

GPU Programming with CUDA (optimizing)

Bryan Mills, PhD

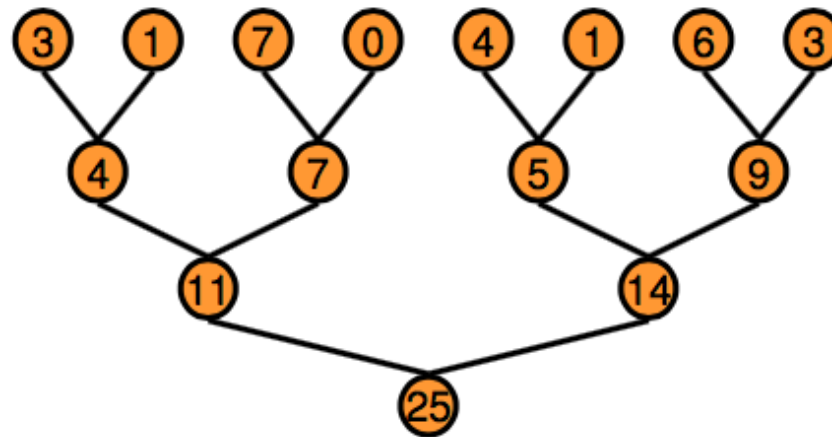
Spring 2017

Parallel Reduction

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

Parallel Reduction

- **Tree-based approach used within each thread block**



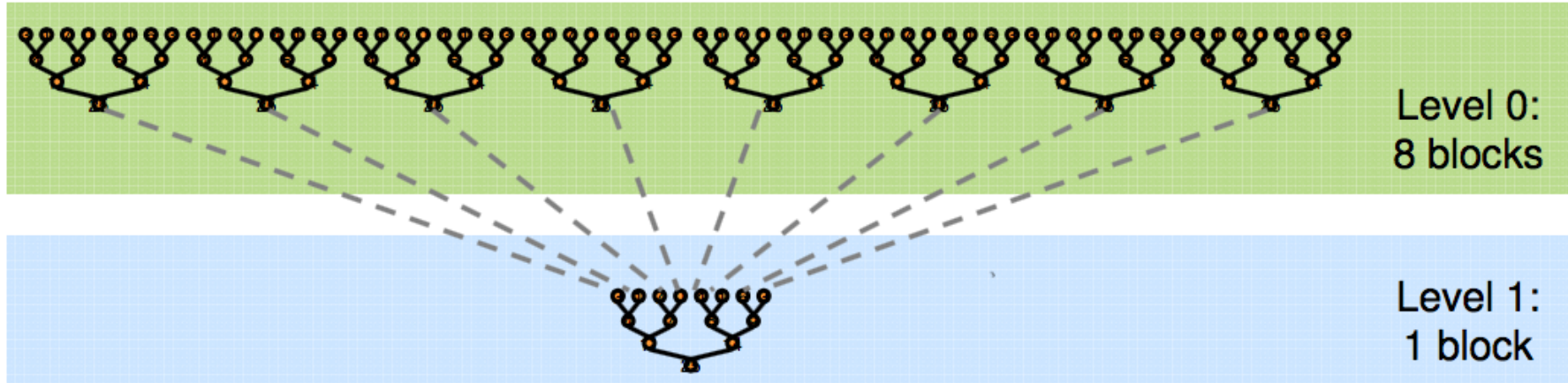
- **Need to be able to use multiple thread blocks**
 - To process very large arrays
 - To keep all multiprocessors on the GPU busy
 - Each thread block reduces a portion of the array
- **But how do we communicate partial results between thread blocks?**

Global Sync?

- **If we could synchronize across all thread blocks, could easily reduce very large arrays, right?**
 - Global sync after each block produces its result
 - Once all blocks reach sync, continue recursively
- **But CUDA has no global synchronization. Why?**
 - Expensive to build in hardware for GPUs with high processor count
 - Would force programmer to run fewer blocks (no more than $\frac{\text{\# multiprocessors}}{\text{\# resident blocks}}$ multiprocessors) 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

Kernel Decomposition

- Avoid global sync by decomposing computation into multiple kernel invocations

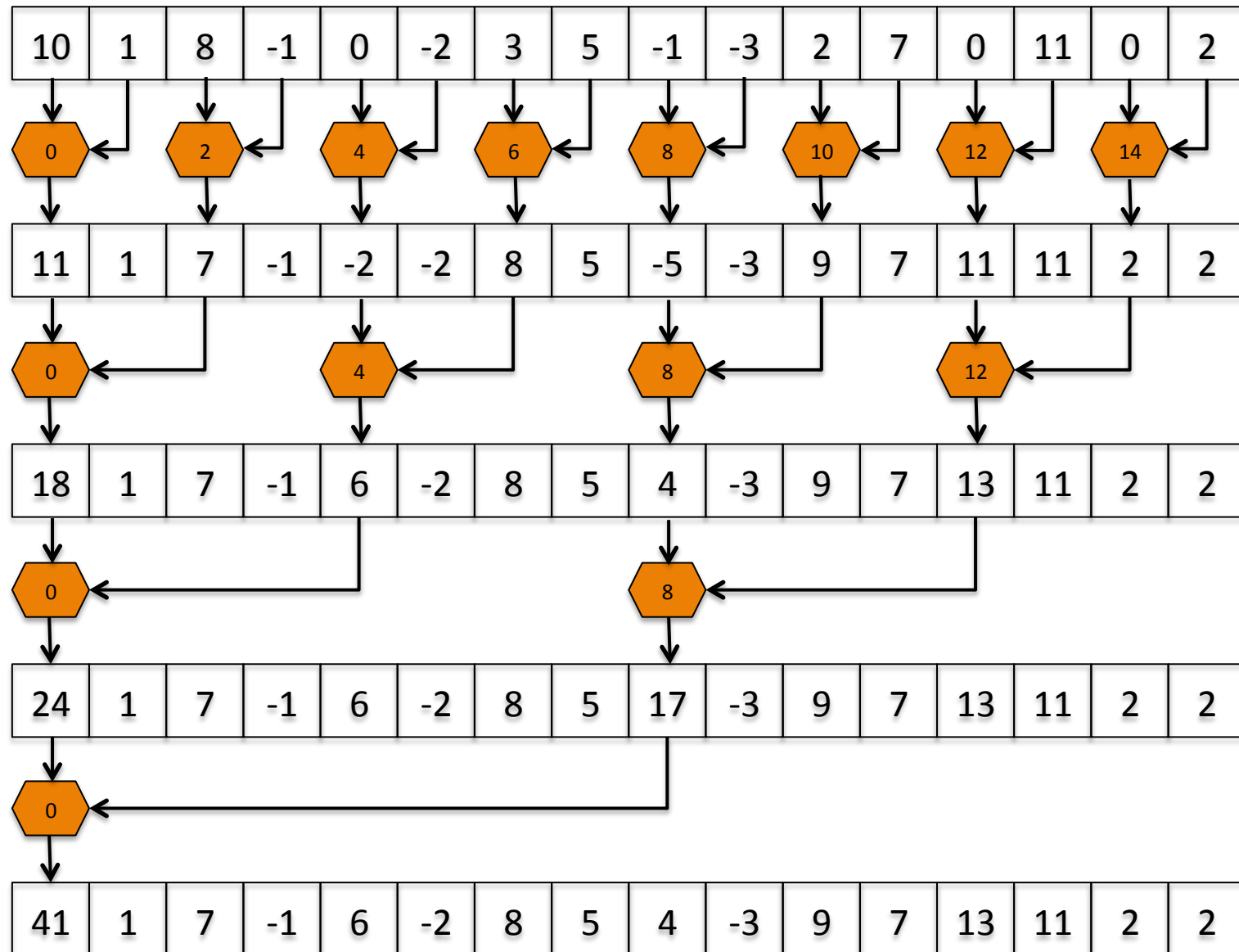


- In the case of reductions, code for all levels is the same
 - Recursive kernel invocation

Interleaved Addresses


```
__global__ void reduce(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // each thread loads 1 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];
}
```

Interleaved Addresses



Interleaved Addresses

```
__global__ void reduce(int *g_idata, int *g_odata) {  
    extern __shared__ int sdata[];  
    // each thread loads 1 element from global to shared mem  
    unsigned int tid = threadIdx.x;  
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;  
    sdata[tid] = g_idata[i];  
    __syncthreads();  
    // do reduction in shared mem  
    for(unsigned int s=1; s < blockDim.x; s *= 2) {  
        if (tid % (2*s) == 0) {  
            sdata[tid] += sdata[tid + s];  
        }  
        __syncthreads();  
    }  
    // write result for this block to global mem  
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];  
}
```



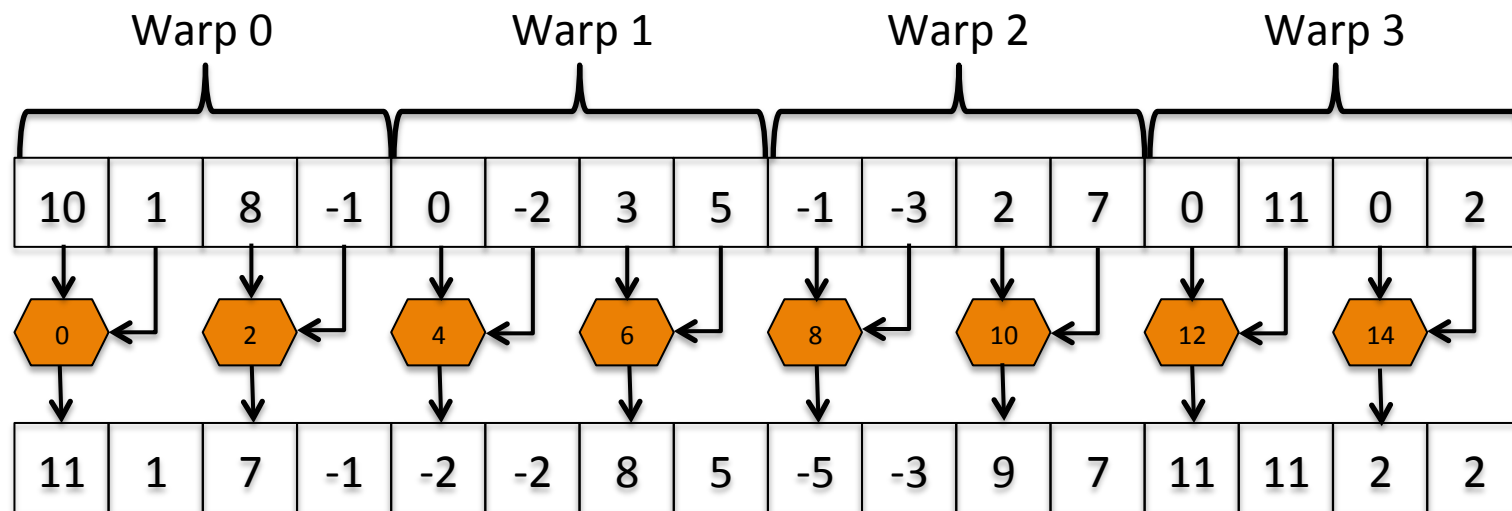
Problem: Highly Divergent Branching

Warps

- Once a block is assigned to an SM, it is divided into units called warps.
 - Thread IDs within a warp are consecutive and increasing
 - Warp 0 starts with Thread ID 0
- Warp size is implementation specific.
- Warp is unit of thread scheduling in SMs

Warp Example

- Assume warp size of 4 in the example.



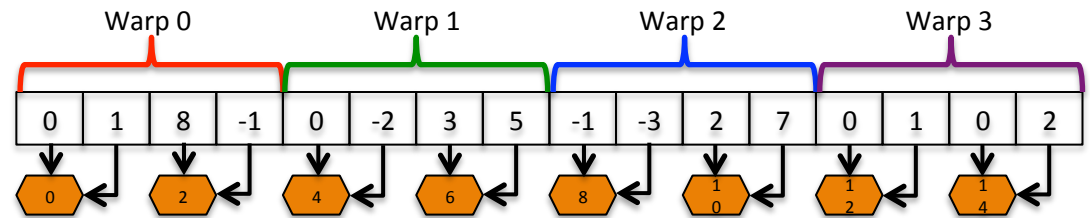
Warps

- Partitioning is always the same
- DO NOT rely on any execution ordering between warps
- Each warp is executed in a SIMD fashion (i.e. all threads within a warp must execute the same instruction at any given time).
 - Problem: branch divergence

Warp Example

$S = 1$

Thread ID	$2 * s$	Conditional
0	2	True
1	2	False
2	2	True
3	2	False
4	2	True
5	2	False
6	2	True
7	2	False
8	2	True
9	2	False
10	2	True
11	2	False
12	2	True
13	2	False
14	2	True
15	2	False

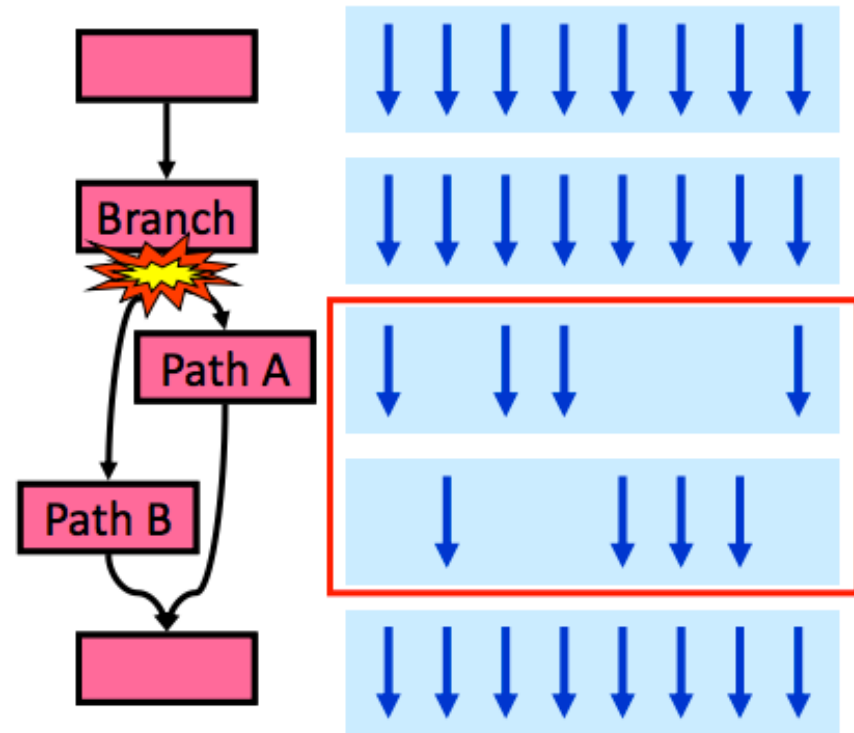


```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

- Each warp has to execute two conditions.
- See that each warp must have both because the conditional is true & false in each warp.

Branch Divergence in Warps

- When threads branch this causes warps to be branched, all threads still execute but some do nothing.



50% reduction in performance

Fixing Branch Divergence

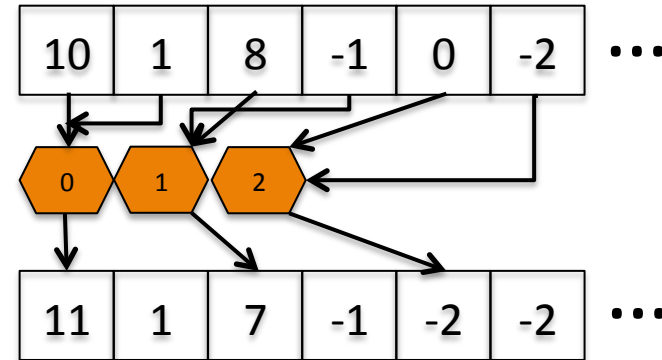
```
for(unsigned int s=1; s < blockDim.x; s *= 2) {  
    if (tid % (2*s) == 0) {  
        sdata[tid] += sdata[tid + s];  
    }  
    __syncthreads();  
}
```

```
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();  
}
```

Branch Divergence

S = 1

tid	2 * s	index	Cond
0	2	0	True
1	2	2	True
2	2	4	True
3	2	6	True
4	2	8	True
5	2	10	True
6	2	12	True
7	2	14	True
8	2	16	False
9	2	18	False
10	2	20	False
11	2	22	False
12	2	24	False
13	2	26	False
14	2	28	False
15	2	30	False



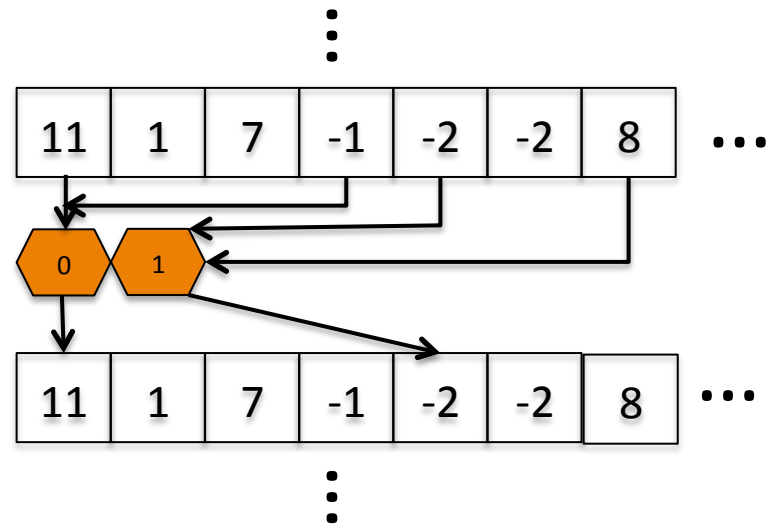
```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

- By shifting work to other threads we can 'group' conditional checks together.
- Allowing warps to not contain branching conditionals
- All threads in warp can not execute same instruction.

Branch Divergence

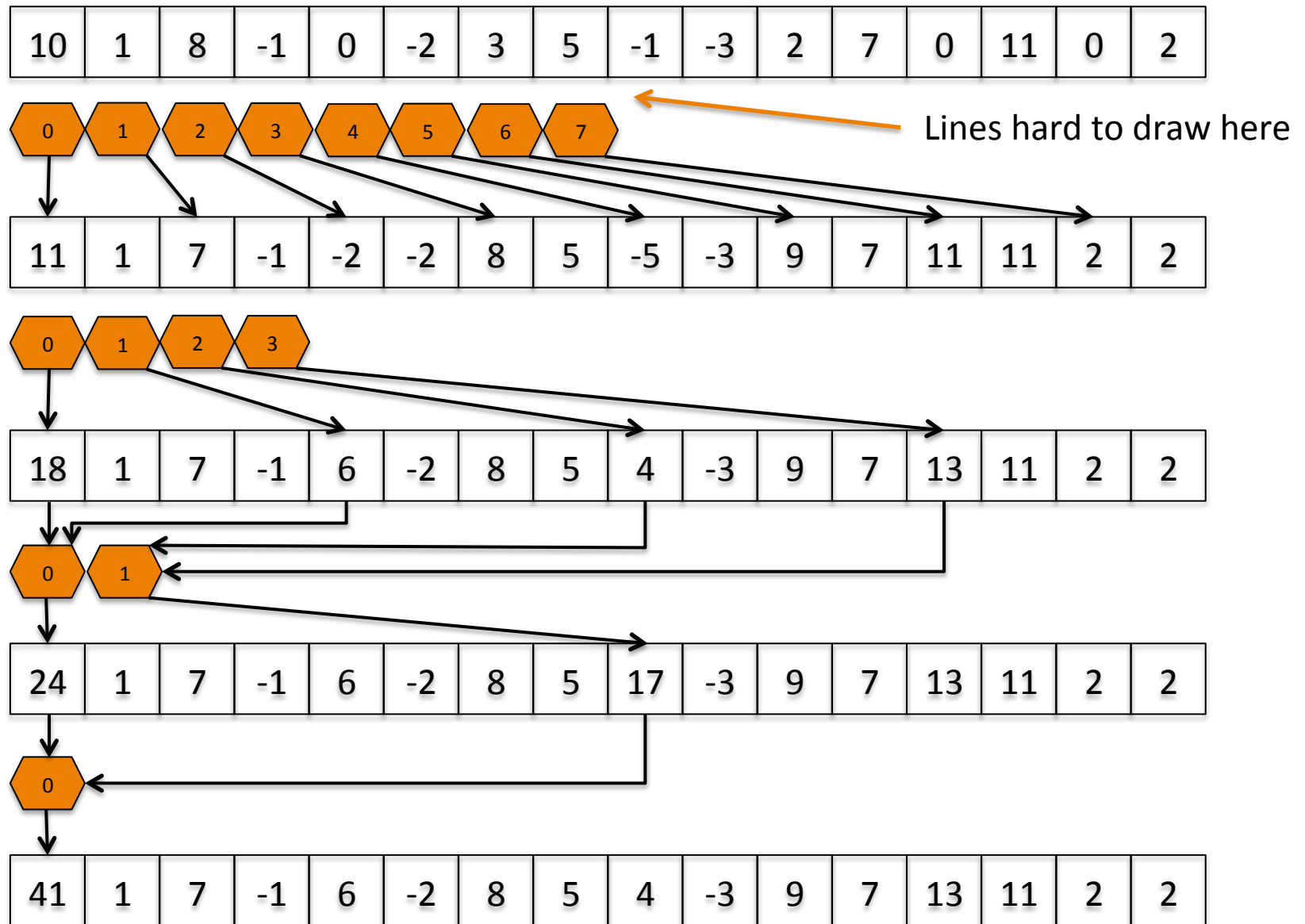
$s = 2$

tid	$2 * s$	index	Cond
0	4	0	True
1	4	4	True
2	4	8	True
3	4	12	True
4	4	16	False
5	4	20	False
6	4	24	False
7	4	28	False
8	4	32	False
9	4	36	False
10	4	40	False
11	4	44	False
12	4	48	False
13	4	52	False
14	4	56	False
15	4	60	False



```
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();
}
```


Fixing Branch Divergence



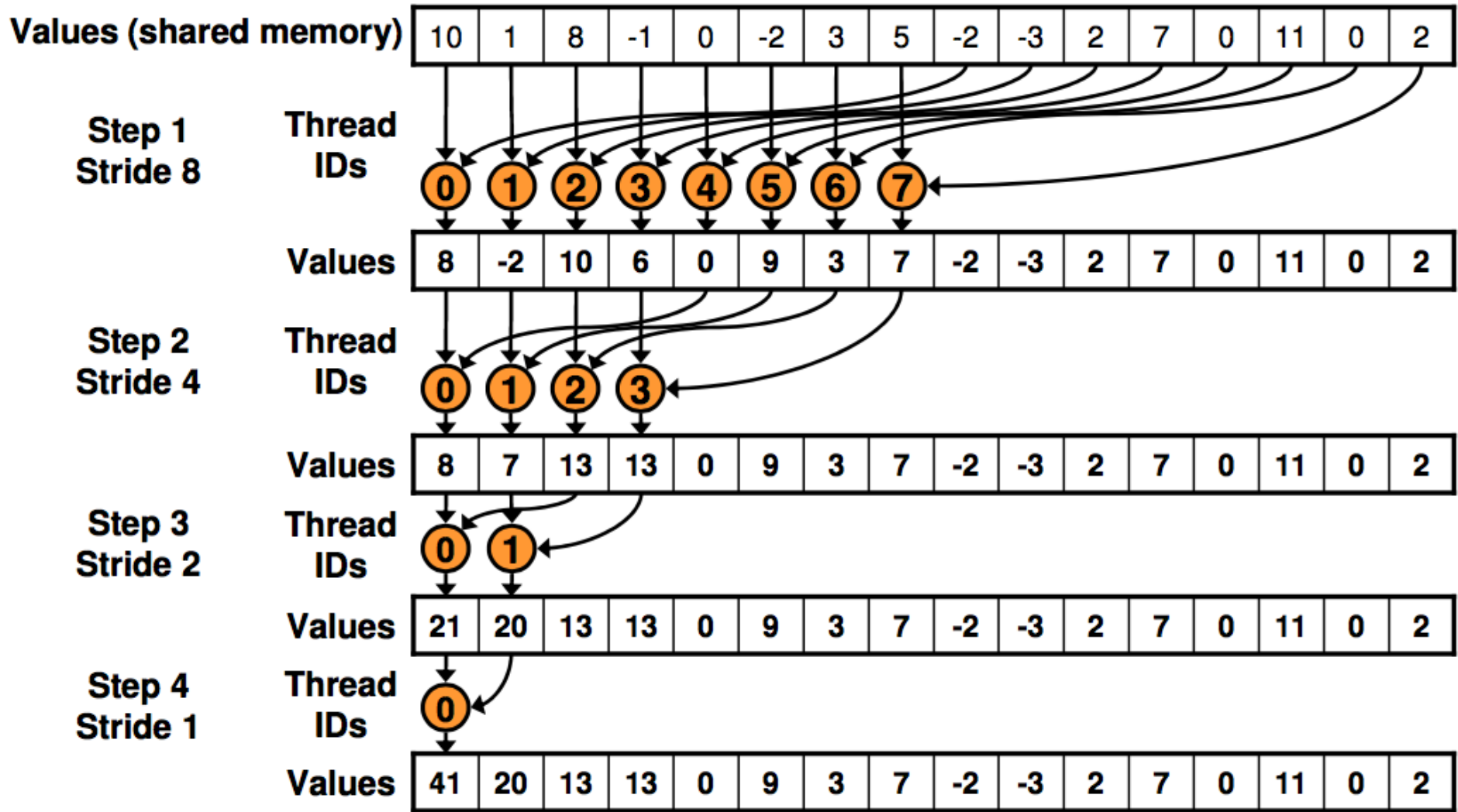
Fixing Branch Divergence



Shared Memory

- Parallel Memory Architecture
 - Memory is divided into banks
- Each bank can service one address per cycle
 - Each bank can be accessed simultaneously
- Multiple simultaneous accesses to the same bank result in a bank conflict.
 - Conflicts = Serialized Access

Sequential Addressing



Sequential Addressing

```
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();  
}
```

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

Idle Threads

Problem:

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

Half of threads are idle on first iteration!

Do Add During Load

```
// each thread loads 1 element from global
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

```
// perform first level of reduction,
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();
```