CS 193G

Lecture 5: Parallel Patterns I

Getting out of the trenches

- So far, we've concerned ourselves with low-level details of kernel programming
  - Mapping of threads to work
  - Launch grid configuration
  - `__shared__` memory management
  - Resource allocation
- Lots of moving parts
- Hard to see the forest for the trees

CUDA Madlibs

```
__global__ void foo(...)
{
  extern __shared__ smem[];
  int i = ???

  // now what???
}
...
int B = ???
int N = ???
int S = ???
foo<<<B,N,S>>>();
```

Parallel Patterns

- Think at a higher level than individual CUDA kernels
- Specify what to compute, not how to compute it
- Let programmer worry about algorithm
- Defer pattern implementation to someone else

Common Parallel Computing Scenarios

- Many parallel threads need to generate a single result
  → Reduce

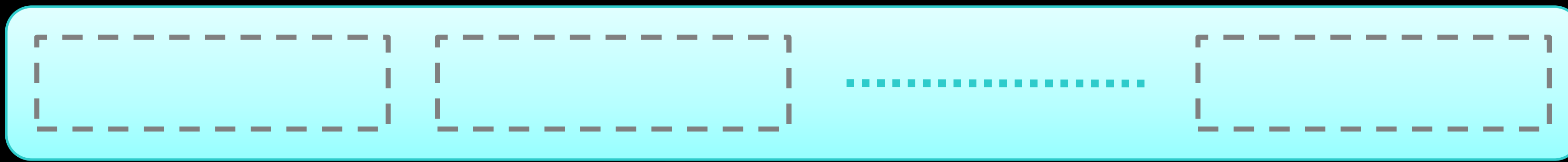
- Many parallel threads need to partition data
  → Split

- Many parallel threads produce variable output / thread
  → Compact / Expand

Primordial CUDA Pattern: Blocking

- Partition data to operate in well-sized blocks
  - Small enough to be staged in shared memory
  - Assign each data partition to a thread block
  - No different from cache blocking!
- Provides several performance benefits
  - Have enough blocks to keep processors busy
  - Working in shared memory cuts memory latency dramatically
  - Likely to have coherent access patterns on load/store to shared memory

Primordial CUDA Pattern: Blocking

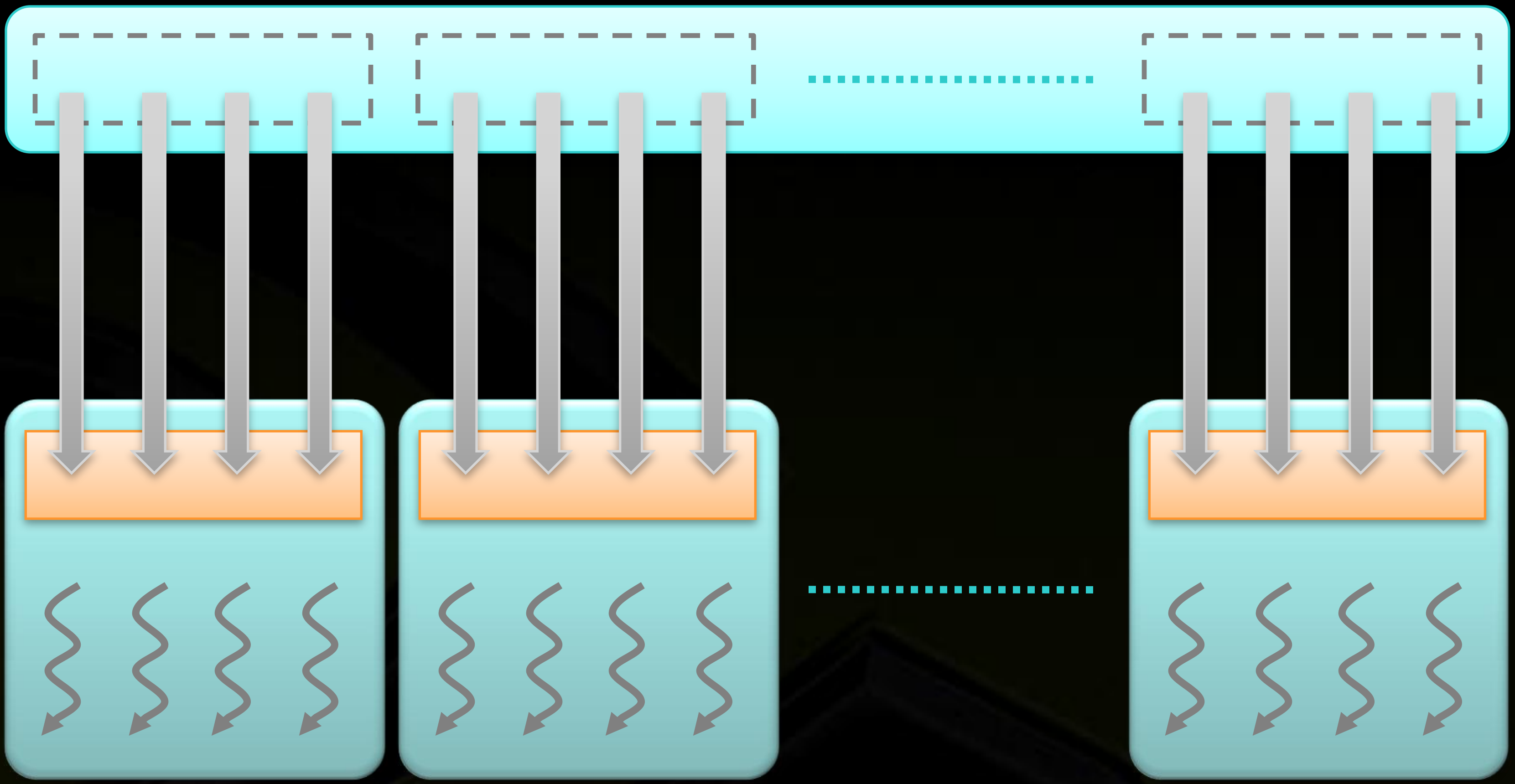


- Partition data into subsets that fit into shared memory

Primordial CUDA Pattern: Blocking

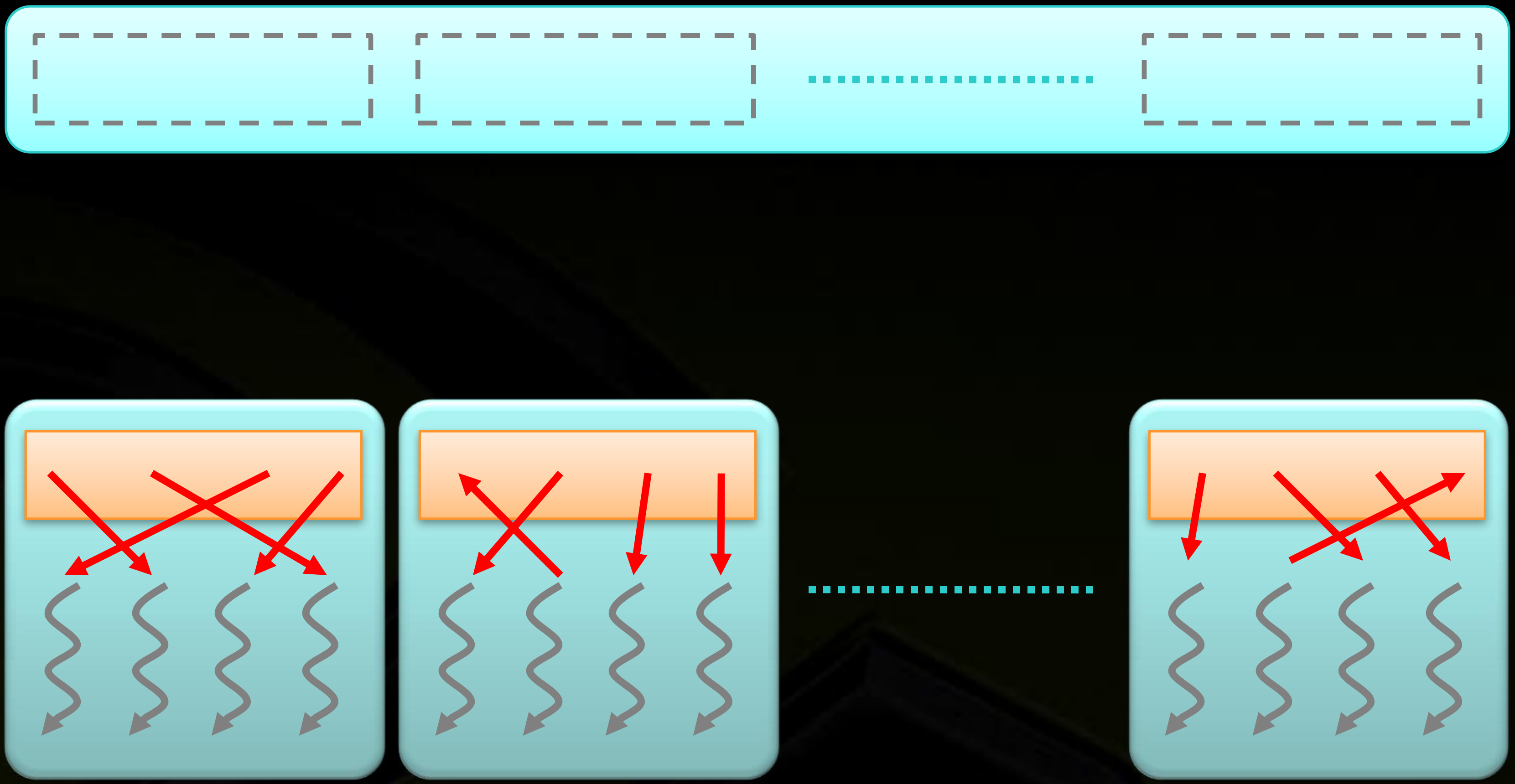
- Handle each data subset with one **thread block**

Primordial CUDA Pattern: Blocking



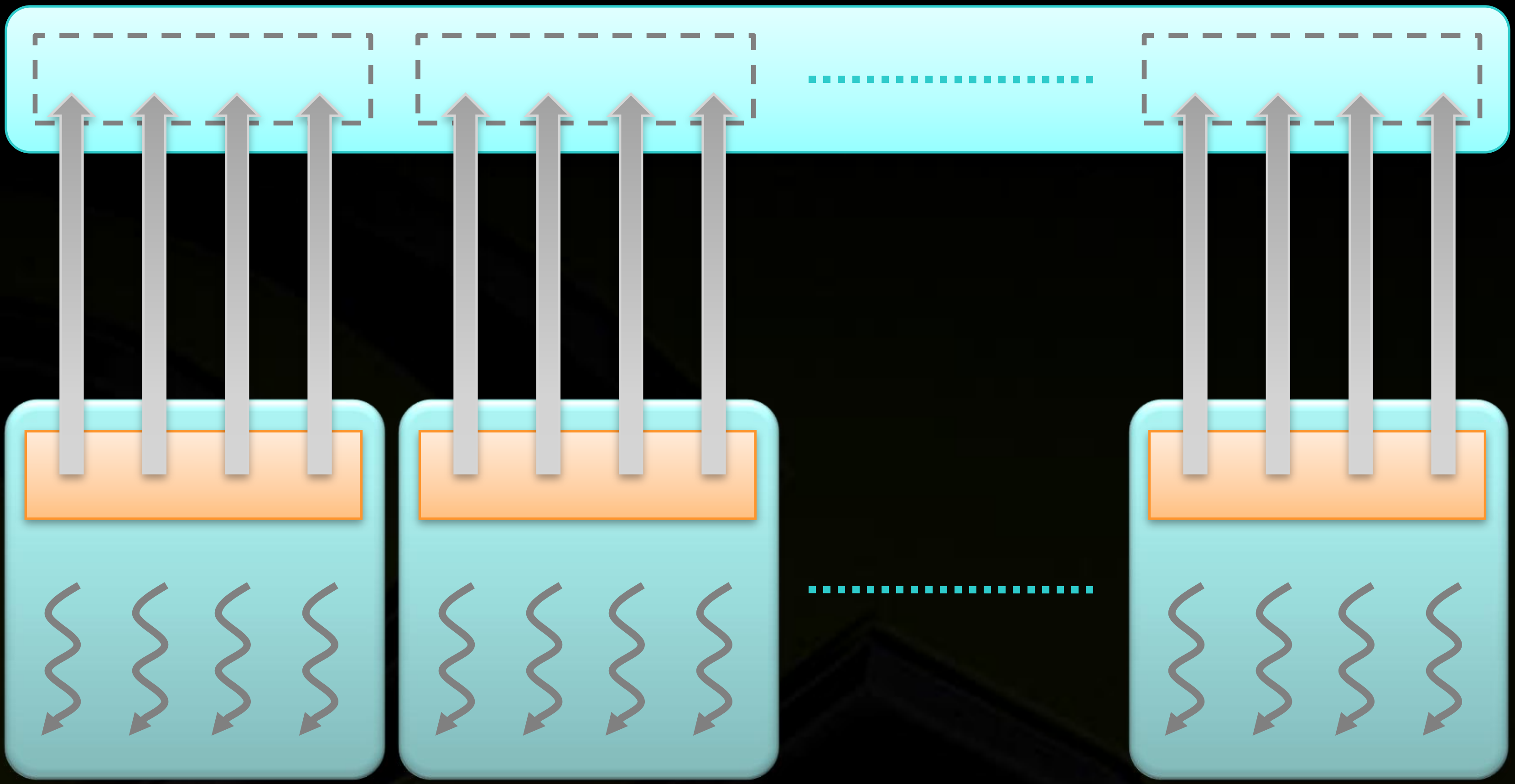
- Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism

Primordial CUDA Pattern: Blocking



- Perform the computation on the subset from **shared memory**

Primordial CUDA Pattern: Blocking



- Copy the result from shared memory back to global memory

Primordial CUDA Pattern: Blocking

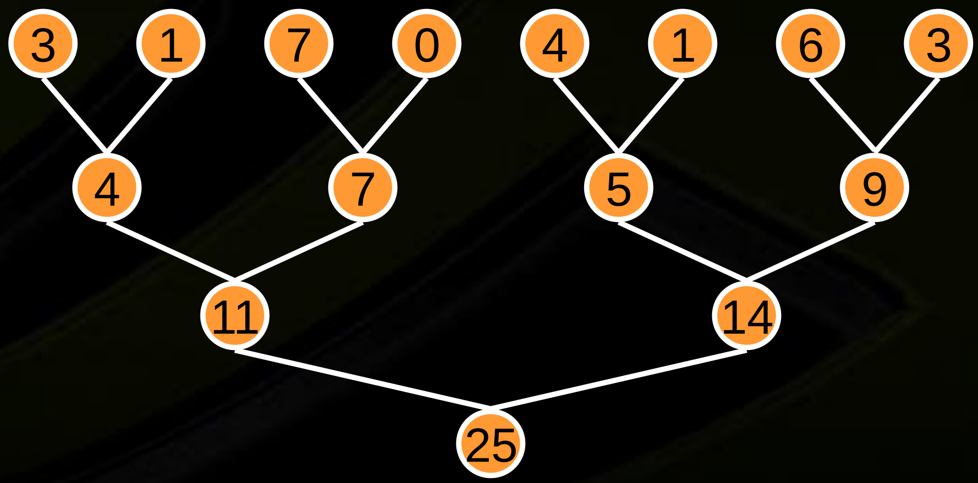
- All CUDA kernels are built this way
  - Blocking may not matter for a particular problem, but you're still forced to think about it
  - Not all kernels require `__shared__` memory
  - All kernels do require registers
- All of the parallel patterns we'll discuss have CUDA implementations that exploit blocking in some fashion

Reduction

- **Reduce vector to a single value**
  - **Via an associative operator (+, *, min/max, AND/OR, …)**
  - **CPU: sequential implementation**

```
for(int i = 0, i < n, ++i) ...
```

  - **GPU: "tree"-based implementation**

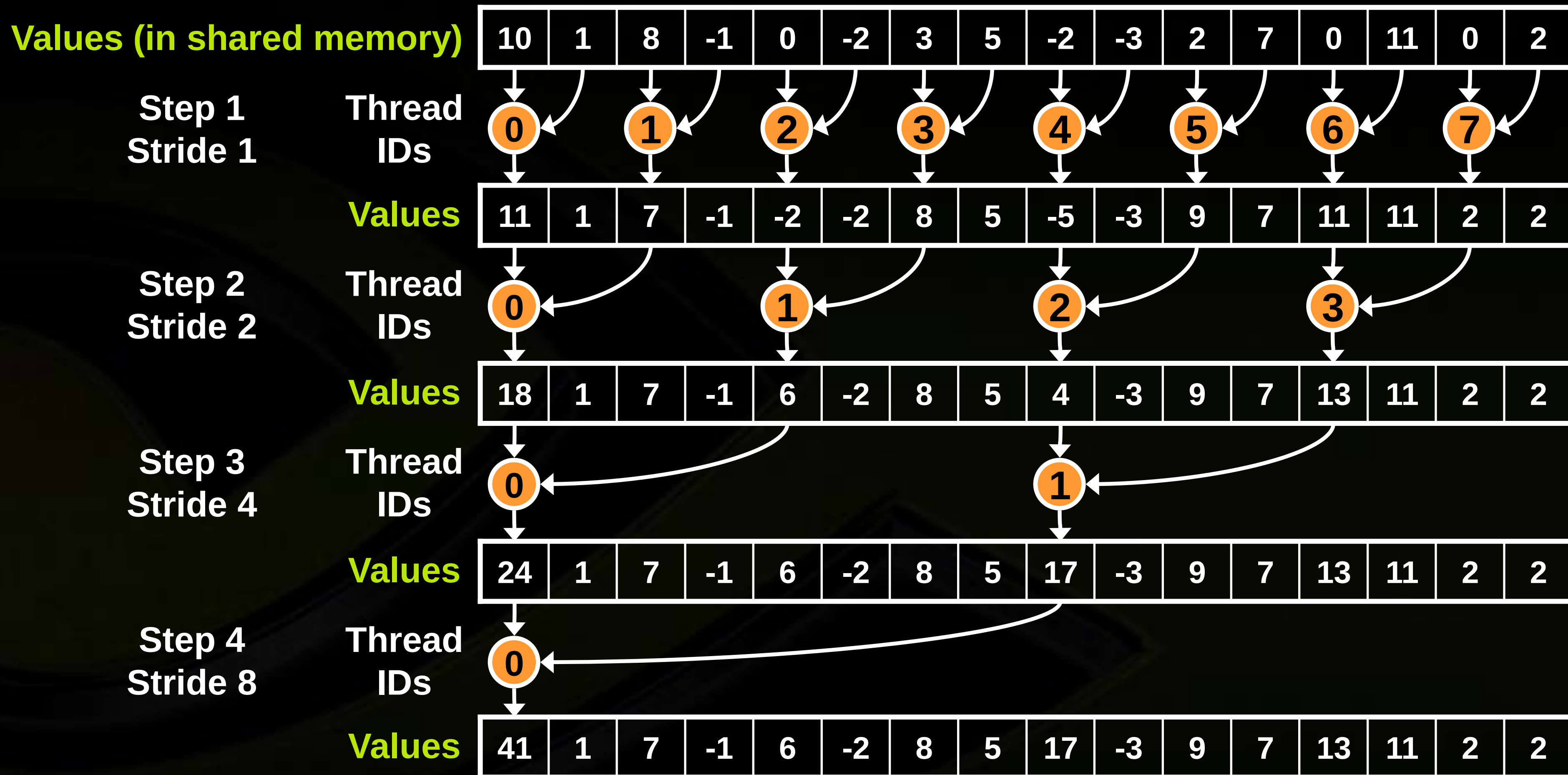


Serial Reduction

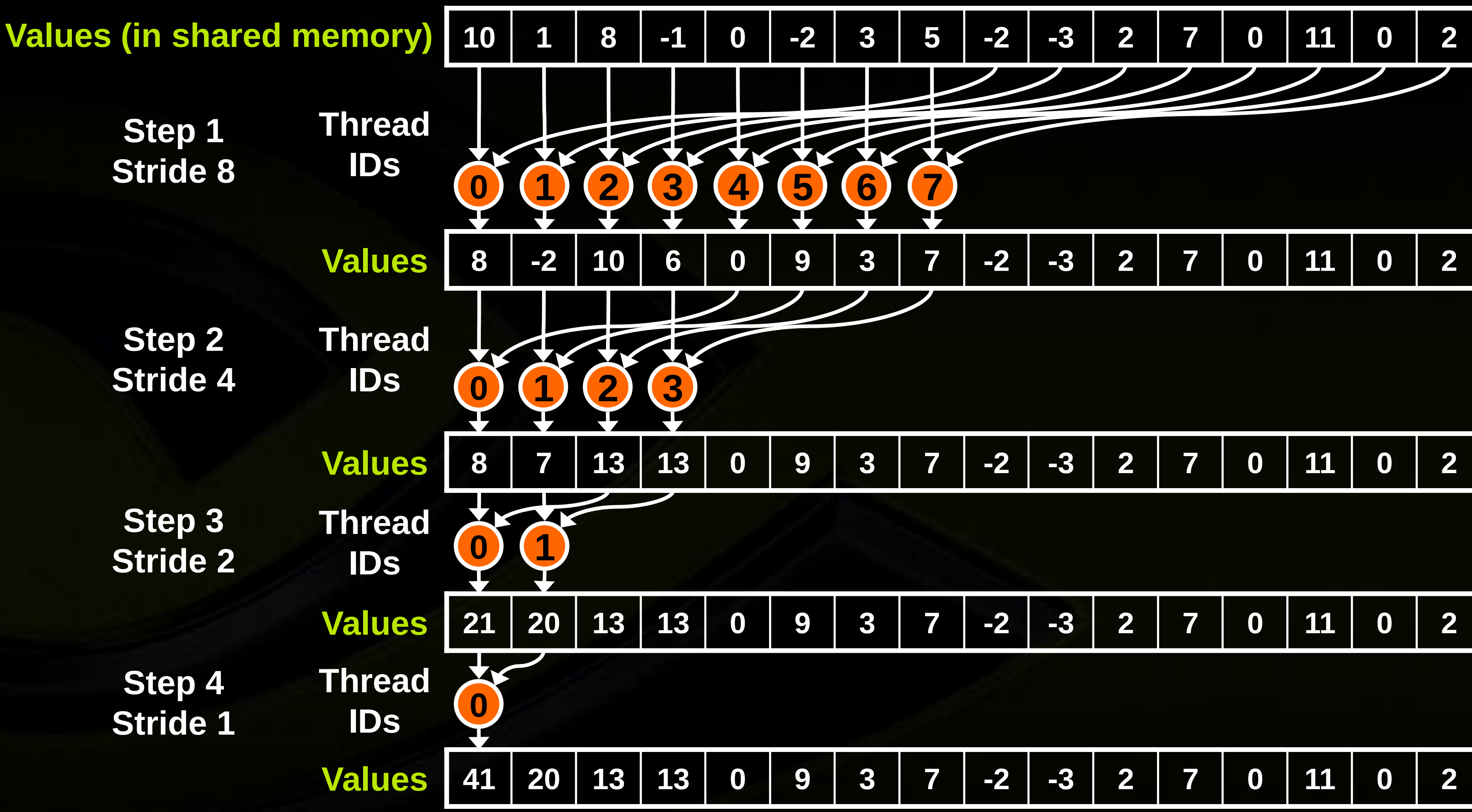
```
// reduction via serial iteration
float sum(float *data, int n)
{
  float result = 0;
  for(int i = 0; i < n; ++i)
  {
    result += data[i];
  }

  return result;
}
```

Parallel Reduction – Interleaved



Parallel Reduction – Contiguous



CUDA Reduction

```
__global__ void block_sum(float *input,
                          float *results,
                          size_t n)
{
  extern __shared__ float sdata[];
  int i = ..., int tx = threadIdx.x;

  // load input into __shared__ memory
  float x = 0;
  if(i < n)
    x = input[i];
  sdata[tx] = x;
  __syncthreads();
```

CUDA Reduction

```
// block-wide reduction in __shared__ mem
for(int offset = blockDim.x / 2;
    offset > 0;
    offset >>= 1)
{
  if(tx < offset)
  {
    // add a partial sum upstream to our own
    sdata[tx] += sdata[tx + offset];
  }
  __syncthreads();
}
```

CUDA Reduction

```
  // finally, thread 0 writes the result
  if(threadIdx.x == 0)
  {
    // note that the result is per-block
    // not per-thread
    results[blockIdx.x] = sdata[0];
  }
}
```

An Aside

```
// is this barrier divergent?
for(int offset = blockDim.x / 2;
    offset > 0;
    offset >>= 1)
{
  ...
  __syncthreads();
}
```

An Aside

```
// what about this one?
__global__ void do_i_halt(int *input)
{
  int i = ...
  if(input[i])
  {
    ...
    __syncthreads(); // a divergent barrier
                     // hangs the machine
  }
}
```

CUDA Reduction

```
// global sum via per-block reductions
float sum(float *d_input, size_t n)
{
  size_t block_size = ..., num_blocks = ...;

  // allocate per-block partial sums
  // plus a final total sum
  float *d_sums = 0;
  cudaMalloc((void**)&d_sums,
    sizeof(float) * (num_blocks + 1));
  ...
```

CUDA Reduction

```
// reduce per-block partial sums
int smem_sz = block_size*sizeof(float);
block_sum<<<num_blocks,block_size,smem_sz>>>
  (d_input, d_sums, n);

// reduce partial sums to a total sum
block_sum<<<1,block_size,smem_sz>>>
  d_sums, d_sums + num_blocks, num_blocks);

// copy result to host
float result = 0;
cudaMemcpy(&result, d_sums+num_blocks, ...);
return result;
```

Caveat Reductor

- What happens if there are too many partial sums to fit into `__shared__` memory in the second stage?
- What happens if the temporary storage is too big?
- Give each thread more work in the first stage
  - Sum is associative & commutative
  - Order doesn't matter to the result
  - We can schedule the sum any way we want
    → serial accumulation before block-wide reduction
- Exercise left to the hacker

Parallel Reduction Complexity

- **Log(*N*) parallel steps, each step *S* does $N/2^S$ independent ops**
  - **Step Complexity is 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**
  - **i.e. 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*)**
  - **Compare to O(*N*) for sequential reduction**

Split Operation

- **Given:array of true and false elements (and payloads)**

Flag	T	F	F	T	F	F	T	F
Payload	3	1	7	0	4	1	6	3

- **Return an array with all true elements at the beginning**

T	T	T	F	F	F	F	F
3	0	6	1	7	4	1	3

- **Examples: sorting, building trees**

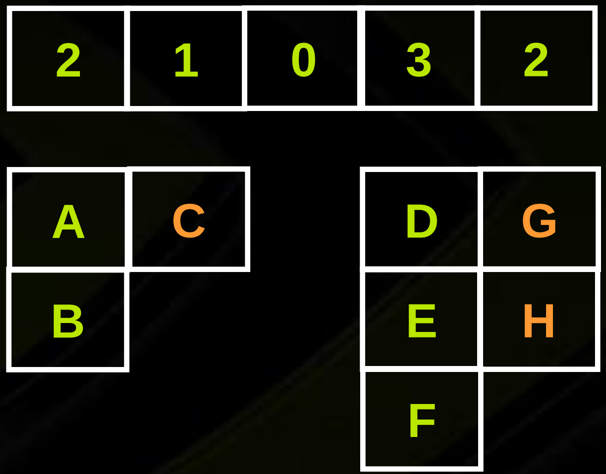
Variable Output Per Thread: Compact

- Remove null elements

- Example: collision detection

Variable Output Per Thread: General Case

- **Reserve Variable Storage Per Thread**



- **Example: binning**

Split, Compact, Expand

- **Each thread must answer a simple question:**

  “Where do I write my output?”

- **The answer depends on what other threads write!**

- ***Scan* provides an efficient parallel answer**

Scan (a.k.a. Parallel Prefix Sum)

- Given an array $A = [a_0, a_1, \ldots, a_{n-1}]$
  and a binary associative operator $\oplus$ with identity I,

  $$\text{scan}(A) = [I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$$

- Prefix sum: if $\oplus$ is addition, then scan on the series

3	1	7	0	4	1	6	3

returns the series

0	3	4	11	11	15	16	22

Applications of Scan

- **Scan is a simple and useful parallel building block for many parallel algorithms:**
  - Radix sort
  - Quicksort (seg. scan)
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Run-length encoding
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Histograms
  - Allocation
  - Etc.
- **Fascinating, since scan is unnecessary in sequential computing!**

Serial Scan

```
int input[8] = {3, 1, 7, 0, 4, 1, 6, 3};
int result[8];
int running_sum = 0;
for(int i = 0; i < 8; ++i)
{
  result[i] = running_sum;
  running_sum += input[i];
}

// result = {0, 3, 4, 11, 11, 15, 16, 22}
```

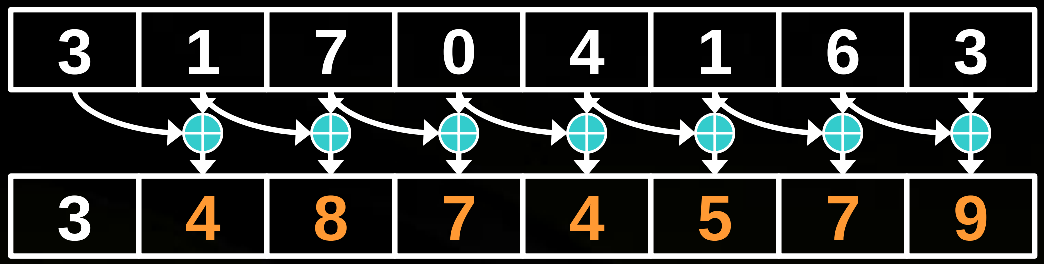
A Scan Algorithm – Preview

3	1	7	0	4	1	6	3

Assume array is already in shared memory

See Harris, M., S. Sengupta, and J.D. Owens. “Parallel Prefix Sum (Scan) in CUDA”, *GPU Gems 3*

A Scan Algorithm – Preview

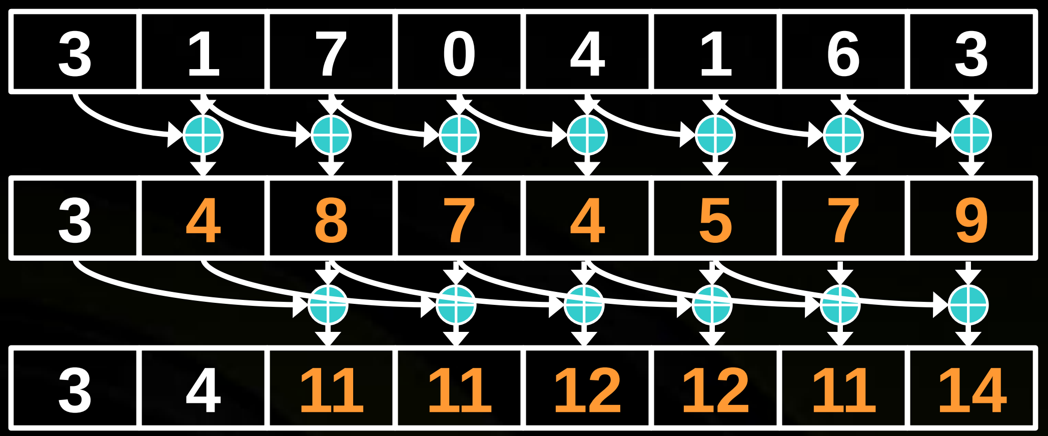


Iteration 0, *n-1* threads

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* elements away to its own value

A Scan Algorithm – Preview

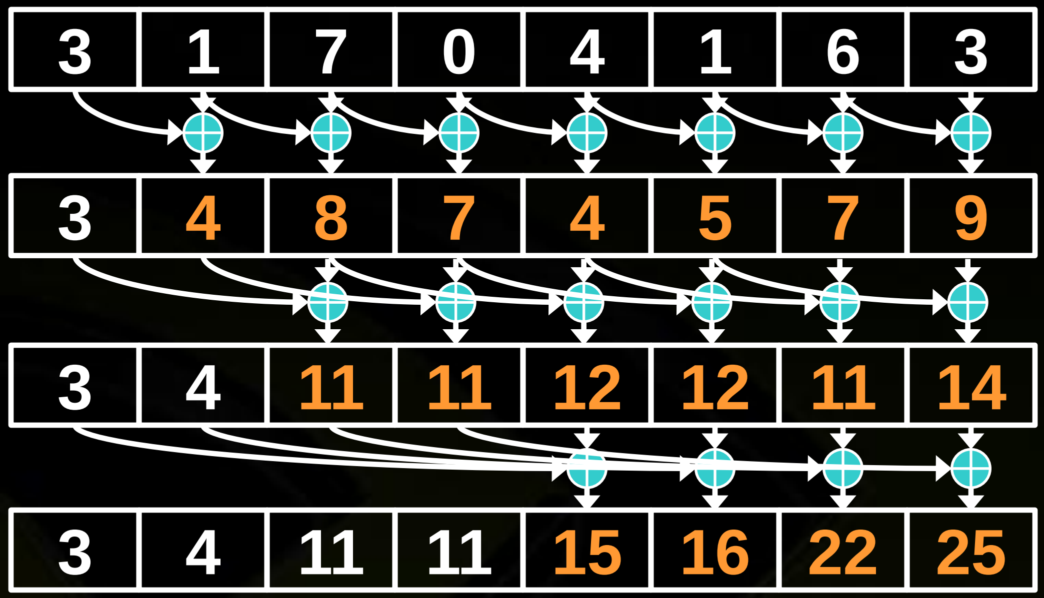


Iteration 1, *n-2* threads

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *offset* elements away to its own value

A Scan Algorithm – Preview



Iteration *i*, *n-2^i* threads

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *offset* elements away to its own value.

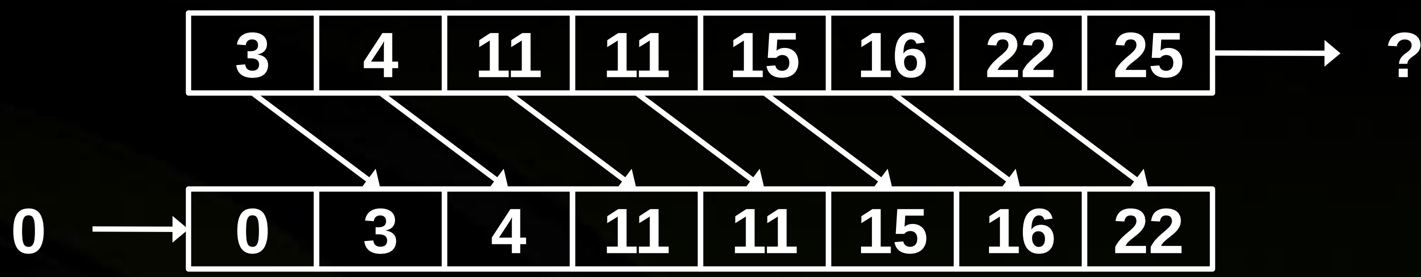
Note that this algorithm operates in-place: no need for double buffering

A Scan Algorithm – Preview

3	4	11	11	15	16	22	25

- We have an inclusive scan result

A Scan Algorithm – Preview



- For an **exclusive** scan, right-shift through `__shared__` memory
- Note that the unused final element is also the sum of the entire array
  - Often called the “carry”
  - Scan & reduce in one pass

CUDA Block-wise Inclusive Scan

```
__global__ void inclusive_scan(int *data)
{
  extern __shared__ int sdata[];

  unsigned int i = ...

  // load input into __shared__ memory
  int sum = input[i];
  sdata[threadIdx.x] = sum;
  __syncthreads();
  ...
```

CUDA Block-wise Inclusive Scan

```
for(int o = 1; o < blockDim.x; o <<= 1)
{
  if(threadIdx.x >= o)
    sum += sdata[threadIdx.x - o];

  // wait on reads
  __syncthreads();

  // write my partial sum
  sdata[threadIdx.x] = sum;

  // wait on writes
  __syncthreads();
}
```

CUDA Block-wise Inclusive Scan

```
    // we're done!
    // each thread writes out its result
    result[i] = sdata[threadIdx.x];
}
```

Results are Local to Each Block

```
Block 0

Input:
 5   5   4   4   5   4   0   0   4   2   5   5   1   3   1   5
Result:
 5  10  14  18  23  27  27  27  31  33  38  43  44  47  48  53



Block 1

Input:
 1   2   3   0   3   0   2   3   4   4   3   2   2   5   5   0
Result:
 1   3   6   6   9   9  11  14  18  22  25  27  29  34  39  39
```

Results are Local to Each Block

- Need to propagate results from each block to all subsequent blocks
- 2-phase scan
  1. Per-block scan & reduce
  2. Scan per-block sums
- Final update propagates phase 2 data and transforms to exclusive scan result
- Details in MP3

Summing Up

- Patterns like reduce, split, compact, scan, and others let us reason about data parallel problems abstractly
- Higher level patterns are built from more fundamental patterns
- Scan in particular is fundamental to parallel processing, but unnecessary in a serial world
- Get others to implement these for you!
  → but not until after MP3