Lec8: Scan

Reference: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda
textbook chapter 11

Introduction: Scan

- Similar to Reduce, **Scan** is another important parallel primitives/kernels. Also called **prefix-sum.**
- Scan: input an array [a0, a1, ..., an];
  output an array [b0, b1, ..., bn]; where:
  b0 = 0
  b1 = a0
  b2 = a0 + a1
  b3 = a0 + a1 + a2
  ...
  bn = a0 + a1 + ... + a[n-1]

Introduction cont’d

- What described previous slide is called **exclusive scan,** meaning b[k] = a[0] + a[1] + …+ a[k-1] which excludes b[k].
- On the other hand, **inclusive scan** includes a[k] in computing b[k]:
  b[k] = a[0] + a[1] + … + a[k-1] + a[k]
- They are not that different; one can convert one to the other with minimal computation.
- We’ll always default to exclusive scan unless otherwise stated.

Potential use scan:

- Sorting
- String comparison
- Part of many parallel algorithms where each process owns variable sized data
- Polynomial evaluation
- Stream compaction
- Building histograms and other data structures like trees, graphs, …

Use of scan, cont'd

- Convert certain sequential programs into parallel one easily. E.g.

```
void seq(float *in, float *out)
{
    out[0] = 0;
    for (int j=1; j<=n; j++)
        out[j] = out[j-1] + f(in[j-1]);
}
```

```
vod para(float *in, float *out, int n)
{
    float temp[n];
    for (int j=0; j<n; j++)
        temp[j] = f(in[j]);
    scan(out, temp); // exclusive scan
}
```

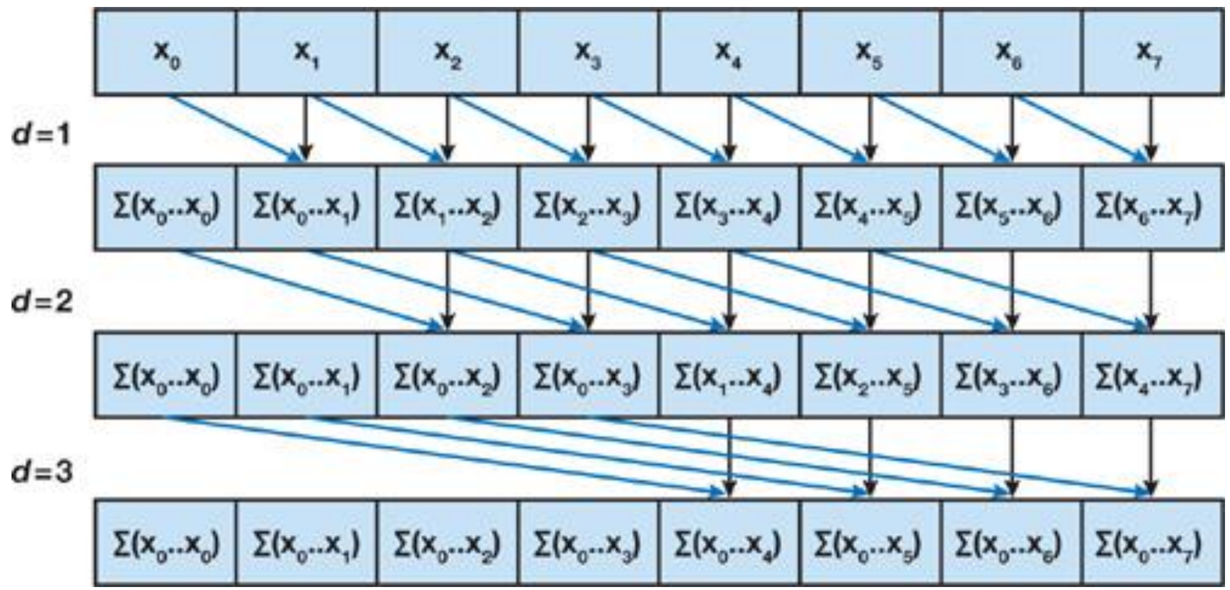
Sequential Algo; work efficiency

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

- Seems inherently sequential?
- But time complexity is pretty good—only n additions for size n input.
- If a parallel algorithm does not more work than the best sequential one (asymptotically), then it's called **work efficient.**
- This sequential algo is O(n). If you go by definition of scan, you get a O(n^2) one

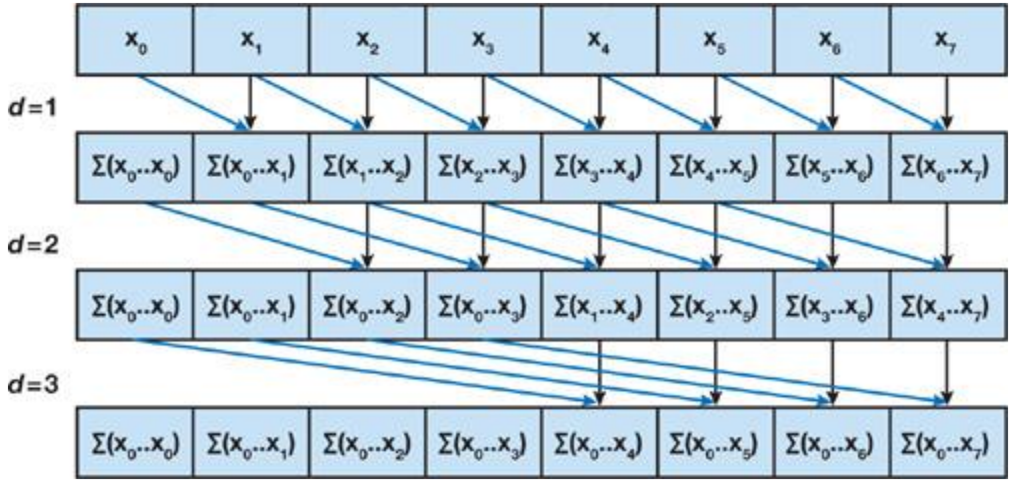
Fixed size input on 1 thread block

Parallel Scan: V0 (Hillis and Steele)



- What's the total work (say additions?)
- Each round there are between [n/2, n] additions.
- There are log_2(n) rounds. Why? (in round d, each element contains at most 2^d x[i]s)
- Total work is therefore O(n log(n))
  Not **work-efficient**

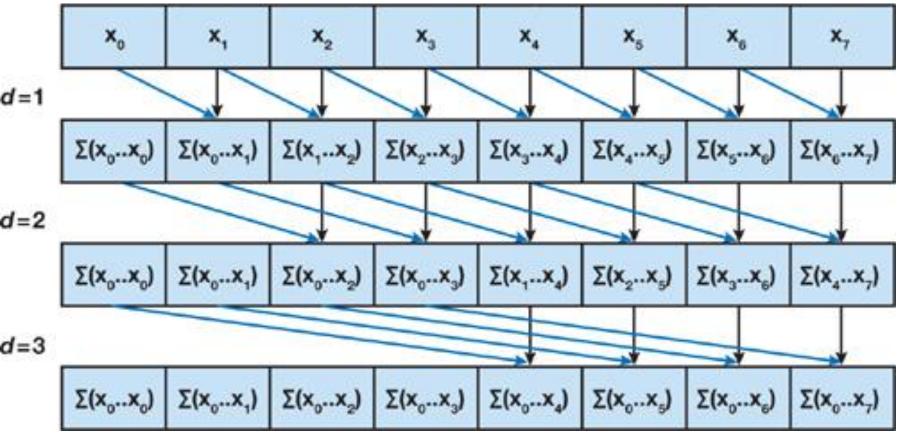
Scan: V0 (Hillis and Steele)




```
for d = 1 to log2(n) do
    m = 2^d
    for all k in parallel do
        if k >= m then
            x[k] = x[k-m/2] + x[k]
```

The algorithm proceeds round by round.
1. In each round, every element in the array got updated by sum of two elements of previous round
2. After round d, each element contains sum of at most 2^d x[.].
3. In the next round d+1, each element combines two 2^d x[.], therefore forming 2^(d+1) x[.] sum.
4. After every round, element k contains only x[i] where i<k.
5. Corollary of previous points, after log2(n) rounds, for all k, elemen k will contains as many x[i] where i<k. This satisfy the definition of exclusive scan.

Scan V0: CUDA implementation (double buffering)

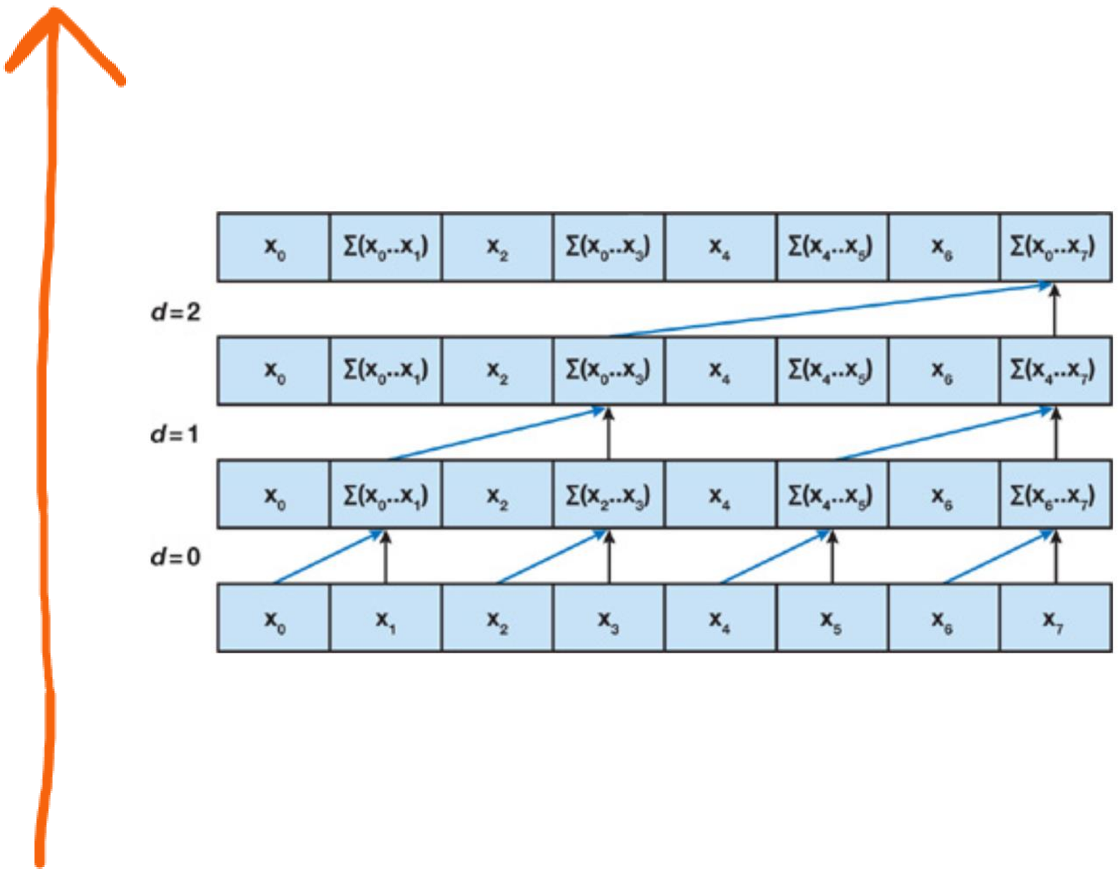



```
// Naive Hillis-Steele exclusive scan using double buffering
__global__ void scan_naive(int *input, int *output, int n) {
    extern __shared__ int temp[];
    int thid = threadIdx.x;
    int pout = 0, pin = 1;

    temp[pout * n + thid] = (thid > 0) ? input[thid - 1] : 0;
    __syncthreads();

    for (int offset = 1; offset < n; offset *= 2) {
        pout = 1 - pout;
        pin = 1 - pout;
        if (thid >= offset) {
            temp[pout * n + thid] = temp[pin * n + thid] + temp[pin * n + (thid - offset)];
        } else {
            temp[pout * n + thid] = temp[pin * n + thid];
        }
        __syncthreads();
    }

    output[thid] = temp[pout * n + thid];
}
```

Scan V1: Work efficient Scan

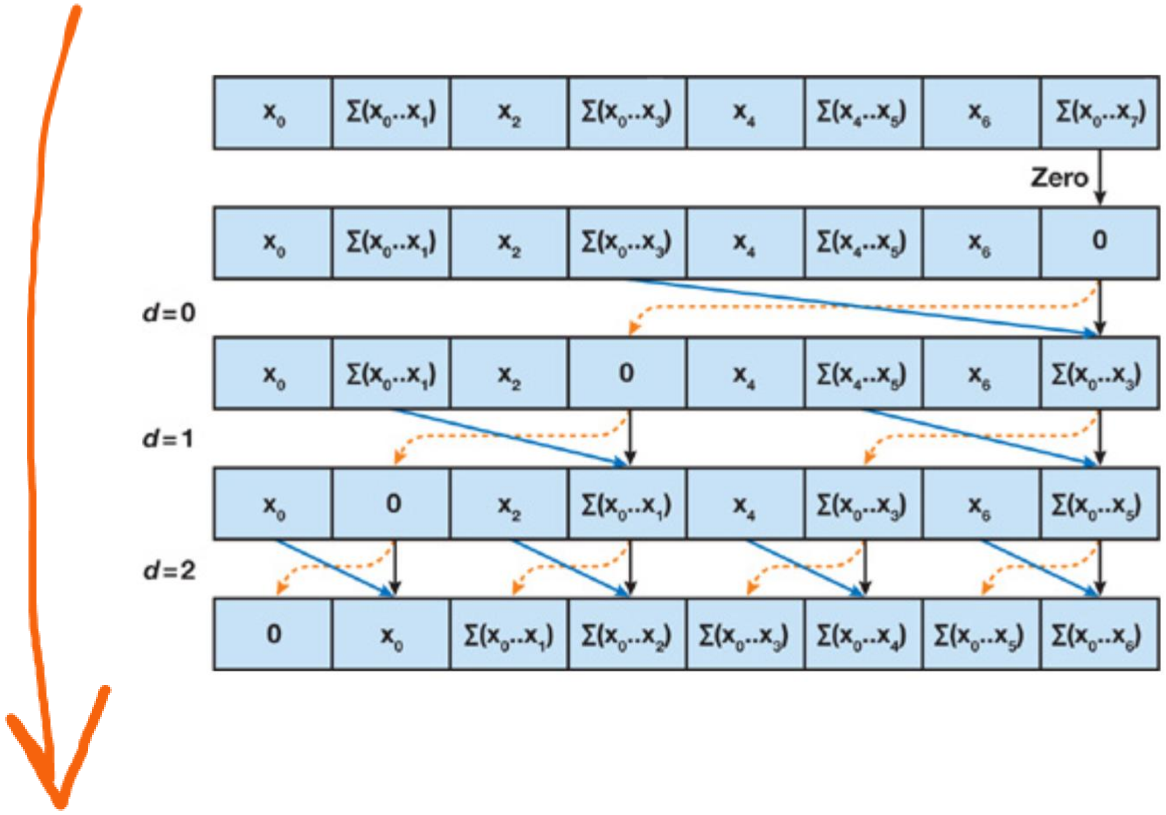
- V0 is not work efficient; how to get rid of the log2(n) factor in the total work?
- (Blelloch 1990) Two phase tree algorithm:
  - Imagine a binary tree, where each input value is a leaf
  - **Up-sweep**: from leaves propagate up to root; each internal node combines its two children.
  - **Down-sweep**: complete the partial scan into full scan

Scan V1: Up-sweep



- From leaves to root, each internal node combines its two children.
- How much work? Each node does one addition, so in total O(n) addition.
- What've got? Partial scans.
- Need a down-sweep to make partial scan full.
- What is missing?

Scan V1: Down-sweep

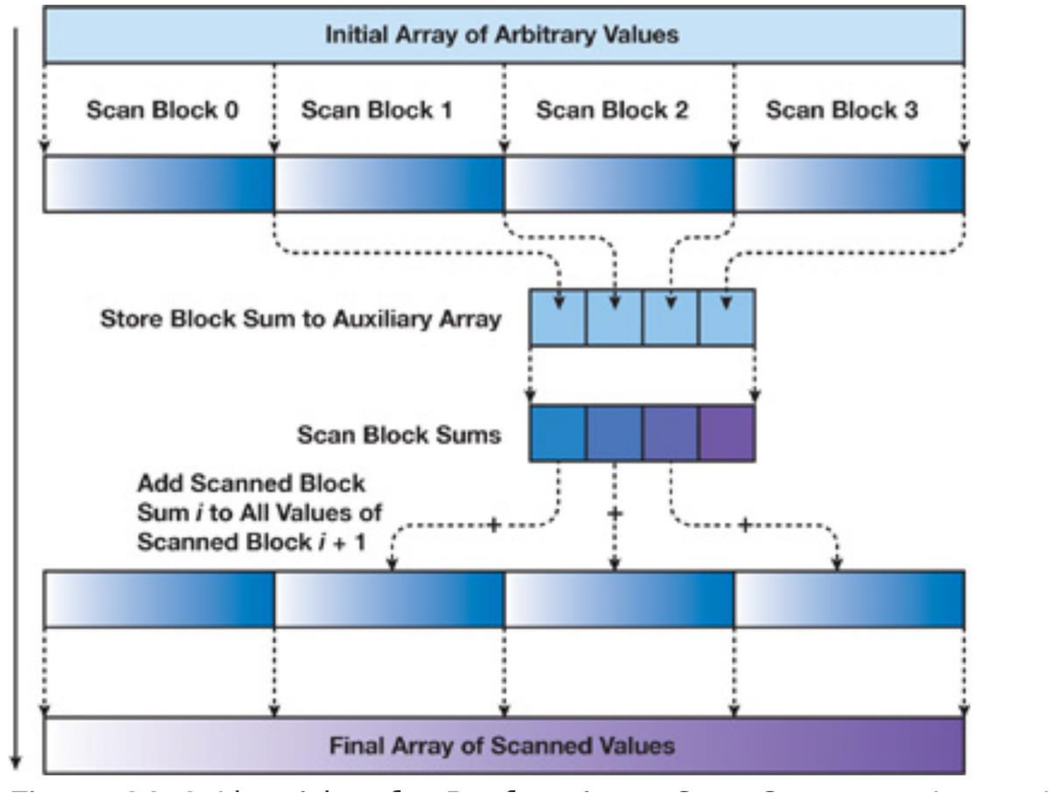


- Traverse back from the root to leaves, level by level.
- Each node:
  - Pass its value to left child
  - Pass its value plus left to right child
- Each node does two things, so down-sweep is O(n) operation.

```
// Work-efficient Blelloch exclusive scan
__global__ void scan_work_efficient(int *input, int *output, int n) {
    extern __shared__ int temp[];
    int thid = threadIdx.x;
    int offset = 1;

    // Load input into shared memory
    temp[2 * thid] = input[2 * thid];
    temp[2 * thid + 1] = input[2 * thid + 1];
    __syncthreads();

    // Up-sweep phase
    for (int d = n >> 1; d > 0; d >>= 1) {
        if (thid < d) {
            int ai = offset * (2 * thid + 1) - 1;
            int bi = offset * (2 * thid + 2) - 1;
            temp[bi] += temp[ai];
        }
        offset *= 2;
        __syncthreads();
    }

    // Clear the last element
    if (thid == 0) {
        temp[n - 1] = 0;
    }
```

```
    // Down-sweep phase
    for (int d = 1; d < n; d *= 2) {
        offset >>= 1;
        if (thid < d) {
            int ai = offset * (2 * thid + 1) - 1;
            int bi = offset * (2 * thid + 2) - 1;
            int t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }
        __syncthreads();
    }

    // Write results to output
    output[2 * thid] = temp[2 * thid];
    output[2 * thid + 1] = temp[2 * thid + 1];
}
```

Arbitrary length input?



- Basic idea: divide and conquer like reduction:
  each thread block “scan” its own chunk of input array.
- Each threadblock/scan contributes one block sum to an intermediate array SUMS
- Scan on the SUMS
- Add SUMS[i] to block i.

```
// Level 1: Scan input into local scans and sums
int numBlocks1 = (numElements + blockSize - 1) / blockSize; // 16,384
int *d_local_scans, *d_sums1;
CHECK_CUDA(cudaMalloc(&d_local_scans, sizeBytes));
CHECK_CUDA(cudaMalloc(&d_sums1, numBlocks1 * sizeof(int)));
scan_naive_blocks<<<numBlocks1, blockSize, 3 * blockSize * sizeof(int)>>>(d_input, d_local_scans, d_sums1, numElements, blockSize);

// Level 2: Scan d_sums1
int numBlocks2 = (numBlocks1 + blockSize - 1) / blockSize; // 17
int *d_local_scans2, *d_sums2;
CHECK_CUDA(cudaMalloc(&d_local_scans2, numBlocks1 * sizeof(int)));
CHECK_CUDA(cudaMalloc(&d_sums2, numBlocks2 * sizeof(int)));
scan_naive_blocks<<<numBlocks2, blockSize, 3 * blockSize * sizeof(int)>>>(d_sums1, d_local_scans2, d_sums2, numBlocks1, blockSize);

// Level 3: Scan d_sums2 with a single block
int *d_sums2_scans;
CHECK_CUDA(cudaMalloc(&d_sums2_scans, numBlocks2 * sizeof(int)));
scan_naive<<<1, numBlocks2, 2 * numBlocks2 * sizeof(int)>>>(d_sums2, d_sums2_scans, numBlocks2);

// Combine Level 2 results
int *d_sums_scans;
CHECK_CUDA(cudaMalloc(&d_sums_scans, numBlocks1 * sizeof(int)));
add_sums<<<numBlocks2, blockSize>>>(d_local_scans2, d_sums2_scans, d_sums_scans, numBlocks1, blockSize);

// Combine Level 1 results into final output
add_sums<<<numBlocks1, blockSize>>>(d_local_scans, d_sums_scans, d_output, numElements, blockSize);
```

Benchmark

```
=== Performance Summary ===
CPU scan:             63.875 ms (1.96 GB/s)
Naive GPU scan:       1.789 ms (69.86 GB/s)
Work-efficient scan: 0.825 ms (151.43 GB/s)
CUB library scan:    0.289 ms (431.87 GB/s)
```

Like reduce, scan performance is capped by global memory bandwidth of 760 GB/s.
Ways to improve it to the CUB level of performance?

Single Pass Scan for Memory Access Efficiency

- In previous solution, partial scans are written back to global memory, and in phase 3 updated again. This read/write whole array multiple times in 3 kernels.
- To reduce this traffic, best to use 1 kernel, fusing the three phases into one kernel.
- Problem is synchronization—phase 3 cannot start until phase 2 (block sum scan) finished. And we know blocks cannot synchronize
- need that's a lie—thread blocks can synchronize, but only in ad-hoc way. We are going ad-hoc synchronization.
- Upon further inspection, we don't actually need a thread block barrier between phase 1 and phase 2; for a thread block i it only needs all blocks <i finish phase 1 before block i can start phase 2. (The same synchronization applies to phase 2 and phase 3)
- This suggests a lighter synchronization-–streaming or domino synchronization–each thread block just waits on its immediate previous block:

  0 -> 1 -> 2 -> … -> tb-1

Adjacent synchronization

```
__shared__ float previous_sum;
if (threadIdx.x == 0){
    // Wait for previous flag
    while(atomicAdd(&flags[bid], 0) == 0) { }
    // Read previous partial sum
    previous_sum = scan_value[bid];
    // Propagate partial sum
    scan_value[bid + 1] = previous_sum + local_sum;
    // Memory fence
    __threadfence();
    // Set flag
    atomicAdd(&flags[bid + 1], 1);
}
__syncthreads();
```

- Ad-hoc wait on signal
- Flags[bid] false: not ready; true: ready.
- Note how to wait until previous block finishes?
- Scheduling problems? Deadlock?

Scheduling & Synchronization

- In terms of scheduling, if we can schedule block 0,1,…, k first, and then schedule k+1,…,2k, … etc, i.e. scheduling blocks in this streaming fashion, then the synchronizations are automatically satisfied.
- But we can't force scheduling threadblocks on GPU. Unless…
- We decouple the static blockIdx with the logical block index. I.e., we dynamically assign blocks an index following 0, 1, 2, … as they are scheduled, **regardless of their blockIdx.**

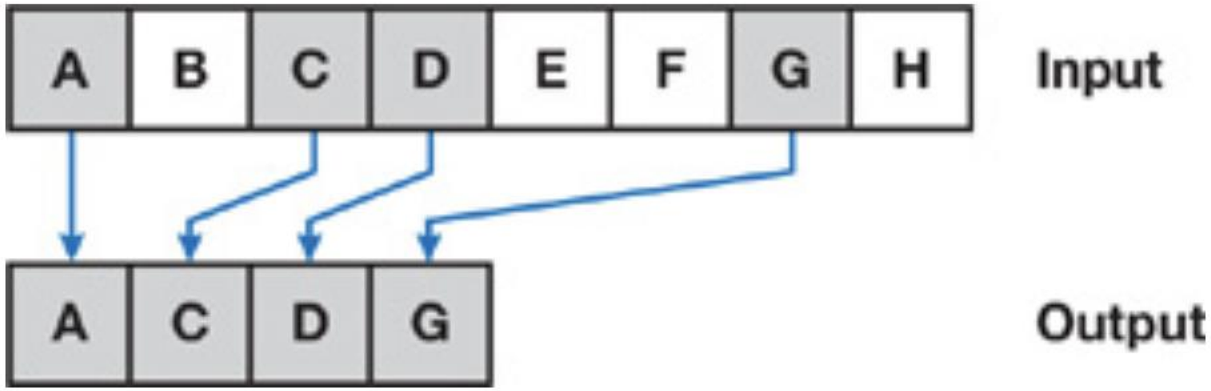
Dynamic block index assignment

```
__shared__ unsigned int bid_s;
if (threadIdx.x == 0) {
    bid_s = atomicAdd(blockCounter, 1);
}
__syncthreads();
unsigned int bid = bid_s;
```

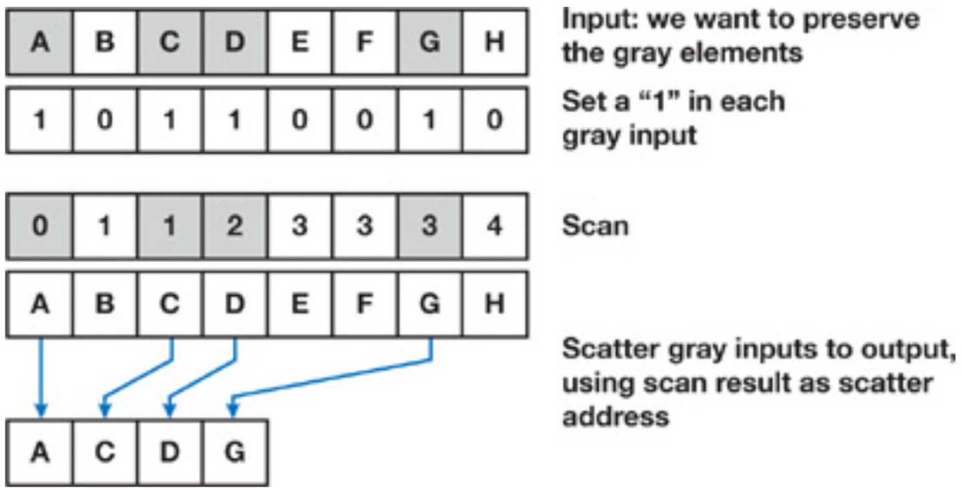
- Note that bid is no longer statically determined by blockIdx.x
- Instead, it's given out as blocks get scheduled.
- This ensures that (logical) thread blocks are scheduled linearly; i.e. 0, 1, …, k will be scheduled first; then k+1, …, 2k, etc.

Use of Scan: Stream Compaction

- Compaction: Input an array, output an array, filtering out unwanted elements determined by predicate p().
- How to do this in parallel? (hint: use scan?)



Stream Compaction



- Step1: map using p() for each input element (data parallel)
- Step2: Scan the 1-0 array of last step
- Step3: Scatter—look up number in scan result to find its location (or should be filtered out)
  Is this step parallel?