

Heterogeneous Parallel Programming

COMP4901D

Parallel Reduction

Overview

- The Reduction Operation
- Sequential Implementation
- Baseline Reduction Kernel
- Improved Reduction Kernel

Reduce (Reduction)

- A commonly used strategy for processing large input data sets
- 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 support this strategy

Reduction in Other Parallel Operations

- Reduction is also needed to clean up 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

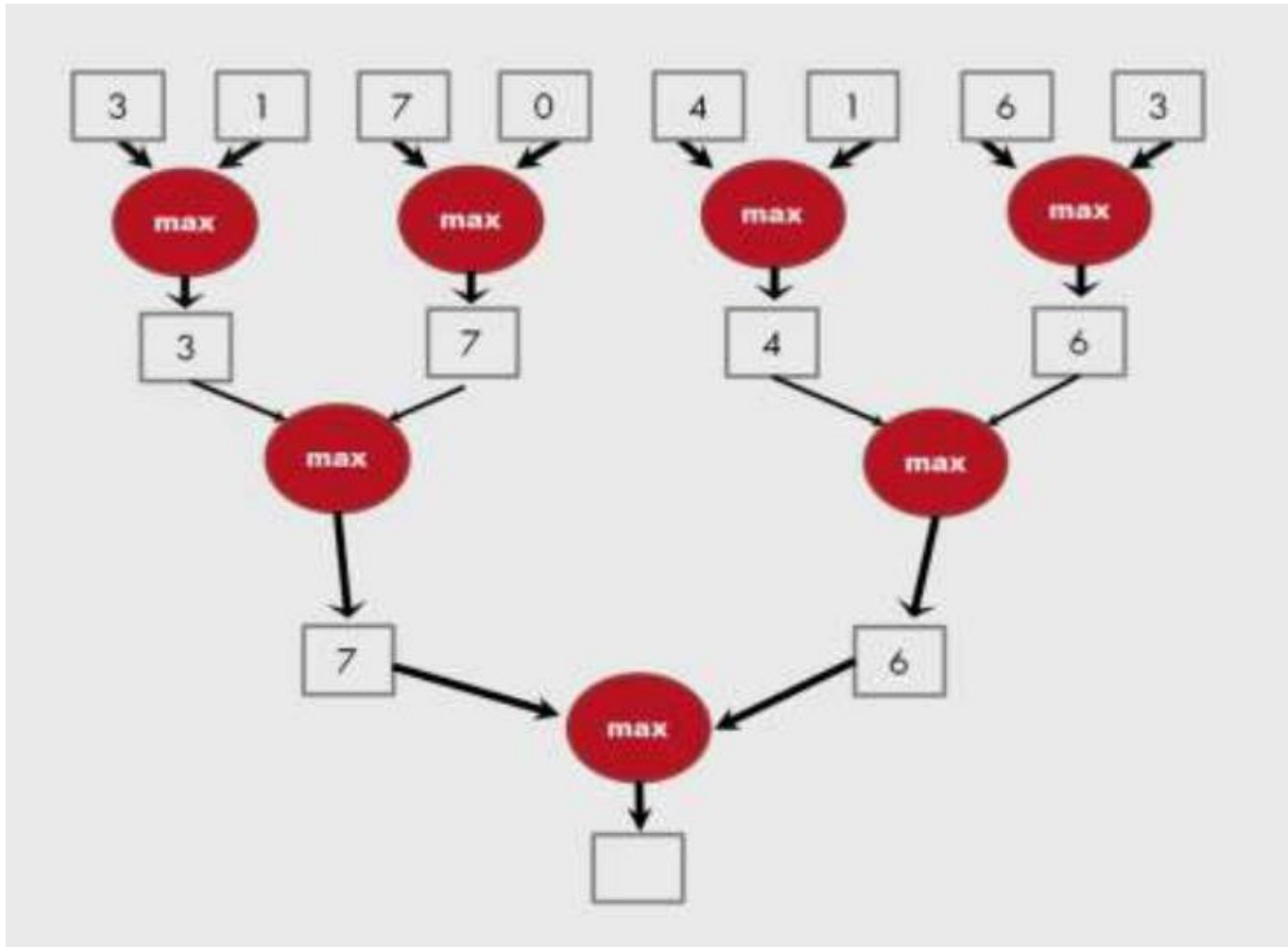
Computation used in Reduction

- Summarize a set of input values into one 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)

Sequential Reduction

- Initialize the result as an identity value 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
 - N reduction operations performed for N input values

A Reduction Tree Algorithm



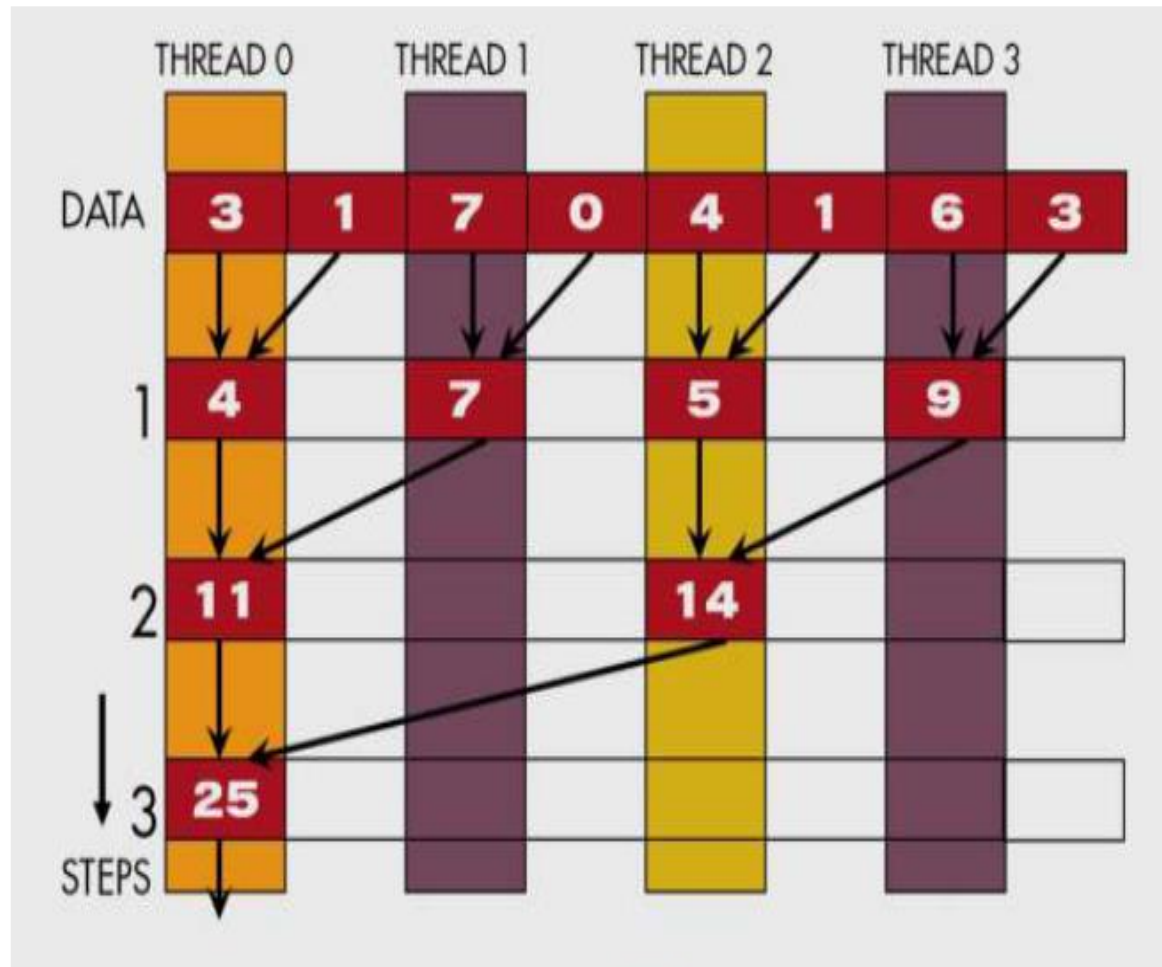
Analysis of Reduction Tree

- For N input values, the reduction tree performs
- $(1/2)N + (1/4)N + (1/8)N + \dots 1 = (1 - (1/N))N = N-1$ operations
- In $\log(N)$ steps – 1,000,000 input values take 20 steps
 - Assuming that we have enough execution resources
- Average Parallelism $(N-1)/\log(N)$
 - For $N = 1,000,000$, average parallelism is 50,000
 - However, peak resource requirement is 500,000!
 - This is not resource efficient.
- This is a work-efficient parallel algorithm
 - The amount of work done is comparable to sequential
 - Many parallel algorithms are not work efficient

Parallel Implementation

- Parallel execution of reduction tree
 - Add two values per thread in each step
 - Halve # of threads for next step
 - Takes $\log(n)$ steps for n elements
 - Requires $n/2$ threads at most in a step
- 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

Example of Parallel Reduction



Baseline Thread to Data Mapping

- Each thread is responsible for an even-index location of the partial sum vector
- After each step, half of the threads are no longer needed
- One of the inputs is always from the location of responsibility
- In each step, one of the inputs comes from an increasing distance away

Simple Thread Block Design

- Each thread block takes $2 * \text{BlockDim.x}$ 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];
```

Reduction

```
for (unsigned int stride = 1; stride <= blockDim.x;
    stride *= 2)
{
    __syncthreads();
    if (t % stride == 0)
        partialSum[2*t] += partialSum[2*t+stride];
}
```

Synchronization Barrier

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

Finishing Up Reduction

- At the end of the kernel, Thread 0 in each thread block writes the sum of the thread block in `partialSum[0]` into a vector indexed by the `blockIdx.x`
- There can be a large number of such sums if the original vector is very large
 - The host code may iterate and launch another kernel
- If there are only a small number of sums, the host can simply transfer the data back and add them together.

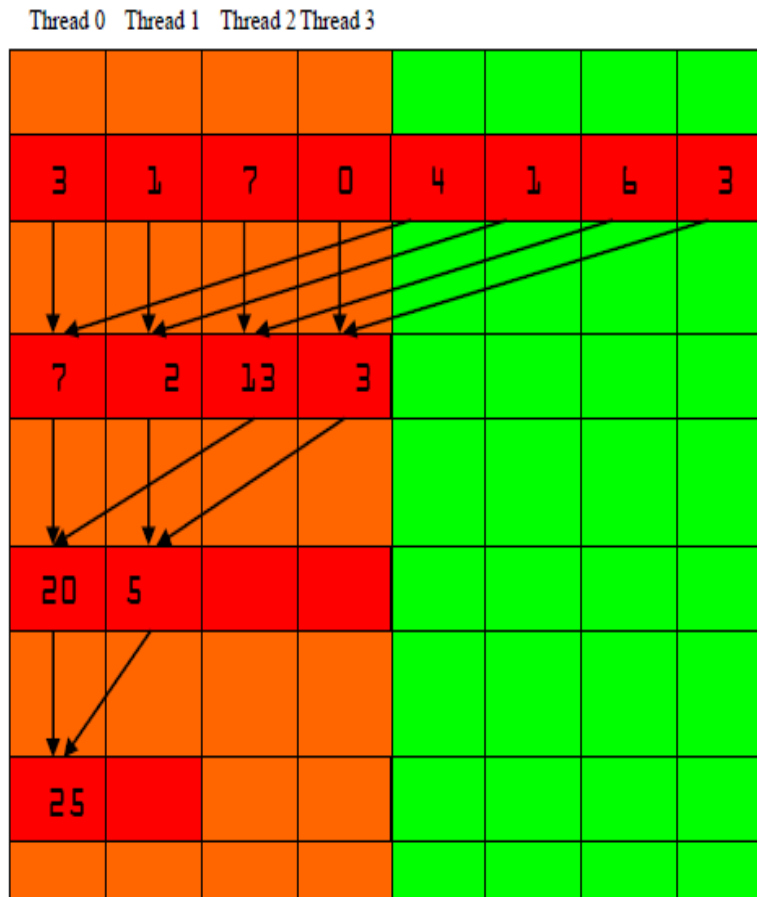
Problems in the Simple Reduction Kernel

- In each iteration, two control flow paths will be sequentially traversed for 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

Thread Index Usage Matters

- In some algorithms, one can shift the index usage to improve the divergence behavior
 - Commutative and associative operators
- Always compact the partial sums into the front locations in the partialSum[] array
- Keep the active threads consecutive

An Example of Four Threads



A Better Reduction Kernel

```
for (unsigned int stride = blockDim.x; stride > 0; stride /= 2)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
6
```

Analysis on the Better Kernel

- 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
 - All threads in each warp either all active or all inactive
 - The final 5 steps will still have divergence

Summary

- Reduction or reduce is also a data-parallel primitive
- Sequential implementation is of $O(n)$ time complexity
- Parallel reduction tree algorithm is work efficient
- Thread index mapping improves reduction kernel performance