ECE408/CS483/CSE408

Applied Parallel Programming

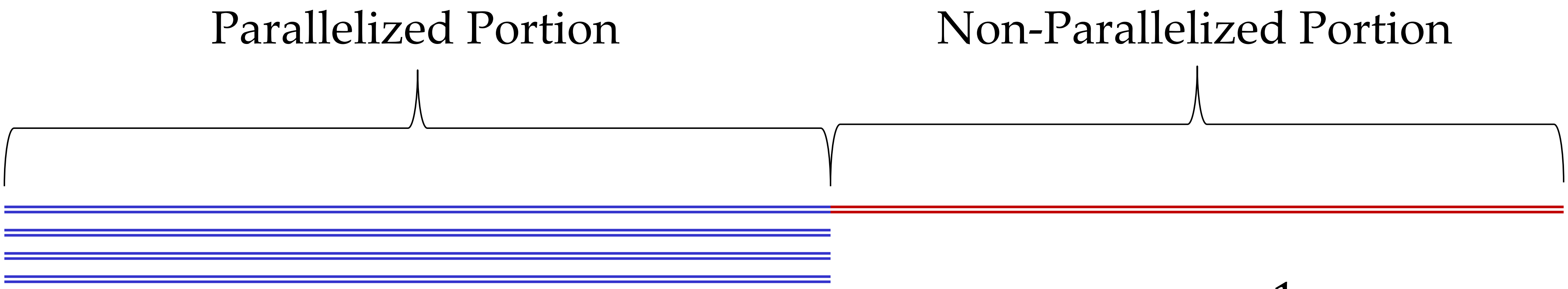
Lecture 15
Parallel Computation Patterns – Reduction Trees

Course Reminders

- Exam 1: Tues Oct 10 @ 7pm
  - Room assignments are available on Canvas
  - 1 sheet of handwritten notes
- Project Milestone 1: CPU Convolution, profiling
  - Due tomorrow (Friday, Oct 13)
  - Project details are posted in GitHub
- Lab 5.1 due Oct 20, will be made available next week

Amdahl's Law

- First articulated by Gene Amdahl, computer pioneer, 1967



$$Speedup = \frac{1}{(1-k) + (\frac{k}{p})}$$

$$k = parallelized\ portion$$
$$p = parallel\ resources$$
$$Assuming\ a\ fixed\ problem\ size$$

Reductions are everywhere… often the final stage of parallel computation

- "Reduce" a set of input values into a singular value
  - Max
  - Min
  - Sum
  - Product
  - Dot Product

- Basically, any operator/function that satisfies the following:
  - Is associative and commutative
  - Has a well-defined identity value (e.g., 0 for sum)

Partition and Summarize

- A commonly used strategy for processing large input data sets
  - If the basic operation is associative and commutative (i.e., reorderable)
    - Partition the data set into smaller chunks
    - Have each thread to process a chunk
    - Use a reduction tree to summarize the results from each chunk into the final answer

- Big Data cloud frameworks (from Google and others) provide support for this pattern

Reduction enables other techniques

- Reduction is often needed as the final stage of parallel computation

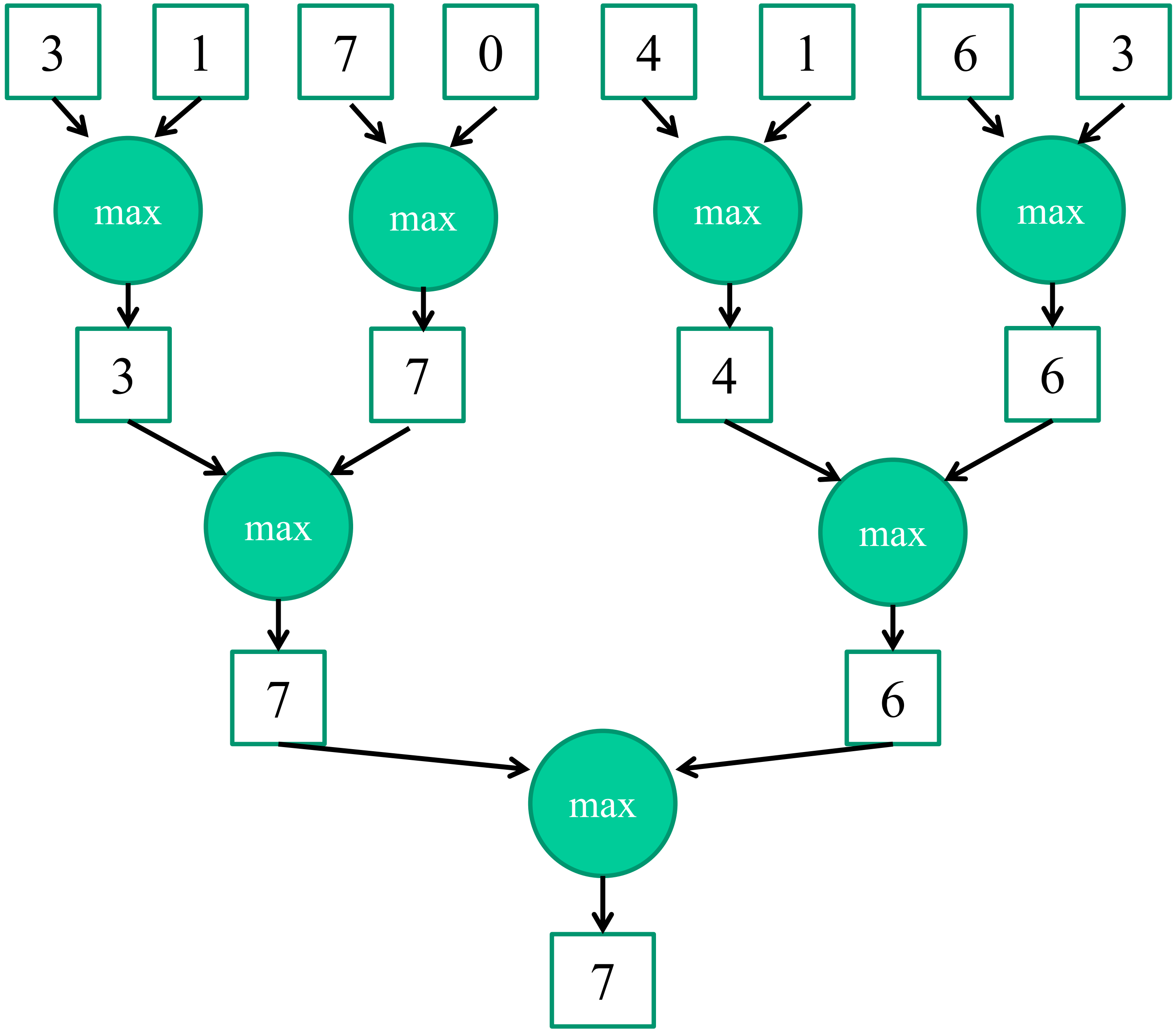
- For example: Privatization
  - Multiple threads write into an output location
  - Replicate the output location so that each thread has a private output location
  - Use a reduction tree to combine the values of private locations into the original output location

An efficient sequential reduction algorithm performs O(N) operations

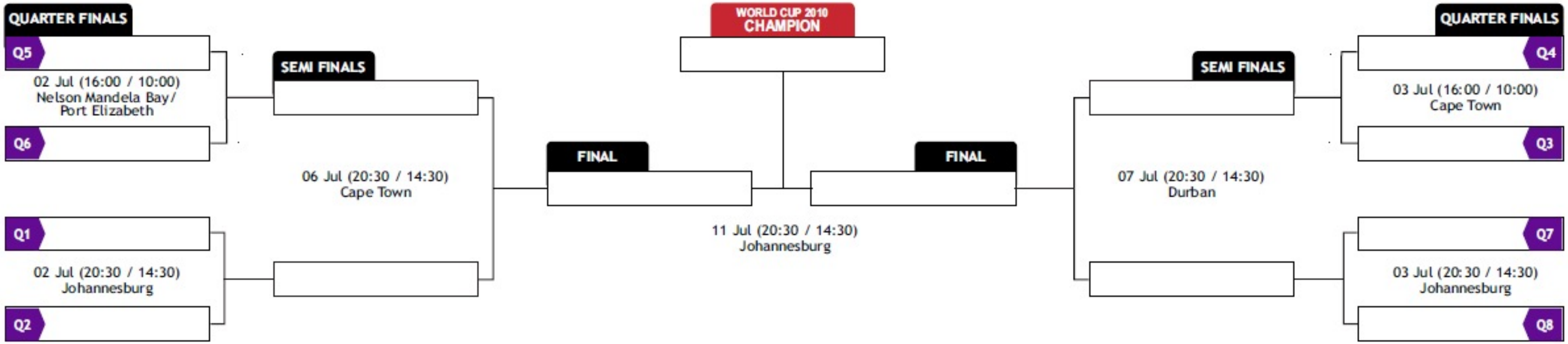
- Initialize the result as an identity value I for the reduction operation □
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction
- Iterate through the input and perform the reduction operation between the result value and the current input value

```
result ← I;
for each value X in input
      result ← result □ X;
```

A Parallel Reduction can be done in log(N) Steps



Tournaments Use Reduction with "max"



(A more artful rendition of the reduction tree.)

The Parallel Algorithm is Work Efficient

For N input values, the number of operations is

$$\frac{1}{2}N + \frac{1}{4}N + \frac{1}{8}N + \cdots + \frac{1}{N}N = \left(1 - \frac{1}{N}\right)N = N - 1.$$

The parallel algorithm shown is work-efficient: requires the same amount of work as a sequential algorithm (overheads might be different)

But requires a lot of resources…

For N input values, the number of steps is log (N).

With enough execution resources,

- N=1,000,000 takes 20 steps!

How much parallelism do we need?

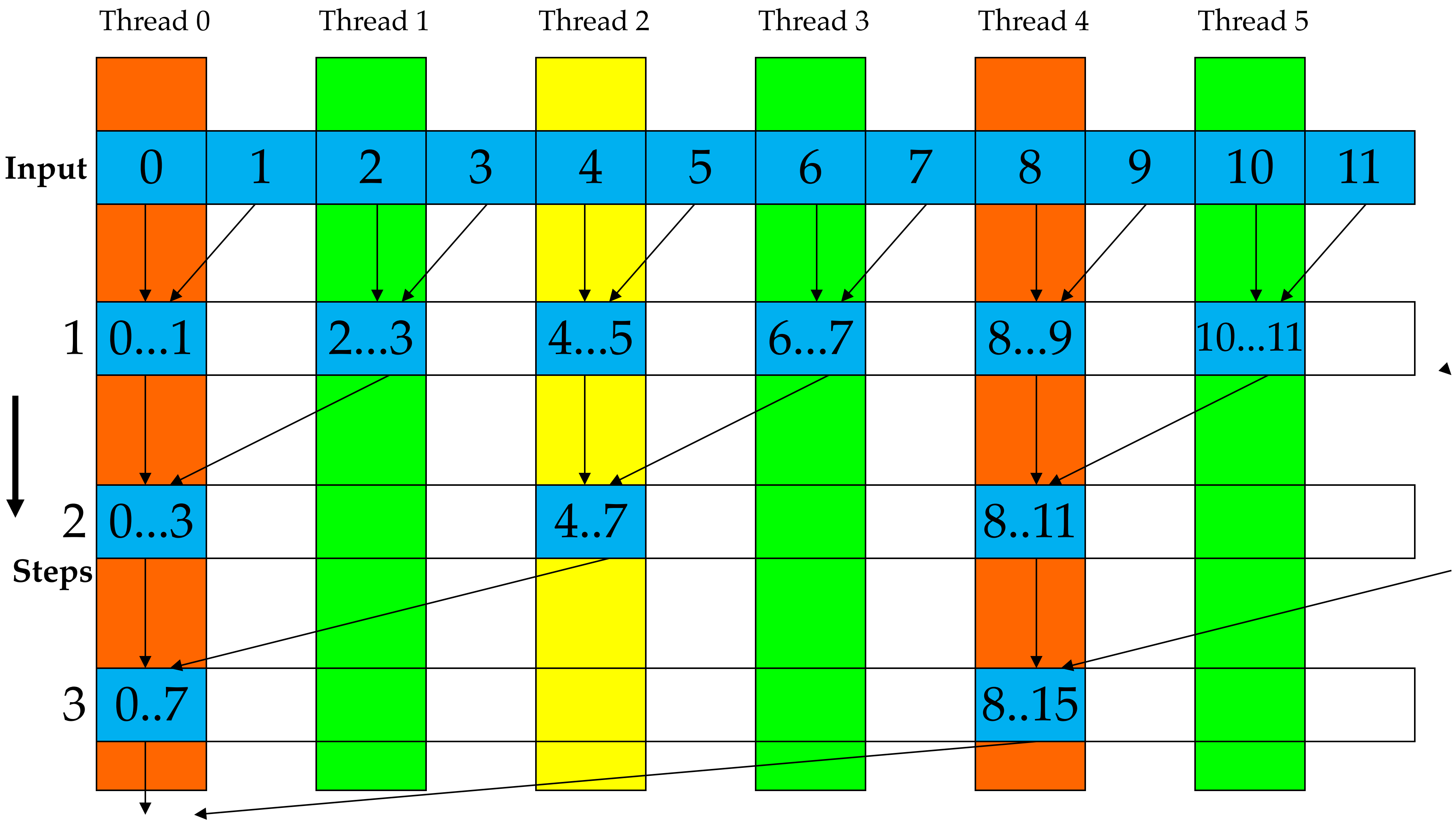
- On average, (N-1)/log(N).
  50,000 in our example.
- But peak is N/2!
  500,000 in our example.

A Sum Reduction Example

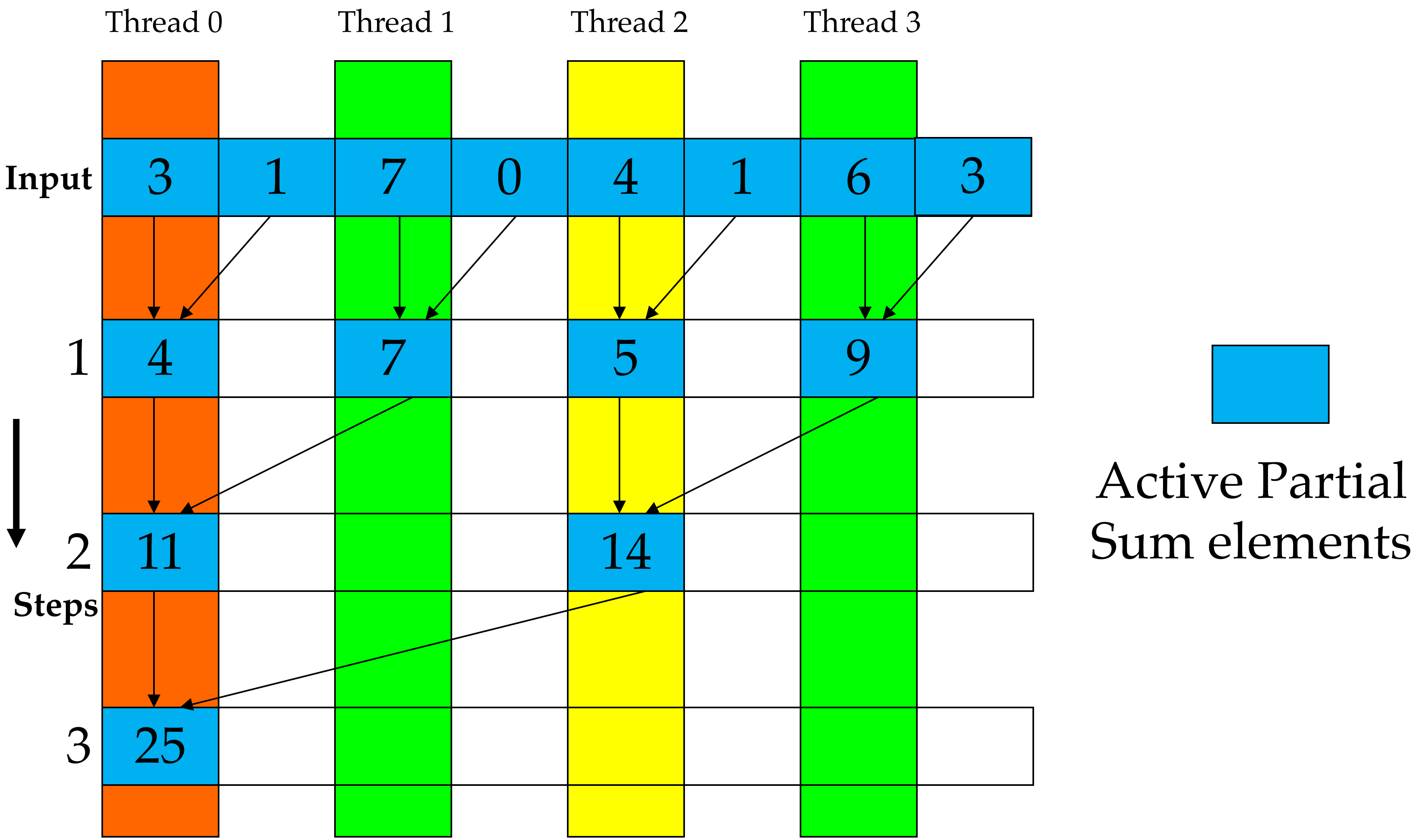
- Parallel implementation:
  - Sum two values in each thread in each step
  - Recursively halve the # of threads
  - Takes log(n) steps for n elements, requires n/2 threads

- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory is used to hold a partial sum vector
  - Each step brings the partial sum vector closer to the sum
  - The final sum will be in element 0
  - Global memory traffic should be minimal due to use of shared memory

Initial Data Mapping for a Reduction



A Sum Example (Values Instead of Indices)

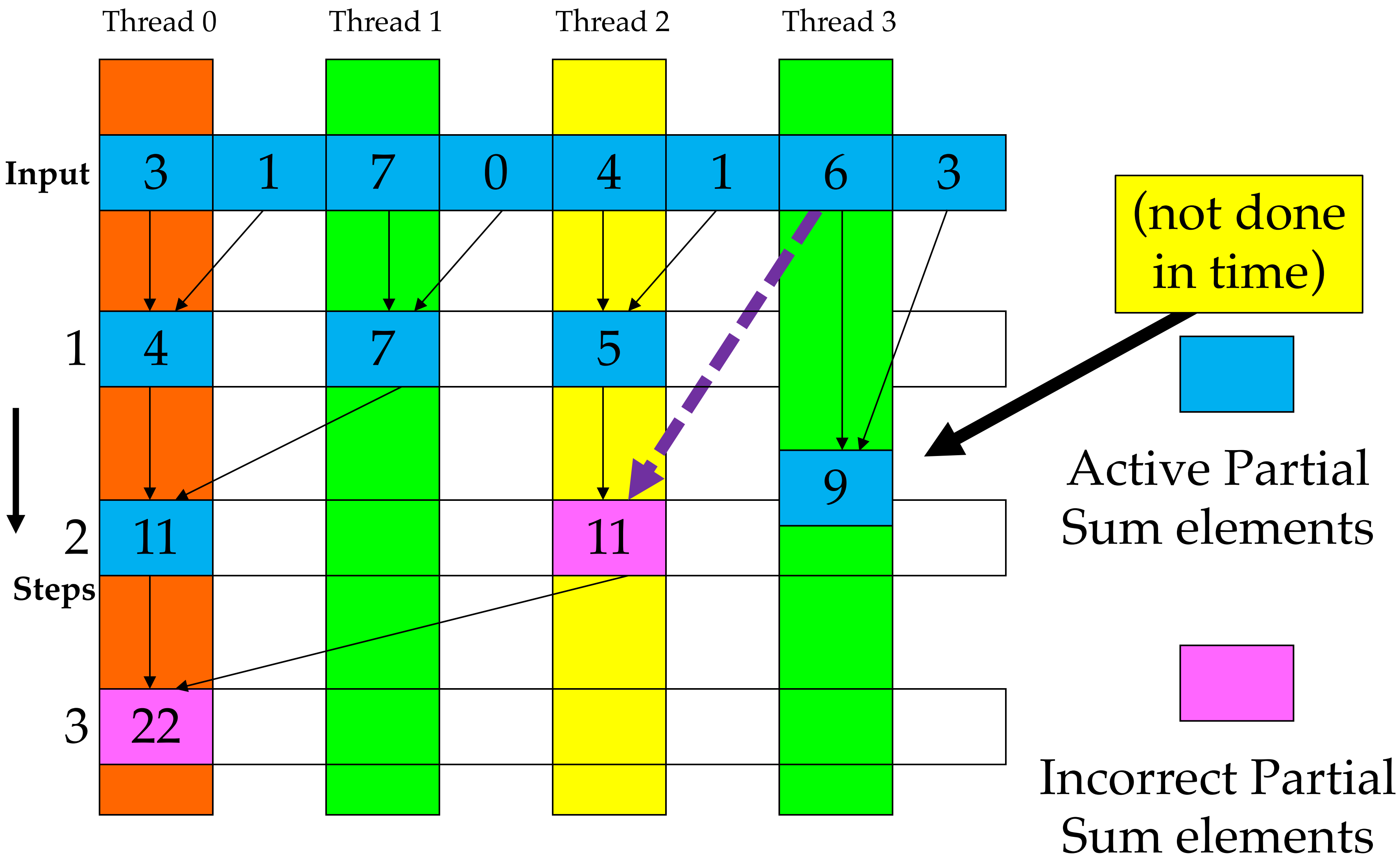


The Reduction Steps

```
// Stride is distance to the next value being
// accumulated into the threads mapped position
// in the partialSum[] array
for (unsigned int stride = 1;
     stride <= blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % stride == 0)
     partialSum[2*t]+= partialSum[2*t+stride];
}
```

Why do we need __syncthreads()?

Example Without __syncthreads



Several Options after Blocks are Done

After all reduction steps, **thread 0**

- **writes** block's **sum** from **`partialSum[0]`**
- **into global vector** indexed by **`blockIdx.x`**.

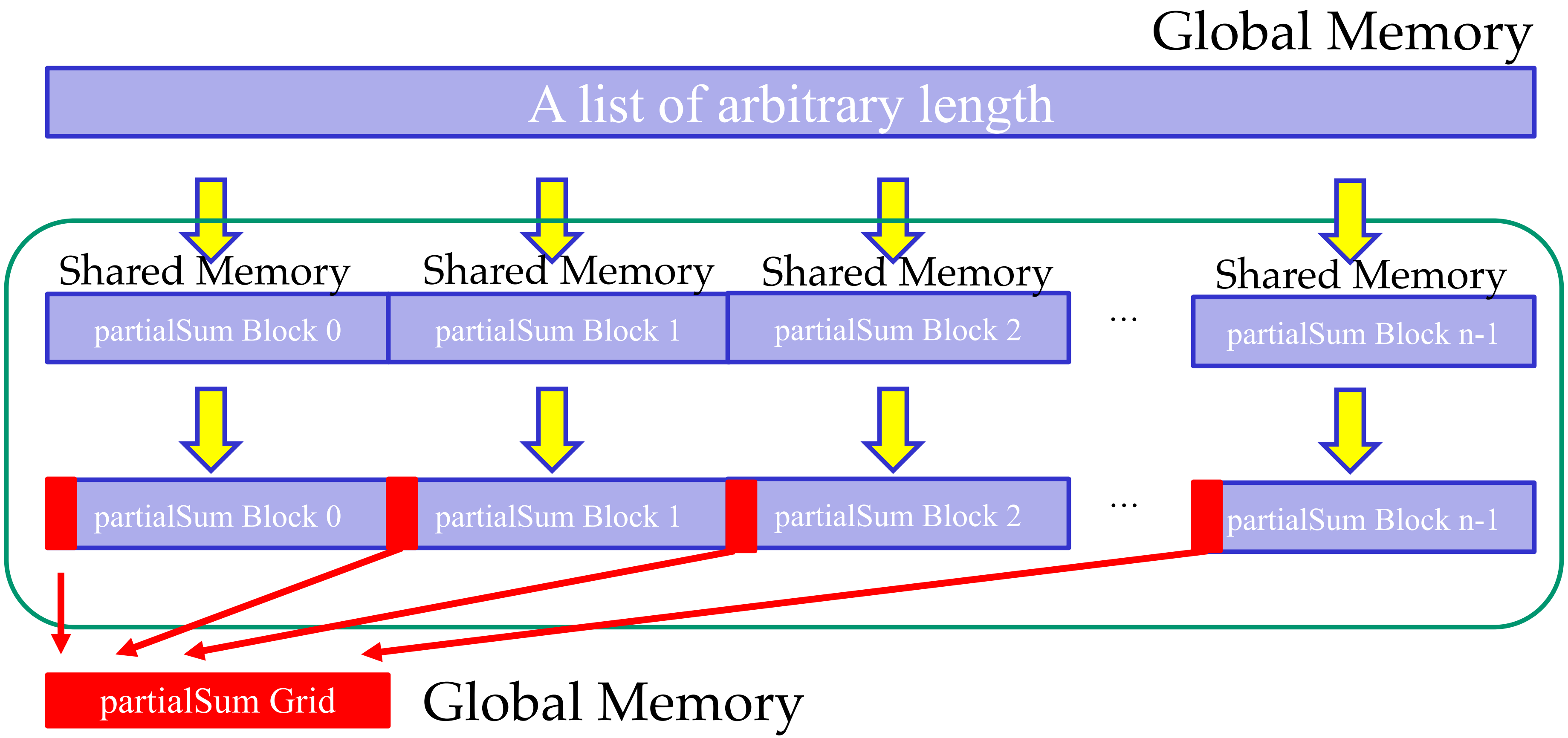
Vector has **length *N* / (2*numBlocks)**.

- If small, **transfer** vector **to host** and **sum** it up **on CPU**.

- If large, **launch kernel again** (and again).

(Kernel can also accumulate to a global sum using atomic operations, to be covered soon.)

"Segmented Reduction"



Copy back to host and host to finish the work.

Analysis of Execution Resources

All threads active in the **first step**.

In all **subsequent steps**, two control flow paths:

- **perform addition, or do nothing**.
- Doing nothing still consumes execution resources.

At most half of threads perform addition after first step

- (all threads with odd indices disabled after first step).
- **After fifth step**, entire warps do nothing:
  poor resource utilization, but no divergence.
- **Active warps have** only **one active thread**.

Up to five more steps (if limited to 1024 threads).

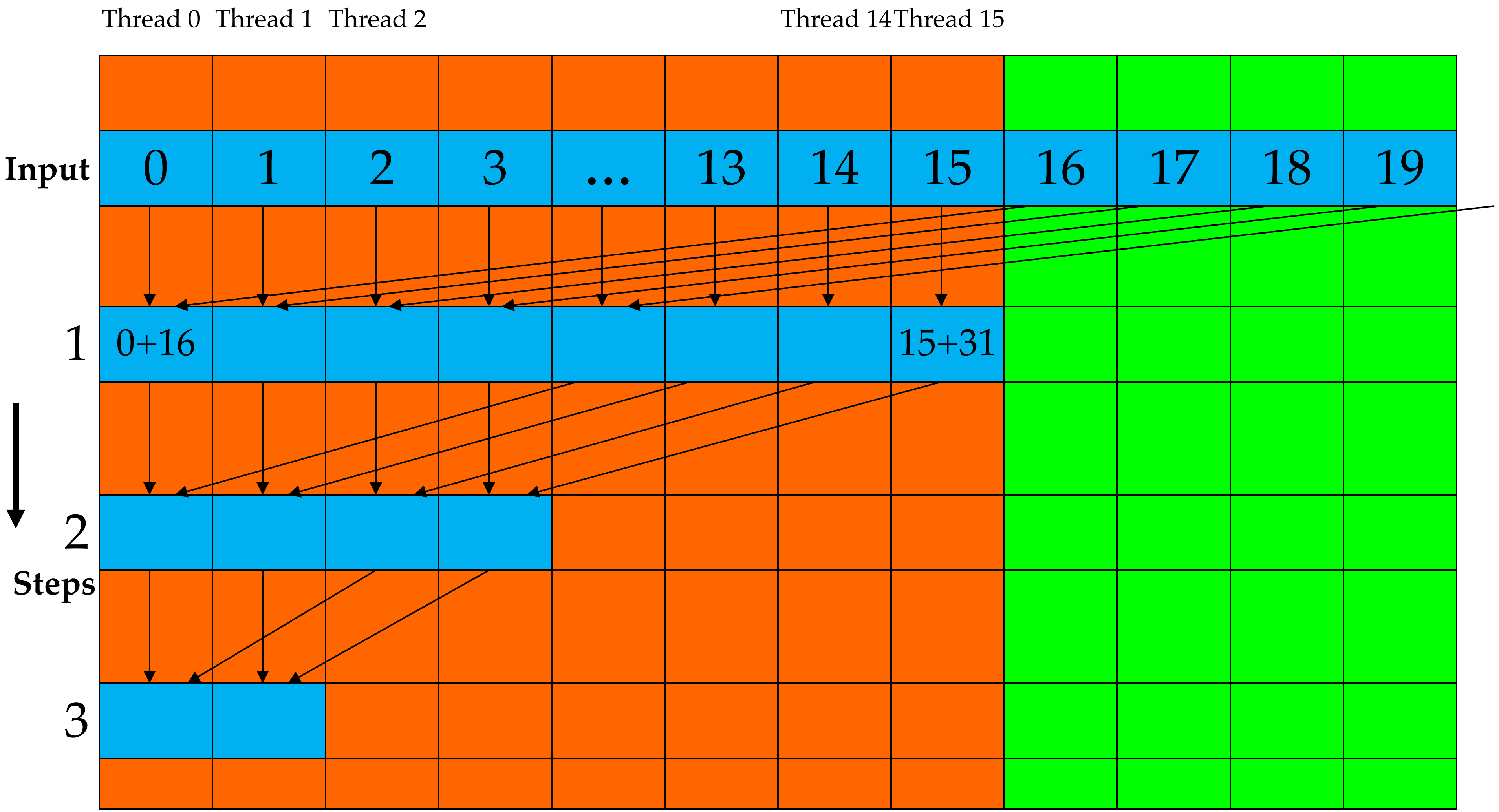
A Better Strategy

Let’s try this approach:

- **in each step**,
- **compact** the partial sums
- **into** the **first locations**
- in the `partialSum` array

Doing so **keeps** the **active threads consecutive**.

Illustration with 16 Threads



A Better Reduction Kernel

```
for (unsigned int stride = blockDim.x; stride >= 1; stride /= 2)
{
  __syncthreads();
  if (t < stride)
     partialSum[t] += partialSum[t+stride];
}
```

Again: Analysis of Execution Resources

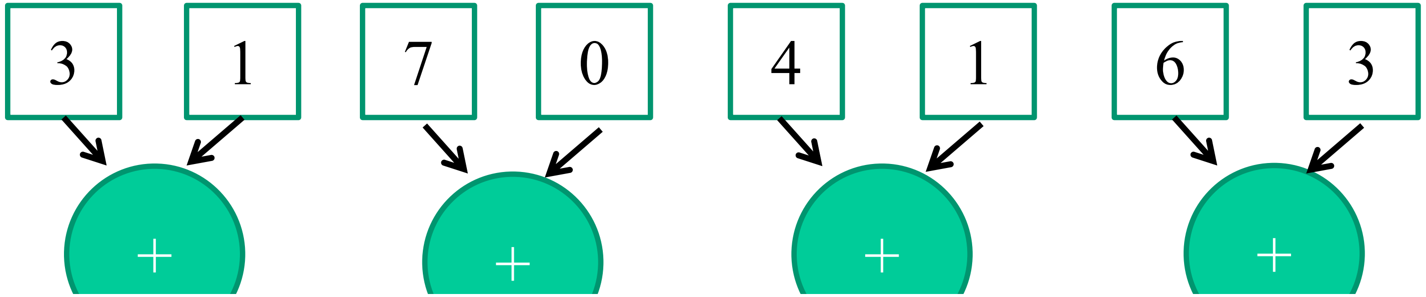
Given 1024 threads,

- Block loads 2048 elements to shared memory.
- **No branch divergence** in the **first six steps**:
  - 1024, 512, 256, 128, 64, and 32 consecutive threads active;
  - threads in each warp either all active or all inactive
- **Last six steps** have **one active warp** (branch divergence for last five steps).

Parallel Algorithm Overhead

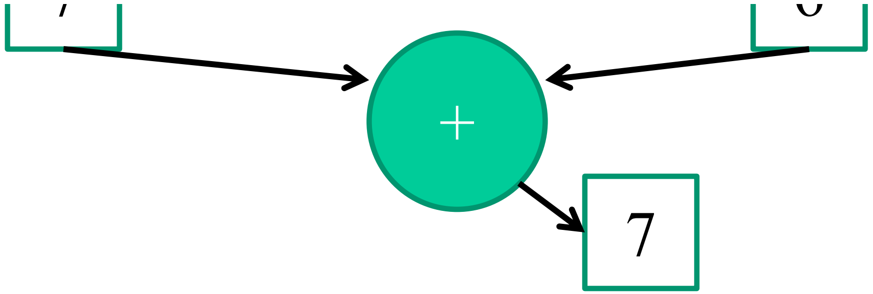
```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start+ blockDim.x+t];
 for (unsigned int stride = blockDim.x; stride >= 1; stride /= 2)
 {
   __syncthreads();
   if (t < stride)
      partialSum[t] += partialSum[t+stride];
 }
```

Parallel Execution Overhead



Although the number of "operations" is N, each "operation involves much more complex address calculation and intermediate result manipulation.

If the parallel code is executed on a single-thread hardware, it would be slower than the code based on the original sequential algorithm.



Further Improvements

The **problem has low compute-intensity**

- **one operation** for every **4B** value **read**;
- so **focus on memory coalescing and** avoiding poor **computational resource use**.
- Experiment with Hyper-parameters:
  - Threads per block
  - Blocks per SM
  - Thread granularity (3-to-1 reduction instead of 2-to-1 reduction)

Dealing with Narrowing Parallelism

Smaller blocks might seem attractive:

- when one warp is active,
- each SM has one warp per block.

But there are probably better ways. For example,

- **stop reducing at 32 elements** (or at 64, or 128), and
- hand off to the **next kernel**.

Work Until the Data is Exhausted!

Say there are 8 SMs, so 16 blocks.

1. **Divide** the whole **dataset into 16 chunks**.
2. **Read** enough **to fill shared memory**.
3. **Compute** … only **until** some **threads not needed.**
4. Then **load more data!**
5. **Repeat** until the data are exhausted,
6. THEN let parallelism drop.

(Gather 16 values on host and reduce them.)

ANY MORE QUESTIONS?
READ CHAPTER 5