CS 6023 - GPU Programming

Parallel Reduce Operation

12/10/2018

Setting and Agenda

- We are looking at some common parallel programming patterns and how to optimize them for GPUs
    - Parallel reduce
- Work-efficient and resource-efficient parallel algorithm

Reduction

- For geometric data decomposition, we can divide data into parallel chunks and later **reduce** them to produce the output
- Reduction = Operation that computes a single result from a set of data
- What are some examples from course?

Reduction

- For geometric data decomposition, we can divide data into parallel chunks and later **reduce** them to produce the output
- Reduction = Operation that computes a single result from a set of data
- What are some examples from course?
  - Histogram computation
  - Convolution in input/output stationary data flows
  - More generally, this is common in other programming interfaces (eg. Map-Reduce by Google)

Examples of reduction operation

- Examples
  - Min
  - Max
  - Product
  - Sum
- When does this work?

Examples of reduction operation

- Examples
  - Min
  - Max
  - Product
  - Sum
- When does this work in parallel?
  - Commutative and associative
  - Identity value for initialization

Trivial solutions

- Consider a reduction (say max) on N numbers. What is the complexity of

- Sequential program

- Parallel program

Trivial solutions

- Consider a reduction (say max) on N numbers. What is the complexity of

- Sequential program

  O(N) - Process each value in one step

- Parallel program

Trivial solutions

- Consider a reduction (say max) on N numbers. What is the complexity of

- Sequential program

  O(N) - Process each value in one step

- Parallel program

  What does complexity mean?

Parallel solutions

- Simple approach - Apply reduction operation recursively to pairs
    - O(log N) time
    - O(N) operations
    - O(N) processors

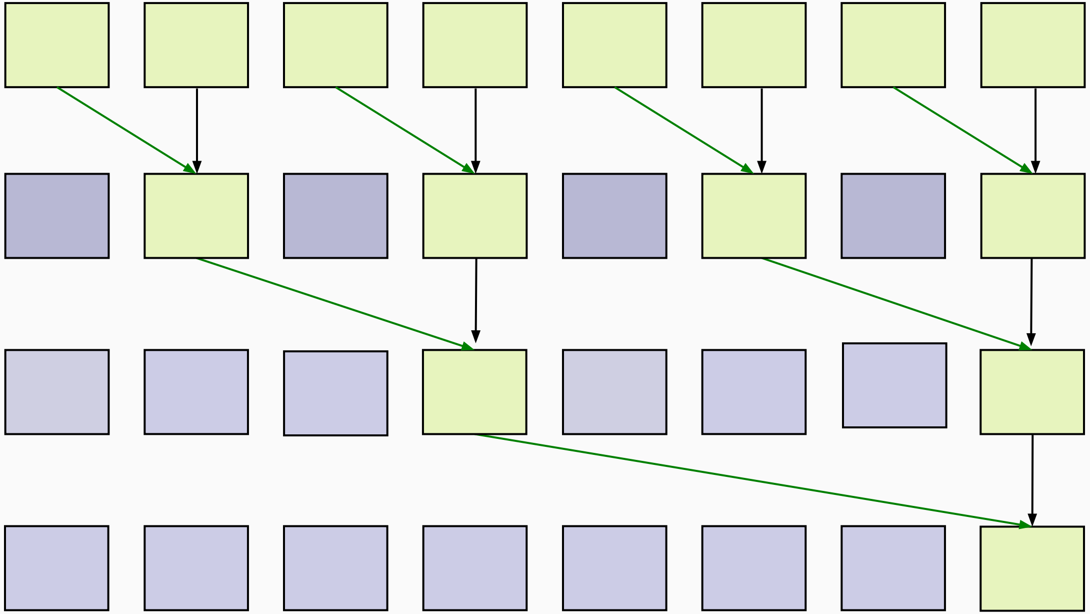
Parallel solutions

- Simple approach - Apply reduction operation recursively to pairs
  - $O(\log N)$ time
  - $O(N)$ operations
  - $O(N)$ processors

- Speed-up vs sequential = $(N - 1) / \log N$

- If N is very large, then parallelism is large, but number of processors required is also large

Parallel reduce - Simple code



```
for i = 0 to log2N - 1
  for all j = 0 to N − 1 by 2^(i+1) in parallel
    x[j + 2^(i+1) − 1] += x[j + 2^i − 1];
```

- In-place modification of data
- Number of threads gets halved in each iteration
- Maximum number of threads is N/2

CUDA code

- Usual questions: What will each thread/block do?

CUDA code

- Usual questions: What will each thread/block do?
- Let a block of N threads read in 2*N values

```
__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];

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

CUDA code

- Usual questions: What will each thread/block do?
- Let a block of N threads read in 2*N values

```
__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];

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

Are we missing something?

CUDA code

- Usual questions: What will each thread/block do?
- Let a block of N threads read in 2*N values

```
__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];

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

CUDA code

- Have we solved the problem?

CUDA code

- Have we solved the problem?
- We have summed up only the values assigned to multiple threads of a block? What about the global sum?

Optimization

- Do you see any performance bugs?

```
__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];

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

Optimization

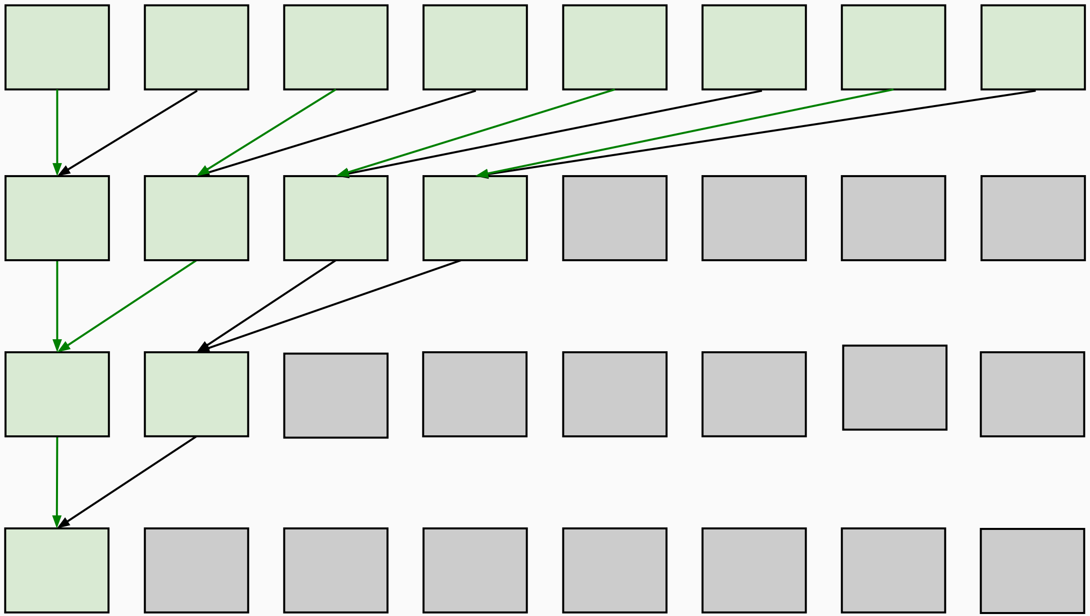
- Do you see any performance bugs?

```
__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];

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

Lot of control divergence on adjacent warps

Optimization

- Remove control divergence amongst consecutive warps by compacting
- Exercise: Try out the CUDA kernel



CUDA code

- Class exercise (solution added at the end)

Exercise

- For a 1024 thread block, what are the number of warps with control divergence in each iteration

Exercise

- For a 1024 thread block, what are the number of warps with control divergence in each iteration
  - First 5 iterations (1024, 512, 256, 128, 64, 32 threads with work) there is NO control divergence
  - For the next 5 iterations 1 warp will have control divergence

Work and resource efficiency

- What is the total work done by the sequential algorithm? N-1 adds
- What is the total work done by the parallel algorithm? N-1 adds
- When both sequential and parallel algorithms do the same work, we say the parallel algorithm is **work-efficient**
- Many parallel algorithms are not work efficient (eg. prefix sum in next class)

Work and resource efficiency

- What is the total work done by the sequential algorithm? N-1 adds
- What is the total work done by the parallel algorithm? N-1 adds
- When both sequential and parallel algorithms do the same work, we say the parallel algorithm is **work-efficient**
- Many parallel algorithms are not work efficient (eg. prefix sum in next class)

- But the parallel reduce algorithm is not resource efficient
- In sequential case: 1 thread runs for N-1 time
- In parallel case: N/2 threads are reserved at the start for log(N) time
- In some algos, trade-off exists between resource usage and work efficiency

CUDA code for compact parallel reduce with lesser control divergence

```
__shared__ float partialSumPing[2*BLOCK_SIZE], partialSumPong[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;

partialSumPing[t] = input[start + t];
partialSumPing[blockDim+t] = input[start + blockDim.x+t];
partialSumPong[t] = 0;
partialSumPong[blockDim+t] = 0;

bool odd = true;

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

Why do we need two arrays?