CSCI 5451: Introduction to Parallel Computing

Lecture 26: Reduction in Cuda

computer science
& engineering

University of Minnesota
Driven to Discover®

Reduction (Recall)

Reductions involve an operation performed over some set of data

Reduction (Recall)

Reductions involve an operation performed over some set of data

Addition

```
sum = 0.0f;
for(i = 0; i < N; ++i) {
    sum += input[i];
}
```

Reduction (Recall)

Reductions involve an operation performed over some set of data

Addition

```
sum = 0.0f;
for(i = 0; i < N; ++i) {
    sum += input[i];
}
```

Generic Operations

```
acc = IDENTITY;
for(i = 0; i < N; ++i) {
    acc = Operator(acc, input[i]);
}
```

Reduction (Recall)

How to Parallelize?

Reductions involve an operation performed over some set of data

Addition

```
sum = 0.0f;
for(i = 0; i < N; ++i) {
    sum += input[i];
}
```

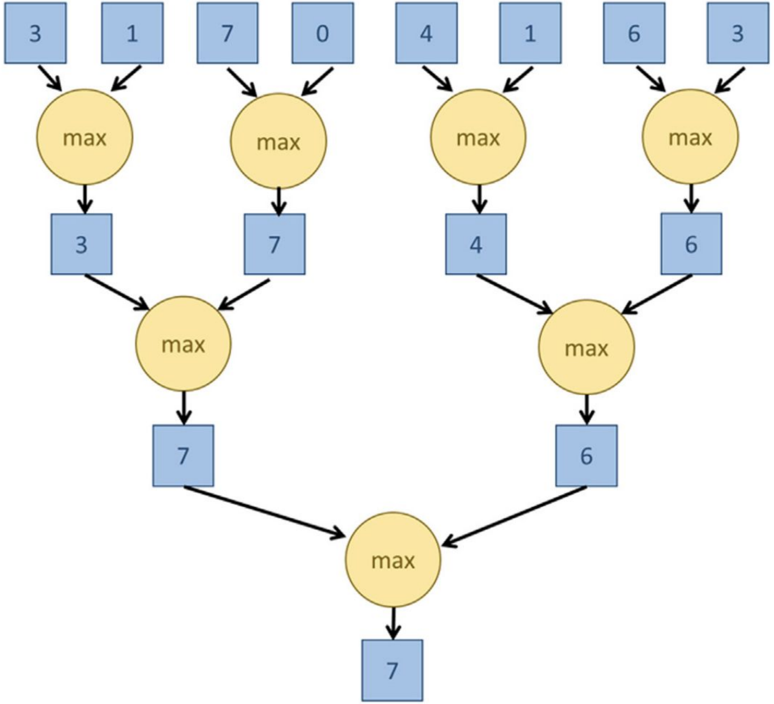
Generic Operations

```
acc = IDENTITY;
for(i = 0; i < N; ++i) {
    acc = Operator(acc, input[i]);
}
```

Reduction (Recall)

How to Parallelize?

Reduction Trees



Reductions involve an operation performed over some set of data

Addition

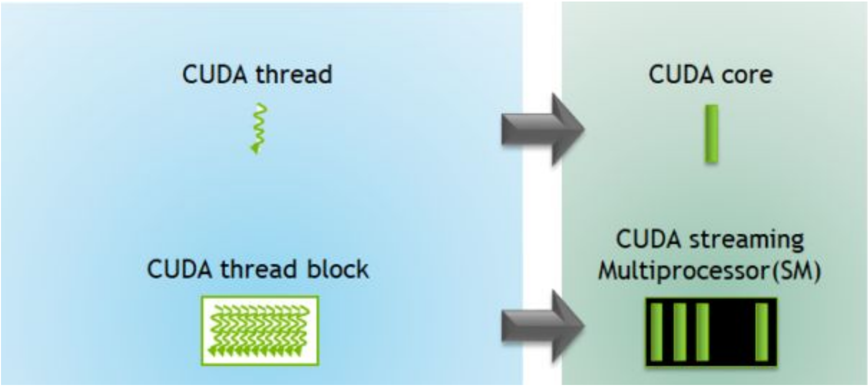
```
sum = 0.0f;
for(i = 0; i < N; ++i) {
    sum += input[i];
}
```

Generic Operations

```
acc = IDENTITY;
for(i = 0; i < N; ++i) {
    acc = Operator(acc, input[i]);
}
```

Simple Kernel Within one Block

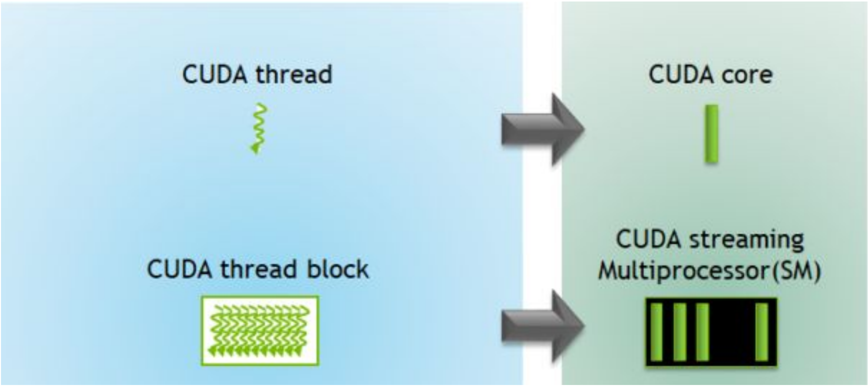
- To start with, let's write a reduction kernel using only a single threadblock
- If we have an array of N elements, what is the maximum amount of concurrency when performing a reduction
  -




```
my_kernel<<<1, num_threads>>>(arg1, arg2);
```

Simple Kernel Within one Block

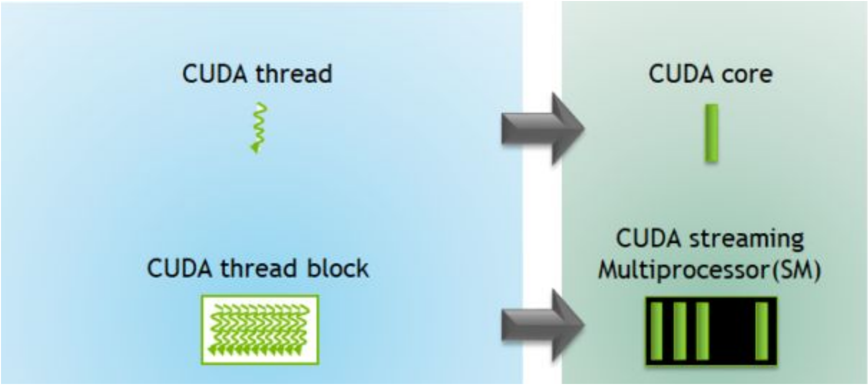
- To start with, let's write a reduction kernel using only a single threadblock
- If we have an array of N elements, what is the maximum amount of concurrency when performing a reduction
  - N/2
- Given this, what is the maximum array size we can perform reduction on in a single threadblock
  -




```
my_kernel<<<1, num_threads>>>(arg1, arg2);
```

Simple Kernel Within one Block

- To start with, let's write a reduction kernel using only a single threadblock
- If we have an array of N elements, what is the maximum amount of concurrency when performing a reduction
  - N/2
- Given this, what is the maximum array size we can perform reduction on in a single threadblock
  - 2048 (i.e. number of threads in a block divided by 2)




```
my_kernel<<<1, num_threads>>>(arg1, arg2);
```

Simple Kernel Within one Block

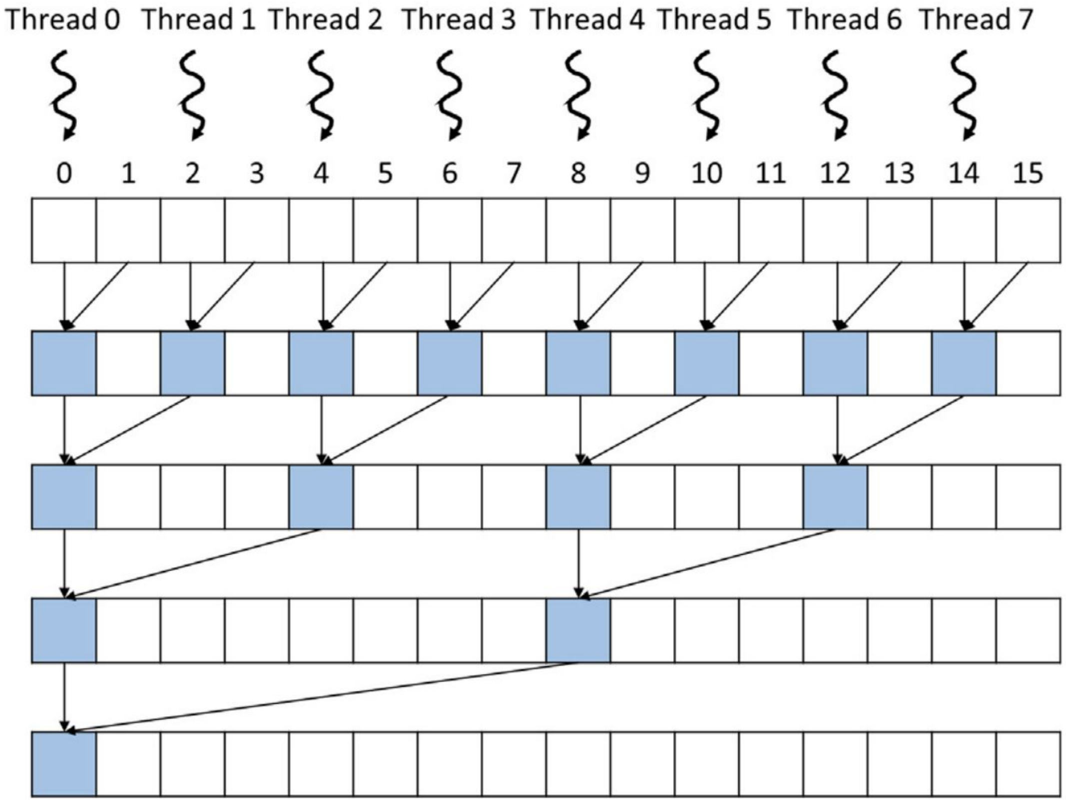
```
__global__ void SimpleSumReductionKernel(float* input, float* output) {
    unsigned int i = 2*threadIdx.x;
    for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
        if (threadIdx.x % stride == 0) {
            input[i] += input[i + stride];
        }
        __syncthreads();
    }
    if(threadIdx.x == 0) {
        *output = input[0];
    }
}
```

Simple Kernel Within one Block

What does this reduction tree look like? Which threads access what elements of memory? Which memory locations are written to at each depth of the tree?

```
__global__ void SimpleSumReductionKernel(float* input, float* output) {
    unsigned int i = 2*threadIdx.x;
    for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
        if (threadIdx.x % stride == 0) {
            input[i] += input[i + stride];
        }
        __syncthreads();
    }
    if(threadIdx.x == 0) {
        *output = input[0];
    }
}
```

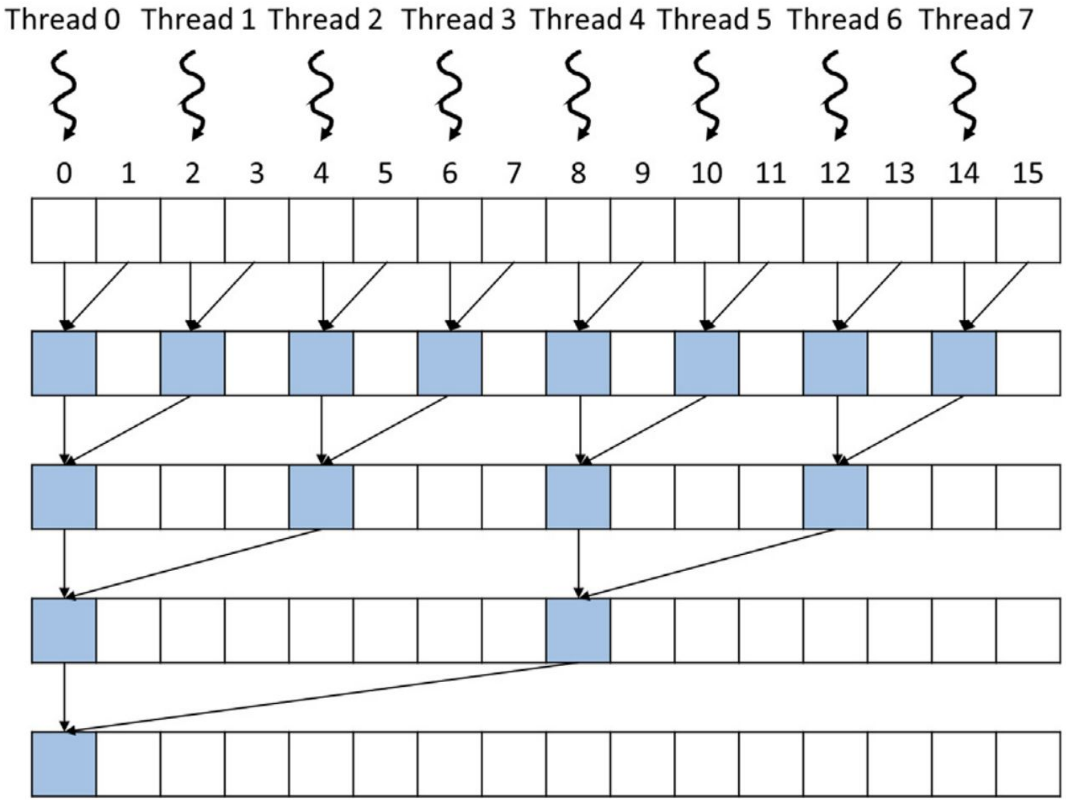
Simple Kernel Within one Block



Simple Kernel Within one Block

What are some problems with this implementation?

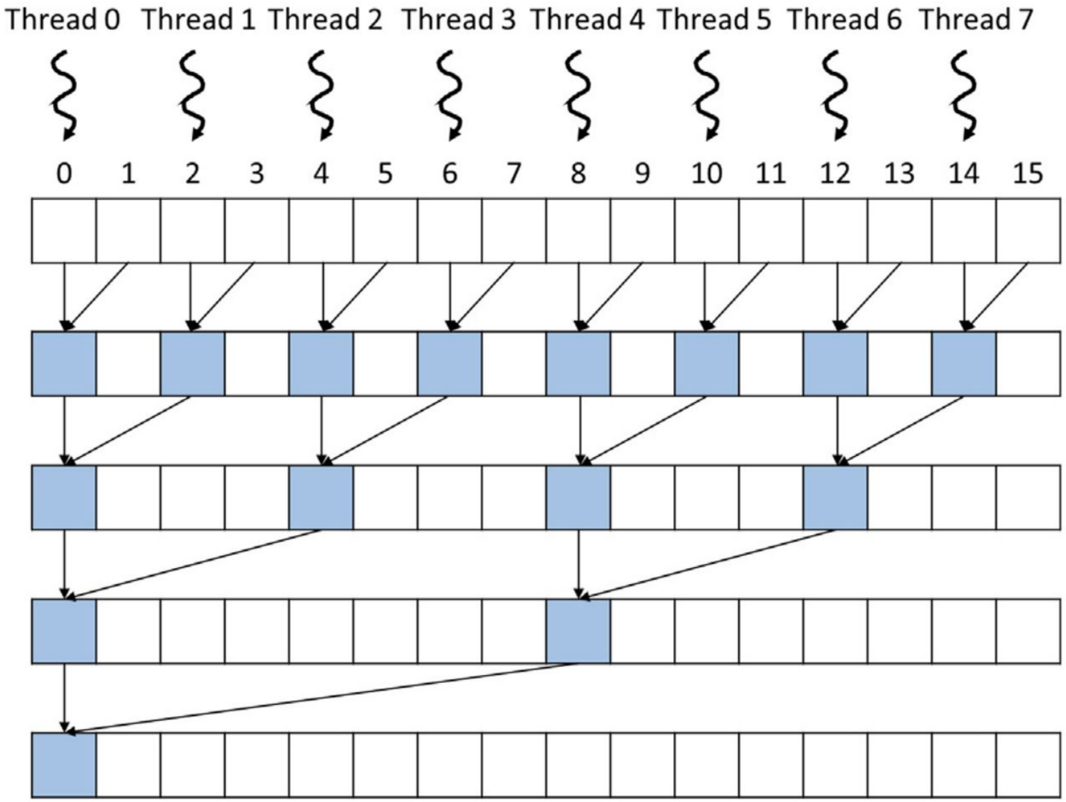
-



Simple Kernel Within one Block

What are some problems with this implementation?

- Warp Divergence.
- Divergent memory accesses (no coalescing)
- Redundant memory loading

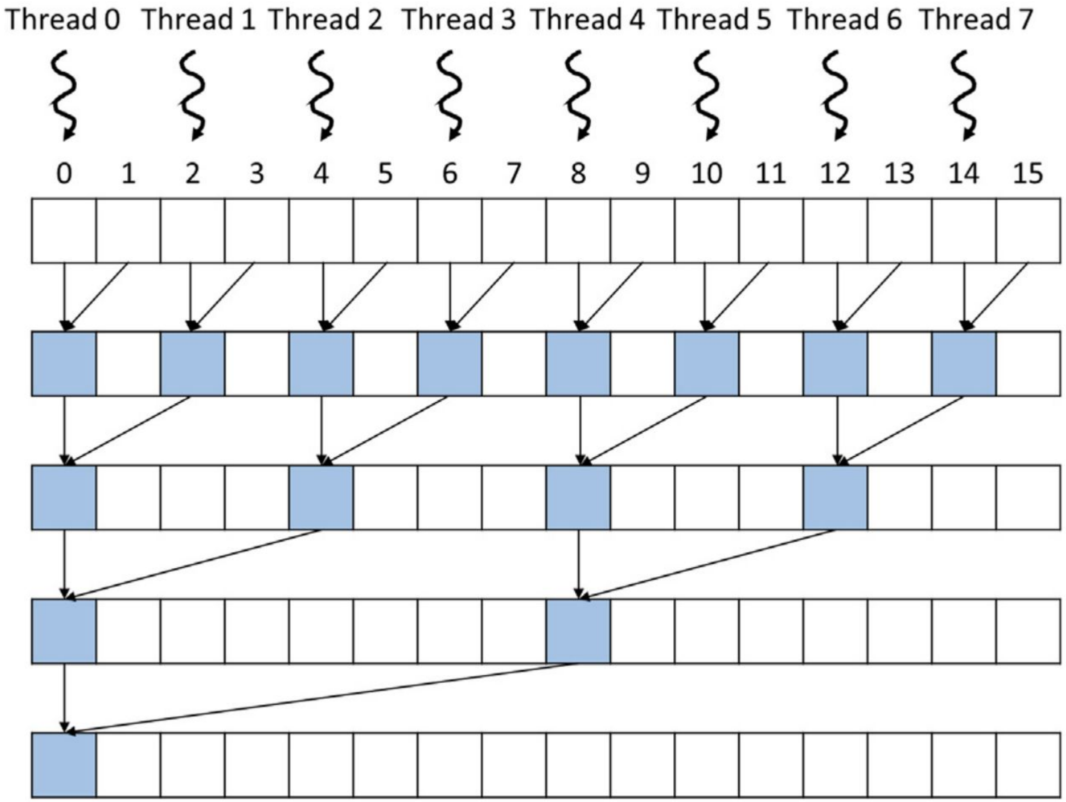


Simple Kernel Within one Block

What are some problems with this implementation?

- **Warp Divergence.**
- **Divergent memory accesses (no coalescing)**
- Redundant memory loading

Let’s fix these first



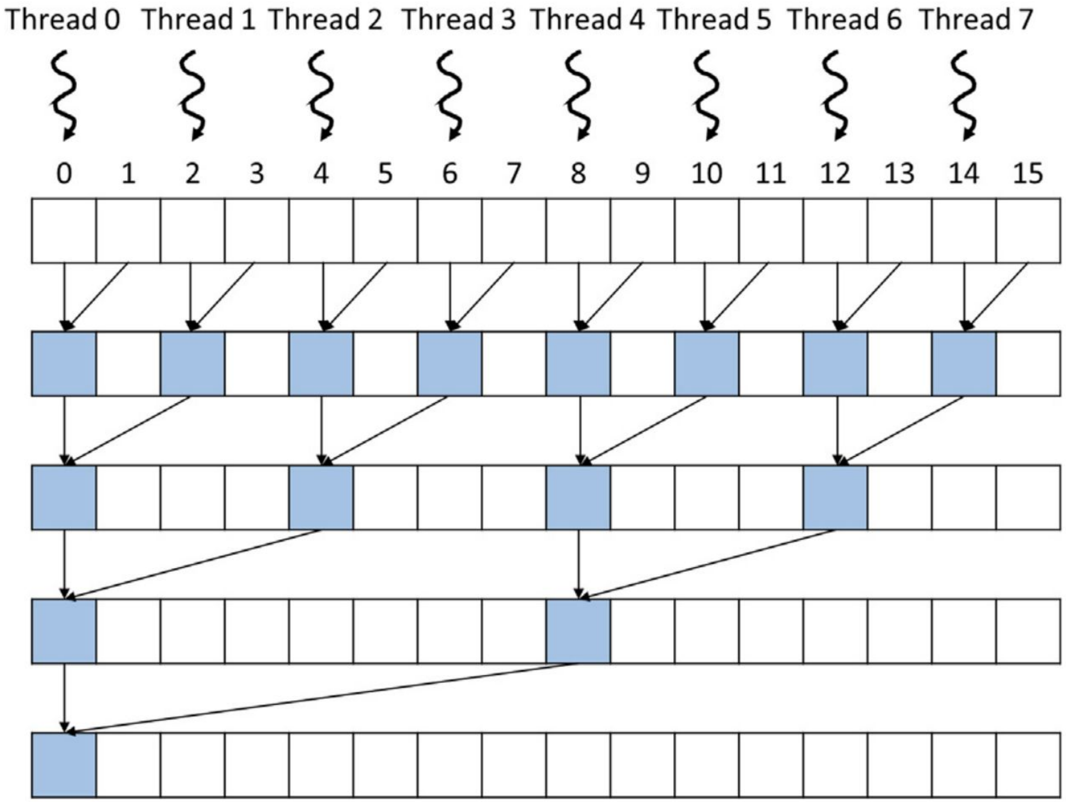
Simple Kernel Within one Block

How to fix this?

What are some problems with this implementation?

- **Warp Divergence.**
- **Divergent memory accesses (no coalescing)**
- Redundant memory loading

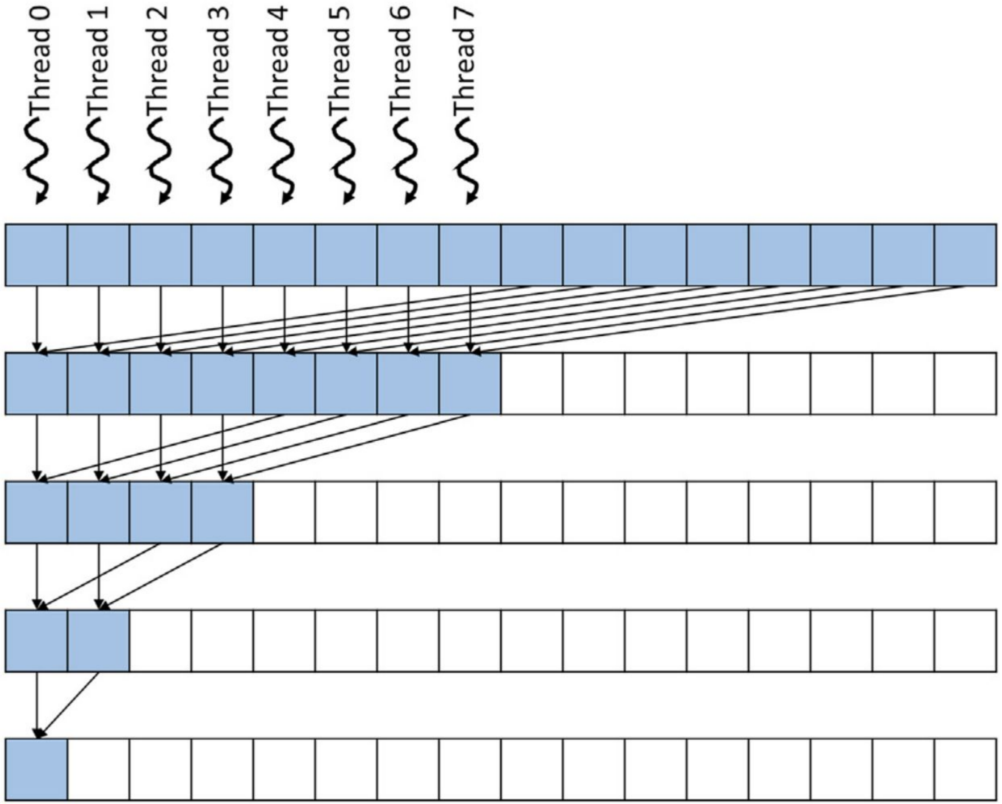
Let’s fix these first



Minimize Divergence Kernel

Threads now compute over adjacent memory locations.

- No warp divergence
- Coalesced reads/writes



Minimize Divergence Kernel

```
__global__ void ConvergentSumReductionKernel(float* input, float* output) {
    unsigned int i = threadIdx.x;
    for (unsigned int stride = blockDim.x; stride >= 1; stride /= 2) {
        if (threadIdx.x < stride) {
            input[i] += input[i + stride];
        }
        __syncthreads();
    }
    if(threadIdx.x == 0) {
        *output = input[0];
    }
}
```

Minimize Divergence Kernel

We have only changed these two lines

```
__global__ void ConvergentSumReductionKernel(float* input, float* output) {
    unsigned int i = threadIdx.x;
    for (unsigned int stride = blockDim.x; stride >= 1; stride /= 2) {
        if (threadIdx.x < stride) {
            input[i] += input[i + stride];
        }
        __syncthreads();
    }
    if(threadIdx.x == 0) {
        *output = input[0];
    }
}
```

Minimize Divergence Kernel

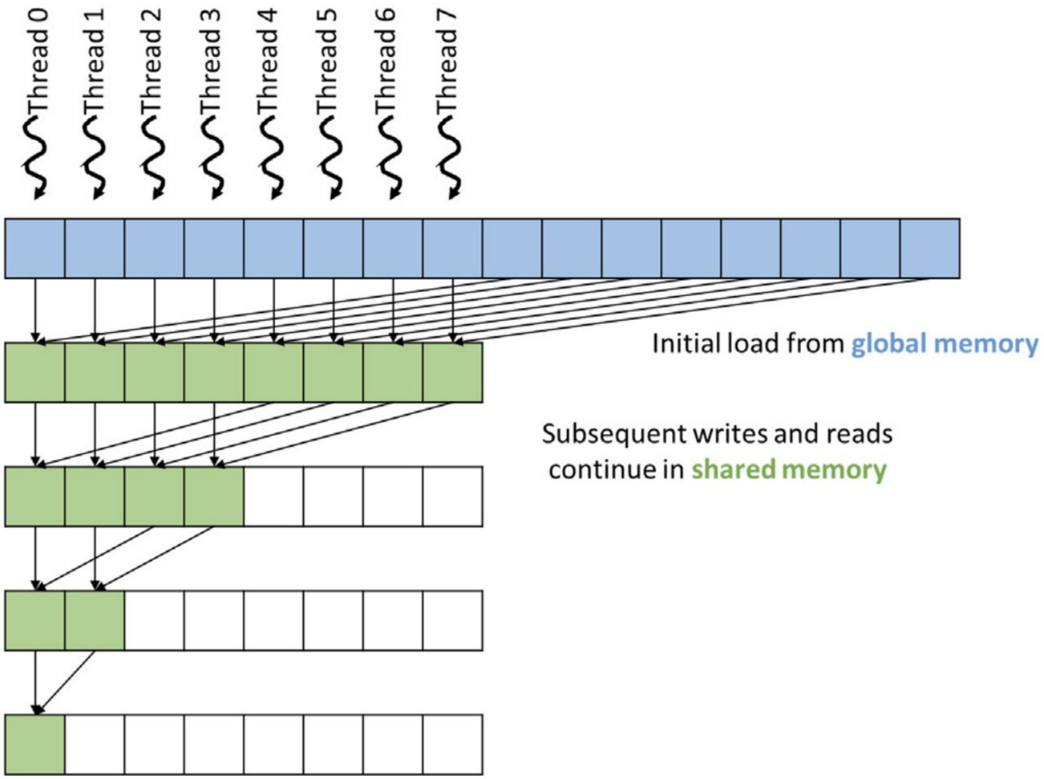
How can we improve this within block kernel further?

We have only changed these two lines

```
__global__ void ConvergentSumReductionKernel(float* input, float* output) {
    unsigned int i = threadIdx.x;
    for (unsigned int stride = blockDim.x; stride >= 1; stride /= 2) {
        if (threadIdx.x < stride) {
            input[i] += input[i + stride];
        }
        __syncthreads();
    }
    if(threadIdx.x == 0) {
        *output = input[0];
    }
}
```

Shared Memory Kernel

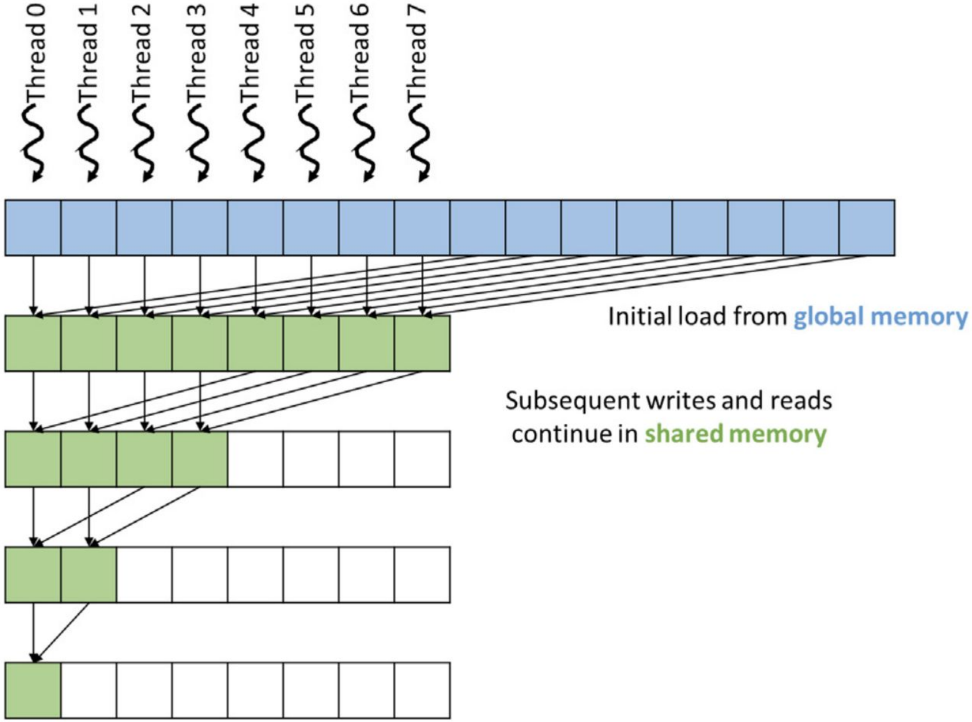
Repeat the same access pattern as in the last kernel, but only the first read and last write are from/to global memory.



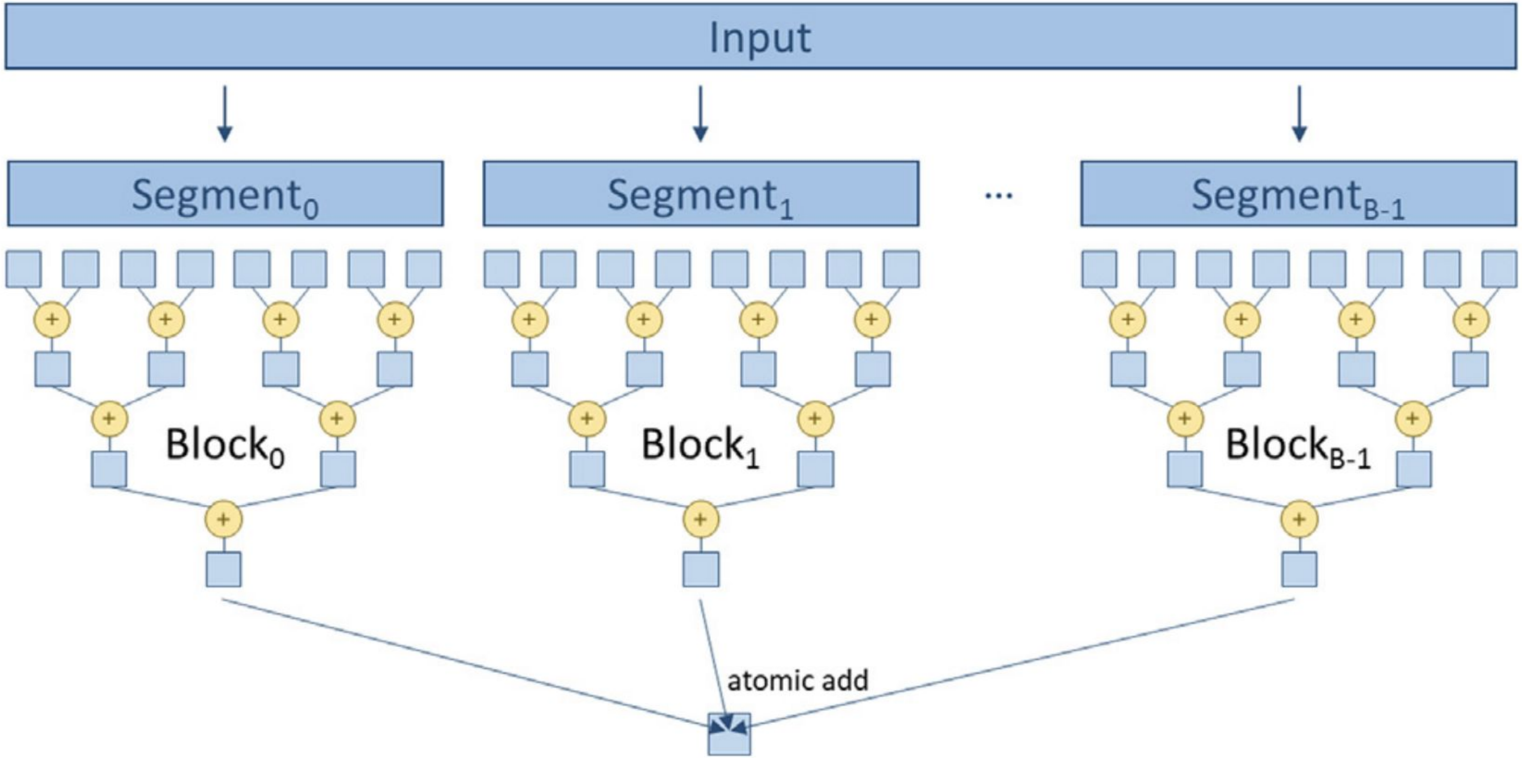
Shared Memory Kernel

```
__global__ void SharedMemorySumReductionKernel(float* input) {
    __shared__ float input_s[BLOCK_DIM];
    unsigned int t = threadIdx.x;
    input_s[t] = input[t] + input[t + BLOCK_DIM];
    for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2) {
        __syncthreads();
        if (threadIdx.x < stride) {
            input_s[t] += input_s[t + stride];
        }
    }
    if (threadIdx.x == 0) {
        *output = input_s[0];
    }
}
```

How can we extend this pattern to multiple blocks when the sequence length is greater than 2048?



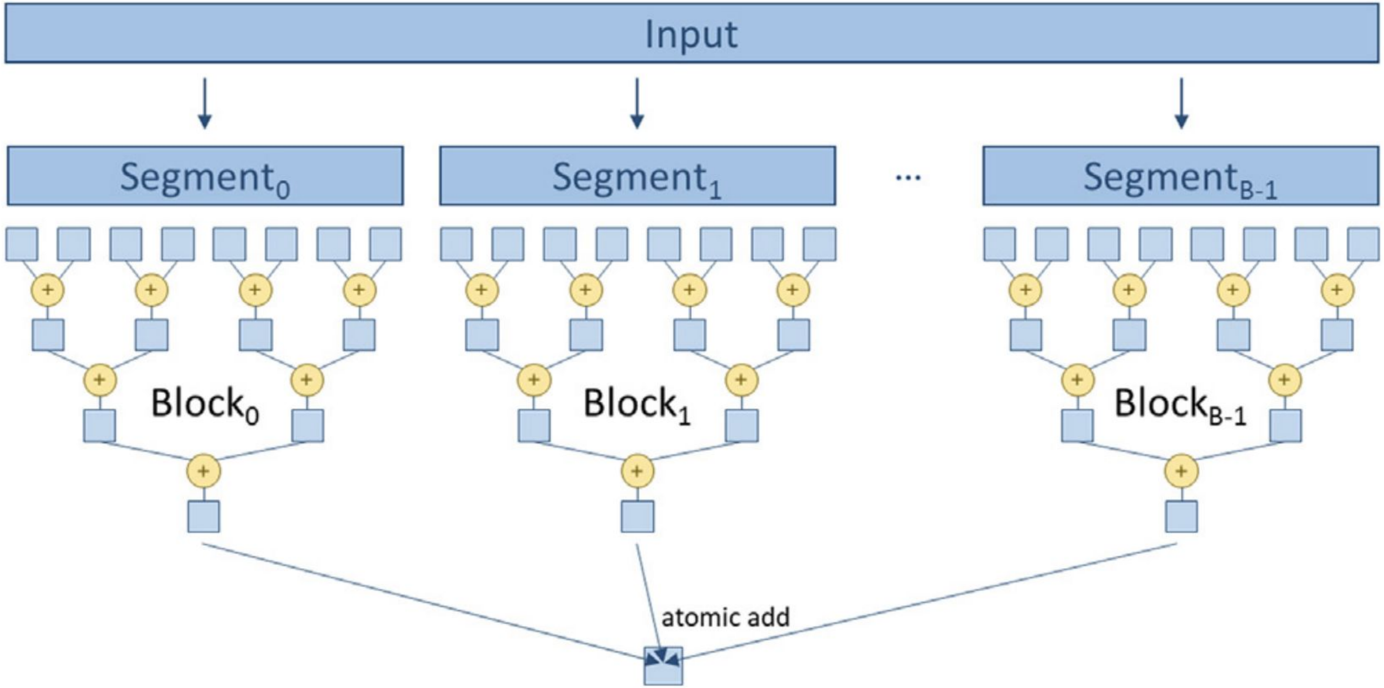
A Hierarchical Reduction Kernel



A Hierarchical Reduction Kernel

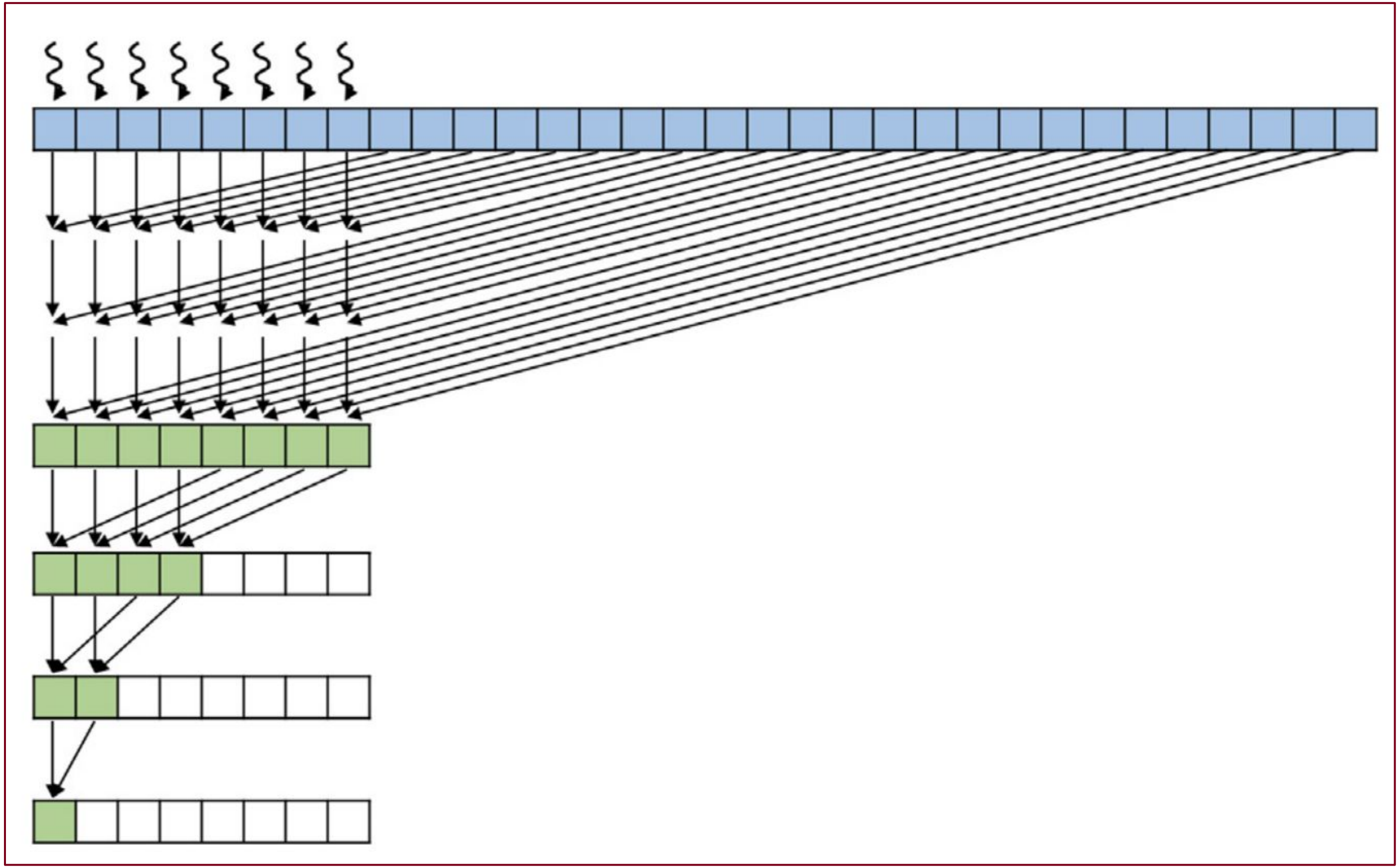
```
__global__ SegmentedSumReductionKernel(float* input, float* output) {
    __shared__ float input_s[BLOCK_DIM];
    unsigned int segment = 2*blockDim.x*blockIdx.x;
    unsigned int i = segment + threadIdx.x;
    unsigned int t = threadIdx.x;
    input_s[t] = input[i] + input[i + BLOCK_DIM];
    for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2){
        __syncthreads();
        if (t < stride) {
            input_s[t] += input_s[t + stride];
        }
    }
    if (t == 0) {
        atomicAdd(output, input_s[0]);
    }
}
```

What further improvements can we make to this access pattern?

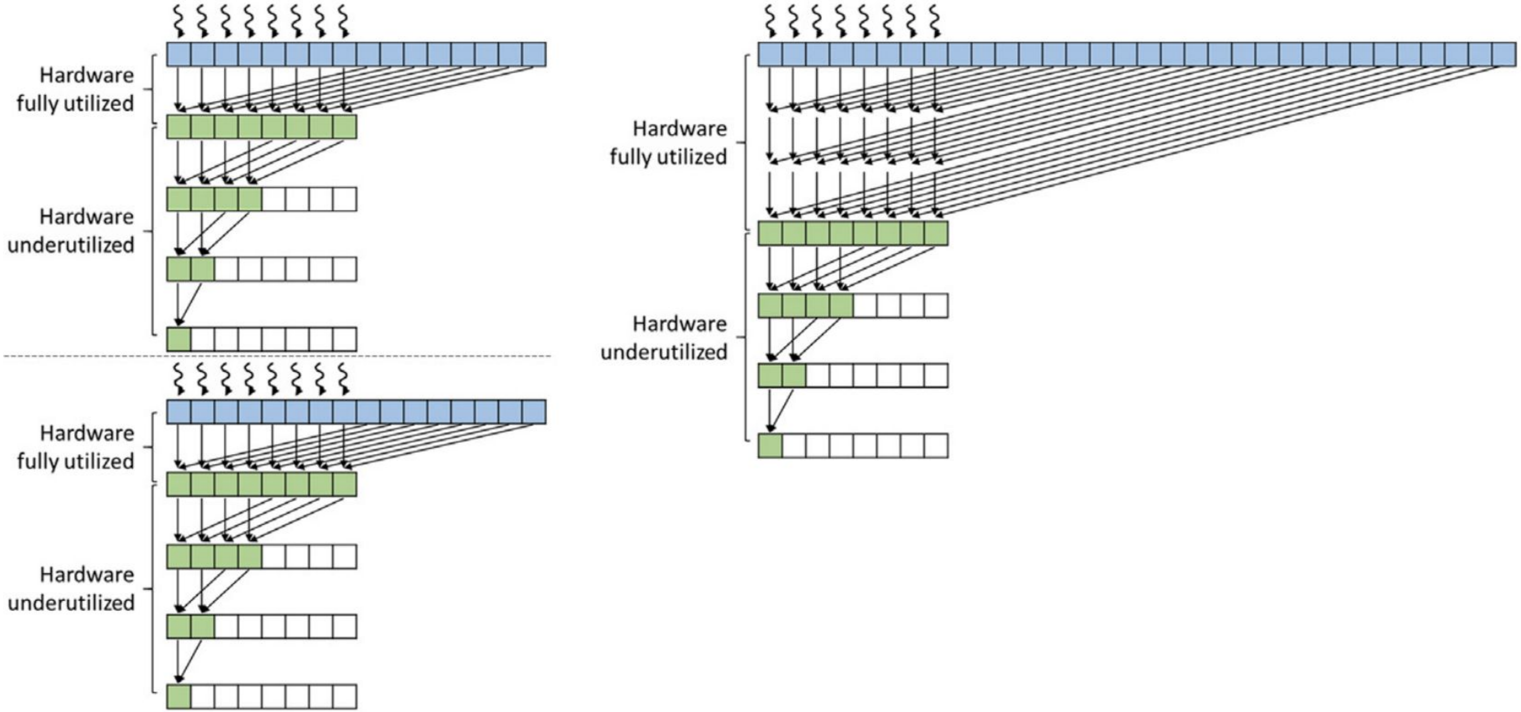


Coarsened Kernel

Each thread computes multiple outputs



Coarsened Kernel



(A) Execution of two original thread blocks serialized by the hardware

(B) Execution of one coarsened thread block doing the work of two original thread blocks

Coarsened Kernel

```
__global__ CoarsenedSumReductionKernel(float* input, float* output) {
    __shared__ float input_s[BLOCK_DIM];
    unsigned int segment = COARSE_FACTOR*2*blockDim.x*blockIdx.x;
    unsigned int i = segment + threadIdx.x;
    unsigned int t = threadIdx.x;
    float sum = input[i];
    for(unsigned int tile = 1; tile < COARSE_FACTOR*2; ++tile) {
        sum += input[i + tile*BLOCK_DIM];
    }
    input_s[t] = sum;
    for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2){
        __syncthreads();
        if (t < stride) {
            input_s[t] += input_s[t + stride];
        }
    }
    if (t == 0) {
        atomicAdd(output, input_s[0]);
      }
}
```