Lecture No.6: Prefix Sum (Scan)

Muhammad Osama Mahmoud, TA

Computer and Systems Eng. Dept. - Faculty of Engineering – University of Minia

Prefix sum

- Prefix sum is essential for many parallel algorithms
  - Radix sort
  - Quicksort
  - Histograms
  - String comparison
  - Tree operation
  - Polynomial evaluation
- Takes a binary associative operator $\oplus$, and an array of N elements

$$[x_0, x_1, x_2, \dots, x_{N-1}]$$

Prefix sum

- Returns the array

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

- For example, assume the associative operator is addition the prefix sum of the input array $[1\ \ 5\ \ -6\ \ 3\ \ 5\ \ 4\ \ -2\ \ 1]$

  will be $[1\ \ 6\ \ 0\ \ 3\ \ 8\ \ 9\ \ 7\ \ 8]$

Inclusive Sequential Addition Scan

Given the input array

$$[x_0 , x_1 , x_2 , \dots , x_{N-1}]$$

Calculate the output

$$[y_0 , y_1 , y_2 , \dots , y_{N-1}]$$

Such that

$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$y_2 = x_0 + x_1 + x_2$$

Using the recursion technique

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

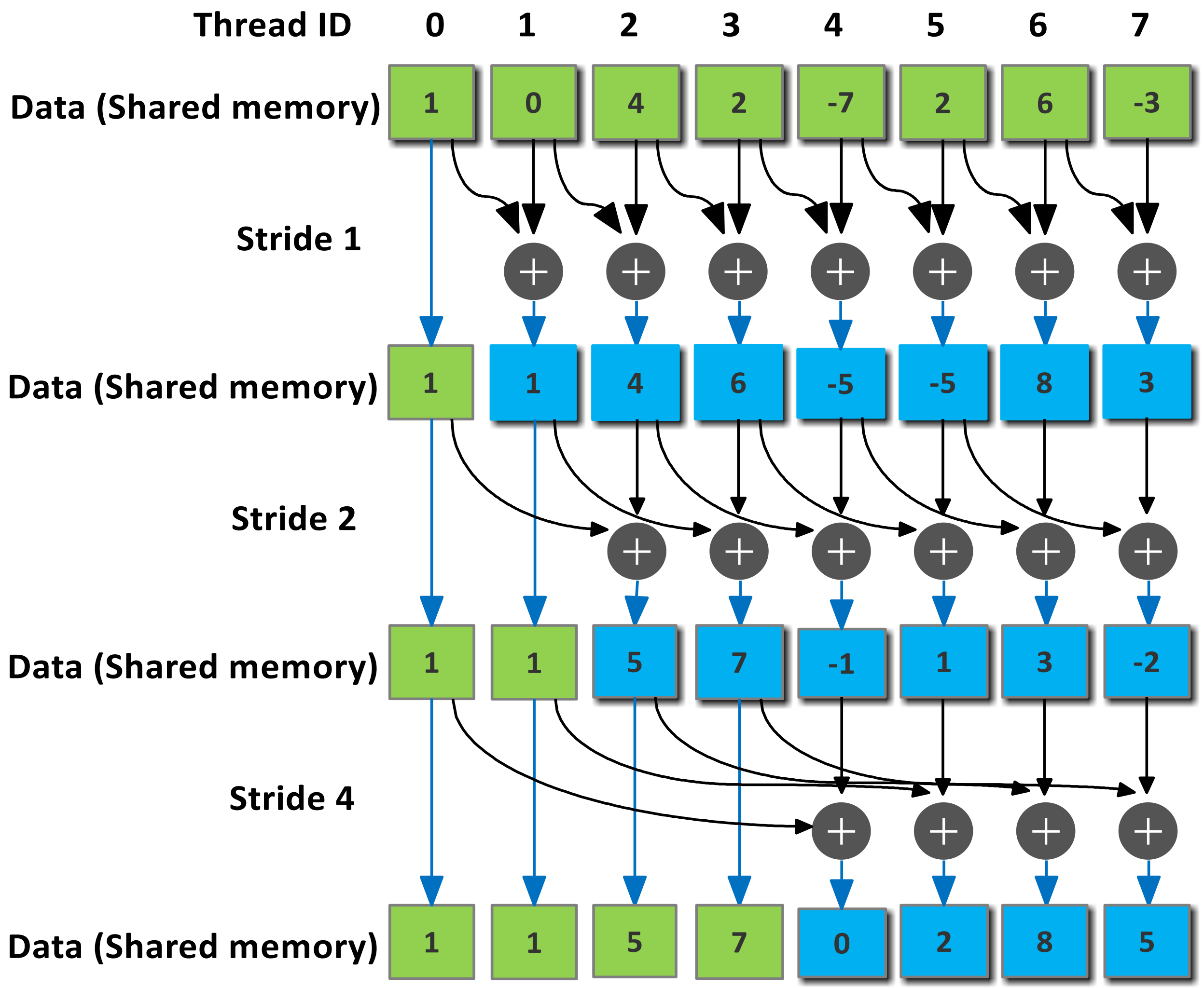
Sequential Implementation

```
void scan( float* output, float* input, int length)  {
        output[0] = 0;
        for (int j = 1; j < length; ++j)  {
                output[j] = input[j-1] + output[j-1];
        }
}
```

- This code performs N operations on N elements, that is a time complexity of $O(N)$
- We would like the parallel version to be work efficient (no more operations than the sequential version)

A Naïve Inclusive Parallel Scan

- Assign a thread to calculate each output y_i
- do all calculations on shared memory



A Naïve Inclusive Parallel Scan Kernel

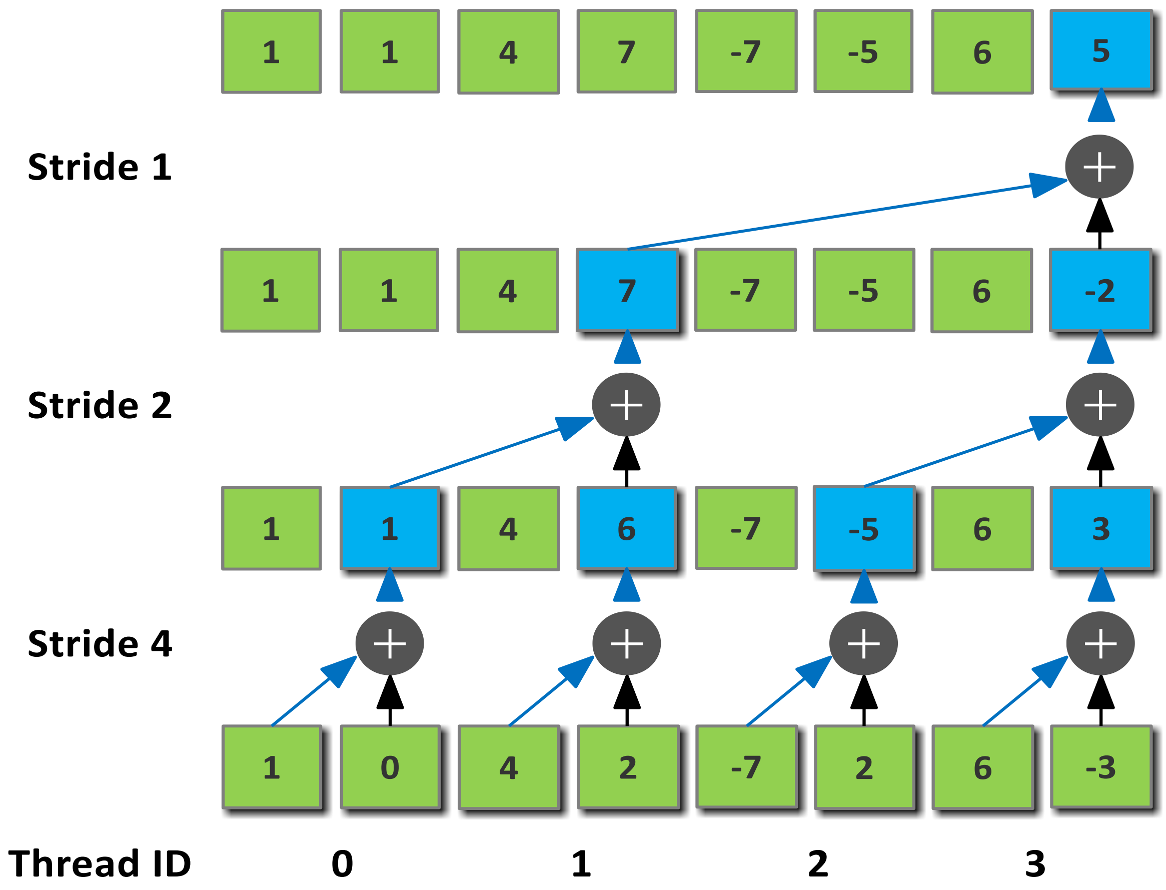
```
__global__ void naive_scan_kernel(float *X, float *Y, int InputSize)  {
        extern  __shared__  float section[ ];
        int tx = threadIdx.x;
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if  (i < InputSize)  section[tx] = X[i];
        else        section[tx] = 0.0;
        // perform iterative scan on section
        for (unsigned int stride = 1; stride <= tx; stride <<= 1)  {
                __syncthreads();
                float inp = section[tx - stride];
                __syncthreads();
                section[tx] += inp;
        }
}
```

Work-efficient Parallel Scan

- The previous kernel performs $\log(N)$ steps, each step do $N-1, N-2, N-4, \ldots, N-N/2$ add operations
- Complexity = $\log(N)\, x\, N-1$ = $O\big(N \log(N)\big)$, that is not work efficient as it do more operations than the sequential algorithm
- Solution: balanced-binary tree with exclusive scan
- Exclusive scan shifts the output values by 1 and pads a zero in the first location
- The work-efficient algorithm has two phases:
  - In-place reduction phase (traversing tree from leaves to root computing the partial sums at the internal nodes of the tree)
  - In-place up-down sweep phase (traverse back down the tree from the root, using the partial sums to build the scan in place on the array)

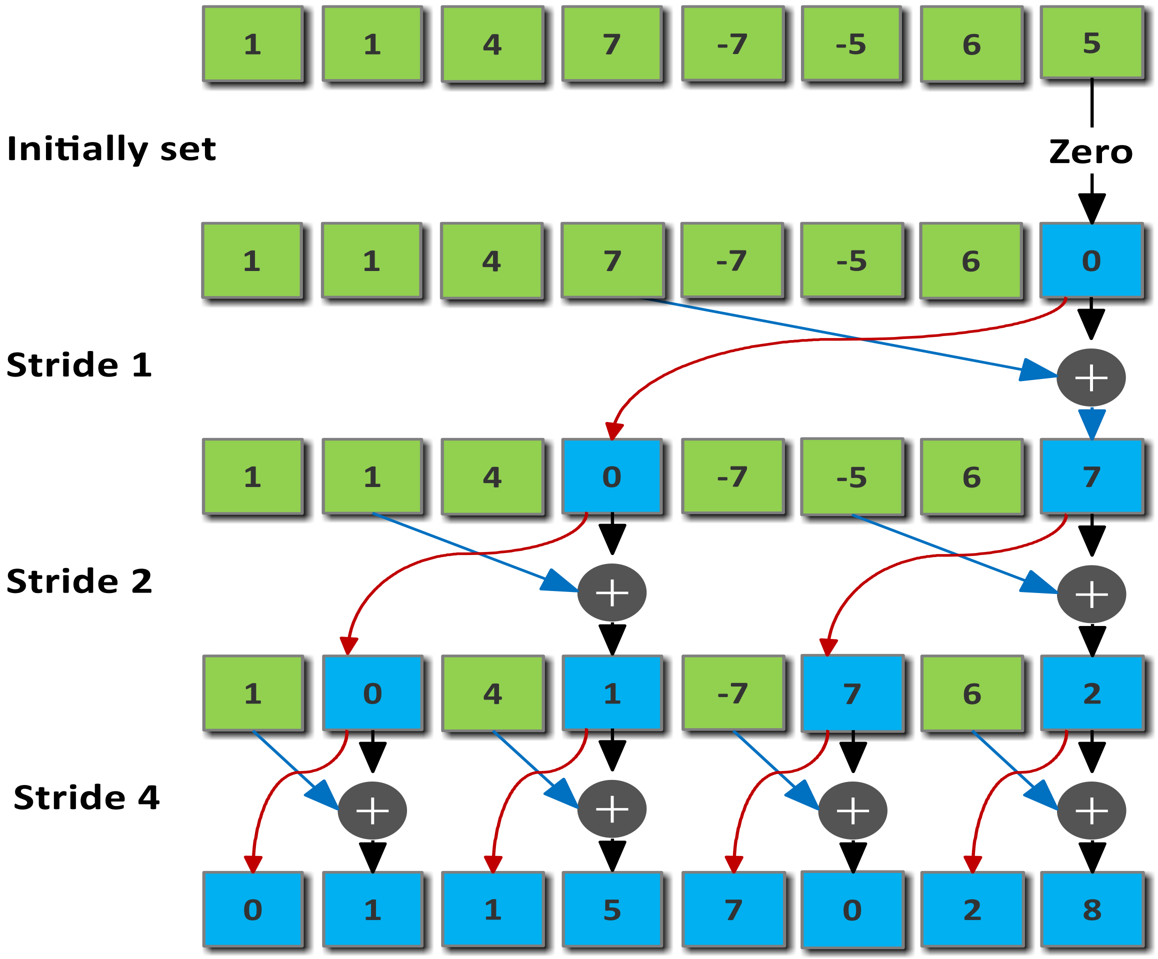
Work-efficient Parallel Scan (Cont.)

- Phase 1



Work-efficient Parallel Scan (Cont.)

- Phase 2



Work-efficient Parallel Scan Kernel

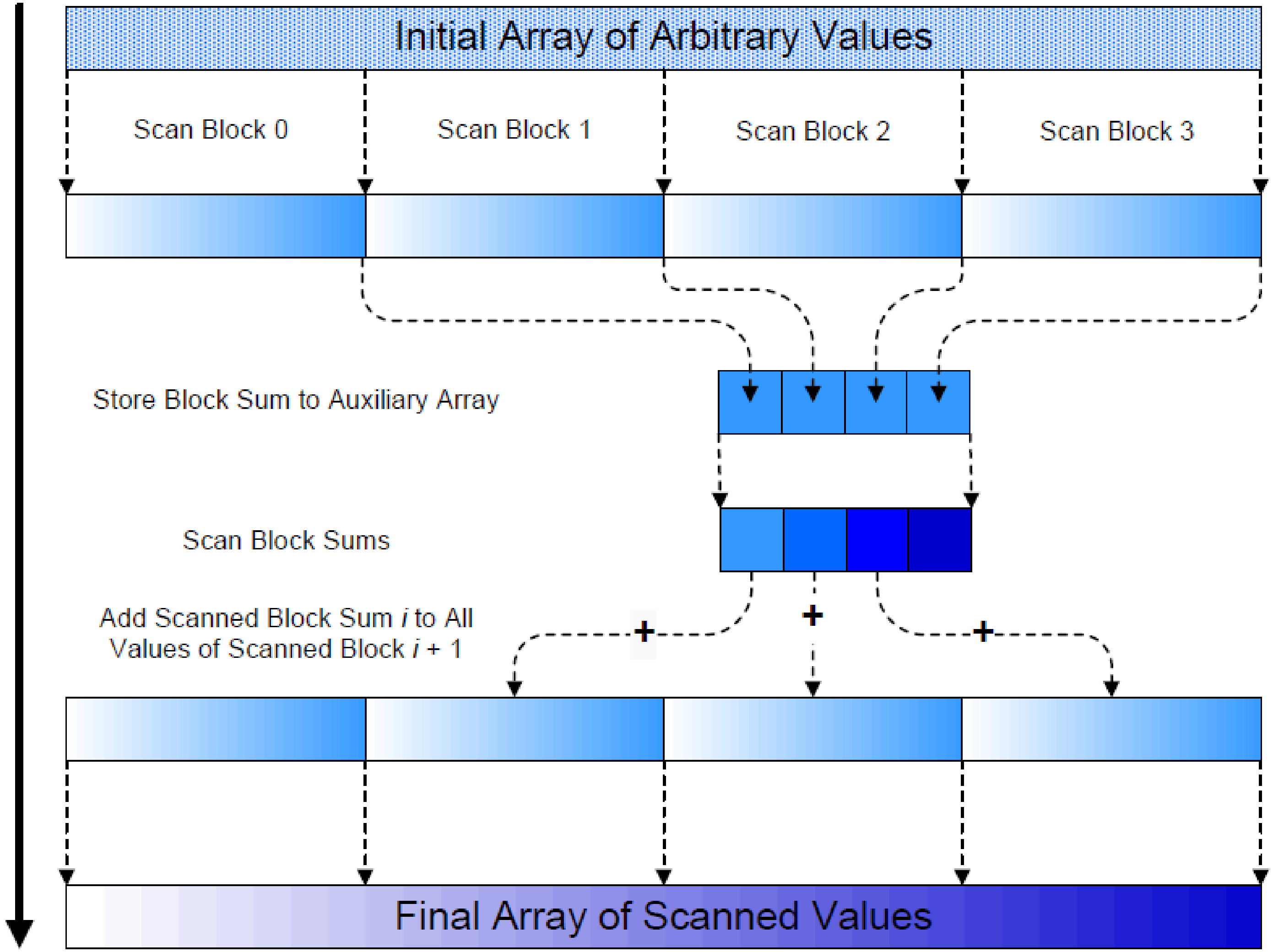
```
__global__ void scan(float *output, float *input, int n) {
        extern  __shared__  float section[ ];
        int tx = threadIdx.x;
        int offset = 1;
        section[2 * tx] = input[2 * tx]; // load input into shared memory
        section[2 * tx + 1] = input[2 * tx + 1];
        // build sum in place up the tree
        for (int s = n >> 1; s > 0; s >>= 1) {
                __syncthreads();
                if (tx < s) {
                        int ai = offset*(2*tx + 1)-1;
                        int bi = offset*(2*tx + 2)-1;
                        section[bi] += section[ai];
                }
                offset *= 2;
        }
        if (tx == 0) { section[n - 1] = 0; } // clear the last element
// rest of kernel in the next slide
```

Work-efficient Parallel Scan Kernel (Cont.)

```
        // traverse down tree & build scan
        for (int s = 1; s < n; S *= 2)  {
                offset >>= 1;
                __syncthreads();
                if (tx < s) {
                        int ai = offset*(2* tx + 1)-1;
                        int bi = offset*(2* tx + 2)-1;
                        float t = section[ai];
                        section [ai] = section [bi];
                        section [bi] += t;
                }
        }
        __syncthreads();
        output[2* tx] = section [2* tx]; // write results to device memory
        output [2* tx +1] = section [2* tx +1];
} // end of kernel
```

Parallel Scan Algorithm for Large Data Sets

- Algorithm for arbitrary array size



References

[1] Wen-mei W. Hwu, “Heterogeneous Parallel Programming”. Online course, 2014. Available: https://class.coursera.org/hetero-002

[2] M. Harris, “Parallel Prefix Sum (Scan) with CUDA”, April 2007.