Parallel Reduction with CUDA

Julien Demouth, Nvidia

Problem

- Input: Array of ints `a[0],a[1], ...,a[n-1]`

1	4	3	2	8	6	3	2	1	0	3	2	1	3	2	3	2	9	1	2	3	4	5	6	1	1	2	3	0	0	2	1

- Output: The sum of the elements

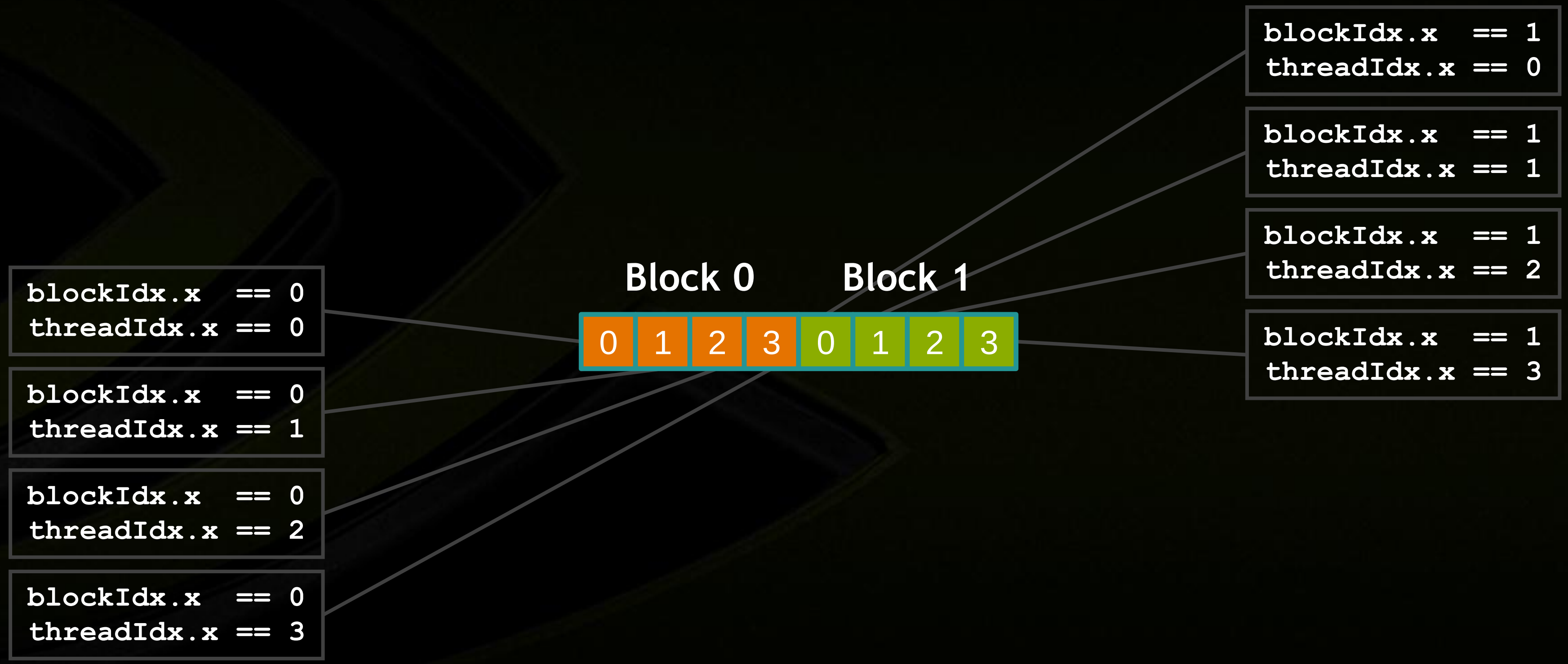
```
int sum = 0;
for( int i = 0 ; i < N ; ++i )
  sum += a[i];
```

Outline of the Algorithm

- 1st Step: Each thread computes its partial sum
- 2nd Step: Each block of threads computes its partial sum
- 3rd Step: One block computes the final sum

Blocks and Threads

- In those slides we use 2 blocks and 4 threads per block



1ST STEP

1st Step: Threads Compute Their Partial Sums

- **Each thread initializes its partial sum to 0**

	Block 0				Block 1			
	0	1	2	3	0	1	2	3
`my_sum:`	0	0	0	0	0	0	0	0

```
int my_sum = 0;
```

- **Each thread reads its element from `a` and update its sum**

Block 0

0	1	2	3
1	4	3	2

Block 1

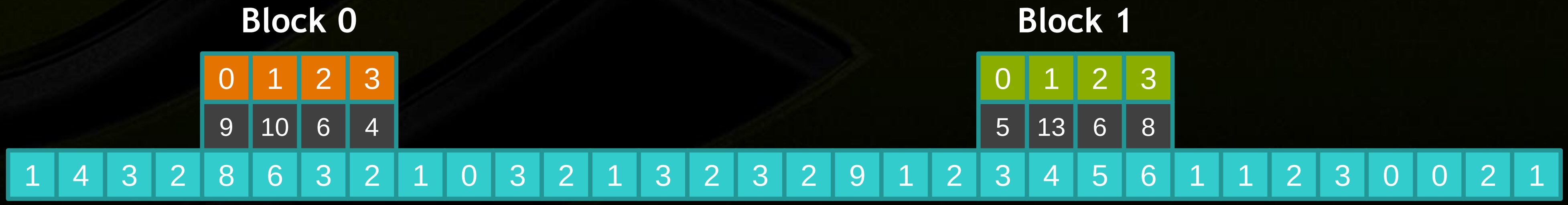
0	1	2	3
2	9	1	2

1	4	3	2	8	6	3	2	1	0	3	2	1	3	2	3	2	9	1	2	3	4	5	6	1	1	2	3	0	0	2	1

1st Step: Threads Compute Their Partial Sums

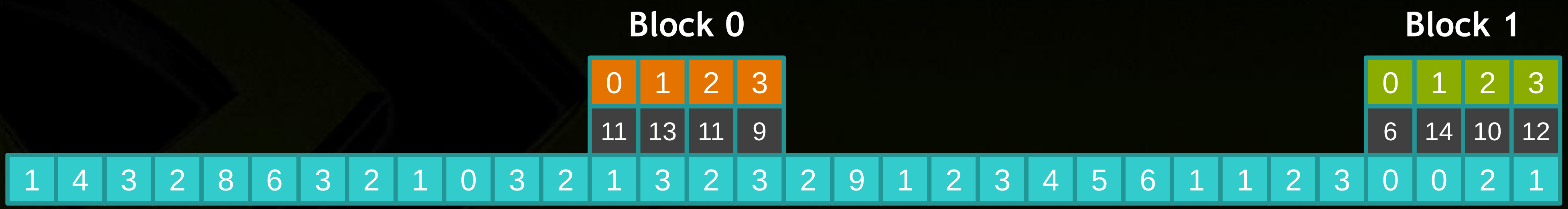
- **Each thread moves to its next element**

- **And updates its partial sum**



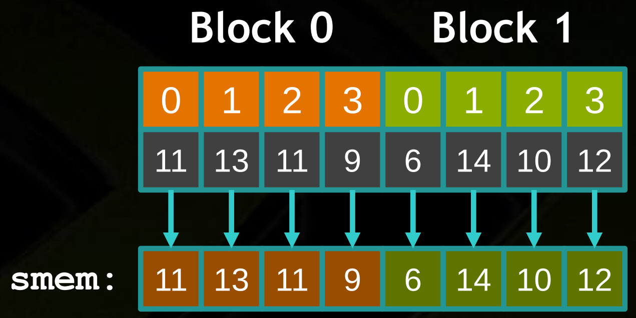
1st Step: Threads Compute Their Partial Sums

- And so on, until all threads have their partial sums



1st Step: Threads Compute Their Partial Sums

- **Each thread writes its partial sum to shared memory**

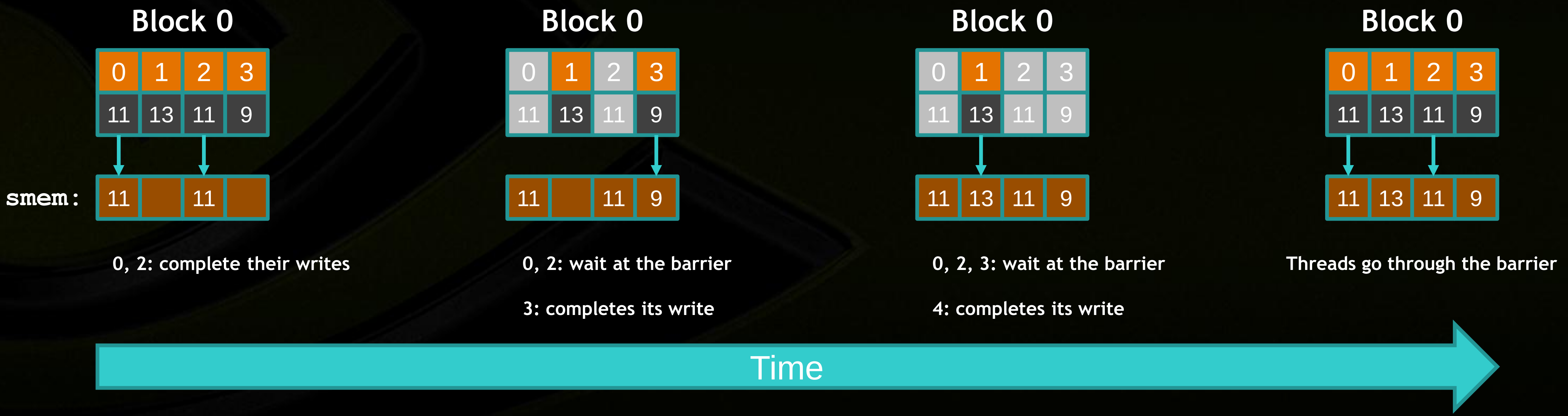


- **Shared memory is shared by threads in the *same* block**
- **No synchronization: *Threads do not know what others are doing***

Advanced note: Inside a block, threads of a same warp are implicitly synchronized. On current HW a warp contains 32 threads.

1st Step: Threads Compute Their Partial Sums

- We want `smem` to contain all the sums of the threads in the block
- We use `__syncthreads()` to create a synchronization barrier



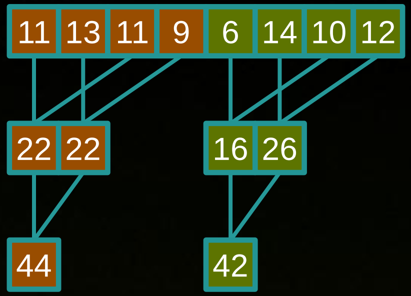
Note: __syncthreads synchronizes the threads of a same block. There is no instruction to synchronize threads of different blocks from a kernel.

1st Step: Threads Compute Their Partial Sums

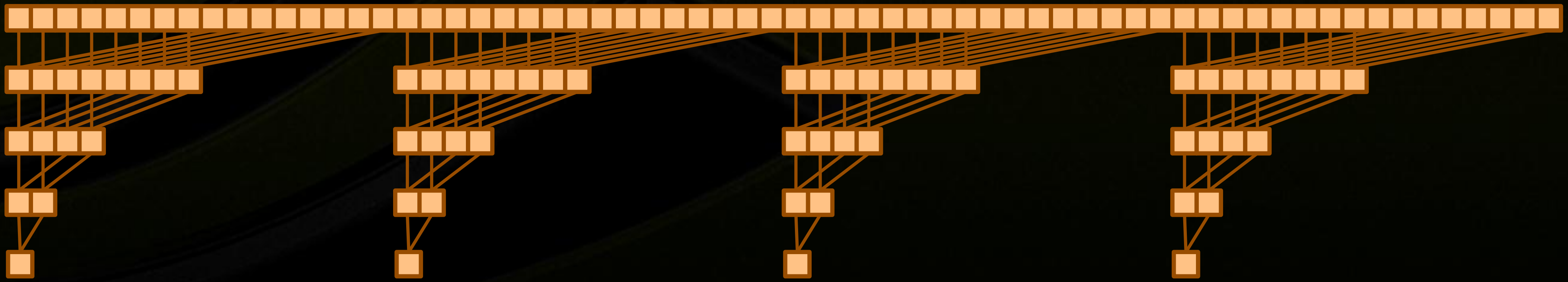
- **Code summary**

2ND STEP

2^{nd} Step: Blocks Compute Their Partial Sums

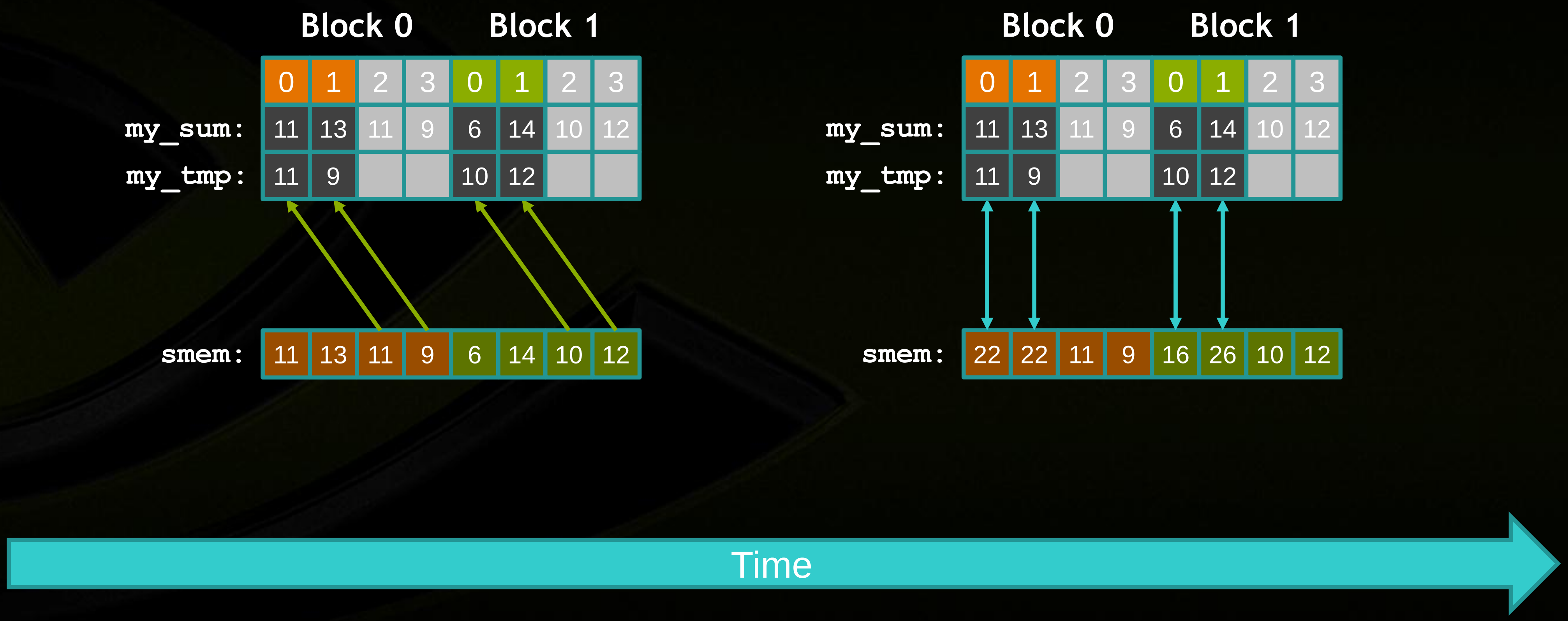


- **With more threads and more blocks**



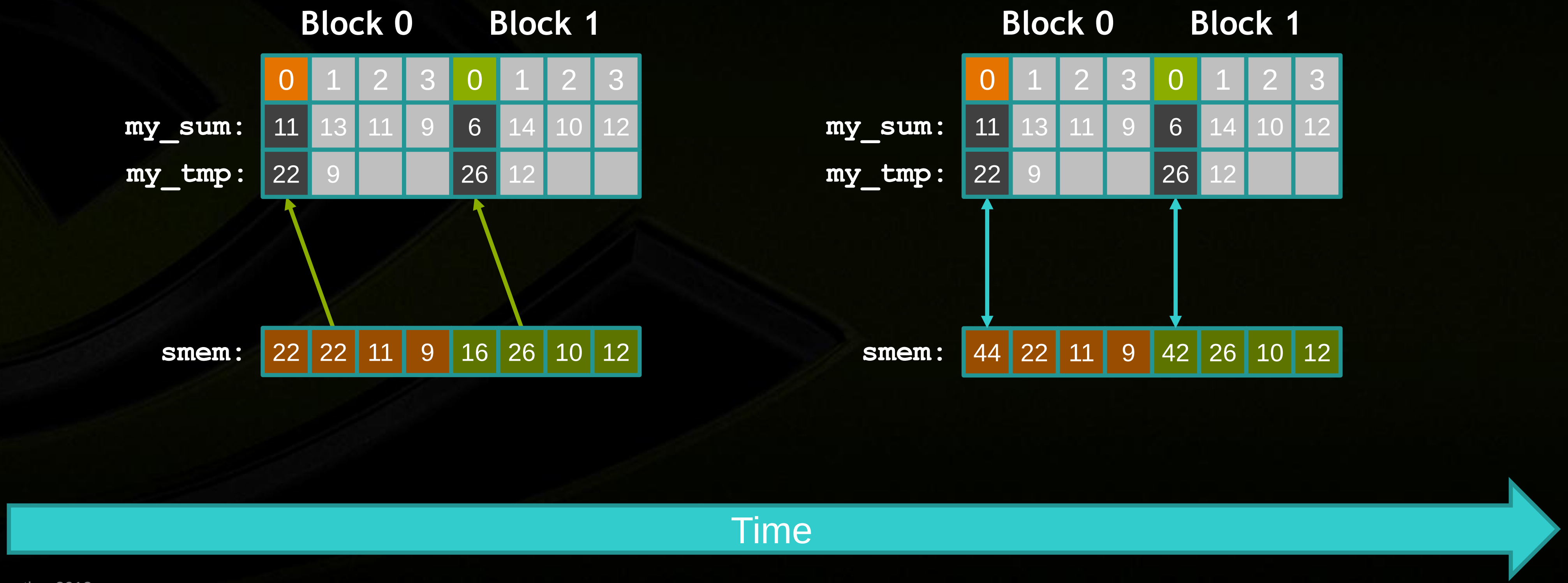
2nd Step: Blocks Compute Their Partial Sums

- Half of the threads updates the sum in `smem`



2nd Step: Blocks Compute Their Partial Sums

- We want `smem` to contain all the sums: `__syncthreads()`
- 1/4th of the threads updates the sum in `smem`



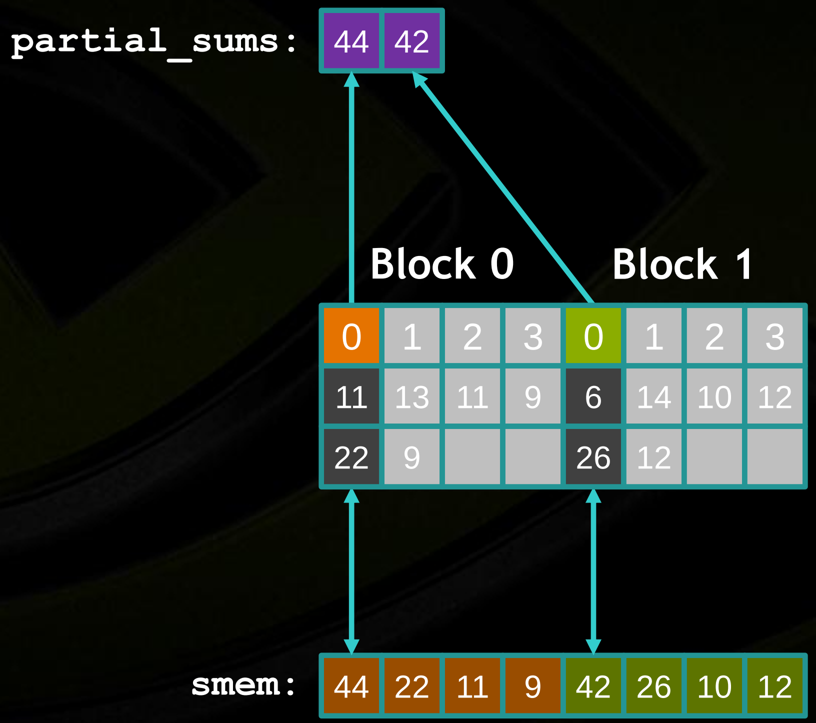
2nd Step: Blocks Compute Their Partial Sums

- We keep doing that until there is no more work to do

- Code summary

2nd Step: Blocks Compute Their Partial Sums

- Block leaders stores the sum of the block in global memory



2^{nd} Step: Blocks Compute Their Partial Sums

- 1^{st} and 2^{nd} steps are implemented in one CUDA kernel

- We launch the kernel

3RD STEP

3^{rd} Step: 1^{st} Block Computes The Sum

- We can call the same kernel with different arguments

```
reduce_kernel<<<1, BLOCK_SIZE, smem_size>>>( grid_size,
                                             partial_sums,
                                             partial_sums,
                                             block_ranges );
```

- The result is in `partial_sums[0]`

Parallel Scan with CUDA

Problem

- Input: Array of ints `a[0],a[1], ...,a[n-1]`

1	4	3	2	8	6	3	2	1	0	3	2	1	3	2	3	2	9	1	2	3	4	5	6	1	1	2	3	0	0	2	1

- Output: The sums `b[i] = a[0] + … + a[i-1]`

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

Outline of the Algorithm

- 1st Step: Each block computes its partial sum
  - Same kernel as the 1st kernel of the reduction
- 2nd Step: One block scans the block sums (global scans)
- 3rd Step: Each block scans its elements and uses its global scan

Scan Primitive: Parallel Scan in a Block

- **Each thread loads its item from GMEM**

```
int my_sum = in_buffer[idx];
```

my_sum:

2	3	1	4

- **We allocate a buffer in SMEM of size 2 x blockDim.x**

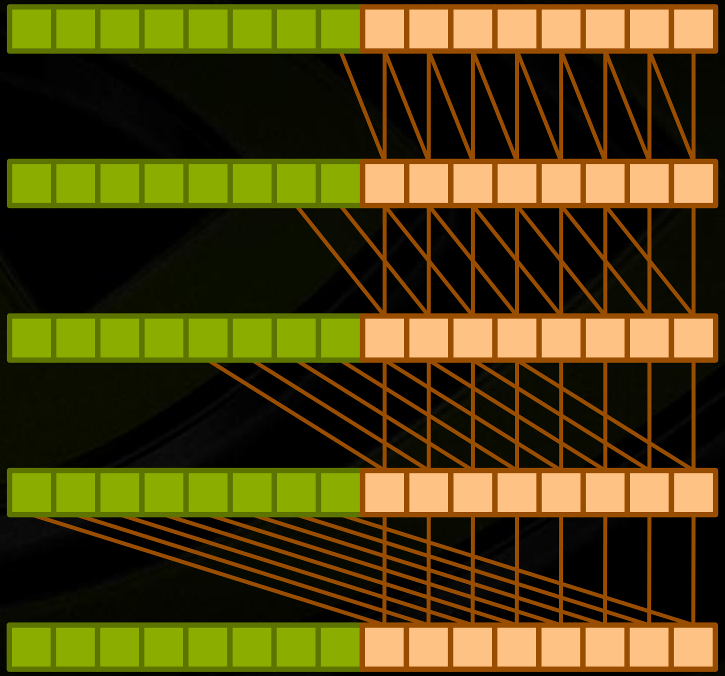
smem:

0	1	2	3	4	5	6	7
0	0	0	0	2	3	1	4

- **Store 0s in the 1st half of `smem` and its items in the 2nd half**

Scan Primitive: Parallel Scan in a Block

- **Parallel scan pattern**



Homework

- **How can you implement inclusive scan?**

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

- **How can you optimize your implementation?**

Parallel Radix Sort with CUDA

Outline of the Algorithm

- Loop on the bits:
- 1^{st} Step: Each block computes its partial sums
  - One sum for each combination of bits
  - Almost the same kernel as the 1^{st} kernel of the reduction
- 2^{nd} Step: One block scans the block sums (global scans)
- 3^{rd} Step: Each block scatters its elements

Sort Primitive: Where to Store Counters

- Each thread has one counter per bit combination

```
int my_counters[NUM_COUNTERS];
```

- Problem: It's not possible to dynamically address registers

```
my_counters[(item & mask) >> i]++;
```

- Solution: Use SMEM to store counters

```
__shared__ int s_counters[NUM_COUNTERS][BLOCK_DIM];

s_counters[(item & mask) >> i][threadIdx.x]++;
```