GPU Teaching Kit

Accelerated Computing

Module 9 – Parallel Computation Patterns (Reduction)

Lecture 9.3 - A Better Reduction Kernel

Objective

- To learn to write a better reduction kernel
  - Improved resource efficiency
  - Improved thread to data mapping
  - Reduced control divergence

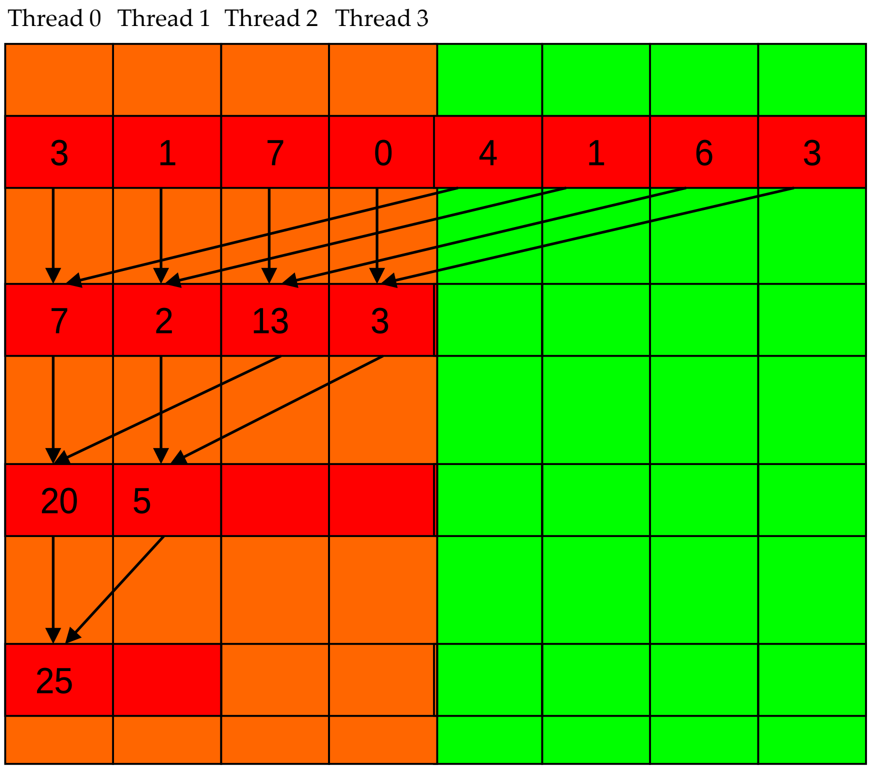
Some Observations on the naïve reduction kernel

- 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

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

GPU Teaching Kit

Accelerated Computing