Prefix Computations

Sathish Vadhiyar

Parallel Algorithm: Prefix computations on arrays

- Array X partitioned into subarrays
- Local prefix sums of each subarray calculated in parallel
- Prefix sums of last elements of each subarray written to a separate array Y
- Prefix sums of elements in Y are calculated.
- Each prefix sum of Y is added to corresponding block of X
- Divide and conquer strategy

Example

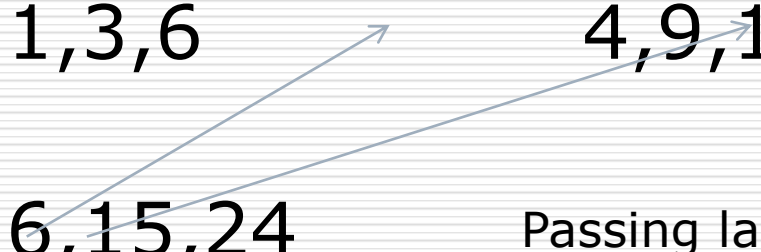
123456789

Divide

123 456 789

Local prefix sum

1,3,6 4,9,15 7,15,24



6,15,24

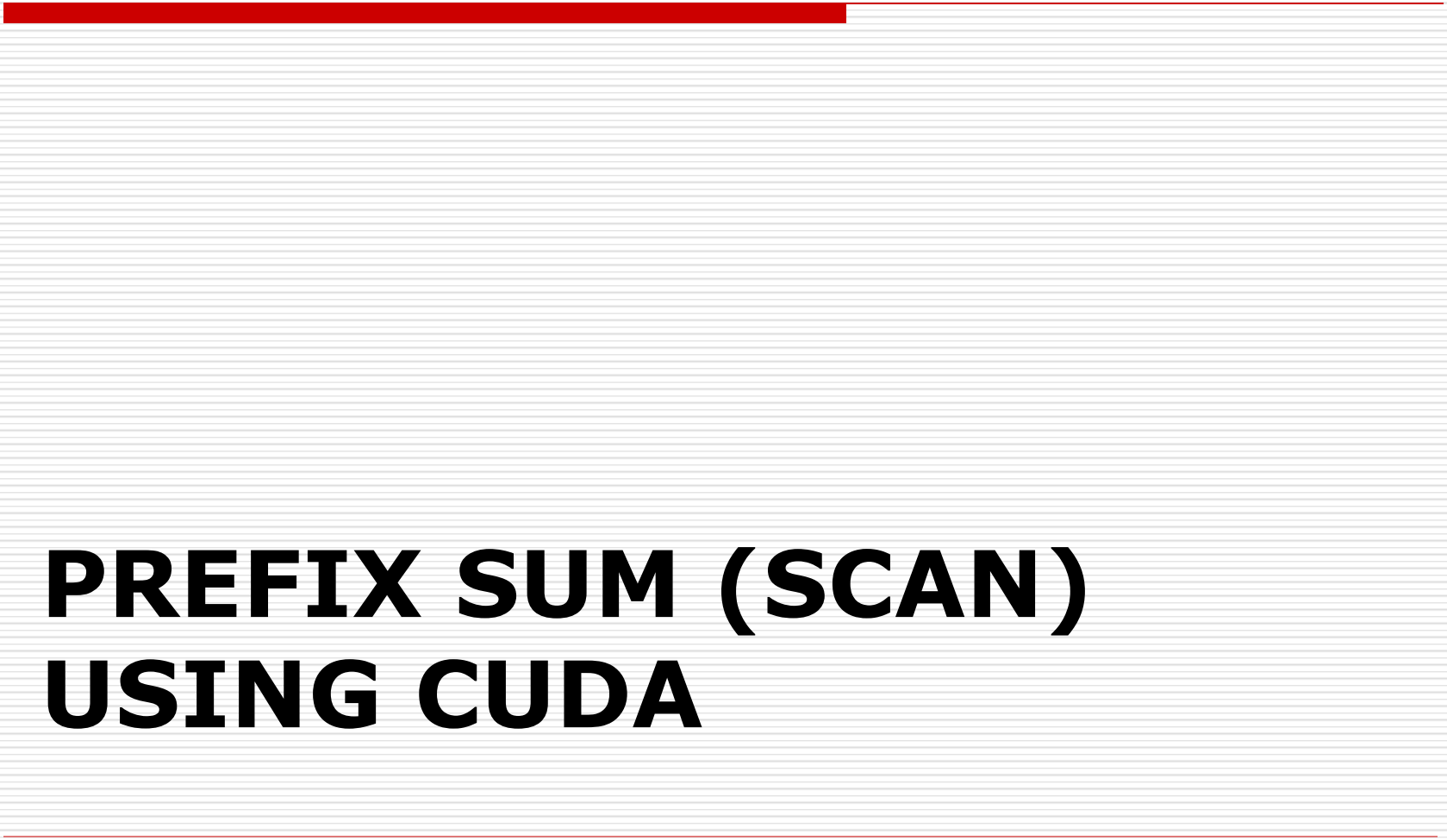
Passing last elements to a processor

6,21,45

Computing prefix sum of last elements on the processor

1,3,6,10,15,21,28,36,45

Adding global prefix sum to local prefix sum in each processor


PREFIX SUM (SCAN)
USING CUDA

Example: Scan or Parallel-prefix sum

Definition: *The* all-prefix-sums *operation takes a binary associative operator* $\oplus$, *and an array of n elements*

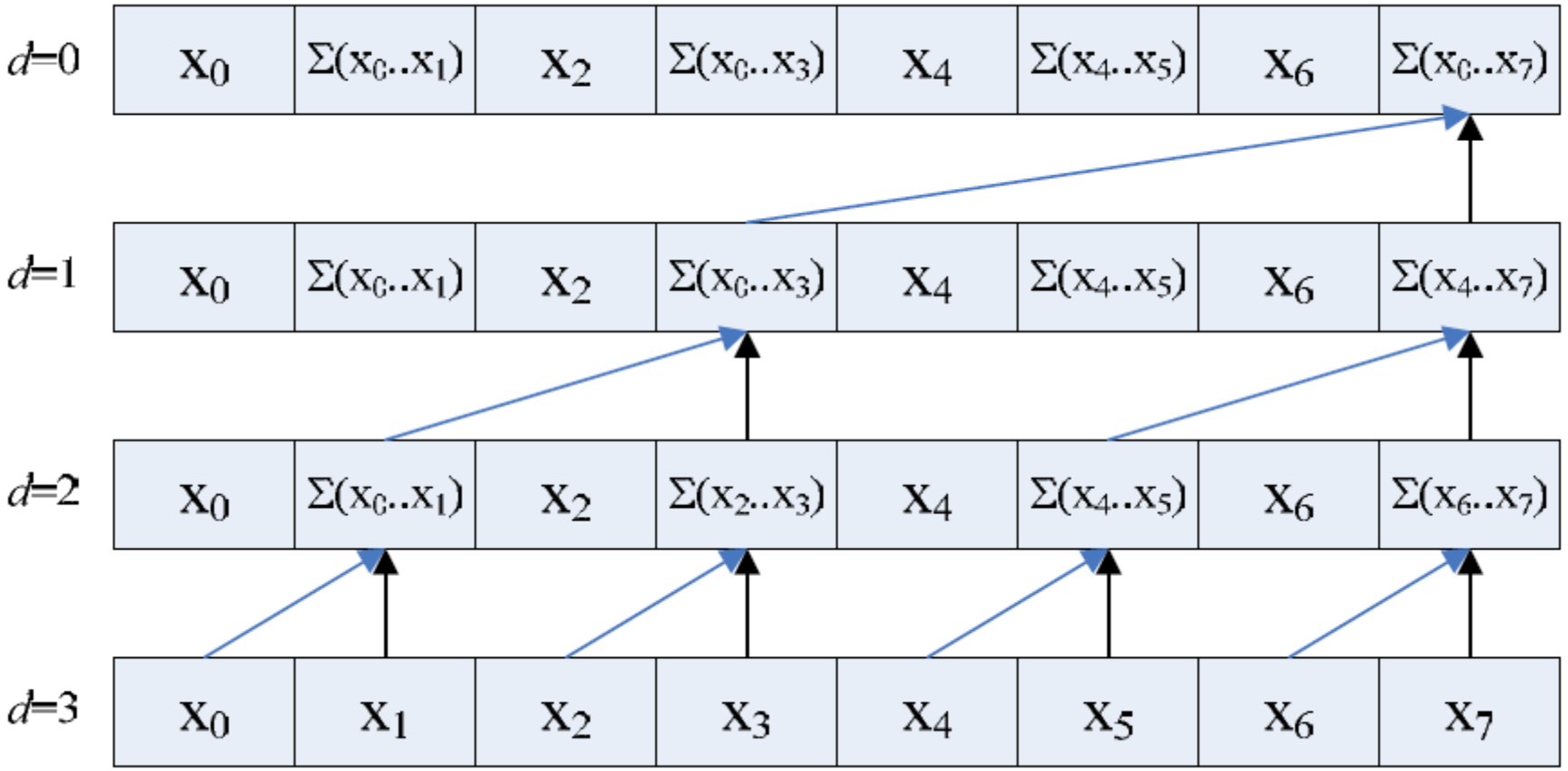
$$[a_0, a_1, \ldots, a_{n-1}],$$

and returns the array

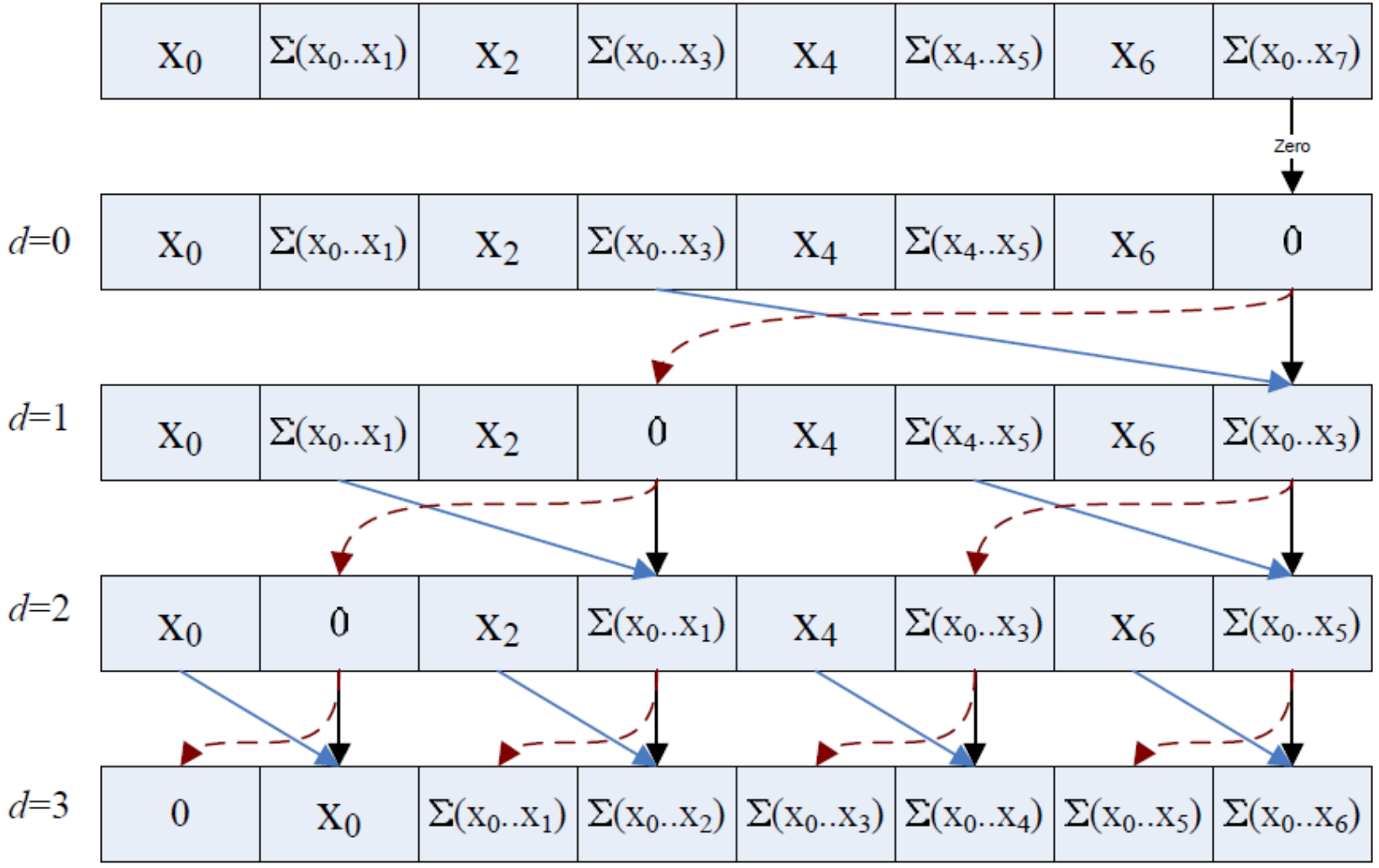
$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})].$$

- Using binary tree
- An upward reduction phase (reduce phase or up-sweep phase)
  - Traversing tree from leaves to root forming partial sums at internal nodes
- Down-sweep phase
  - Traversing from root to leaves using partial sums computed in reduction phase

Up Sweep


d=0
x0
Σ(x0..x1)
x2
Σ(x0..x3)
x4
Σ(x4..x5)
x6
Σ(x0..x7)
d=1
x0
Σ(x0..x1)
x2
Σ(x0..x3)
x4
Σ(x4..x5)
x6
Σ(x4..x7)
d=2
x0
Σ(x0..x1)
x2
Σ(x2..x3)
x4
Σ(x4..x5)
x6
Σ(x6..x7)
d=3
x0
x1
x2
x3
x4
x5
x6
x7

Down Sweep


x_0 $\Sigma(x_0..x_1)$ x_2 $\Sigma(x_0..x_3)$ x_4 $\Sigma(x_4..x_5)$ x_6 $\Sigma(x_0..x_7)$
Zero
d=0
x_0 $\Sigma(x_0..x_1)$ x_2 $\Sigma(x_0..x_3)$ x_4 $\Sigma(x_4..x_5)$ x_6 0
d=1
x_0 $\Sigma(x_0..x_1)$ x_2 0 x_4 $\Sigma(x_4..x_5)$ x_6 $\Sigma(x_0..x_3)$
d=2
x_0 0 x_2 $\Sigma(x_0..x_1)$ x_4 $\Sigma(x_0..x_3)$ x_6 $\Sigma(x_0..x_5)$
d=3
0 x_0 $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_0..x_4)$ $\Sigma(x_0..x_5)$ $\Sigma(x_0..x_6)$

Host Code

- int main(){
- const unsigned int num_threads = num_elements / 2;
- /* cudaMalloc d_idata and d_odata */
- cudaMemcpy(d_idata, h_data, mem_size, cudaMemcpyHostToDevice));
- dim3 grid(256, 1, 1); dim3 threads(num_threads, 1, 1);
- scan<<< grid, threads>>> (d_odata, d_idata);
- cudaMemcpy(h_data, d_odata[i], sizeof(float) * num_elements, cudaMemcpyDeviceToHost
- /* cudaFree d_idata and d_odata */
- }

Device Code

```
__global__ void scan_workefficient(float *g_odata, float
    *g_idata, int n)
{
   // Dynamically allocated shared memory for scan
    kernels
   extern  __shared__  float temp[];

   int thid = threadIdx.x;    int offset = 1;

   // Cache the computational window in shared memory
   temp[2*thid]   = g_idata[2*thid];
   temp[2*thid+1] = g_idata[2*thid+1];
```

Device Code

```
// build the sum in place up the tree
for (int d = n>>1; d > 0; d >>= 1)
{
    __syncthreads();

    if (thid < d)
    {
        int ai = offset*(2*thid+1)-1;
        int bi = offset*(2*thid+2)-1;

        temp[bi] += temp[ai];
    }

    offset *= 2;
}
```

Device Code

```
// scan back down the tree

// clear the last element
if (thid == 0)        temp[n - 1] = 0;
// traverse down the tree building the scan in place
for (int d = 1; d < n; d *= 2)
{
    offset >>= 1;
    __syncthreads();
    if (thid < d)
    {
        int ai = offset*(2*thid+1)-1;
        int bi = offset*(2*thid+2)-1;
```

Device Code

```
        float t   = temp[ai];
        temp[ai]  = temp[bi];
        temp[bi] += t;
    }
  }

  __syncthreads();

  // write results to global memory
  g_odata[2*thid]   = temp[2*thid];    g_odata[2*thid+1] =
   temp[2*thid+1];
}
```