ECE408/CS483/CSE408 Fall 2022

Applied Parallel Programming

Lecture 15
Parallel Computation Patterns – Reduction Trees

Course Reminders

- Midterm 1
  - We are grading the coding part of the exam
  - Median for multiple choice questions is 48/70
- Project Milestone 1
  - Baseline CPU implementation is due Friday October 14th
- MP 5.1
  - Is due next week, will be posted this Friday

Objectives

- To learn the basic concept of reductions, one of the most widely used parallel computation patterns
- To learn simple strategies for parallelization of reductions
- To understand the performance issues involved with performing reductions on GPUs

Important Enough to Use in Theory

"… scan operations, also known as prefix computations,

can execute in no more time than …

parallel memory references …

greatly simplify the description of many [parallel] algorithms, and are

significantly easier to implement than memory references."

—Guy Blelloch, 1989*

*G. Blelloch, "Scans as Primitive Parallel Operations," IEEE Transactions on Computers, 38(11):1526-1538, 1989.
The idea behind scans for computation goes back another 30+ years.

Trying to Bridge Theory and Practice

A generic parallel algorithm,

- in which parallel threads access memory arbitrarily,
- is likely to produce an extremely slow access pattern.

Scans

- can be implemented quickly in hardware, and
- form a useful alternative to arbitrary memory accesses.

(His hope was to enable theory
without knowledge of microarchitecture.)

Example Use: Summarizing Results

1. Start with a **large set of things** (examples: integers, social networking user information)
2. **Process** each thing **independently to produce** some **value** (examples: number of friends, timeline posts in last two weeks)
3. **Summarize**!
   - Typically, with an associative and commutative operation (+, *, min, max, …)
   - since things in the set are unordered and independent.

Focus on Reduction Using a Tree

Pattern is so common that

- people have built frameworks around it!
- examples: Google and Hadoop MapReduce

Let's focus on the summarization, called a **reduction**:

- – **no required order** for processing the values (operator is associative and commutative), so
- – **partition the data set** into smaller chunks,
- – have each thread to process a chunk, and
- – **use a tree to compute the final answer**.

Reduction Enables Parallelization

Reduction enables common parallel transformations.

example: **privatization of output**

- **Loop iterations sum into a single output** (examples: inner loops in matrix multiply and convolution).
- To parallelize iterations, must **make private copies** of the output!
- **Use reduction** to sum private copies into the original output.

What Exactly is a Reduction?

Reduce a set of inputs to a single value

- using a binary operator, such as
- sum, product, minimum, maximum,
- or a user-defined reduction operation
  - must be associative and commutative
  - and have an identity value (example: 0 for sum)

Available in most parallel libraries as **collective operations** (like barriers).

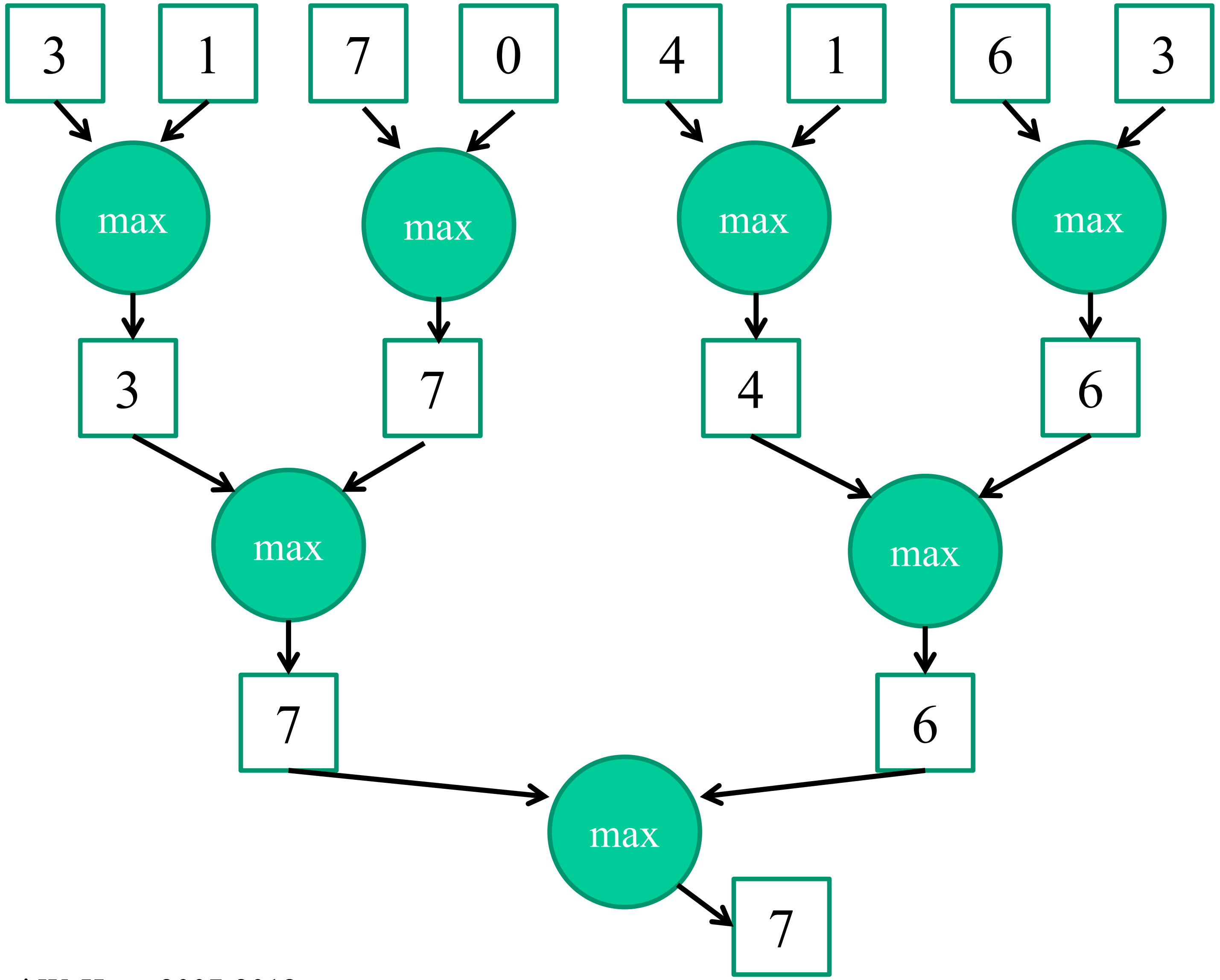
Sequential Reduction is O(N)

Given binary operator $\leftrightarrow$ and an identity value $I^{\leftrightarrow}$

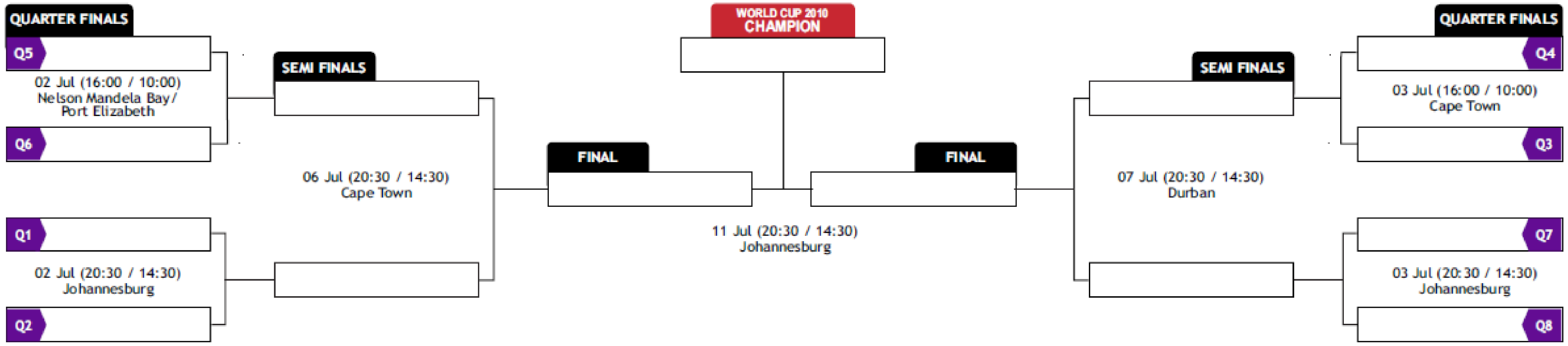
- $I^{\leftrightarrow}$ = 0 for sum
- $I^{\leftrightarrow}$ = 1 for product
- $I^{\leftrightarrow}$ = largest possible value for min
- $I^{\leftrightarrow}$ = smallest possible value for max

```
result ← I↔
for each value X in input
    result ← result ↔X
```

Example: Parallel Max Reduction in log(N) Steps



Tournaments Use Reduction with “max”



(A more artful rendition of the reduction tree.)

Algorithm is Work Efficient

For ***N* input values**, the **number of operations** is

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

The parallel algorithm shown is **work-efficient**:

- requires the **same amount of work as a sequential algorithm**
- (constant overheads, but nothing dependent on ***N***).

Fast if Enough Resources are Available

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

With enough execution resources,

- **N=1,000,000 takes 20 steps!**
- Sounds great!

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.

Diminishing Parallelism is Common

In our **parallel reduction**,

- the **number of operations**
- **halves in every step**.

This kind of **narrowing parallelism is common**

- from combinational logic circuits
- to basic blocks
- to high-performance applications.

CUDA kernels allow only a fixed number of threads.

Parallel Strategy for CUDA

Let's start simple: ***N*** values in device global memory.

Each **thread block** of ***M*** threads

- uses shared memory,
- to **reduce chunk of 2M** values to one value
- (2M << N to produce enough thread blocks).

Blocks operate **within shared memory**

- to reduce global memory traffic, and
- **write one value back** to global memory.

CUDA Reduction Algorithm

1. **Read** block of **2M values** into shared memory.

2. For each of **log(2M)** steps,
   - **combine two values** per thread in each step,
   - **write result** to shared memory, and
   - **halve** the number of **active threads**.

3. **Write final result** back to global memory.

A Simple Mapping of Data to Threads

Each **thread**

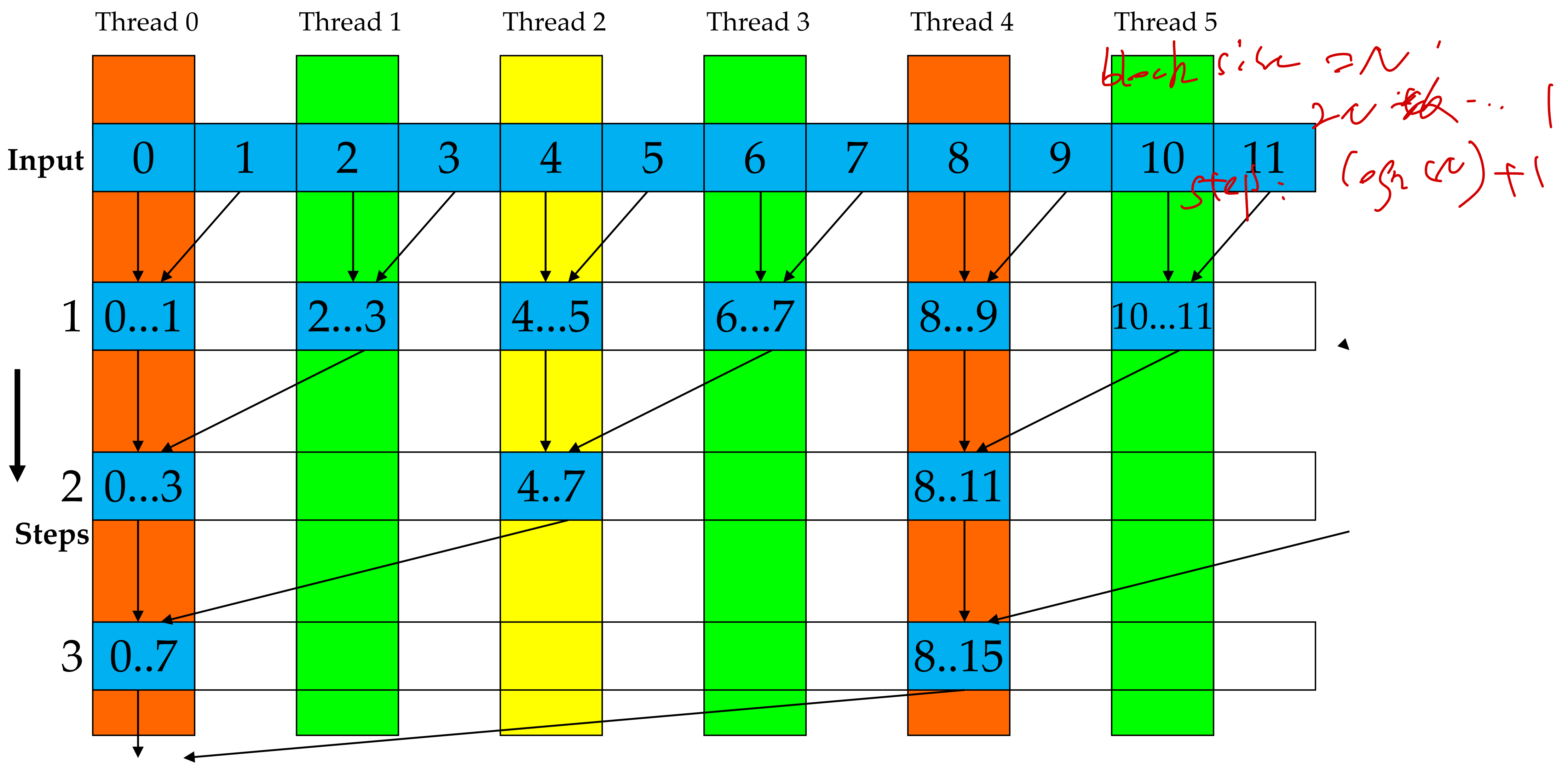
- **begins with** two **adjacent locations** (**stride of 1**),
- **even index (first)** and an odd index (second).
- Thread 0 gets 0 and 1, Thread 1 gets 2 and 3, …
- Write **result** back **to** the **even index**.

After each step,

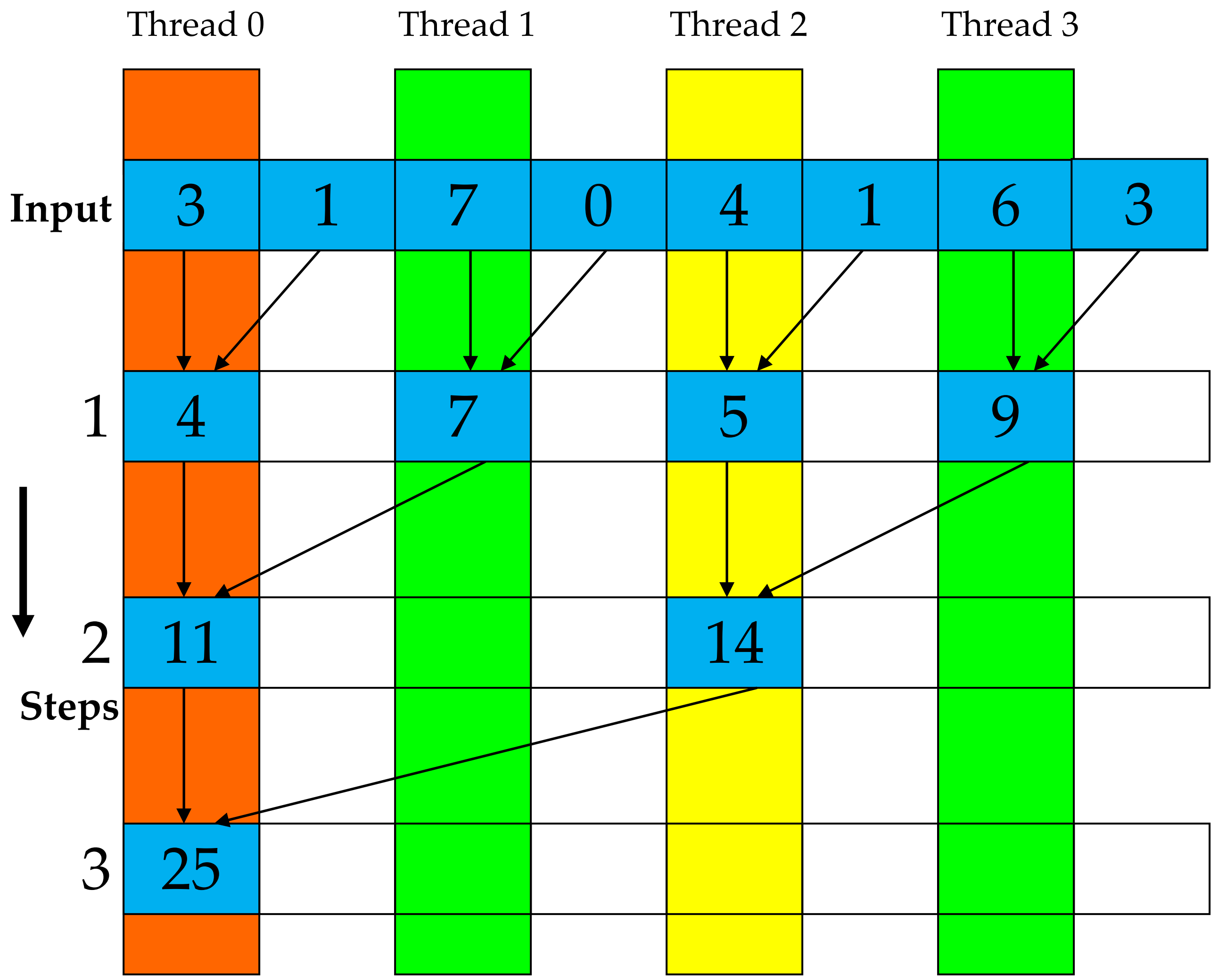
- **half of** active **threads** are **done**.
- **Double the stride**.

At the end, **result is at index 0**.

Naïve Data Mapping for a Reduction



A Sum Example (Values Instead of Indices)



🟦 Active Partial Sum elements

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()?

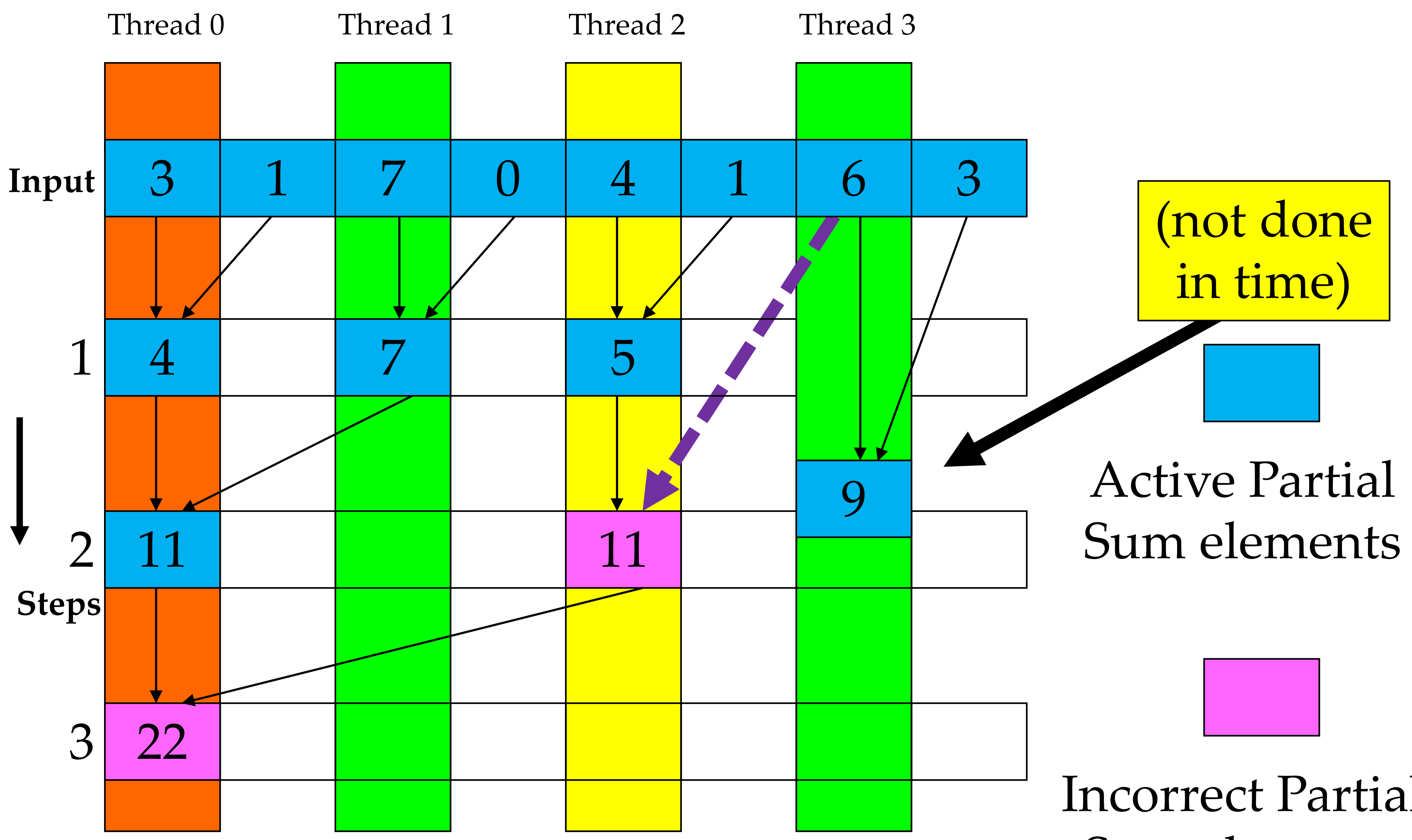
Barrier Synchronization

__syncthreads() ensures

- **all elements** of partial sum **generated**
- **before** the **next step uses them**.

Why do we not need __syncthreads() at the end of the reduction loop?

Example Without __syncthreads



🟦 Active Partial Sum elements

🟪 Incorrect Partial Sum elements

Several Options after Blocks are Done

After all reduction steps, **thread 0**

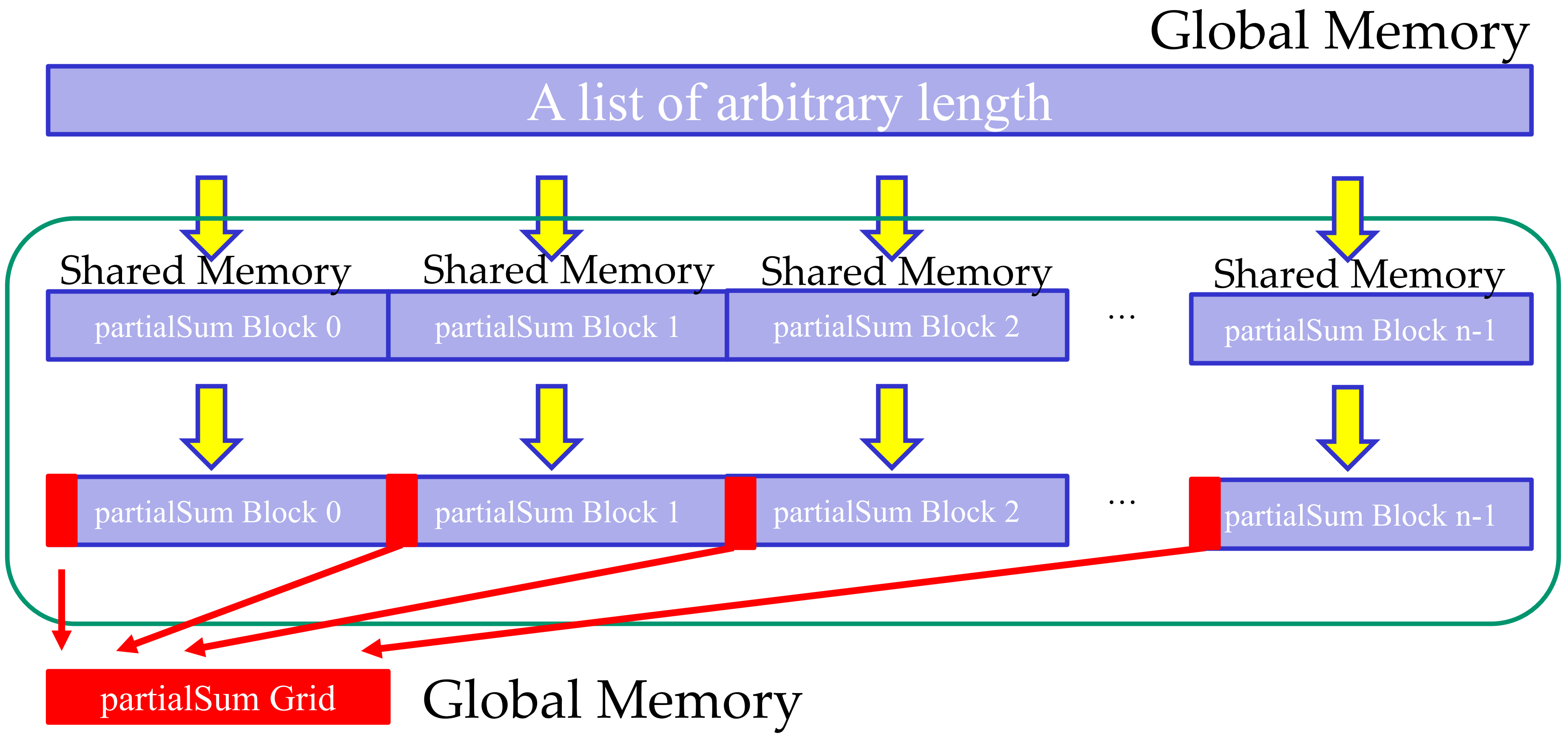
- **writes** block's **sum** from **`partialSum[0]`**
- **into global vector** indexed by **`blockIdx.x`**.

Vector has **length *N* / 2*M***.

- 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).

Improve Performance by Reassigning Data

Can we do better?

Absolutely!

How we assign data to threads makes a difference in some algorithms, including reduction.

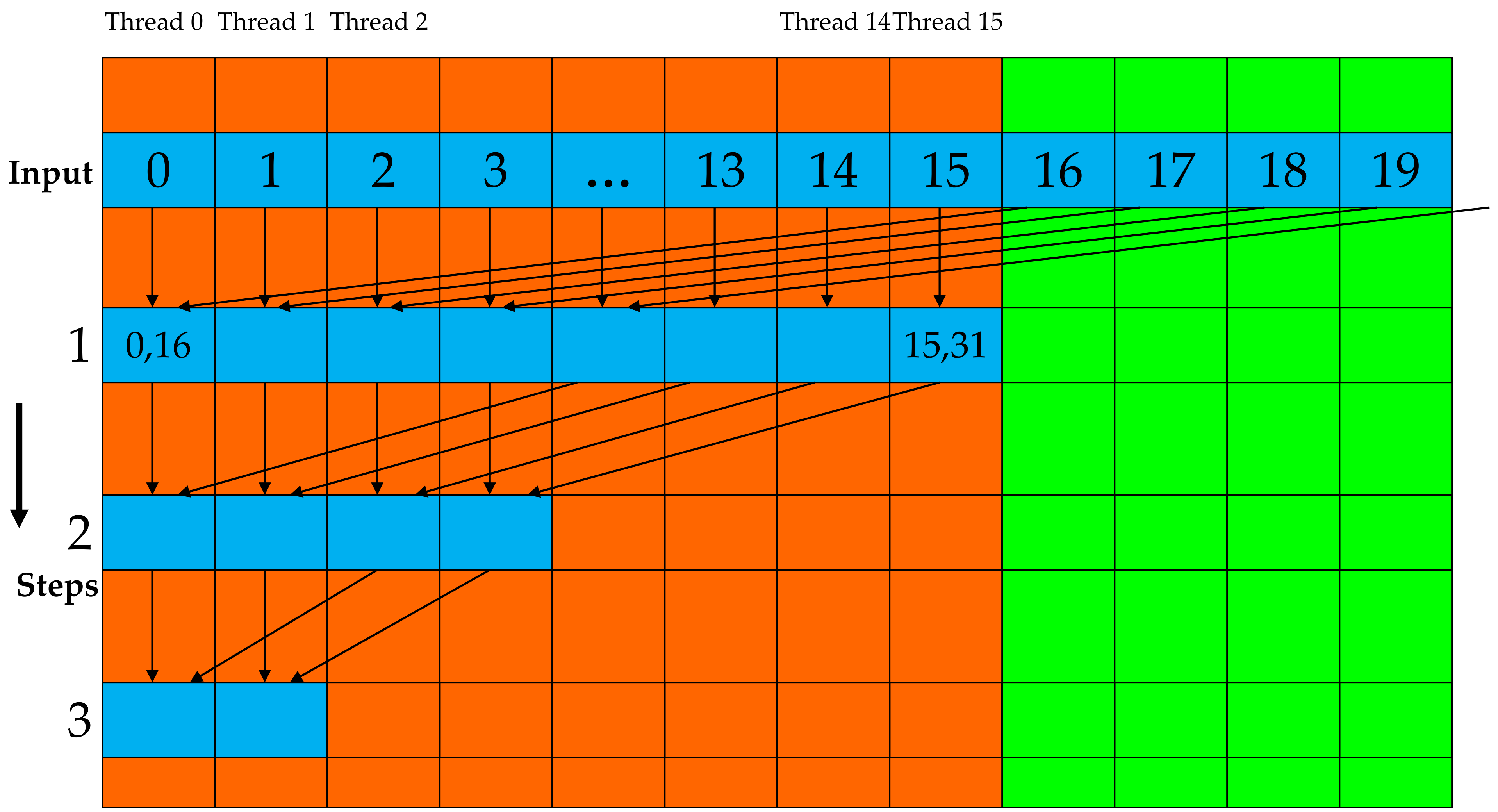
A Better Strategy

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).

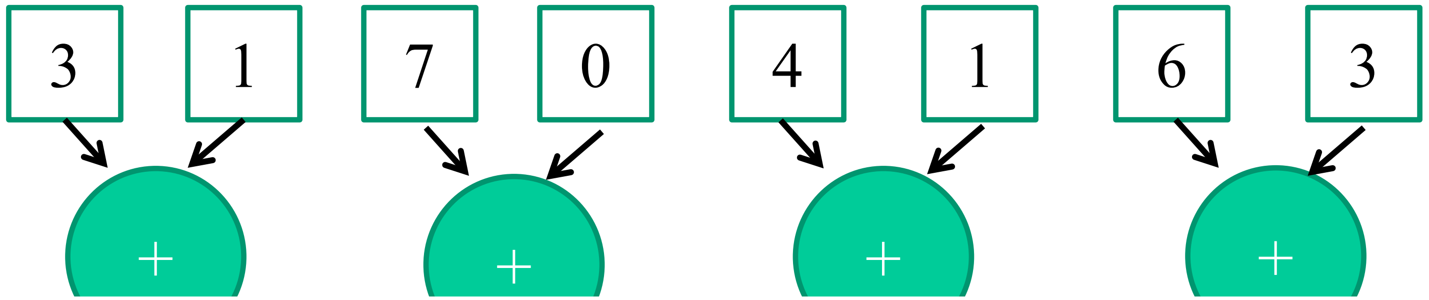
Parallel Algorithm Overhead

```
__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 >>= 1)
 {
   __syncthreads();
   if (t < stride)
      partialSum[t] += partialSum[t+stride];
 }
```

Parallel Algorithm Overhead

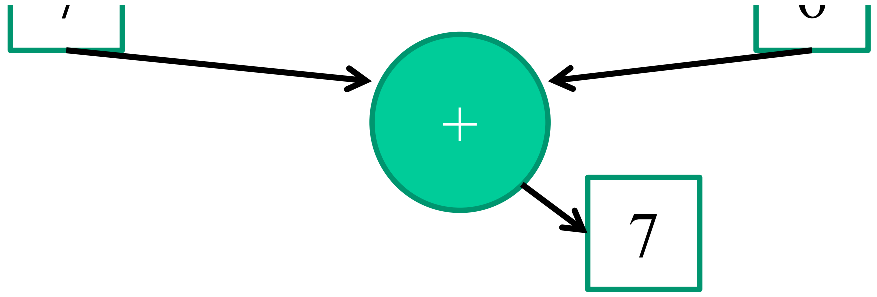
```
__shared__ float partialSum[2*BLOCK_SIZE];

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 >>= 1)
 {
   __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 significantly slower than the code based on the original sequential algorithm.



Further Improvements

Can we further improve reduction?

The **problem is memory-bound**:

- **one operation** for every **4B** value **read**;
- so **focus on memory coalescing and** avoiding poor **computational resource use**.

Make Use of Shared Memory

How much shared memory are we using?

Each block of **1,024** threads reads **2,048** values.

- Let's say **two** blocks per SM,
- so **16 kB** (**= 2,048 × 2 × 4B**).

Could **read 4,096 or 8,192** values

- (with **64 kB** per SM)
- **to** slightly **increase parallelism**.

(For **48 kB** per SM, use **6,144** values and have all threads do a 3-to-1 reduction before the current loop.)

Eliminate the Narrow Parallelism

What about 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**.

Get Rid of the Overhead

Launching kernels is expensive.

- Why bother tearing down and setting up the same blocks on the same SMs?
- Makes no sense.
- Remember that reduction operators are associative and commutative.

Let's **be compute-centric**:

- put **2048 threads** (as two blocks) **on each SM**, and
- just keep them there **until we're done**!

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.)

Caveat

I didn't try these ideas.

I'll leave them

- for those of you who feel motivated
- to **try in MP5.1**.

Do

- **save a copy of** your **simpler solution**, though, as
- you will need the partial sums for scan (MP5.2).

ANY MORE QUESTIONS? READ CHAPTER 5