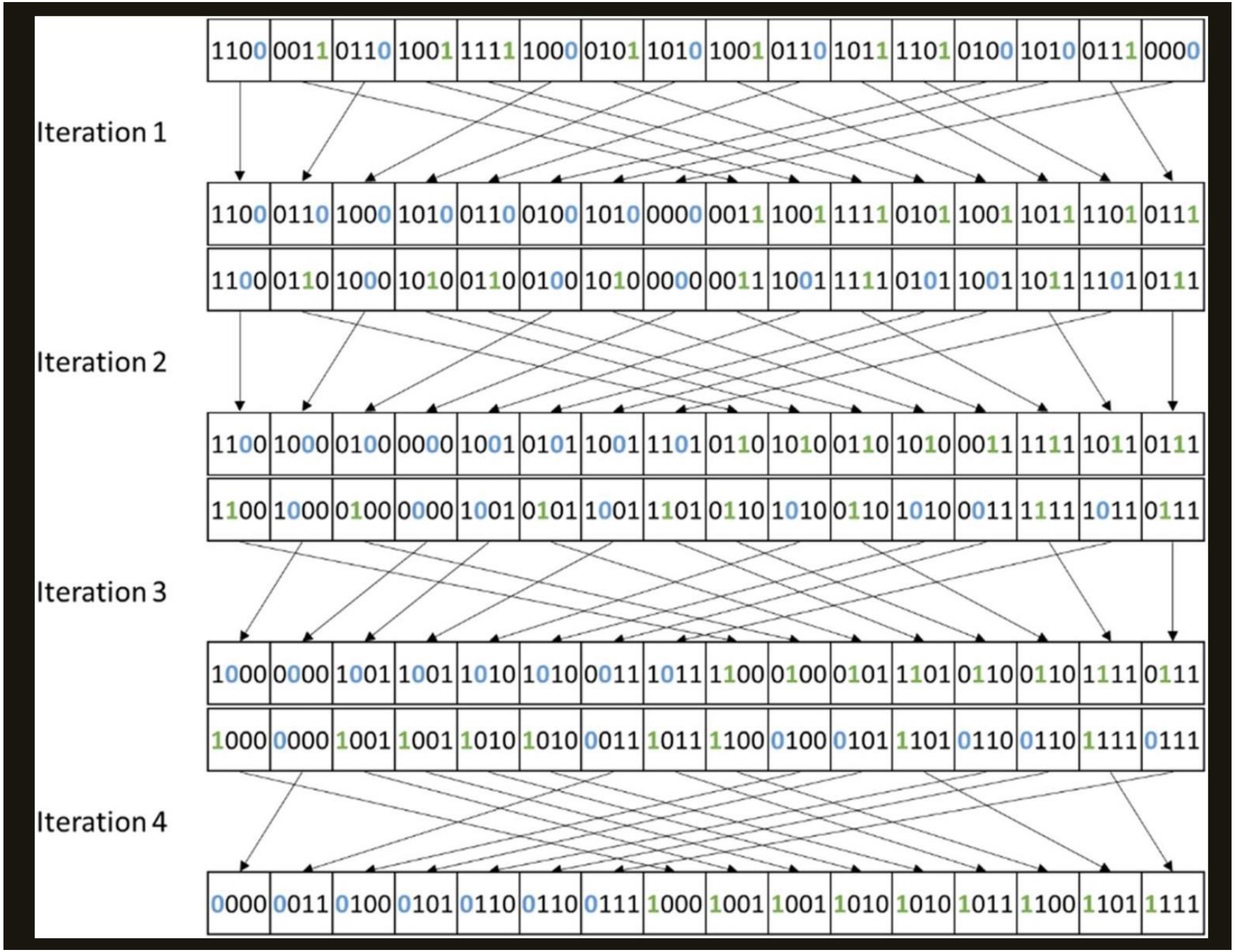
Lec11: Sorting

Two parallel sorting algorithms

- Non-comparison based: Radix sort
- Comparison based: merge sort

Radix sort algorithm

- Non comparison based; limited to certain keys (mostly integers)
- Sorting algorithms proceeds position by position, sorting from Least Significant Bit (LSB) to Most Significant Bit (MSB)
- **Invariant property:** after round k, the array is **sorted** according to the last k-bits.
- After m rounds (m is the max # bits of elements in the array), the whole array is completely sorted.
  Time complexity? O(m*n) (with fixed sized integers like 64 bit, it's effective O(n) linear time)
- We can use binary representation for radix sort, so that sorting by 1 position is simply putting 0s before 1s


1100 0011 0110 1001 1111 1000 0101 1010 1001 0110 1011 1101 0100 1010 0111 0000
Iteration 1
1100 0110 1000 1010 0110 0100 1010 0000 0011 1001 1111 0101 1001 1011 1101 0111
1100 0110 1000 1010 0110 0100 1010 0000 0011 1001 1111 0101 1001 1011 1101 0111
Iteration 2
1100 1000 0100 0000 1001 0101 1001 1101 0110 1010 0110 1010 0011 1111 1011 0111
1100 1000 0100 0000 1001 0101 1001 1101 0110 1010 0110 1010 0011 1111 1011 0111
Iteration 3
1000 0000 1001 1001 1010 1010 0011 1011 1100 0100 0101 1101 0110 0110 1111 0111
1000 0000 1001 1001 1010 1010 0011 1011 1100 0100 0101 1101 0110 0110 1111 0111
Iteration 4
0000 0011 0100 0101 0110 0110 0111 1000 1001 1001 1010 1010 1011 1100 1101 1111

Parallel Radix Sort

- There is sequential dependencies between rounds, so that can't be parallelized
- We must do parallelization **within each round**
- How? Let's try input decompositions: each thread takes one input element and **determine its output position**.
- OK, let's say thread i takes input[i] which has 0 bit for the round. Where should we put it?

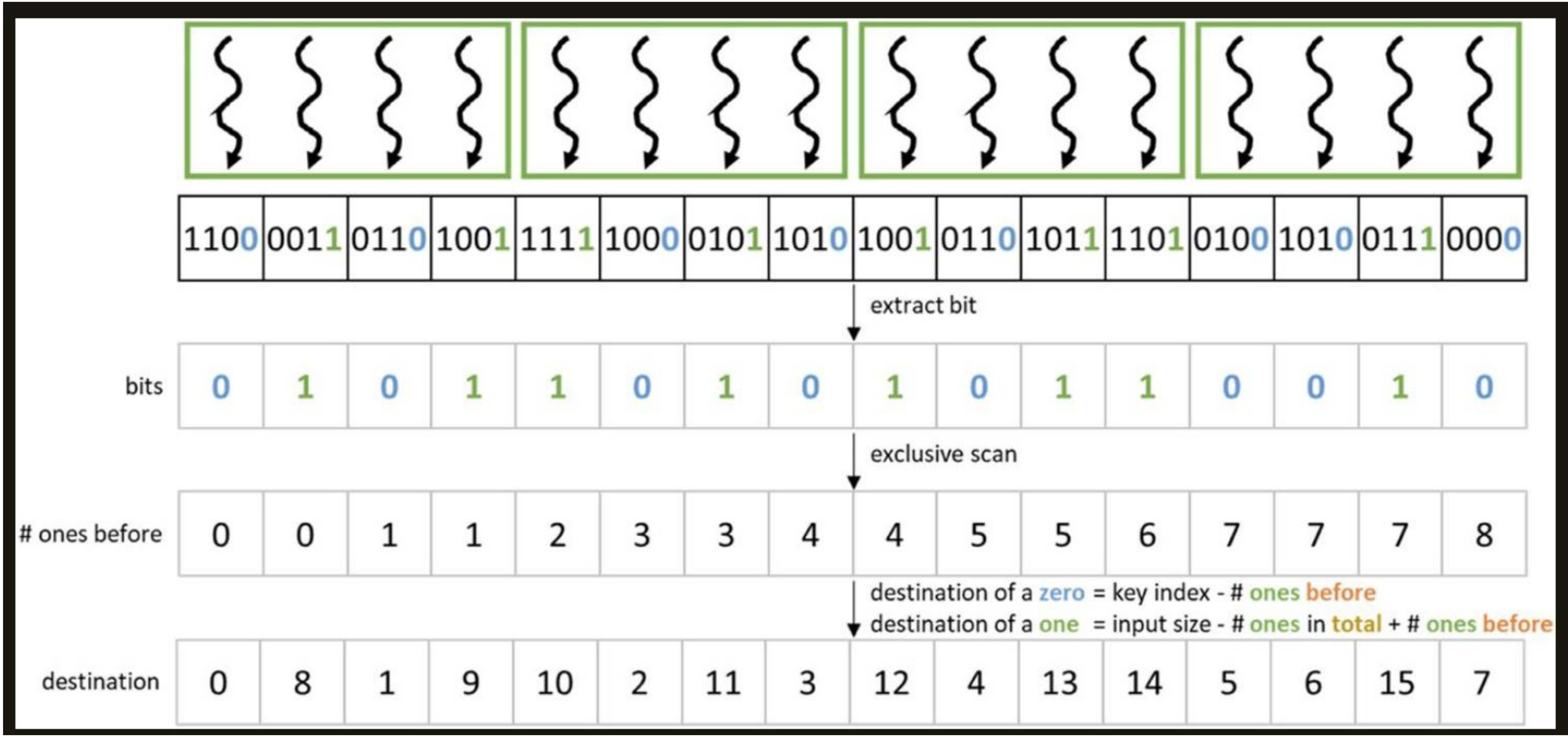
destination of a zero	= # zeros before
	= # keys before - # ones before
	= key index - # ones before

Parallel Radix-sort

- What if input[i] has value of 1 in the round bit?

destination of a one = # zeros in total + # ones before
= (# keys in total - # ones in total) + # ones before
= input size - # ones in total + # ones before

- OK, it seems if we can get
  #ones before (every index i)
  then we can simply calculate the destination of input[i]
- How to compute #ones_before_index(i) for all i=0, 1, ..., n-1?
  Does it look related to **scan?**



1100 0011 0110 1001 1111 1000 0101 1010 1001 0110 1011 1101 0100 1010 0111 0000
extract bit
bits
0 1 0 1 1 0 1 0 1 0 1 1 0 0 1 0
exclusive scan
ones before
0 0 1 1 2 3 3 4 4 5 5 6 7 7 7 8
destination of a zero = key index - # ones before
destination of a one = input size - # ones in total + # ones before
destination
0 8 1 9 10 2 11 3 12 4 13 14 5 6 15 7

```
__global__ void radix_sort_iter(unsigned int* input, unsigned int* output,
                    unsigned int* bits, unsigned int N, unsigned int iter) {
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int key, bit;
    if(i < N) {
        key = input[i];
        bit = (key >> iter) & 1;
        bits[i] = bit;
    }
    exclusiveScan(bits, N);
    if(i < N) {
        unsigned int numOnesBefore = bits[i];
        unsigned int numOnesTotal = bits[N];
        unsigned int dst = (bit == 0)?(i - numOnesBefore)
                                     :(N - numOnesTotal - numOnesBefore);
        output[dst] = key;
    }
}
```

FIGURE 13.4 Radix sort iteration kernel code.

Performance Considerations

- First thing first, what would be the performance metric?
  - Sorting – O(n log n) operations, minimum data from/to global memory is O(n)
  - Memory bandwidth is likely the hard bottleneck;
  - Therefore achieved global memory throughput GB/s 2*(bytes_of_input)/run_time
- Issues of the previous parallel Radix sort algorithm
  - Memory Coalesced?
  - # passes to read/write the whole array?
- What would be the expected performance of previous Radix sort in terms of global memory GB/s?

PerfOpt1: Write coalescing

- The previous radix sort algorithm, when threads write key back to their sorted positions, they are essentially "scattered" around
  - Not coalesced
  - Is this a big deal? Well the main cost of sorting seems to be (several passes) reading and shuffling/writing the whole array from/to global memory.
  - Half of the memory access (write) is not coalesced; we could have up to <2x speedup if we can make it coalesced
- How to make write coalescing?
  - Shared memory?
  - Locally sort in each thread block and store results in shared memory
  - And then write back to memory in coalescing.

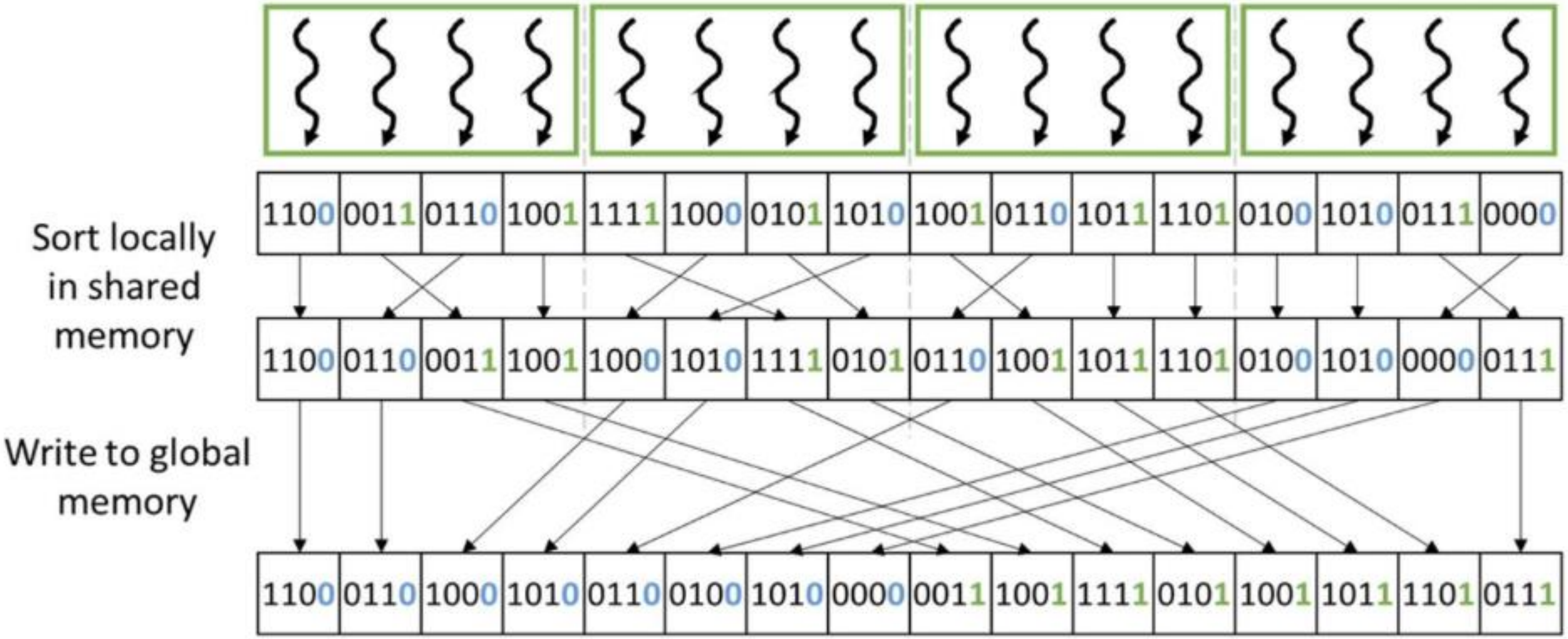


FIGURE 13.5 Optimizing for memory coalescing by sorting locally in shared memory before sorting into the global memory.

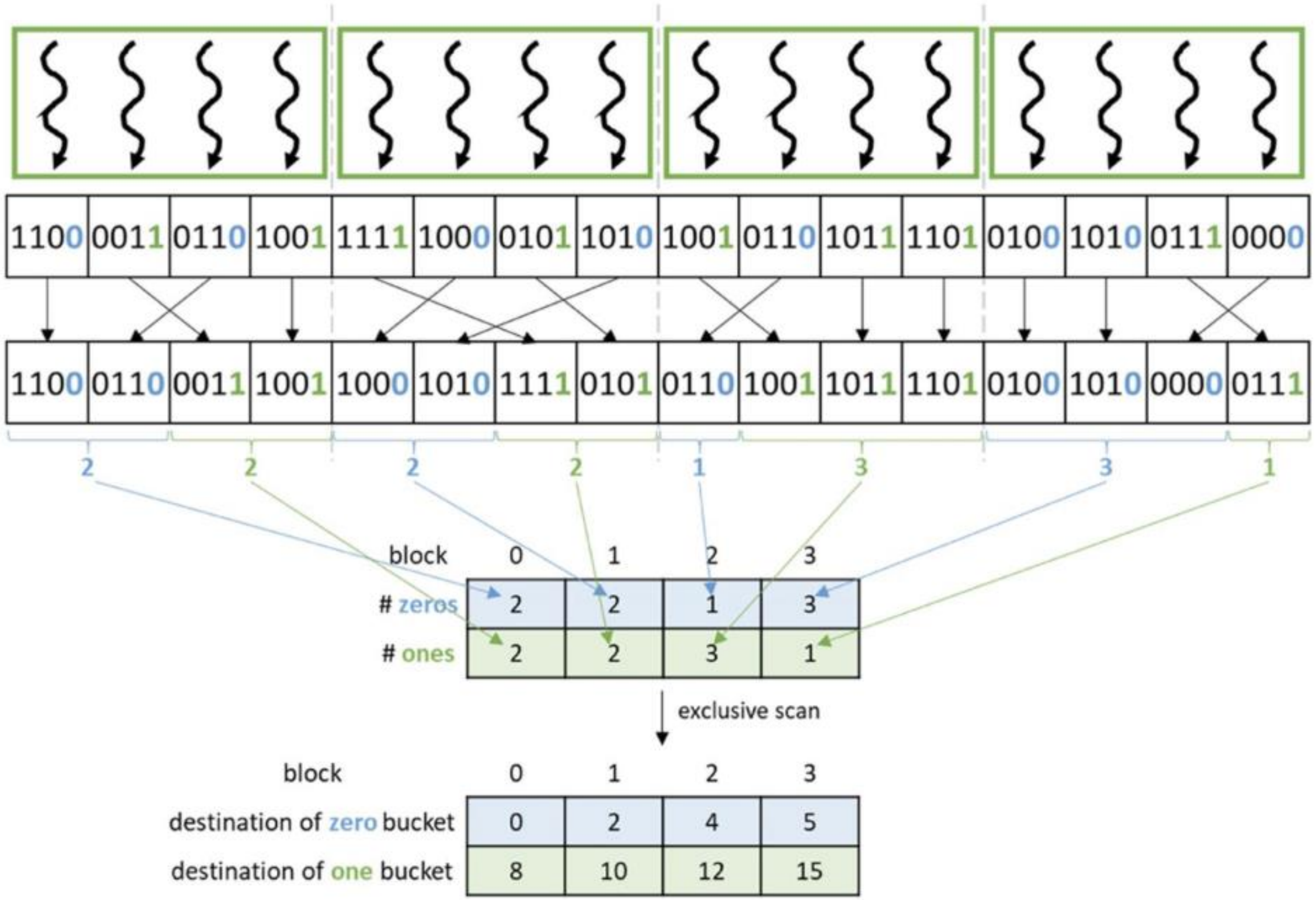


FIGURE 13.6 Finding the destination of each thread block's local buckets.

PerfOpt2: Reducing #pass

- With 1-bit radix sort, to sort 64-bit integer arrays, there are 64 passes, one pass per bit.
- Each pass will require at least one read pass and write pass of the whole array.
- So achieved GB/s would be 1/64 of peak hardware GB/s
- Big impact (potentially proportionally) to reduce the #passes
- How? Use multi-bit radix sort
  - Each pass, sort not by a single bit, but rather a few grouped bits.

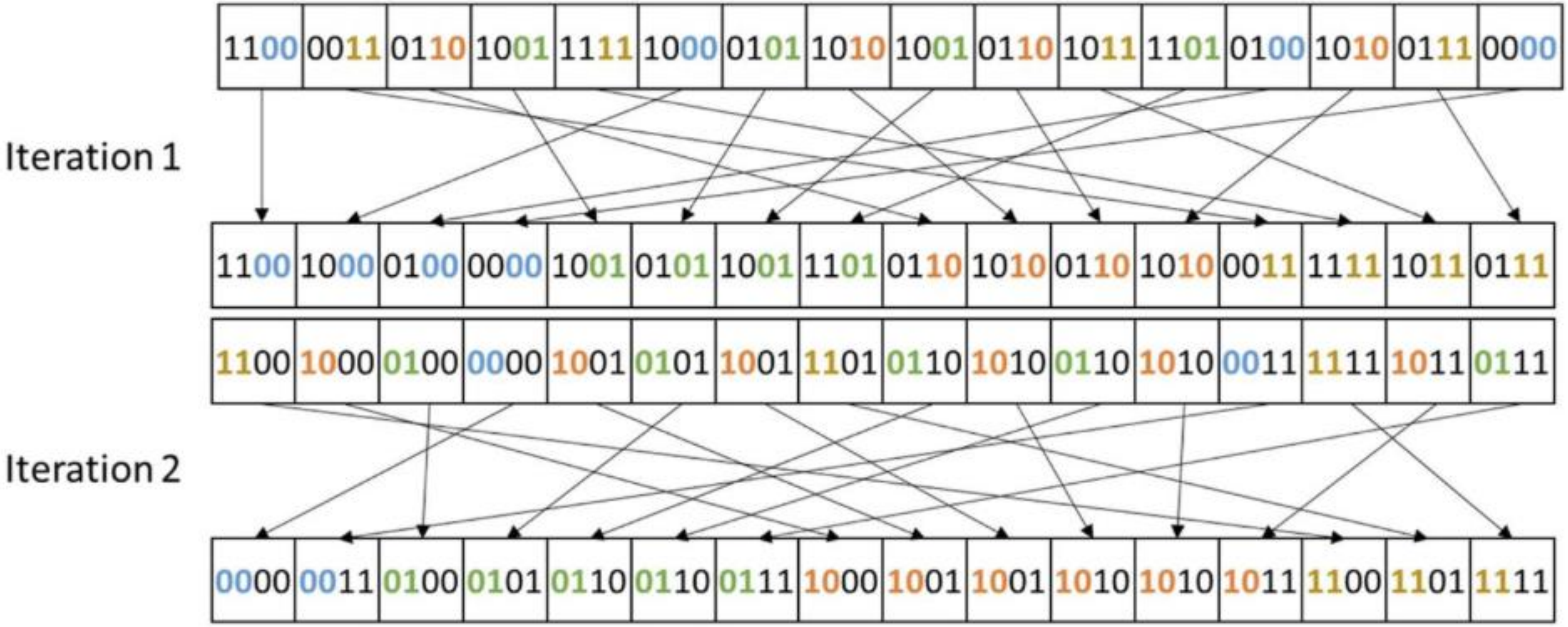
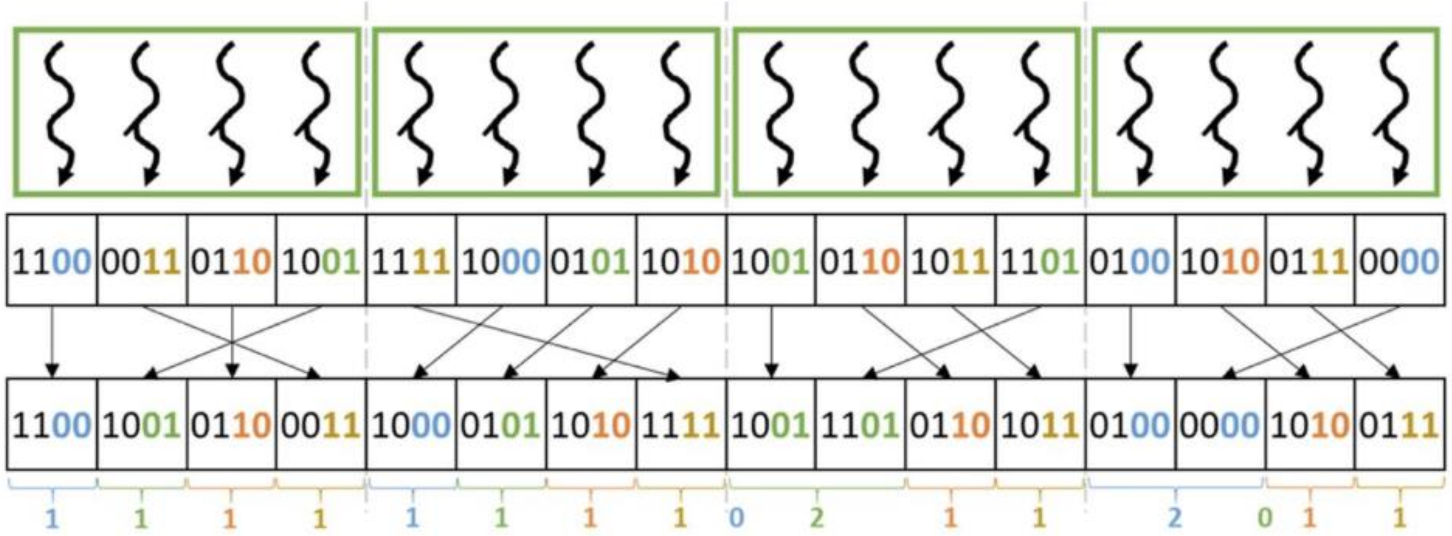


FIGURE 13.7 Radix sort example with 2-bit radix.

2-bit Radix sort

- How to sort keys into 4 buckets (00, 01, 10, 11) instead of 2 buckets (0, 1) in the 1-bit radix sort?
- Well, we can internally do a 1-bit radix sort within each thread block!
- This plays well with the shared-memory idea where results are first cached in shared memory and then at the end written back to global memory.
- OK next question, once each thread blocks finishes 4 buckets radix sort, how to assemble them in global memory?



block	0	1	2	3
# 00	1	1	0	2
# 01	1	1	2	0
# 10	1	1	1	1
# 11	1	1	1	1

↓ exclusive scan

block	0	1	2	3
destination of 00 bucket	0	1	2	2
destination of 01 bucket	4	5	6	8
destination of 10 bucket	8	9	10	11
destination of 11 bucket	12	13	14	15

FIGURE 13.9 Finding the destination of each block's local buckets for a 2-bit radix.

Pros and Cons of k-bit Radix sort

- Pros
  - Fewer #passes!
- Cons
  - Scan got bigger!
  - More complicated code
  - Write coalescing becomes worse

- Tradeoffs need to be regarding the best k-bit radix sort.

Parallel MergeSort

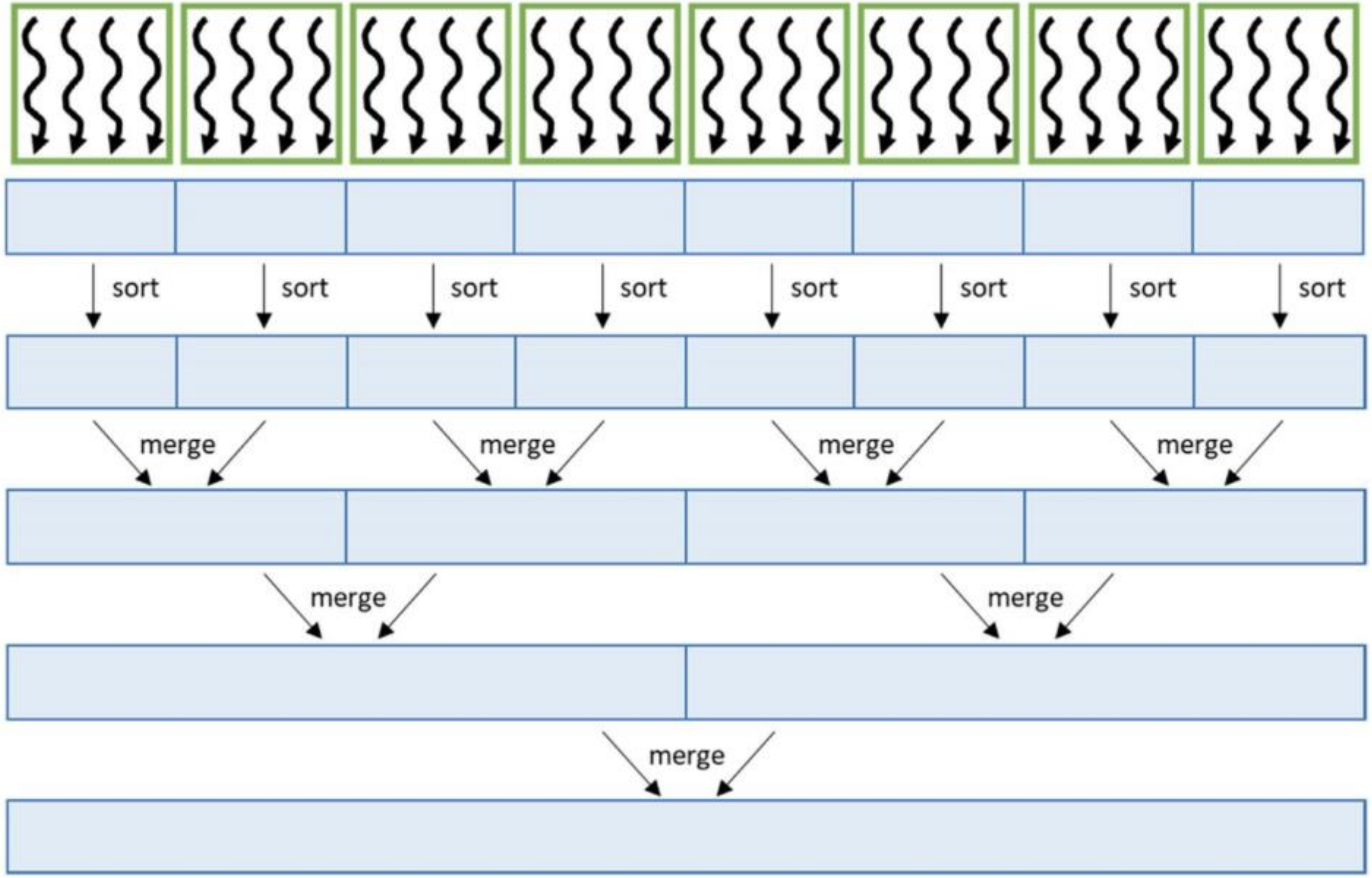


FIGURE 13.11 Parallelizing merge sort.

References and Further Reading

- CUB MergeSort : https://nvidia.github.io/cccl/cub/api/structcub_1_1DeviceMergeSort.html
- CUB RadixSort: https://nvidia.github.io/cccl/cub/api/structcub_1_1DeviceRadixSort.html