CENG443
Heterogeneous Parallel Programming

Scan

Inclusive Scan

The scan operation takes a binary associative operator ⊕ (pronounced as circle plus), and an array of n elements,

$$[x_0, x_1, \ldots, x_{n-1}],$$

and returns the array

$$[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})]$$

If ⊕ is addition, then scan operation on the array would return

[3 1 7 0 4 1 6 3], [3 4 11 11 15 16 22 25]

Scan Application Example

Assume that we have a 100-inch sandwich to feed 10 people

We know how much each person wants in inches

[3 5   2   7   28 4 3 0 8 1]

How do we cut the sandwich quickly?

How much will be left?

Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.

Method 2: calculate prefix sum (scan):

[3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)

Parallel Scan as a Primitive Operation

Quicksort

String comparison

Polynomial evaluation

Solving recurrences

Histograms

Sequential Addition Scan

Given a sequence $[x_0, x_1, x_2, \ldots]$

Calculate output $[y_0, y_1, y_2, \ldots]$

Such that

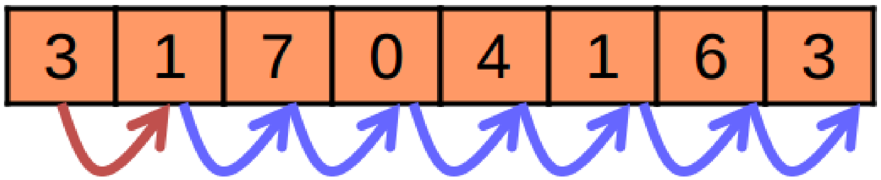
$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2 \ldots$$

Using a recursive definition

$$y_i = y_{i-1} + x_i$$



C Implementation

```
void scan( float* y, float* x, int length)
{
    y[0] = x[0];
    for(int i = 1; i < length; i++)
        y[i] = y[i-1] + x[i];
}
```

n -1 additions needed for n elements - O(n)

Naive Parallel Scan

Assign one thread to calculate each y element

Have every thread to add up all x elements needed for the y element

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2 \ldots$$

A Better Parallel Scan

Read input from device memory to shared memory

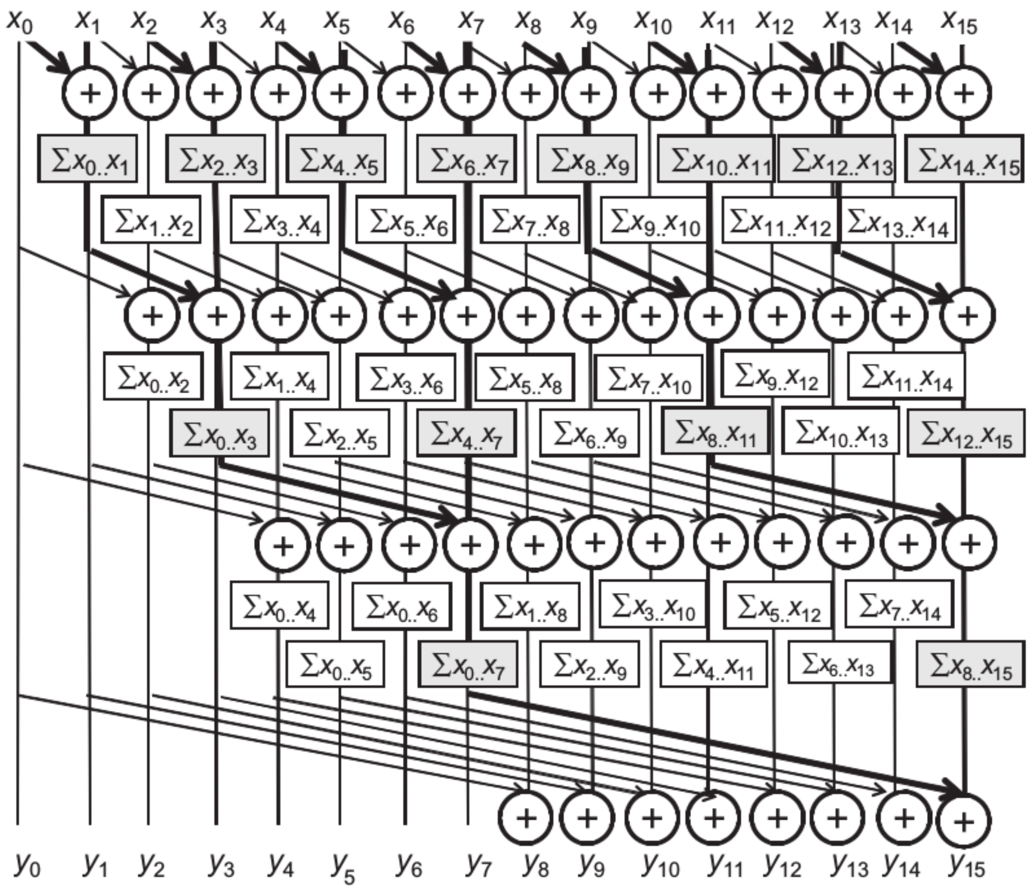
- Each thread reads one value from the input array in device memory into shared memory array

Iterate log(n) times

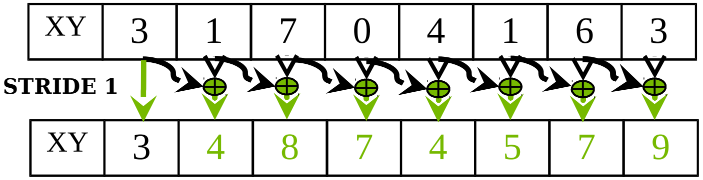
- Threads stride 1 to n
- Add pairs of elements stride elements apart
- Double stride at each iteration

Write output from shared memory to device memory

A Better Parallel Scan

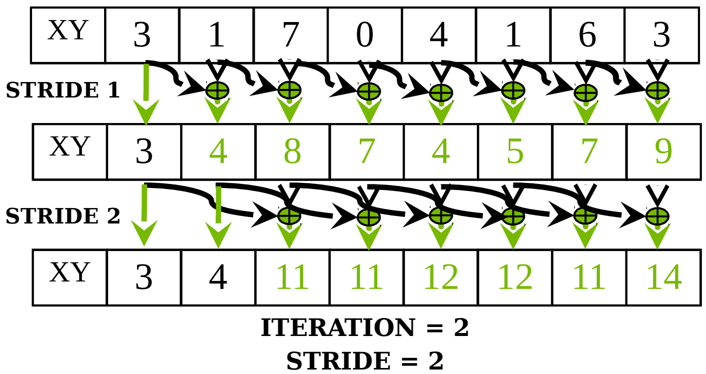


A Better Parallel Scan

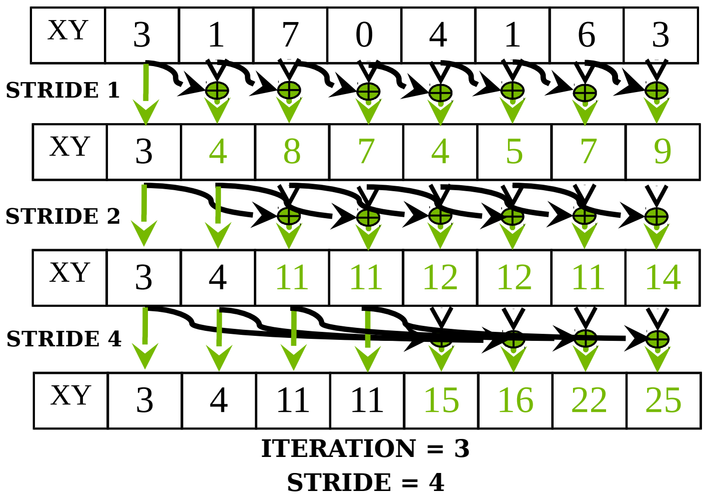


ITERATION = 1
STRIDE = 1

A Better Parallel Scan



A Better Parallel Scan



Kogge-Stone Scan Kernel

```
__global__ void kogge_stone_scan_kernel(float *X, float *Y, int InputSize) {
    __shared__ float XY[SECTION_SIZE];//assume it is equal to block size
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < InputSize)
        XY[threadIdx.x] = X[i];
    // the code below performs iterative scan on XY
    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
         __syncthreads();
         if (threadIdx.x >= stride)
             XY[threadIdx.x] += XY[threadIdx.x-stride];
    }
    if (i < InputSize)
        Y[i] = XY[threadIdx.x];
}
```

Work Efficiency Analysis

Sequential scan: n add operations

- 1024 add operations for 1024 elements → 1024 time units

Kogge-stone: log(n) parallel iterations; (n-1), (n-2), (n-4), ..., (n-n/2) add operations per each iteration

- O(n*log(n)) work
- 1024 threads, 32 execution units for 1024 elements → (1024*10)/32 = 320 time units, speedup=3.2
- 4 execution units → (1024*10)/4= 2560 time units, slower

Can be slower than sequential if resources are low

- at least 8 times more execution units than the sequential machine just to break even

Work-inefficient!

Improving Efficiency

Sharing intermediate results to streamline the operations performed

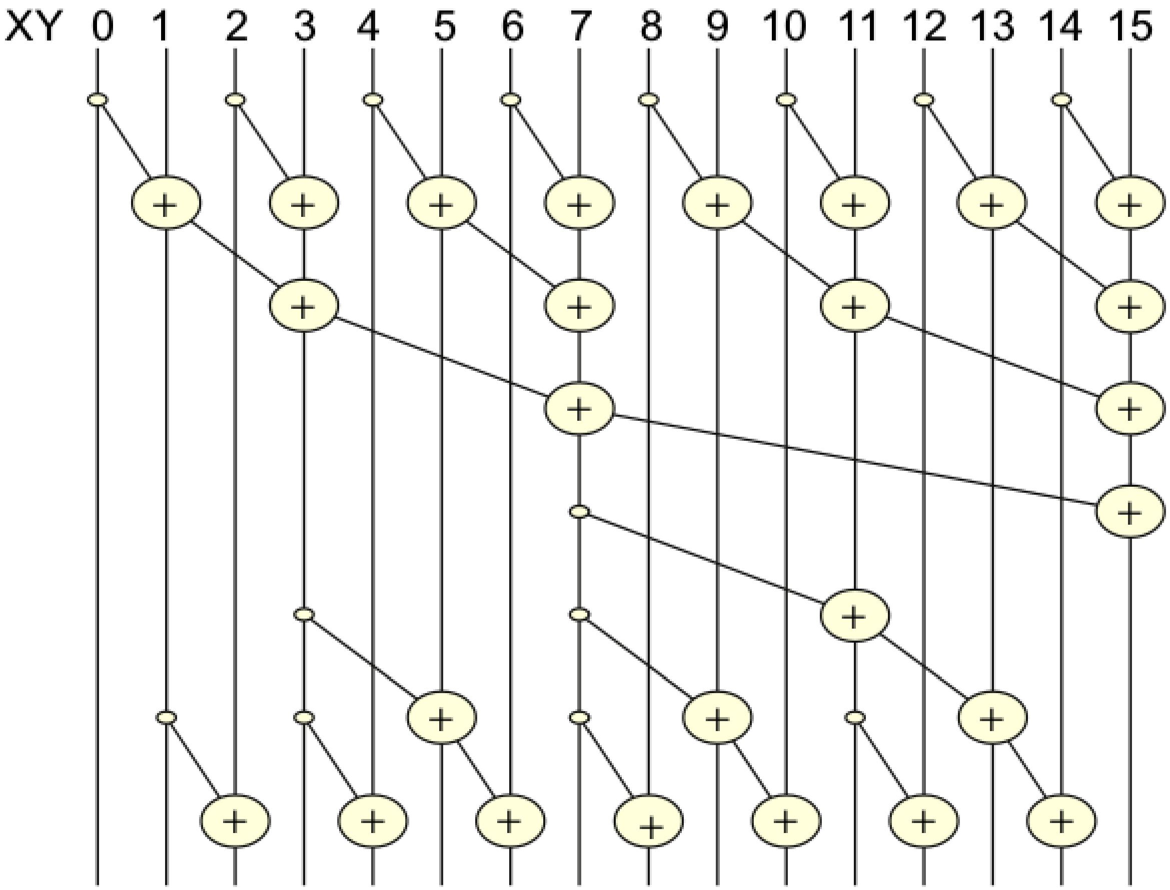
Strategically calculate the intermediate results to be shared and then readily distribute them to different threads in order to allow more sharing across multiple threads

Reduction tree

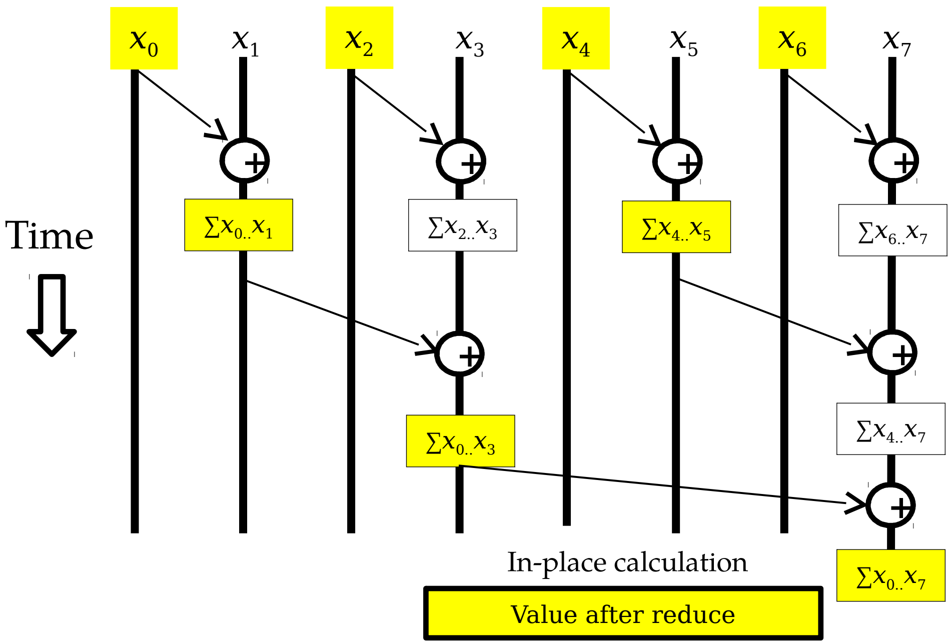
Traverse down from leaves to the root building partial sums at internal nodes in the tree

Traverse back up the tree building the output from the partial sums

Brent–Kung Scan



Parallel Scan – Reduction Phase

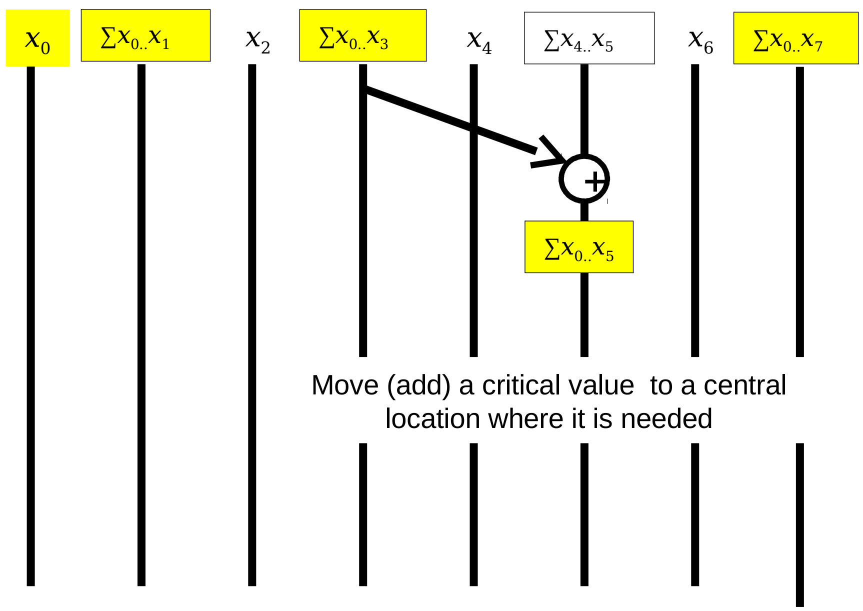


Reduction Phase Kernel Code

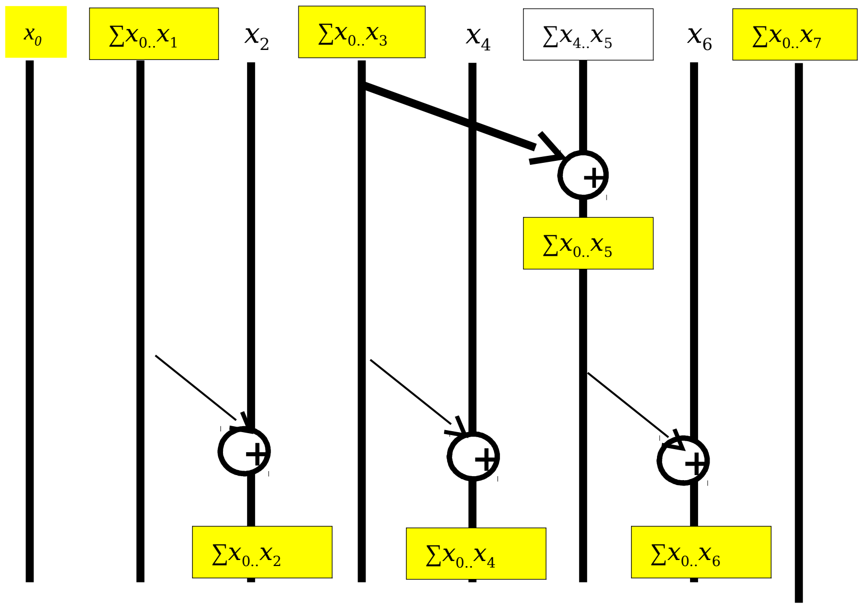
```
// XY[2*BLOCK_SIZE] is in shared memory

for (unsigned int stride = 1;stride <= BLOCK_SIZE; stride *= 2) {
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < 2*BLOCK_SIZE)
        XY[index] += XY[index-stride];
    __syncthreads();
}
```

Parallel Scan – Post Reduction Reverse Phase



Parallel Scan – Post Reduction Reverse Phase



Post Reduction Phase Kernel Code

```
for (unsigned int stride = BLOCK_SIZE/2; stride > 0; stride /= 2) {
        __syncthreads();
        int index = (threadIdx.x+1)*stride*2 - 1;
        if(index+stride < 2*BLOCK_SIZE) {
          XY[index + stride] += XY[index];
        }
  }
    __syncthreads();
  if (i < InputSize) Y[i] = XY[threadIdx.x];
```

Work Efficiency Analysis

log(n) parallel iterations in the reduction step

- n/2, n/4, ..., 1 add operations
- Total add operations: n-1 O(n) work

log(n)-1 parallel iterations in the post-reduction reverse step

- 2-1, 4-1, n/2-1 add operations
- Total add operations: (n-2) – (log(n)-1) O(n) work