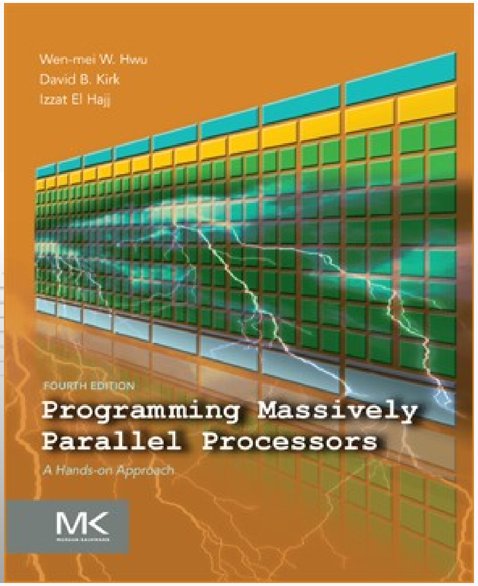
Programming Massively Parallel Processors

A Hands-on Approach

CHAPTER 11 (PART 1) › Prefix Sum (Scan)



Scan

- A **scan** operation:
  - Takes:
    - An input array $[x_0, x_1, \ldots, x_{n-1}]$
    - An associative operator $\oplus$
      - e.g., sum, product, min, max
  - Returns:
    - An output array $[y_0, y_1, \ldots, y_{n-1}]$ where
      - **Inclusive scan**: $y_i = x_0 \oplus x_1 \oplus \ldots \oplus x_i$
      - **Exclusive scan**: $y_i = x_0 \oplus x_1 \oplus \ldots \oplus x_{i-1}$

Scan Example

- Addition example:

3	6	7	4	8	2	1	9
3	9	16	20	28	30	31	40

Inclusive Scan

3	6	7	4	8	2	1	9
0	3	9	16	20	28	30	31

Exclusive Scan

- In general:

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
x_0	$x_0..x_1$	$x_0..x_2$	$x_0..x_3$	$x_0..x_4$	$x_0..x_5$	$x_0..x_6$	$x_0..x_7$

Inclusive Scan

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
ID	x_0	$x_0..x_1$	$x_0..x_2$	$x_0..x_3$	$x_0..x_4$	$x_0..x_5$	$x_0..x_6$

Exclusive Scan

Sequential Scan

- Sequential scan for sum:

```
output[0] = input[0];
for(i = 1; i < N; ++i) {
    output[i] = output[i-1] + input[i];
}
```

Inclusive Scan

```
output[0] = 0.0f;
for(i = 1; i < N; ++i) {
    output[i] = output[i-1] + input[i-1];
}
```

Exclusive Scan

- In general:

```
output[0] = input[0];
for(i = 1; i < N; ++i) {
    output[i] = f(output[i-1], input[i]);
}
```

Inclusive Scan

```
output[0] = IDENTITY;
for(i = 1; i < N; ++i) {
    output[i] = f(output[i-1], input[i-1]);
}
```

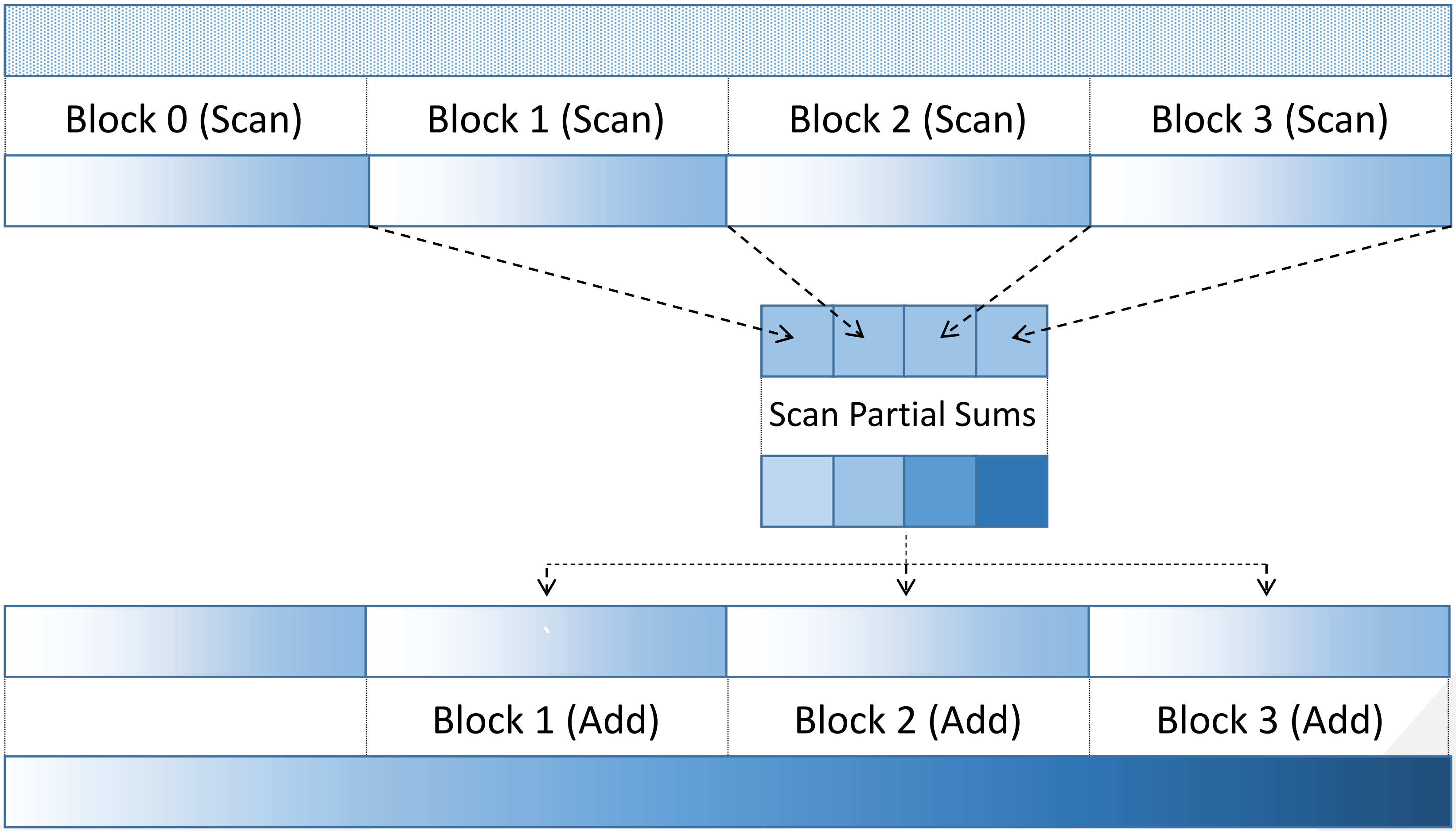
Exclusive Scan

Segmented Scan

- Similar to reduction, threads must synchronize to perform scan
  - Cannot synchronize across threads in different blocks

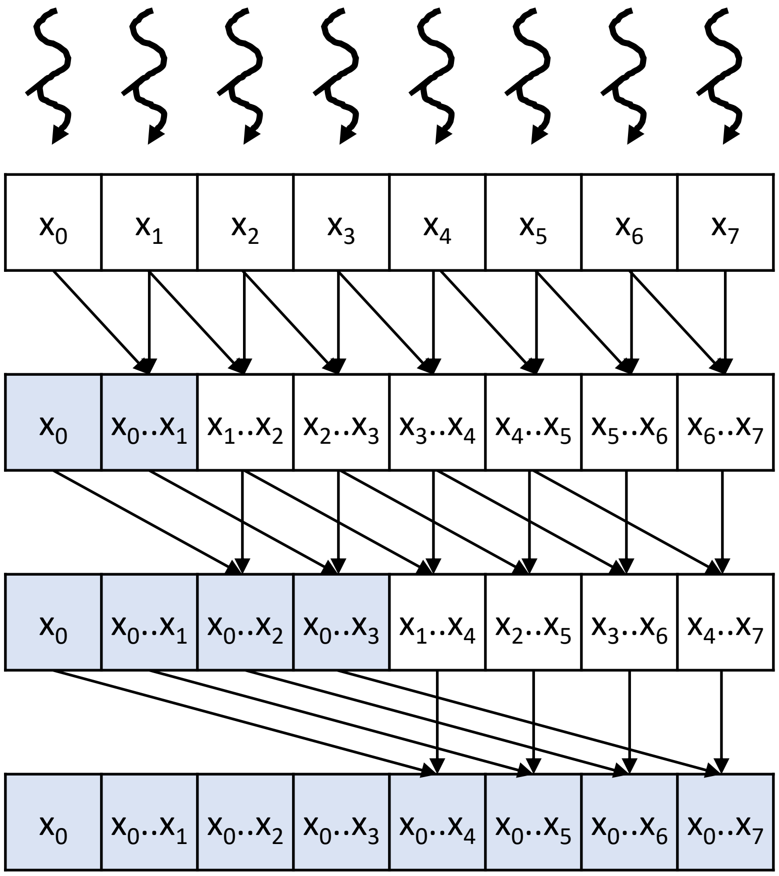
- Solution: **segmented scan** (or **hierarchical scan**)
  - Every thread block scans a segment
  - Scan the partial sums
  - Update segments based on partial sums

Segmented Scan Example



Still need to implement a parallel in each block

Kogge-Stone Parallel (Inclusive) Scan



Intuition: equivalent to having a reduction tree for each element, with overlap between trees

Kogge-Stone Parallel (Inclusive) Scan Code

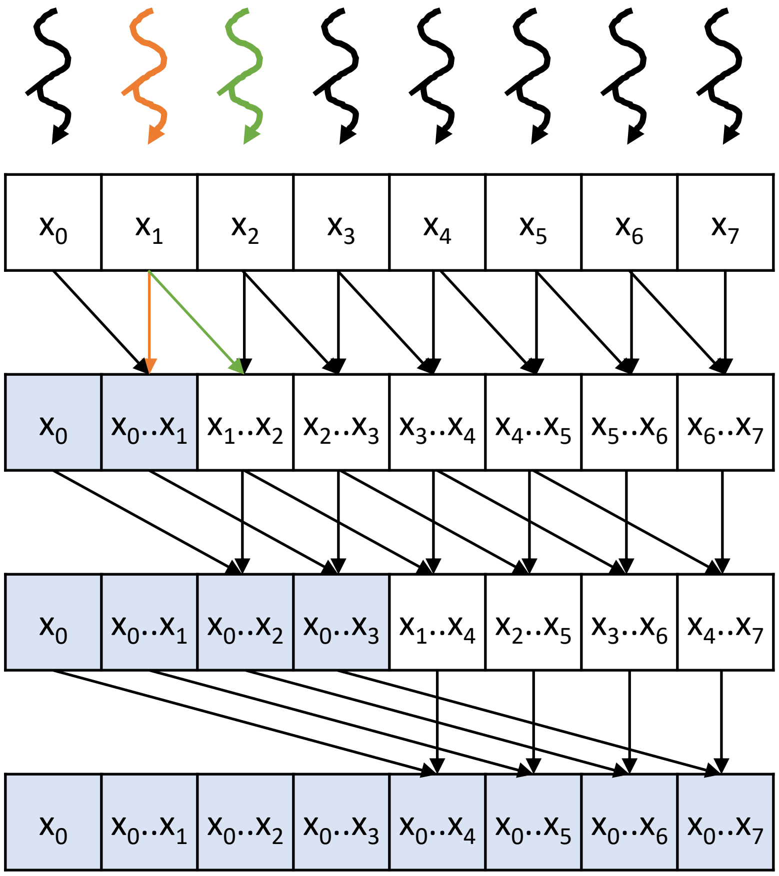
```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

output[i] = input[i];

__syncthreads();

for(unsigned int stride = 1; stride <= BLOCK_DIM/2; stride *= 2) {
    if(threadIdx.x >= stride) {
        output[i] += output[i - stride];
    }
    __syncthreads();
}

if(threadIdx.x == BLOCK_DIM - 1) {
    partialSums[blockIdx.x] = output[i];
}
```

Incorrect!
Different threads are reading and writing the same data location without synchronizing

Kogge-Stone Parallel (Inclusive) Scan

Thread 1 may **update value at index 1** before **thread 2 reads it**

Solution: wait for everyone to read before updating



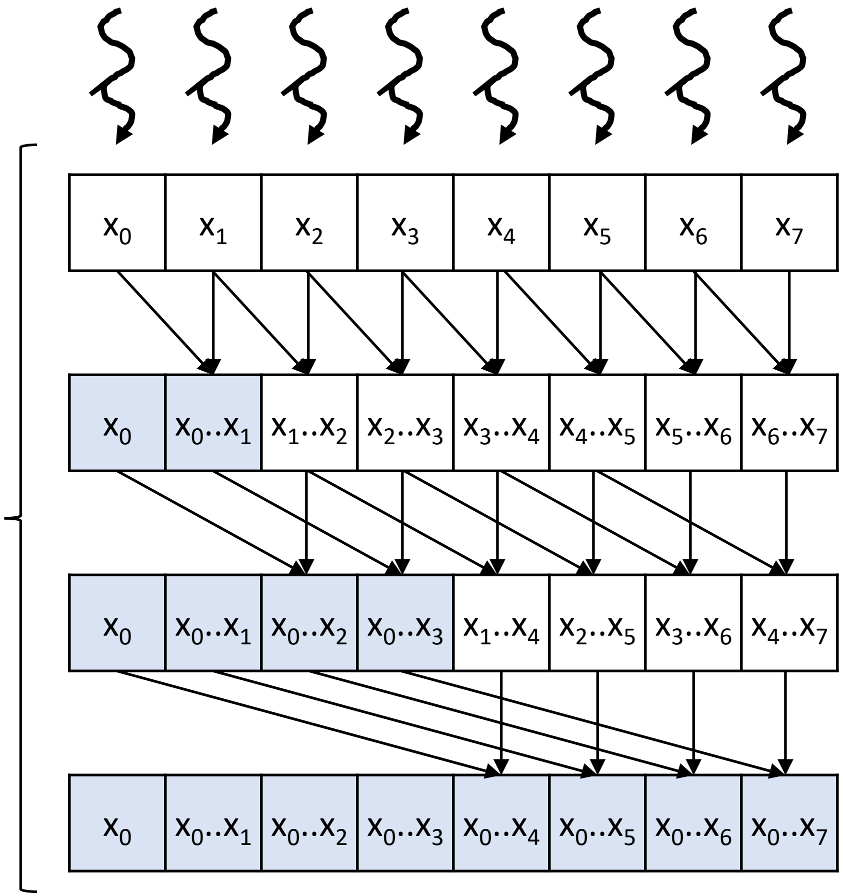
Kogge-Stone Parallel (Inclusive) Scan Code

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

output[i] = input[i];

__syncthreads();

for(unsigned int stride = 1; stride <= BLOCK_DIM/2; stride *= 2) {
    float v;
    if(threadIdx.x >= stride) {
        v = output[i - stride];
    }
    __syncthreads();
    if(threadIdx.x >= stride) {
        output[i] += v;
    }
    __syncthreads();
}

if(threadIdx.x == BLOCK_DIM - 1) {
    partialSums[blockIdx.x] = output[i];
}
```

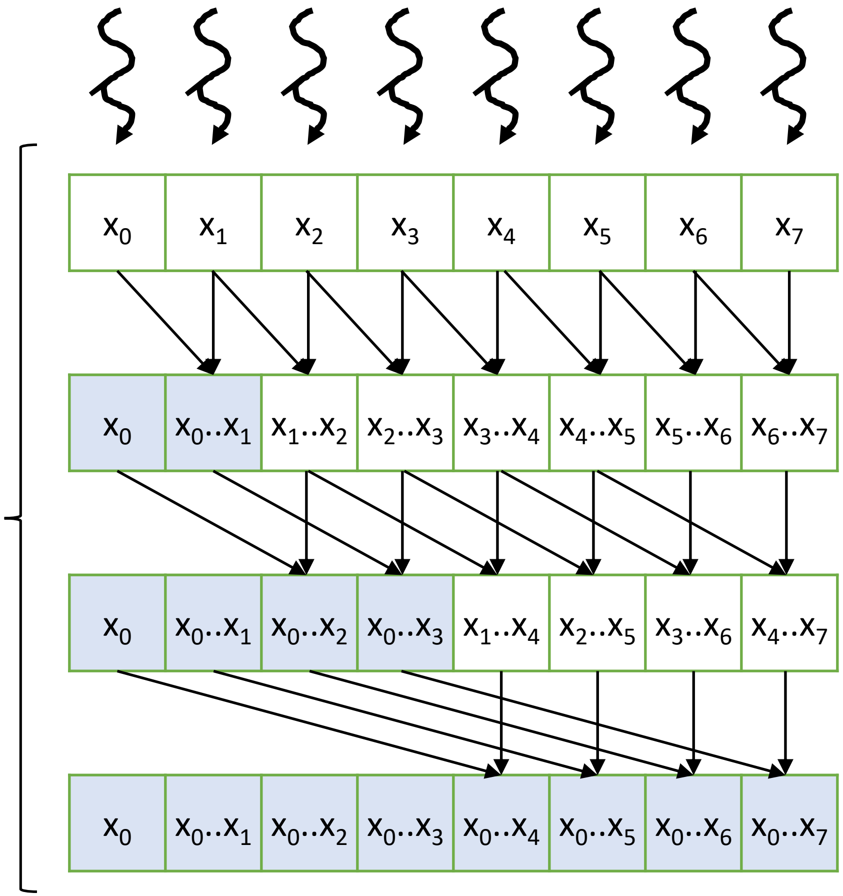
Wait for everyone to read before updating

Kogge-Stone Parallel (Inclusive) Scan

Observation: memory locations are reused



Using Shared Memory

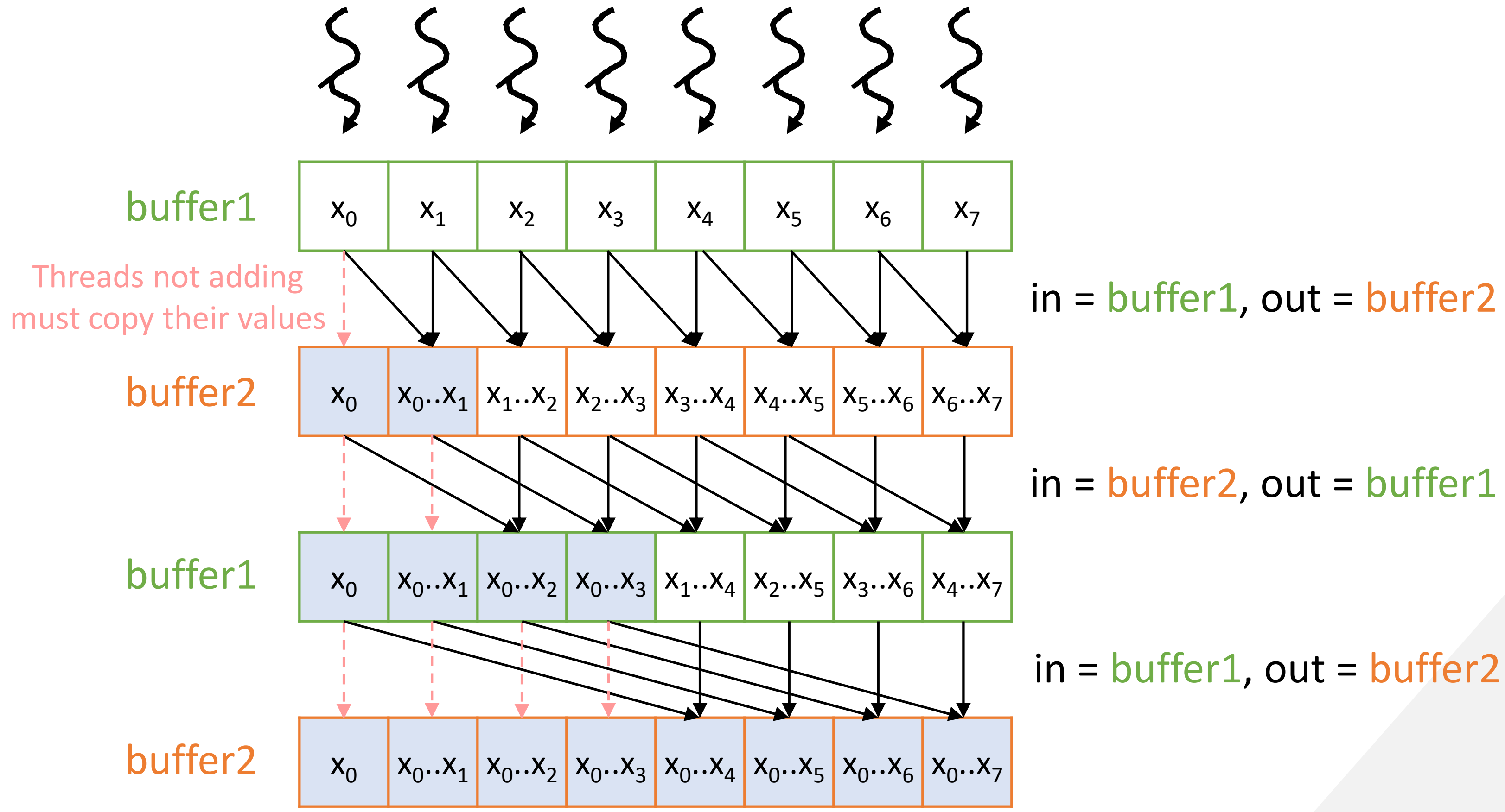


Optimization: load once to a **shared memory** buffer and perform successive reads and writes to the same array can be done in shared memory

Using Shared Memory Code

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

__shared__ float buffer_s[BLOCK_DIM];
buffer_s[threadIdx.x] = input[i];
__syncthreads();

for(unsigned int stride = 1; stride <= BLOCK_DIM/2; stride *= 2) {
    float v;
    if(threadIdx.x >= stride) {
        v = buffer_s[threadIdx.x - stride];
    }
    __syncthreads();
    if(threadIdx.x >= stride) {
        buffer_s[threadIdx.x] += v;
    }
    __syncthreads();
}

if(threadIdx.x == BLOCK_DIM - 1) {
    partialSums[blockIdx.x] = buffer_s[threadIdx.x];
}

output[i] = buffer_s[threadIdx.x];
```

Double Buffering



Optimization: eliminate one synchronization by using two buffers and alternating them as the input/output buffer (called **double buffering**)

Double Buffering

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

__shared__ float buffer1_s[BLOCK_DIM];
__shared__ float buffer2_s[BLOCK_DIM];
float* inBuffer_s = buffer1_s;
float* outBuffer_s = buffer2_s;
inBuffer_s[threadIdx.x] = input[i];
__syncthreads();

for(unsigned int stride = 1; stride <= BLOCK_DIM/2; stride *= 2) {
    if(threadIdx.x >= stride) {
        outBuffer_s[threadIdx.x] =
                inBuffer_s[threadIdx.x] + inBuffer_s[threadIdx.x - stride];
    } else {
        outBuffer_s[threadIdx.x] = inBuffer_s[threadIdx.x];
    }
    __syncthreads();
    float* tmp = inBuffer_s;
    inBuffer_s = outBuffer_s;
    outBuffer_s = tmp;
}

if(threadIdx.x == BLOCK_DIM - 1) {
    partialSums[blockIdx.x] = inBuffer_s[threadIdx.x];
}

output[i] = inBuffer_s[threadIdx.x];
```

Exclusive Scan

- Formulate as inclusive scan:
    - Shift elements by one when loading them from shared memory (skips the last element)
    - Fetch last element when computing partial sum

Exclusive Scan

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

__shared__ float buffer1_s[BLOCK_DIM];
__shared__ float buffer2_s[BLOCK_DIM];
float* inBuffer_s = buffer1_s;
float* outBuffer_s = buffer2_s;
if(threadIdx.x == 0) {
    inBuffer_s[threadIdx.x] = 0.0f;
} else {
    inBuffer_s[threadIdx.x] = input[i - 1];
}
__syncthreads();

for(unsigned int stride = 1; stride <= BLOCK_DIM/2; stride *= 2) {
    if(threadIdx.x >= stride) {
        outBuffer_s[threadIdx.x] =
                inBuffer_s[threadIdx.x] + inBuffer_s[threadIdx.x - stride];
    } else {
        outBuffer_s[threadIdx.x] = inBuffer_s[threadIdx.x];
    }
    __syncthreads();
    float* tmp = inBuffer_s;
    inBuffer_s = outBuffer_s;
    outBuffer_s = tmp;
}

if(threadIdx.x == BLOCK_DIM - 1) {
    partialSums[blockIdx.x] = inBuffer_s[threadIdx.x] + input[i];
}

output[i] = inBuffer_s[threadIdx.x];
```

Work Efficiency

- A parallel algorithm is **work-efficient** if it performs the same amount of work as the corresponding sequential algorithm
- Scan work efficiency
  - Sequential scan performs *N* additions
  - Kogge-Stone parallel scan performs:
    - log(N) steps, N - 2^{step} operations per step
    - Total: (N-1) + (N-2) + (N-4) + … + (N-N/2)
      = N*log(N) - (N-1) = O(N*log(N)) operations
    - Algorithm is not work efficient
- If resources are limited, parallel algorithm will be slow because of low work efficiency

References

- Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.