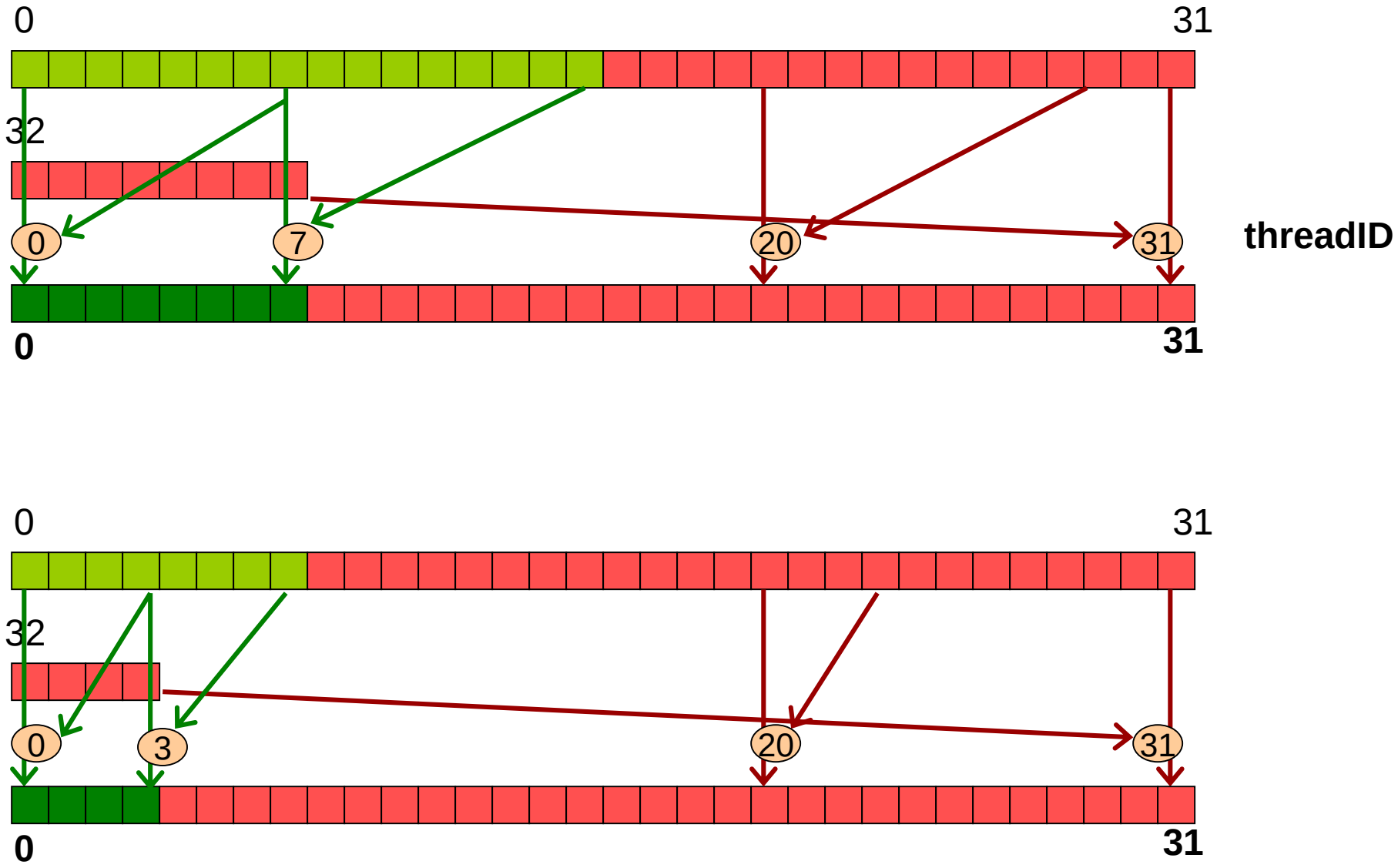
Unrolling the Last Warp: A Closer Look

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

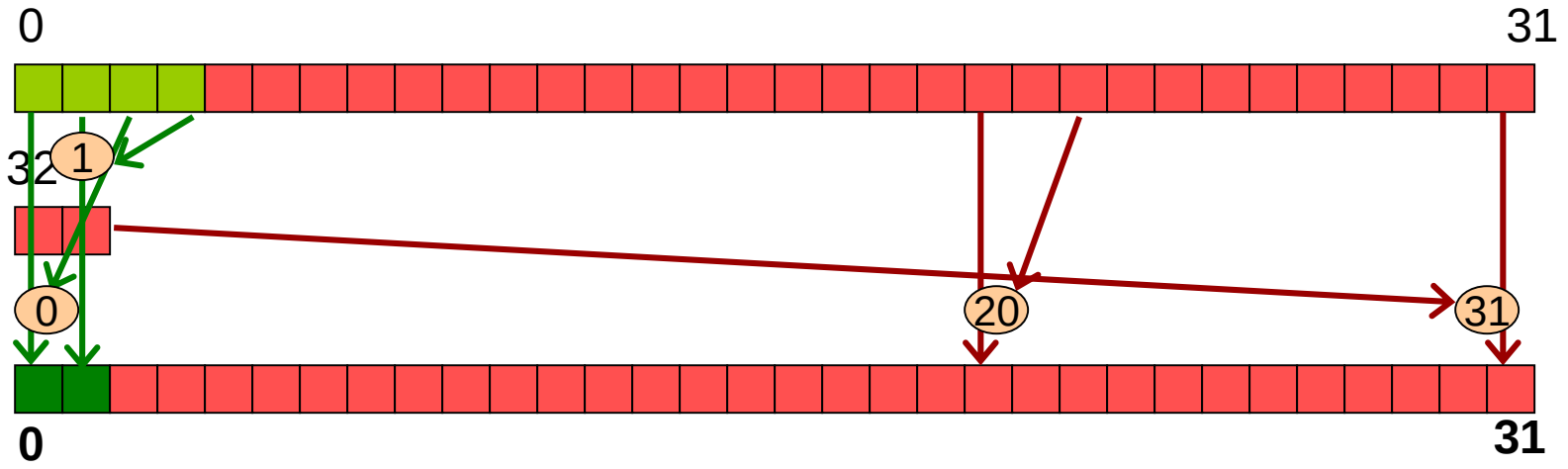


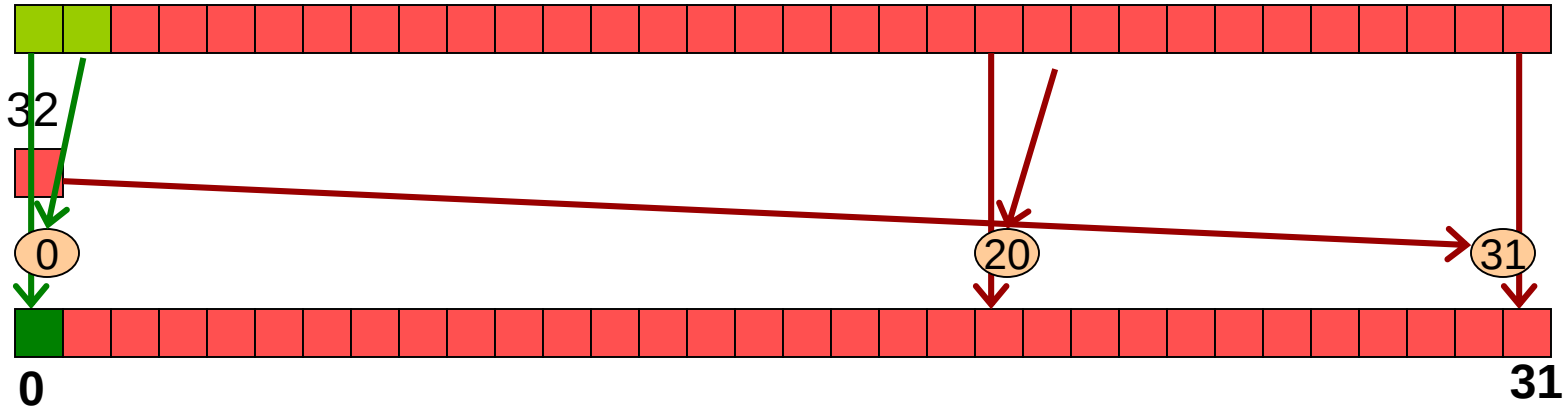
  - Half of the threads, 16-31, are doing useless work
  - At the end they “destroy” elements 16-31
  - Elements 16-31 are inputs to threads 0-14
  - But threads 0-15 read them before they get written by threads 16-31
    - All reads proceed in “parallel” first
    - All writes proceed in “parallel” last
  - So, no correctness issues
  - But, threads 16-31 are doing useless work
    - The units and bandwidth are there → no harm (only power)

Unrolling the Last Warp: A Closer Look



Unrolling the Last Warp: A Closer Look





Performance for 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 non-divergent branching	**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 step 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**

Reduction #6: Complete Unrolling

- If we knew the number of iterations at compile time, we could completely unroll the reduction
  - 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 complile time?
- C++ Templates
  - CUDA supports C++ templates on device and host functions

Unrolling Using Templates

- Specify block size as a function template parameter:

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

Reduction #6: Completely Unrolled

```
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

What if block size is not known at compile time?

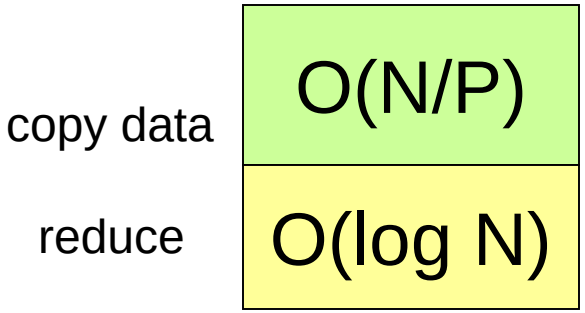
- There are “only” 10 possibilities:

```
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;
}
```

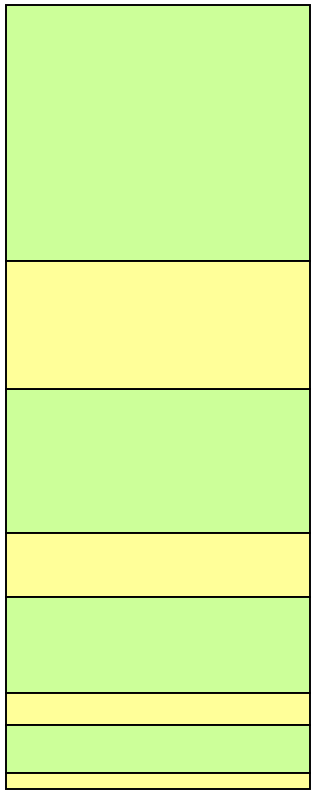
Performance for 4M element reduction

	Time (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 non-divergent branching	**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 step 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: Complete Unroll	**0.381 ms**	**43.996 GB/s**	**1.41x**	**21.16x**

Can we improve?



If we can choose P --> can we make this O(log N)



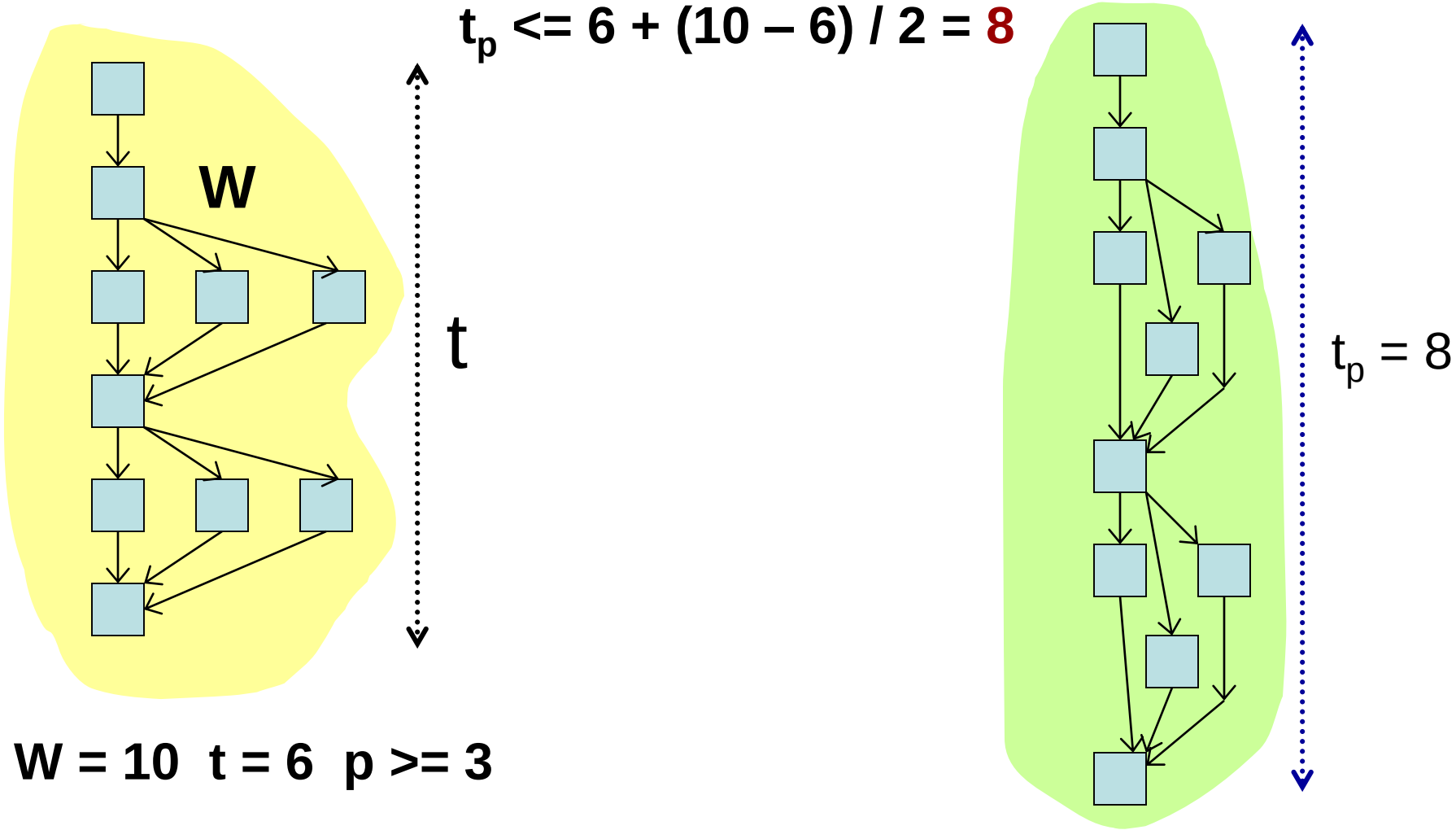
Can we improve?

- Two parts:
  - Loading elements + pair-wise reduction
    - **O(*N*/*P*)**
  - Tree-like reduction
    - **O(log *N*)**
- Overall: O(N/P + log N)
- Can we make the O(N/P) part O(log N)?
- What should be P?
  - N/log N
- So, first step should be:
  - load N/log N elements
  - do the first N/log reductions

Brent's theorem

$$t_p <= t + (W - t) / p$$

$t_p <= 6 + (10 - 6) / 2 = 8$



W = 10 t = 6 p >= 3

If given infinite processors an algorithm takes t steps
then given p processors it should take at most t_p steps

Can we improve the algorithm?

- **Work** or "**element complexity**": **W**
  - total number of operations performed collectively by all processors
  - Time to execute on a single processor
- **Time Step:** All operations that have no unresolved dependencies are performed in parallel
- **Depth** or "**step complexity**": **t**
  - time to complete all time steps
  - time to execute if we had infinite processors
- **Computation time on P Processors:** t_p
  - time to execute when there are P processors

Brent's Theorem

$$t_p <= t + (W - t) / p$$

- p =1, t_p = W (seq. algorithm)
- p→ inf, t_p → t (limit)
- if sequential --> t = W

- **Work** or "**element complexity**": **W**
- **Depth** or "**step complexity**": **t**
- **Computation time on P Processors: t_p**

What is Brent's theorem useful for?

- Tells us whether an algorithm can be improved for sure
- Example:
  - summing the elements of an array
    - for (i=0; i < N); i++) sum = sum + a[i];
    - W = N, t = N
    - $t_p = N + (N – N) / p = N$
    - No matter what, this will take N time

Brent's theorem example application

- Summing the elements recursively
  - ((a[0] + a[1]) + ((a[2] + a[3])) …
  - t = log N
  - W = N

- **T = log(N) + (N - log(N))/p**

- Conclusions:
  - No implementation can run faster than O(log *N*).
  - Given *N* processors, there is an algorithm of log *N* time
  - Given *N* / log *N* processors, there is an algorithm of ~2 x log *N* = O(log N) time
  - Given 1 processor, there is an algorithm that takes *N* time

Brent's theorem contd.

- Cost: P x Time Complexity
  - P = 1 x O(N) time
  - P = log *N* x O(log N) time
  - P = *N* x O(log N) time
  - The implementations with 1 or log N processors, therefore are **cost optimal**,
  - The implementation with N processors is not.
- It is important to remember that Brent's theorem does not tell us how to implement any of these algorithms in parallel
  - it merely tells us what is possible
  - The Brent's theorem implementation may be hideously ugly compared to the naive implementation

Parallel Reduction Complexity

- *Log(N)* parallel steps, each step *S* does $N/2^S$ independent operations
  - Step Complexity: 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
  - 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*)
  - N/P for global to shared
  - log N for the calculations
  - Compare to O(N) for sequential reduction

What about Cost?

- Cost of parallel algorithm
  - Processors x Time Complexity
    - Allocate threads instead of processors: O(*N*) threads
    - Time complexity is O(log *N*), so cost is O(*N* log *N*)
      - Not cost efficient
- Brent's theorem suggests O(*N*/log *N*) threads
  - Each thread does O(log *N*) sequential work
  - Then all O(*N*/log *N*) threads cooperate for O(log *N*) steps
  - Cost = O((*N*/log *N*) * log *N*) = O(*N*) → cost efficient
- Sometimes called ***algorithm cascading***
  - Can lead to significant speedups in practice

Algorithm Cascading

- Combine sequential and parallel reduction
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory
- Brent's theorem says each thread should sum O(log N) elements
  - i.e., 1024 to 2048 elements per block vs. 256
- In Mike Harris' 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 two elements

```
// each thread loads two elements from global to shared mem
// end performs the first step of the reduction
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 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

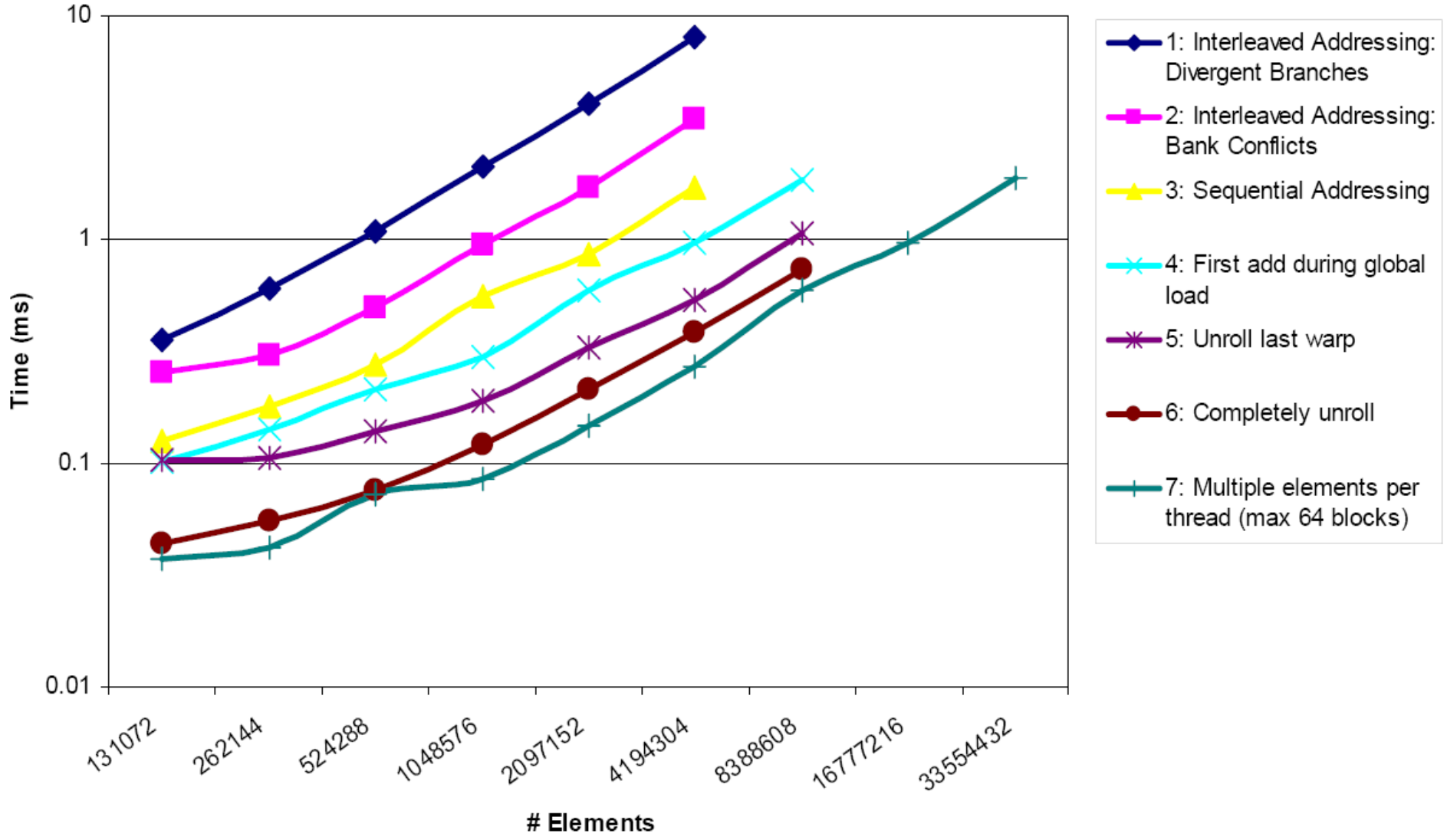
Performance for 4M element reduction

	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 non-divergent branching	**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 step 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: Complete Unroll	**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)
{
    __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];
}
```

Performance comparison on G80



Optimization types

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

- Code optimizations:
  - Loop unrolling
  - 2.54x speedup

Summary

- 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

THANK YOU