Tutorial 11 – GPU algorithms design

Parallel primitives – Scan operation

Scan primitive

Definition:

Scan($\oplus,\varepsilon,[a_0,a_1,a_2,\ldots,a_{n-1}]$**) =** $[\varepsilon, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots, a_0 \oplus \ldots \oplus a_{n-2}]$

Example:

Scan(+,0,[3 1 7 0 4 1 6 3]) **=** [0 3 4 11 11 14 16 22]

Specification:

The defined Scan is an *exclusive* scan

An *inclusive* Scan returns $[a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots, a_0 \oplus \ldots \oplus a_{n-1}]$

Implementation:

Coming next - for Scan(+,0,…), **also called Prefix-Sum.**

Sequential implementation

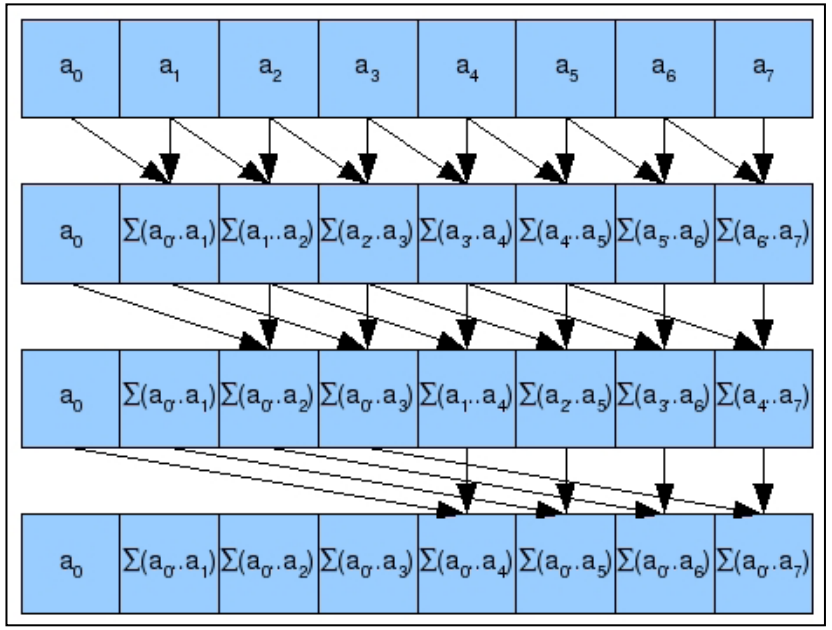
```
void scan(int* in, int* out, int n)
{
        out[0] = 0;
        for (int i = 1; i < n; i++)
                out[i] = in[i-1] + out[i-1];
}
```

Complexity: O(n)

Parallel implementation - naïve

```
for (d = 1; d < log2n; d++)
   for all k in parallel
     if( k ≥ 2^d )
        x[out][k] = x[in][k – 2^(d-1)] + x[in][k]
     else
        x[out][k] = x[in][k]
```

x is Double-Buffered.
x[in] ➔ x[out]



Complexity: $O(n \log_2 n)$

Q: Why Double-Buffer?

Parallel implementation

- **The naïve implementation is less efficient than the sequential. We want O(n) operations in total.**

Solution:

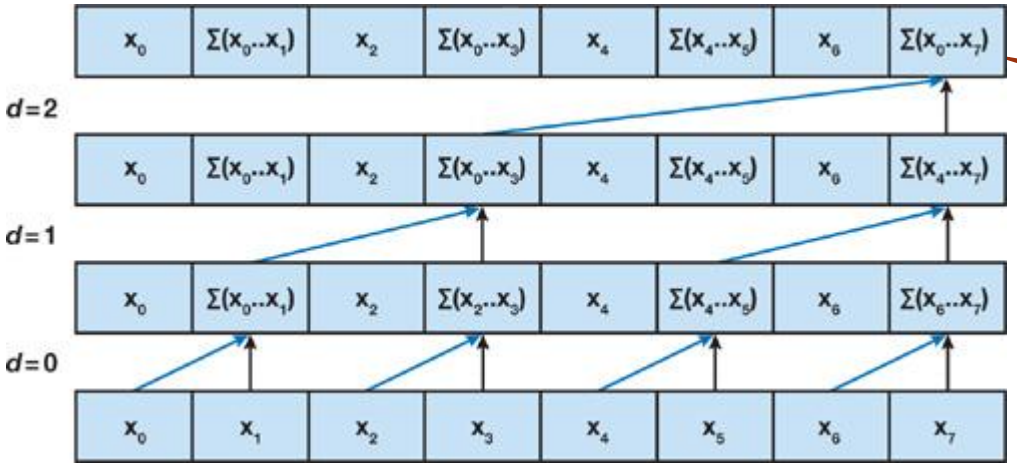
- **Balanced trees. Build a balanced binary tree on the input data and sweep it to and from the root to compute the prefix sum.**
- **Complexity: O(n).**

- **The algorithm consists of two phases:**
  - ***Reduce (also known as the up-sweep)***
  - ***Down-sweep***

Up-Sweep

Traverse the tree from leaves to root computing partial sums at internal nodes of the tree.

```
for d := 0 to log2n - 1 do
    for k from 0 to n – 1 by 2^(d+1) in parallel do
        x[k + 2^(d+1) - 1] := x[k + 2^d - 1] + x[k + 2^(d+1) - 1]
```

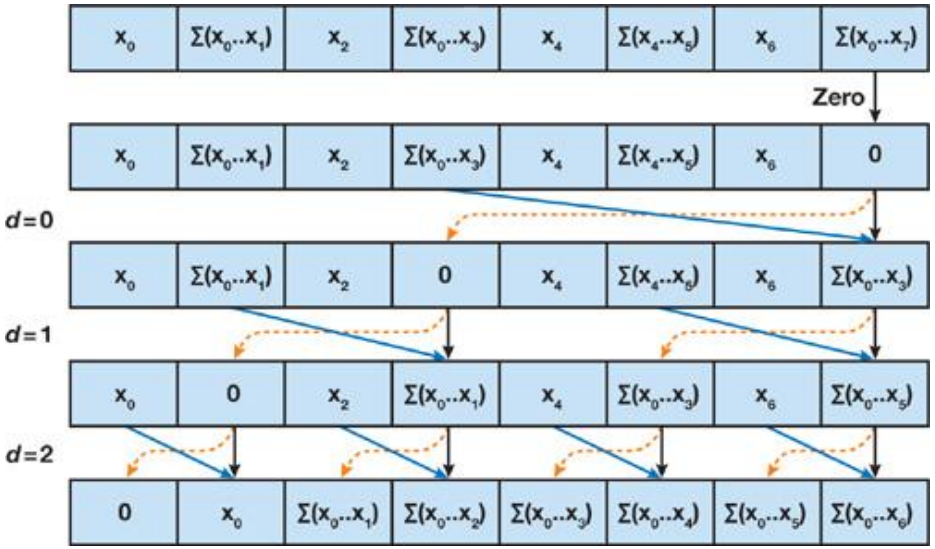


Sum of all elements. What collective operation that we know returns this value?

Down-Sweep

Traverse back up the tree from the root, using the partial sums.

```
x[n - 1] := 0
for d := log2n down to 0 do
    for k from 0 to n – 1 by 2^(d+1) in parallel do
        t := x[k + 2^d - 1]
        x[k + 2^d - 1] := x[k + 2^(d+1) - 1]
        x[k + 2^(d+1) - 1] := t + x[k + 2^(d+1) - 1]
```



Replace last element with 0.
Do we need that value?

CUDA GPU Execution model

- SIMD Execution of threads in a *warp* (currently 32 threads).
- A group of warps executing a common task is a *thread-block*. The size of a thread-block is limited (currently to 1024 threads).
- Threads in the same thread-block are:
  - Executed on the same Multi-Processor (GPU core)
  - Enumerated – locally (in thread-block) and globally
  - Share data and synchronize
- Threads in different thread-blocks cannot cooperate
- Number of concurrent thread-blocks is (practically) unlimited.

CUDA Implementation (1/2)

```
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
   extern __shared__ float temp[];  // points to shared memory
   int thid = threadIdx.x;
   int offset = 1;
   temp[2*thid] = g_idata[2*thid]; // load input into shared memory
   temp[2*thid+1] = g_idata[2*thid+1];
   for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
   {
     __syncthreads();
     if (thid < d)
     {
        int ai = offset*(2*thid+1)-1;
        int bi = offset*(2*thid+2)-1;
        temp[bi] += temp[ai];
     }
     offset *= 2;
   }
   // continues on next slide
```

for $d := 0$ **to** $\log_2 n - 1$ **do**
  for k **from** 0 **to** $n-1$ **by** 2^{d+1} **in parallel do**
    $x[k + 2^{d+1} - 1] := x[k + 2^d - 1] + x[k + 2^{d+1} - 1]$

CUDA Implementation (2/2)

```
    if (thid == 0) { temp[n - 1] = 0; } // clear the last element
    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
      offset >>= 1;
      __syncthreads();
      if (thid < d)
      {
          int ai = offset*(2*thid+1)-1;
          int bi = offset*(2*thid+2)-1;
          float t = temp[ai];
          temp[ai] = temp[bi];
          temp[bi] += t;
       }
    }
    __syncthreads();
    g_odata[2*thid] = temp[2*thid]; // write results to device memory
    g_odata[2*thid+1] = temp[2*thid+1];
}
```

$x[n-1] := 0$
for $d := \log_2 n$ **down to** 0 **do**
  for k **from** 0 **to** $n-1$ **by** 2^{d+1} **in parallel do**
    $t := x[k + 2^d - 1]$
    $x[k + 2^d - 1] := x[k + 2^{d+1} - 1]$
    $x[k + 2^{d+1} - 1] := t + x[k + 2^{d+1} - 1]$

Analysis

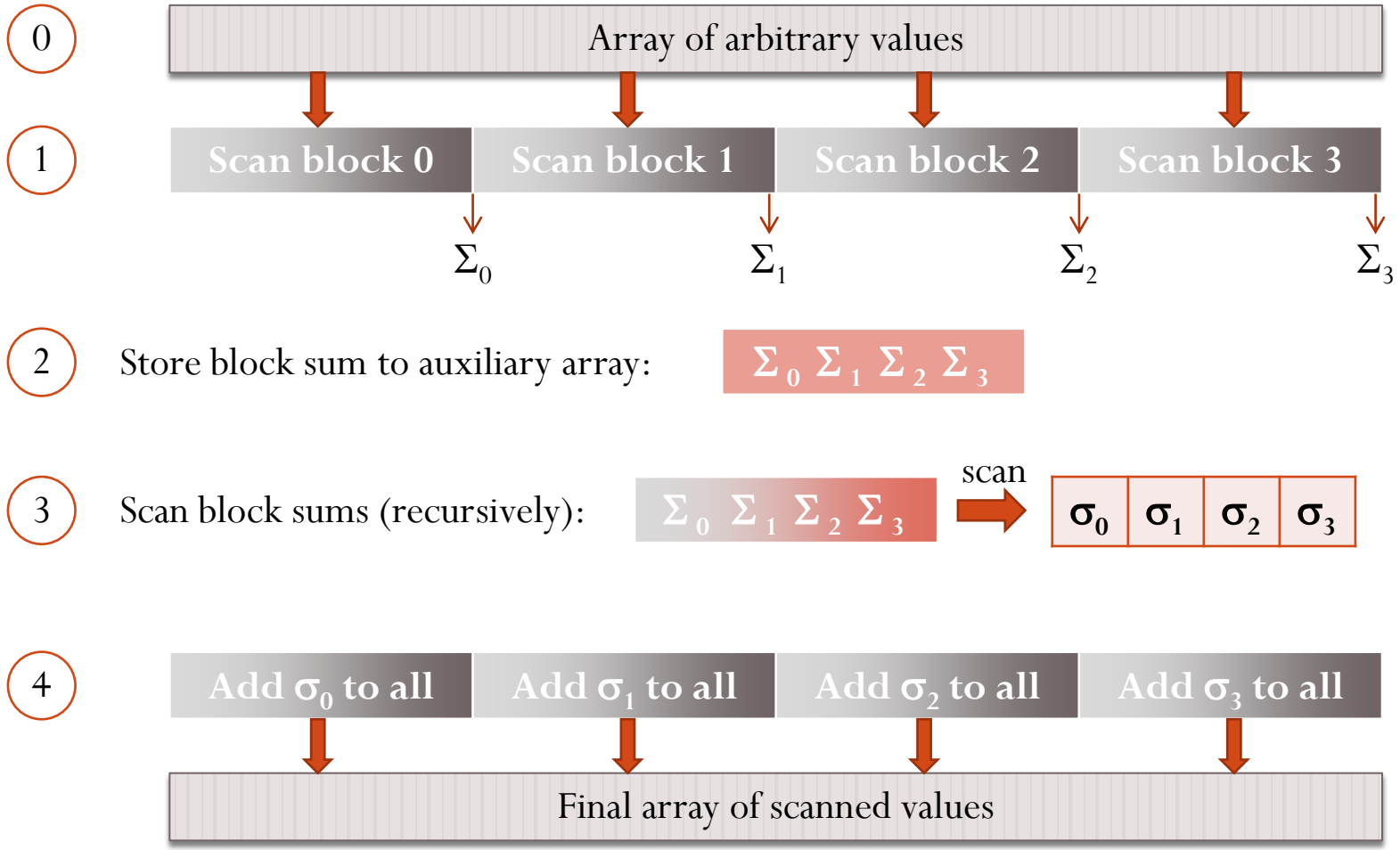
```
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
   extern __shared__ float temp[];  // points to shared memory
   int thid = threadIdx.x;
   int offset = 1;
   temp[2*thid] = g_idata[2*thid]; // load input into shared memory
   ...
```

Restriction: Limited to a single thread block

- **threadIdx.x is an identifier inside the thread block**
- **Shared memory is defined in the scope of a thread block only**

We need to find a solution that supports arrays of arbitrary size.

Scan - arbitrary array size



Scan - arbitrary array size

1. Divide the large array into blocks that each can be scanned by a single thread block
2. Scan the blocks, and write the total sum of each block to another array of block sums
3. Scan the block sums, generating an array of block increments that are added to all elements in their respective blocks

Performance

# elements	CPU Scan (ms)	GPU Scan (ms)	Speedup
1024	0.002231	0.079492	0.03
32768	0.072663	0.106159	0.68
65536	0.146326	0.137006	1.07
131072	0.726429	0.200257	3.63
262144	1.454742	0.326900	4.45
524288	2.911067	0.624104	4.66
1048576	5.900097	1.118091	5.28
2097152	11.848376	2.099666	5.64
4194304	23.835931	4.062923	5.87
8388688	47.390906	7.987311	5.93
16777216	94.794598	15.854781	5.98

Table 2: Performance of the work-efficient, bank conflict free Scan implemented in CUDA compared to a sequential scan implemented in C++. The CUDA scan was executed on an NVIDIA GeForce 8800 GTX GPU, the sequential scan on a single core of an Intel Core Duo Extreme 2.93 GHz.

Table source: **Parallel Prefix Sum (Scan) with CUDA (2007), Mark Harris.**

References

- **Parallel Prefix Sum (Scan) with CUDA (2007), Mark Harris.**
- **Parallel Prefix Sum (Scan) with CUDA, GPU Gems 3 Chapter 39.**