www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Reductions

Antonio J. Peña

Based on material from NVIDIA's GPU Teaching Kit

Barcelona, July 4-6 2016

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

PARALLEL REDUCTION

Partition and Summarize

- 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
- We will focus on the reduction tree step for now

Reduction Enables Other Techniques

- 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

What is a Reduction Computation?

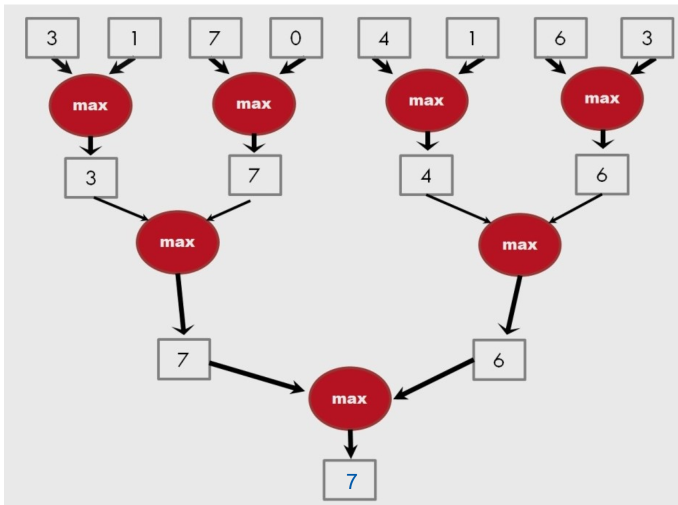
- Summarize a set of input values into one value using a "reduction operation"
  - Max
  - Min
  - Sum
  - Product
- Often used with a user defined reduction operation function as long as the operation:
  - Is associative and commutative
  - Has a well-defined identity value (e.g., 0 for sum)
  - For example, the user may supply a custom "max" function for 3D coordinate data sets where the magnitude for the each coordinate data tuple is the distance from the origin.

An example of "collective operation"

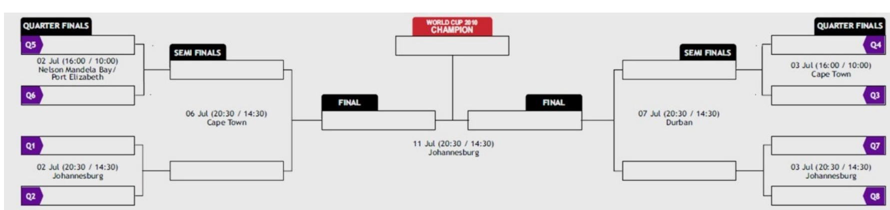
An Efficient Sequential Reduction O(N)

- 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
  - Each input value is only visited once – an O(N) algorithm
  - This is a computationally efficient algorithm.

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



A tournament is a reduction tree with "max" operation



A Quick Analysis

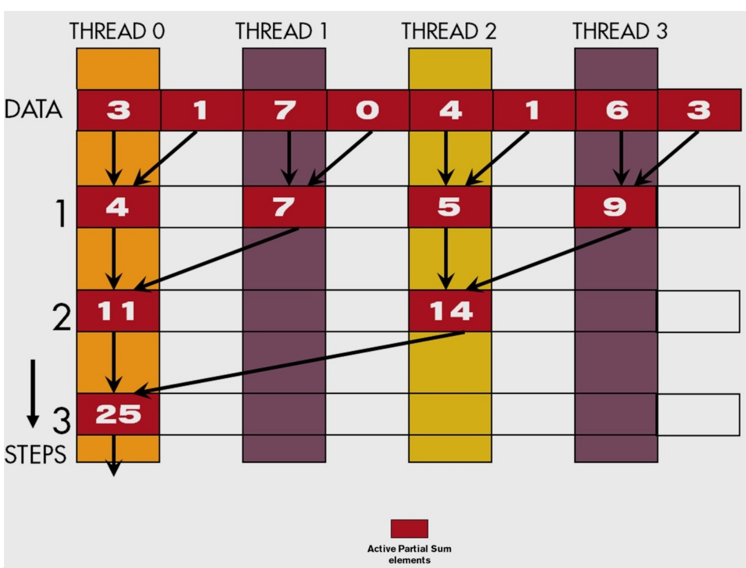
- For N input values, the reduction tree performs
  - (1/2)N + (1/4)N + (1/8)N + … (1)N = (1- (1/N))N = N-1 operations
  - In Log (N) steps – 1,000,000 input values take 20 steps
    - Assuming that we have enough execution resources

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Parallel Sum Reduction

- 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
- 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 of the partial sum vector
  - Reduces global memory traffic due to partial sum values
  - Thread block size limits n to be less than or equal to 2,048

A Parallel Sum Reduction Example



A Naive Thread to Data Mapping

- Each thread is responsible for an even-index location of the partial sum vector (location of responsibility)
- 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

A Simple Thread Block Design

- Each thread block takes 2*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+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()?

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

15

Barrier Synchronization

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

16

Back to the Global Picture

- 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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

17

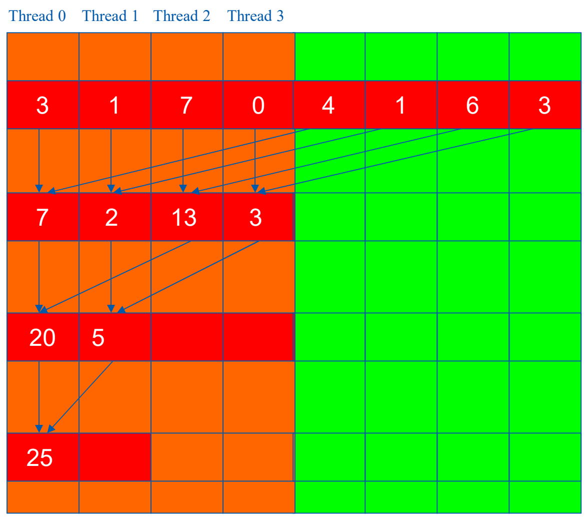
Some Observations on the Naïve Reduction Kernel

- 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 4 threads



A Better Reduction Kernel

```
for (unsigned int stride = blockDim.x;
      stride > 0;  stride /= 2)
{
  __syncthreads();
  if (t < stride)
      partialSum[t] += partialSum[t+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
    - All threads in each warp either all active or all inactive
  - The final 5 steps will still have divergence

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

23

Question 1

- For the following basic reduction kernel code fragment, if the block size is 1024 and warp size is 32, how many warps in a block will have divergence during the iteration where stride is equal to 1?

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

a) 0
b) 1
c) 16
d) 32

Question 1 - Answer

- For the following basic reduction kernel code fragment, if the block size is 1024 and warp size is 32, how many warps in a block will have divergence during the iteration where stride is equal to 1?

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

a) 0
b) 1
c) 16
d) 32

Explanation: During the first iteration, all threads in each warp are active. There is no control divergence.

Question 2

- In the same code, how many warps in a block will have divergence during the iteration where stride is equal to 16?

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

a) 0
b) 1
c) 16
d) 32

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Question 2 - Answer

- In the same code, how many warps in a block will have divergence during the iteration where stride is equal to 16?

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


a) 0
b) 1
c) 16
d) 32

Explanation: During each iteration, 1/stride of the threads in each warp are active. When stride is 16, every warp will have 32/16= 2 active threads that execute the addition statement. All 32 warps will have control divergence.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Question 3

- In the same code, how many warps in a block will have divergence during the iteration where stride is equal to 64?

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

a) 0
b) 1
c) 16
d) 32

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

29

Question 3 - Answer

- In the same code, how many warps in a block will have divergence during the iteration where stride is equal to 64?

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

a) 0
b) 1
c) 16
d) 32

Explanation: There will be one active thread in every 64 threads or 2 warps. So, 1/2 of the warps or 16 warps have divergence. The other ¾ of the warps have only inactive threads and thus no divergence.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

30

Question 4

- For the following improved reduction kernel, if the block size is 1024 and warp size is 32, how many warps will have divergence during the iteration where stride is equal to 16?

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

a) 0
b) 1
c) 16
d) 32

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Question 4 – Answer

- For the following improved reduction kernel, if the block size is 1024 and warp size is 32, how many warps will have divergence during the iteration where stride is equal to 16?

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

a) 0
b) 1
c) 16
d) 32

Explanation: In each iteration, there are stride consecutive active threads. During the iteration where stride is 16, there are 16 consecutive active threads, all in the same warp. All other warps have only inactive threads. So one warp has control divergence and 31 will not.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Question 5

- In the same code, how many warps in a block will have divergence during the iteration where stride is 64?

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

a) 0
b) 1
c) 16
d) 32

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Question 5 - Answer

- In the same code, how many warps in a block will have divergence during the iteration where stride is 64?

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

a) 0
b) 1
c) 16
d) 32

Explanation: There are 64 consecutive active threads, which is a multiple of warp size. So two warps will have all their threads active and 30 warps will have all their threads inactive. None of them will have control divergence.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact
antonio.pena@bsc.es