NVIDIA

GPU Teaching Kit

Accelerated Computing

西南石油大學 计算机科学学院

SCHOOL OF COMPUTER SCIENCE, SOUTHWEST PETROLEUM UNIVERSITY

Module 5.2 – Parallel Computation Patterns (Reduction)

Parallel Reduction

Objective

- **To learn the parallel reduction pattern**
  - An important class of parallel computation
  - **Work efficiency** analysis
  - **Resource efficiency** analysis

“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
- E.G., **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(复制) the output location so that each thread has a private output location (privatization)
  - Use a **reduction tree** to combine the values of private locations into the original output location

NVIDIA

What is a reduction computation?

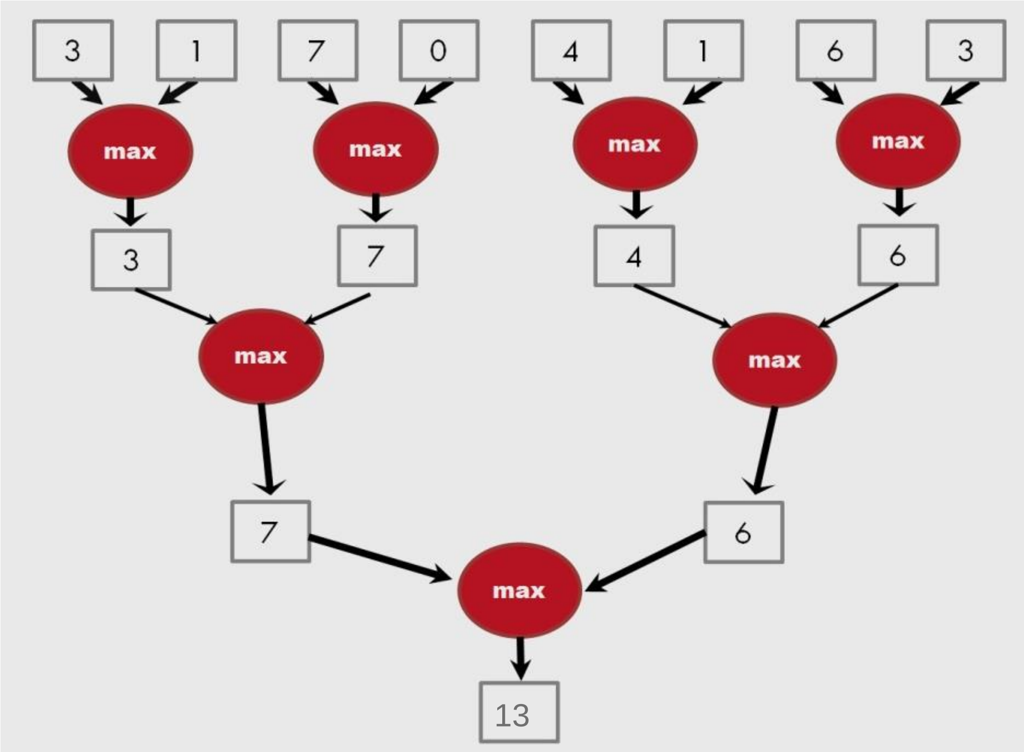
- Summarize **a set of input values** into one value using a "**reduction operation**"
  - Max
  - Min
  - Sum
  - Product
- Often used with a user defined reduction operation function as long as the operation:
  - Is associative and commutative
  - Has a well-defined(明确的) identity value (e.g., 0 for 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(集合操作)"

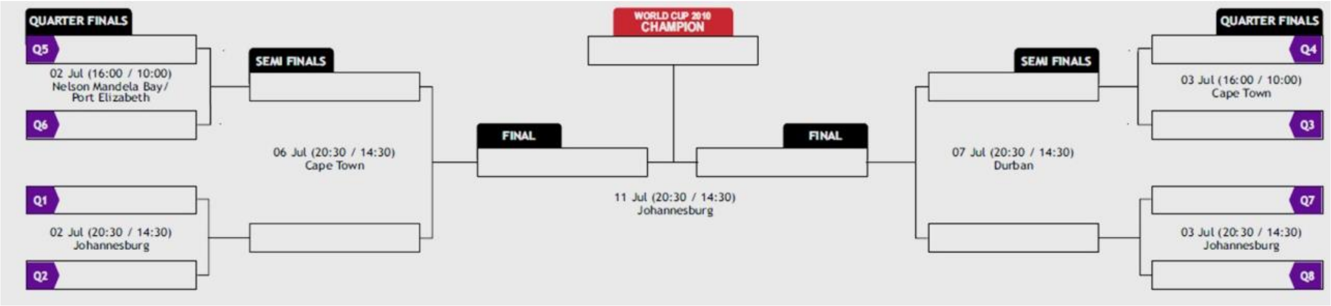
An Efficient Sequential Reduction O(N)

- 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 <u>the result value</u> and <u>the current input value</u>**
  - 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** – 1,048,576(2^20) input values take 20 steps
    - Assuming that we have enough execution resources
  - **Average Parallelism (N-1)/Log(N))**
    - For N = 1,048,576, average parallelism is approximately 52,000
    - 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 the an efficient sequential algorithm
  - Many parallel algorithms are not work efficient

GPU Teaching Kit

Accelerated Computing

Module 9 – Parallel Computation Patterns (Reduction)

Lecture 9.2 - A Basic Reduction Kernel

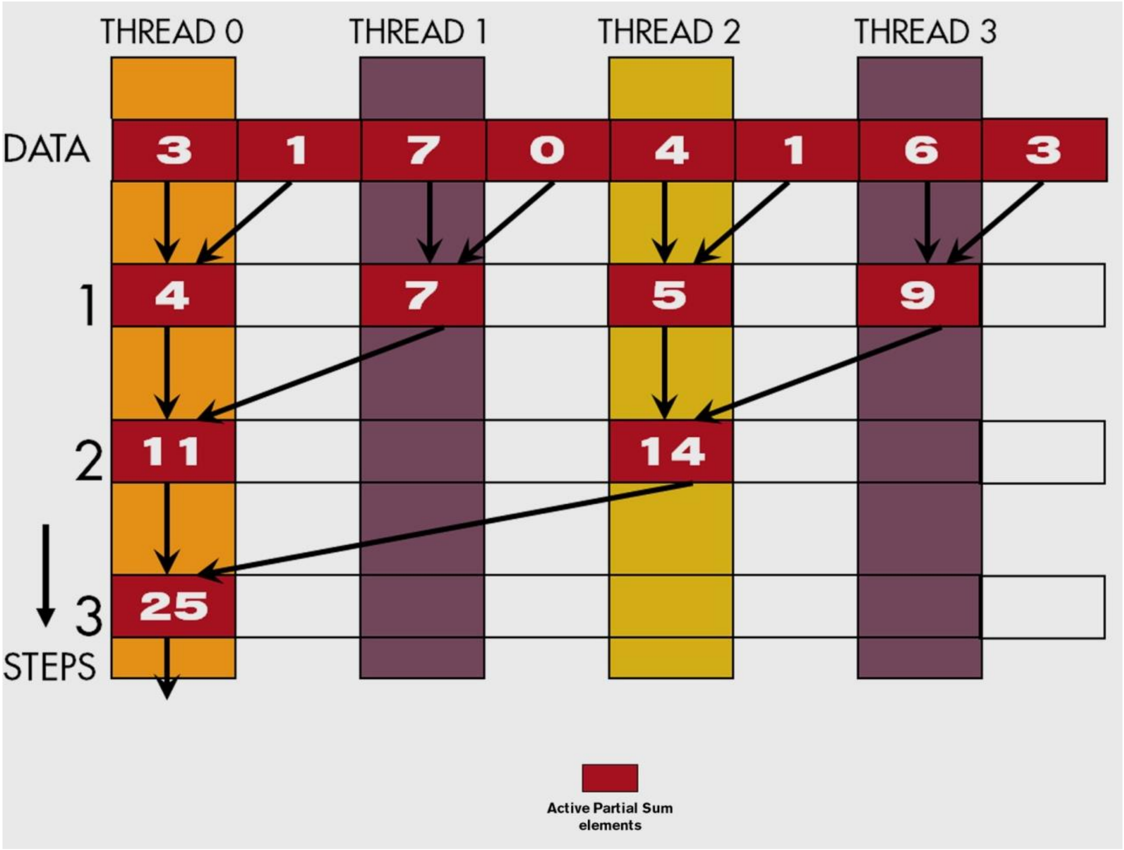
Objective

- **To learn to write a basic reduction kernel**
  - Thread to data mapping
  - Turning off threads
  - Control divergence

Parallel Sum Reduction

- Parallel implementation
  - **Each thread adds two values** in each step
  - Recursively halve # of threads
  - Takes **log(n) steps** for **n elements**, requires **n/2 threads**
- Assume an in-place(就地) reduction using shared memory
  - The original vector is in **device global memory**
  - **The shared memory** is used to hold a partial sum vector
  - Initially, the partial sum vector is simply the original 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 reading and writing 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 tx = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[tx] = input[start + tx];
partialSum[blockDim+tx] = input[start + blockDim.x+tx];
```

The Reduction Steps

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

Why do we need __syncthreads()?

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

Back to the Global Picture

- At the end of the kernel, **Thread 0** in each 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

- Alternatively, Thread 0 of each block could use **atomic operations** to accumulate into a global sum variable.

NVIDIA®

GPU Teaching Kit

Accelerated Computing

西南石油大學计算机科学学院
SCHOOL OF COMPUTER SCIENCE, SOUTHWEST PETROLEUM UNIVERSITY

Module 9 – Parallel Computation Patterns (Reduction)

Lecture 9.3 - A Better Reduction Kernel

Objective

- **To learn to write a <u>better reduction kernel</u>**
  - Improved resource efficiency
  - Improved thread to data mapping
  - Reduced control divergence

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

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

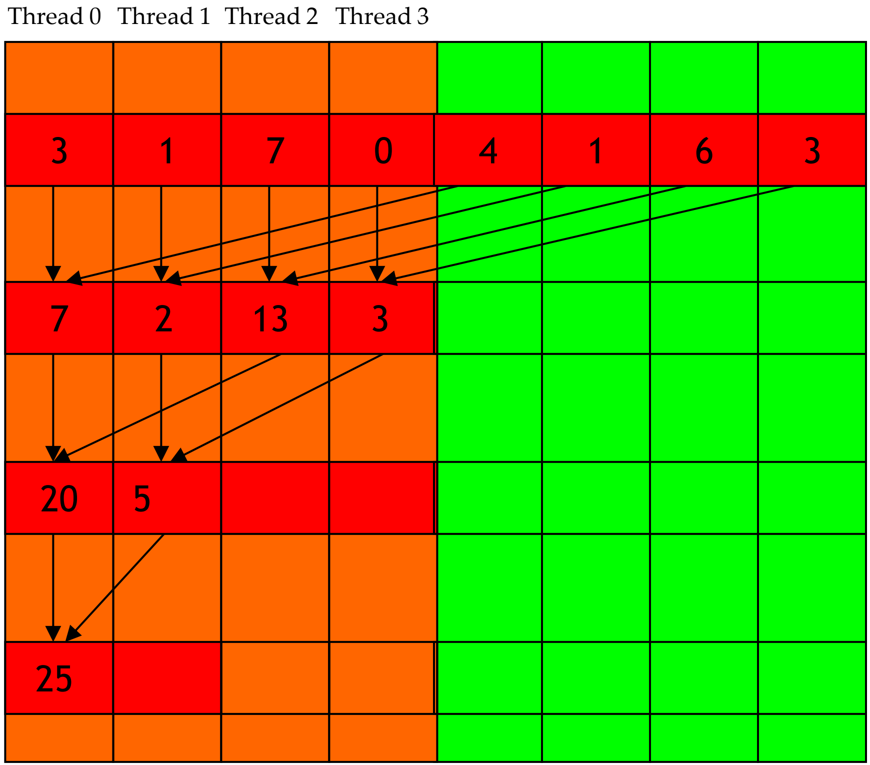
Thread Index Usage Matters

- 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 <u>in the first 5 steps</u>**
    - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
    - All threads in each warp either all active or all inactive
  - **<u>The final 5 steps</u> will still have divergence**
    - 16, 8, 4, 2, 1 consecutive threads are active in each step
    - Fewer than 32 threads in each warp

GPU Teaching Kit

Accelerated Computing