Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Reductions

Marc Jordà, Antonio J. Peña

Montevideo, 21-25 October 2019

Partition and Summarize

- A commonly used strategy for processing large input data sets
  - Assume there is no required order of processing elements in a data set (associative and commutative)
  - 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
- Google and Hadoop MapReduce frameworks are examples of this pattern
- We will focus on the reduction tree step.

Reduction enables other techniques

- Reduction is also needed as a final step after some commonly used parallelizing transformations

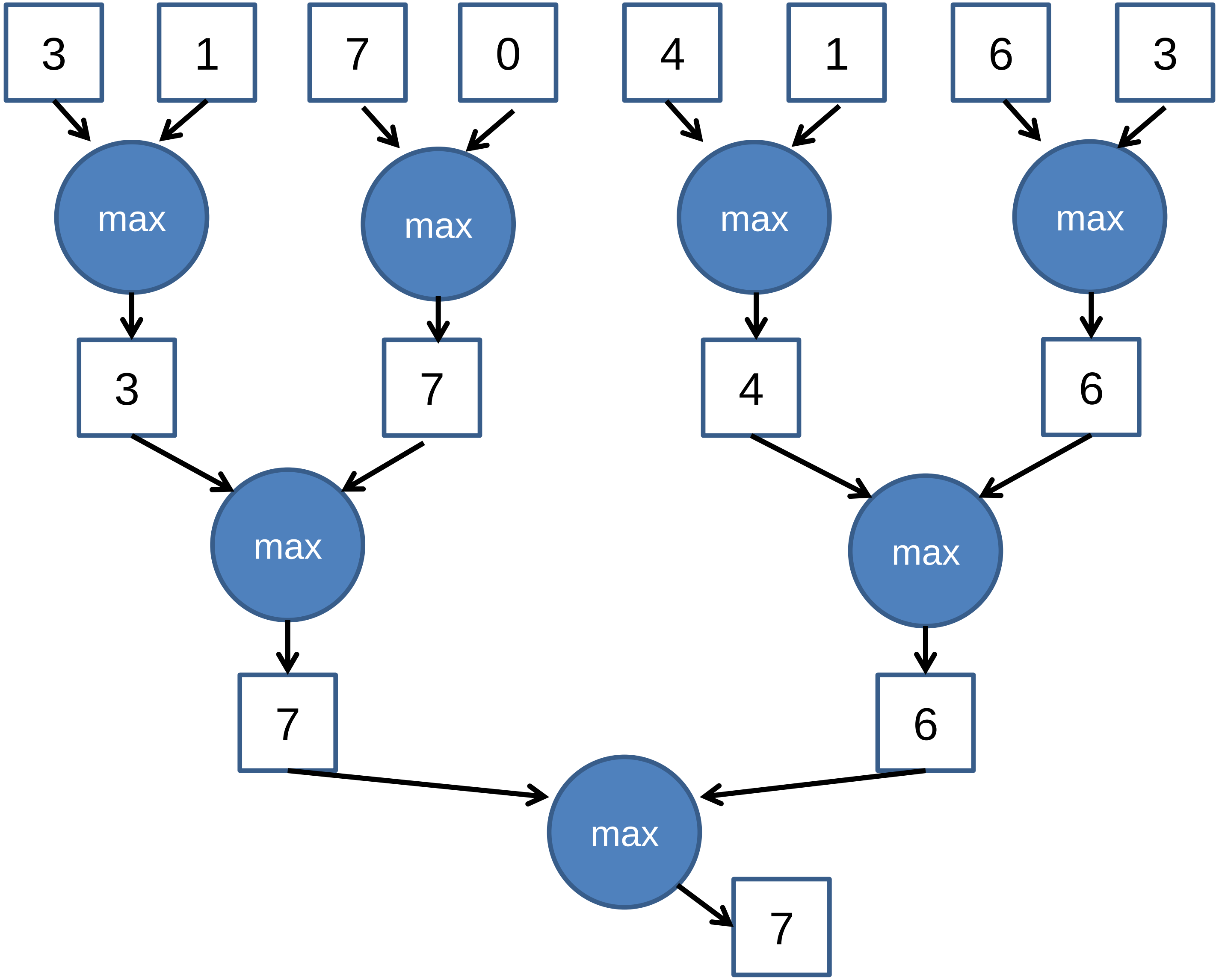
- Privatization
  - Multiple threads write into an output location
  - Replicate the output location so that each thread has a private output location
  - Use a reduction tree to combine the values of private locations into the original output location

What is a reduction computation ?

- Summarize a set of input values into one single value using a *reduction operation*
  - Max
  - Min
  - Sum
  - Product
  - User defined reduction operation function, as long as the operation
    - Is associative and commutative
    - Has a well-defined identity value (e.g., 0 for sum, 1 for product)

- Reduction is an example of "collective operations"

A parallel reduction tree algorithm performs N-1 Operations in log(N) steps

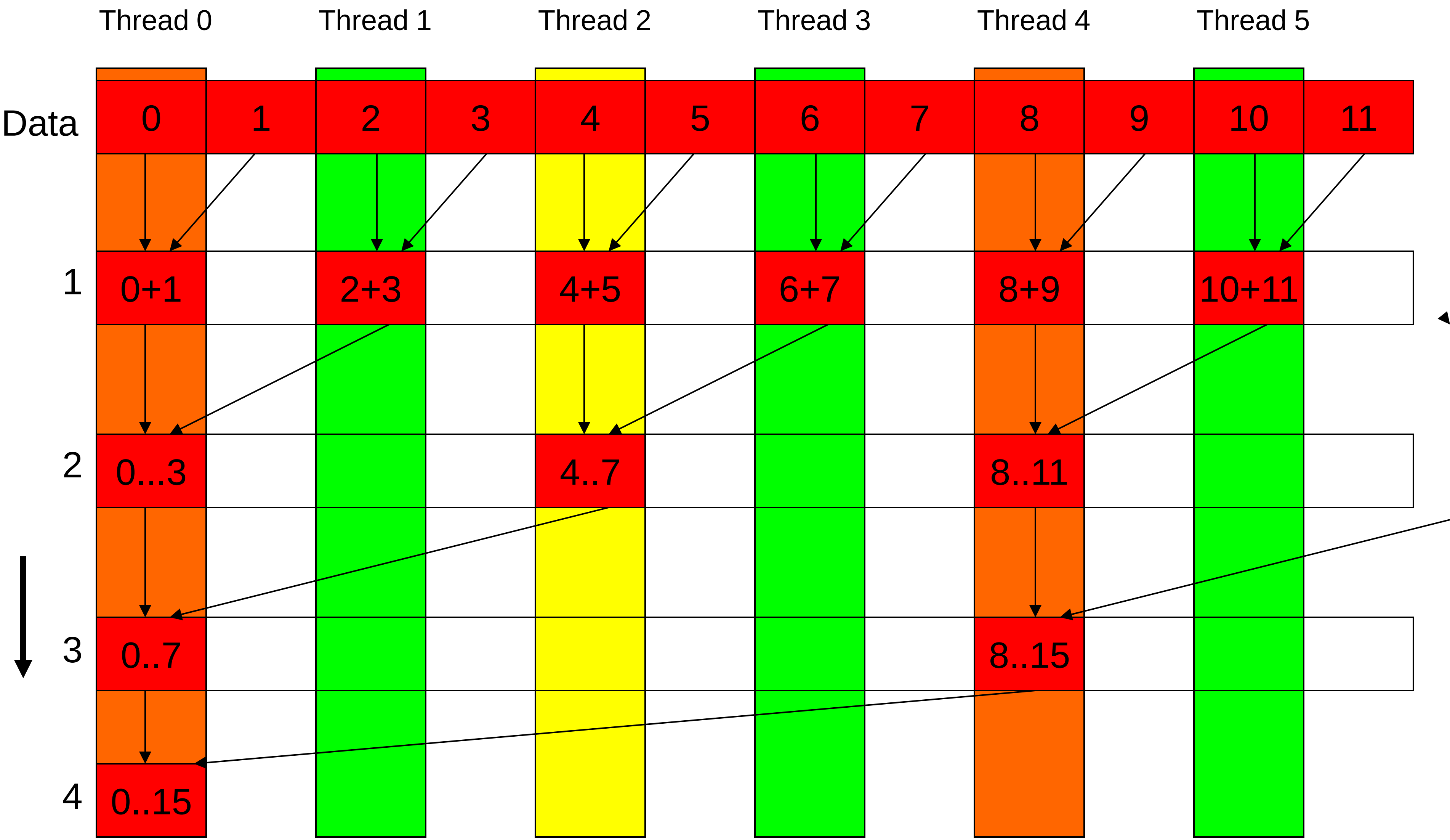


A Parallel Sum Reduction Example

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

- Example: 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
  - Reduces global memory traffic due to partial sum values

Vector Reduction with Branch Divergence



Simple Thread Index to Data Mapping

- Each thread is responsible of an even-index location of the partial sum vector
  - One input is the location of responsibility

- After each step, half of the threads are no longer needed

- In each step, one of the inputs comes from an increasing distance away

A Simple Thread Block Design

- Each thread block takes 2* BlockDim input elements
- Each thread loads 2 elements into shared memory

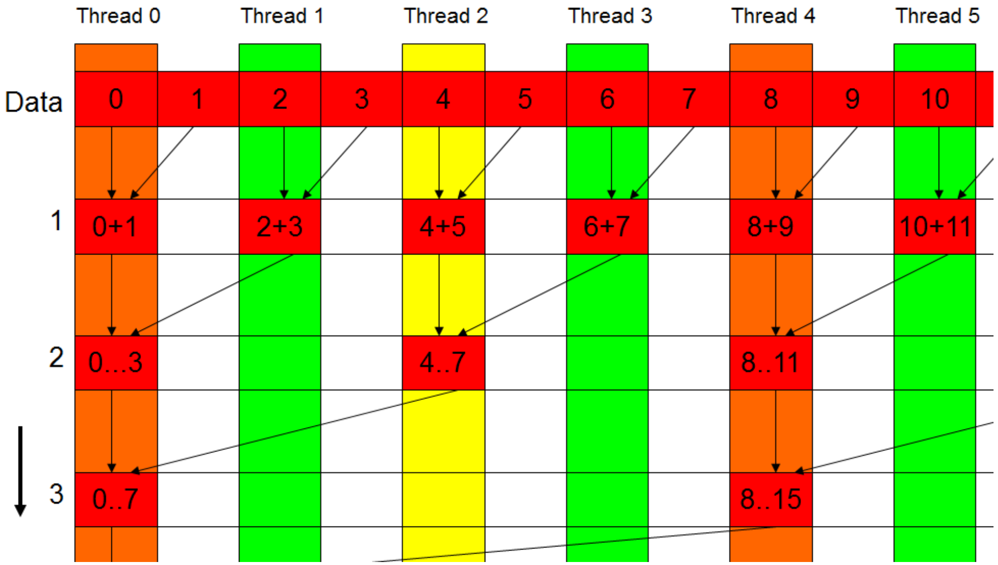
```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim.x + t] = input[start + blockDim.x + t];
```

The Reduction Steps

```
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() are needed to ensure that all elements of each version of partial sums have been generated before we proceed to the next step

Back to the Global Picture

- Thread 0 of each thread block writes the partial sum computed by the thread block into a vector indexed by the blockIdx.x

- There can be a large number of reduction iterations if the original vector is very large
  - The thread grid may iterate over the input, or
  - The host code may iterate and launch another kernel

- When there are only a small number of elements to reduce, the host can simply transfer the data back and add them together serially.

Some Observations

- In each iteration, two control flow paths will executed in each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition still consume execution resources

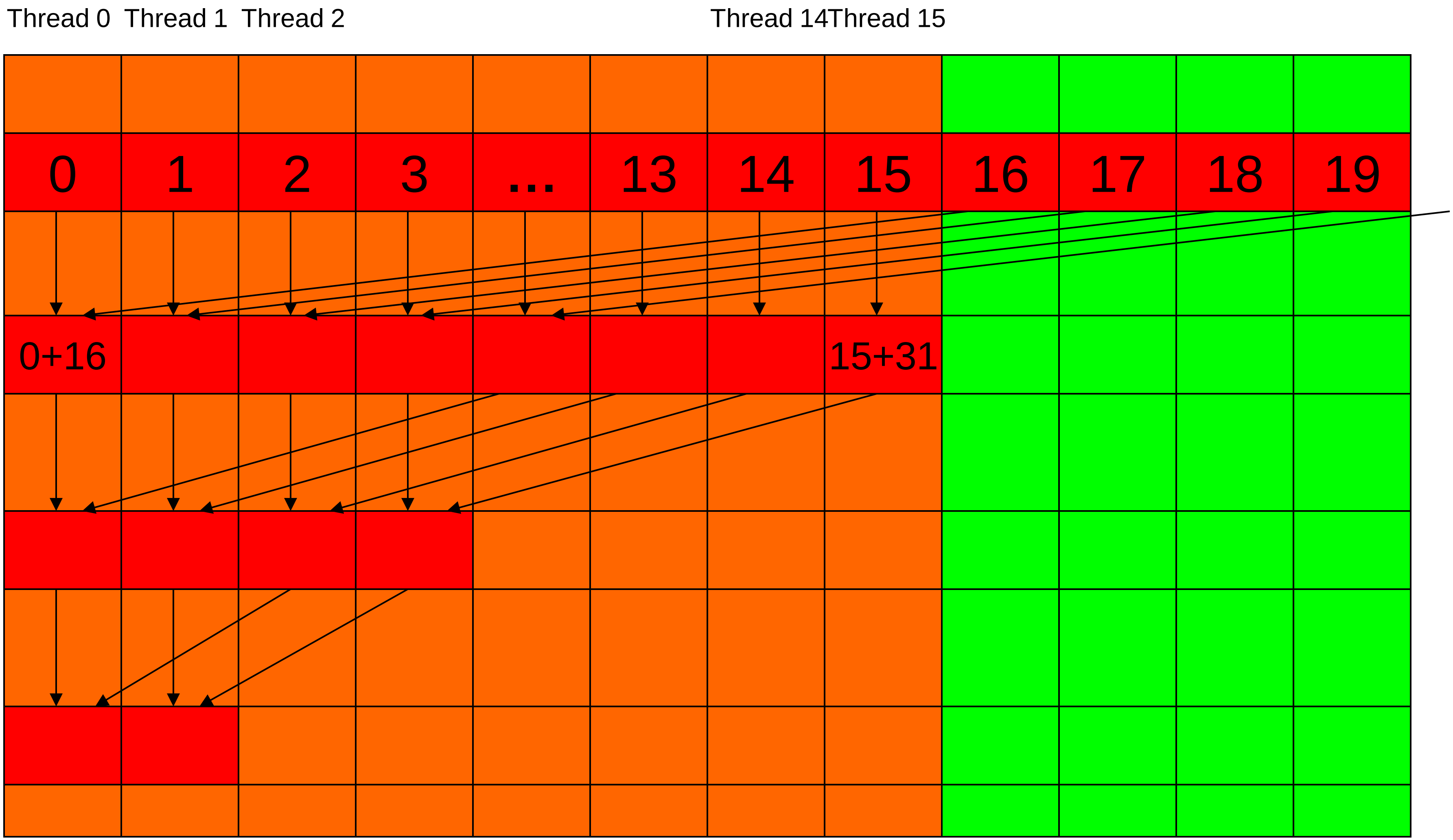
- Half or fewer of threads will be executing after the first step
  - All odd index threads are disabled after first step
  - After the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence
    - This can go on for a while, up to 6 more steps (stride = 32, 64, 128, 256, 512, 1024), where each active warp only has one productive thread until all warps in a block retire

A Better Strategy

- Always compact the partial sums into the first locations in the partialSum[] array

- Keep the active threads consecutive

An Example of 16 threads



A Better Reduction Kernel

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

unsigned int tx = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[tx] = input[start + tx];
partialSum[blockDim.x + tx] = input[start + blockDim.x + tx];

 for (unsigned int stride = blockDim.x;
                           stride > 0;  stride /= 2)
 {
   __syncthreads();
   if (tx < stride)
      partialSum[tx] += partialSum[tx + stride];
 }
```

A Quick Analysis

- For a 1024 thread block
  - No divergence in the first 5 steps
  - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
  - The final 5 steps will still have divergence
    - Only the last warp will be running, no efficiency issues

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact

marc.jorda@bsc.es, antonio.pena@bsc.es