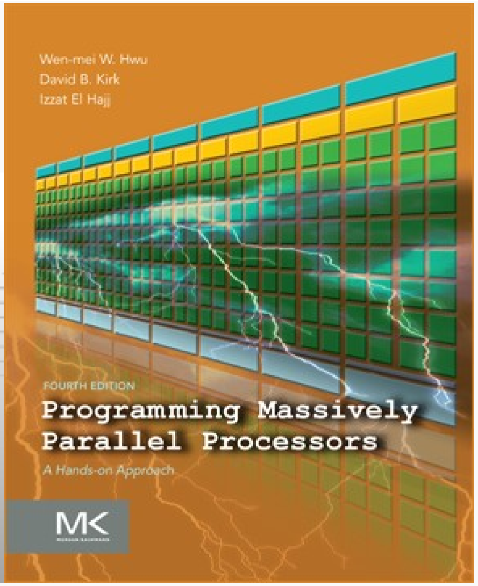
Programming Massively Parallel Processors

A Hands-on Approach

CHAPTER 13 › Sorting



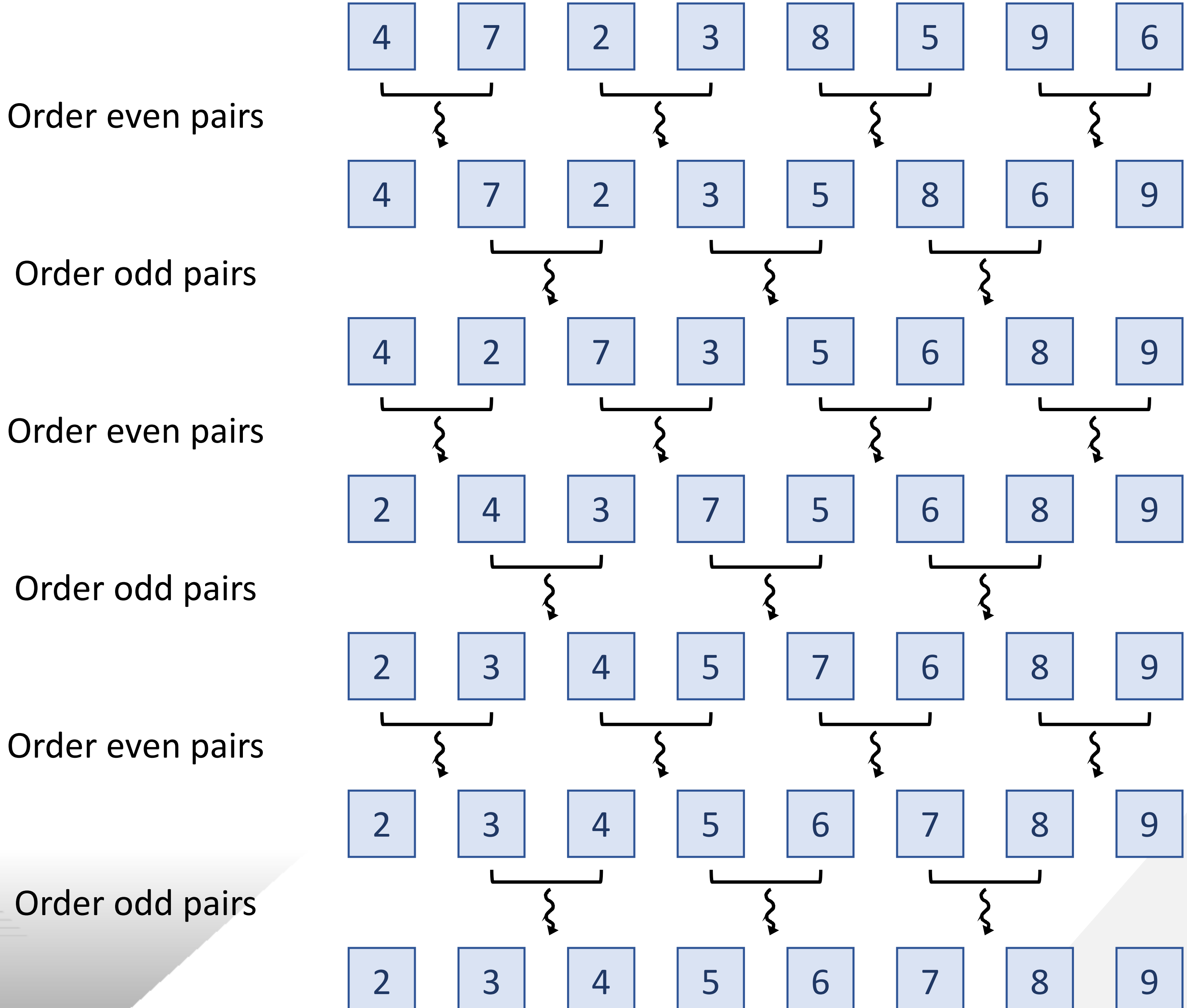
Odd-Even Sort

- **Odd-even** sort is similar to bubble sort in that it repeatedly compares and orders adjacent pairs of elements until the entire list is sorted

- Parallelization approach: each thread is responsible for ordering one pair of elements in each iteration

- To avoid race conditions, iterations alternate between processing only odd pairs (pairs where the first element is odd) or only even pairs (pairs where the first element is even) such that each element is accessed by only one thread

Odd-Even Sort Example



Odd-Even Sort Code

Kernel code:

```
__global__ void sort_kernel(unsigned int* data, unsigned int* hasChanged, unsigned int N,
unsigned int isOddStep) {
    unsigned int i = 2*(blockIdx.x*blockDim.x + threadIdx.x) + (isOddStep ? 1 : 0);
    if(i < N - 1) {
        if(data[i] > data[i + 1]) {
            unsigned int tmp = data[i];
            data[i] = data[i + 1];
            data[i + 1] = tmp;
            *hasChanged = 1;
        }
    }
}
```

Host code:

```
    unsigned int *hasChanged_d;
    cudaMalloc((void**) &hasChanged_d, sizeof(unsigned int));
    unsigned int hasChanged;
    do {
        cudaMemset(hasChanged_d, 0, sizeof(unsigned int));
        sort_kernel <<< (N + 2048 - 1)/2048 + 1, 1024 >>> (data_d, hasChanged_d, N, 0);
        sort_kernel <<< (N + 2048 - 1)/2048 + 1, 1024 >>> (data_d, hasChanged_d, N, 1);
        cudaMemcpy(&hasChanged, hasChanged_d, sizeof(unsigned int), cudaMemcpyDeviceToHost);
    } while(hasChanged);
    cudaFree(hasChanged_d);
```

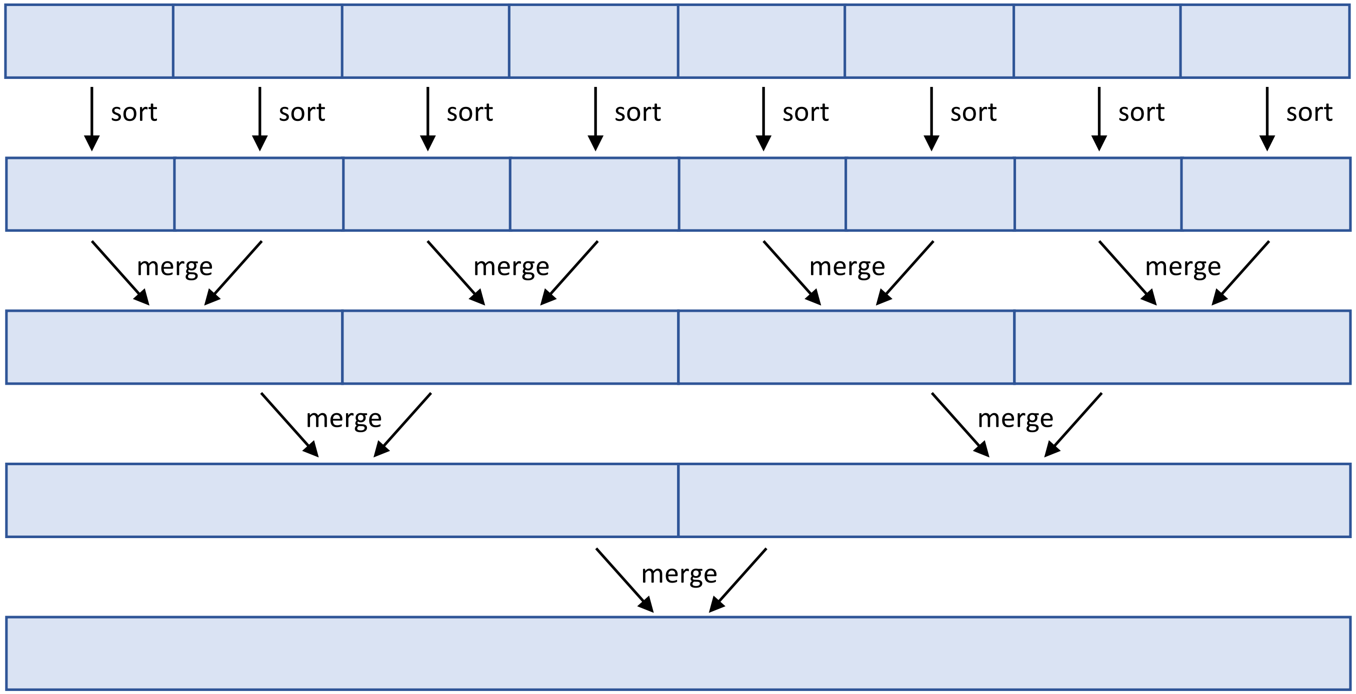
Odd-Even Sort Complexity

- Odd-even sort has a complexity of $O(n^2)$
  - Similar to bubble sort

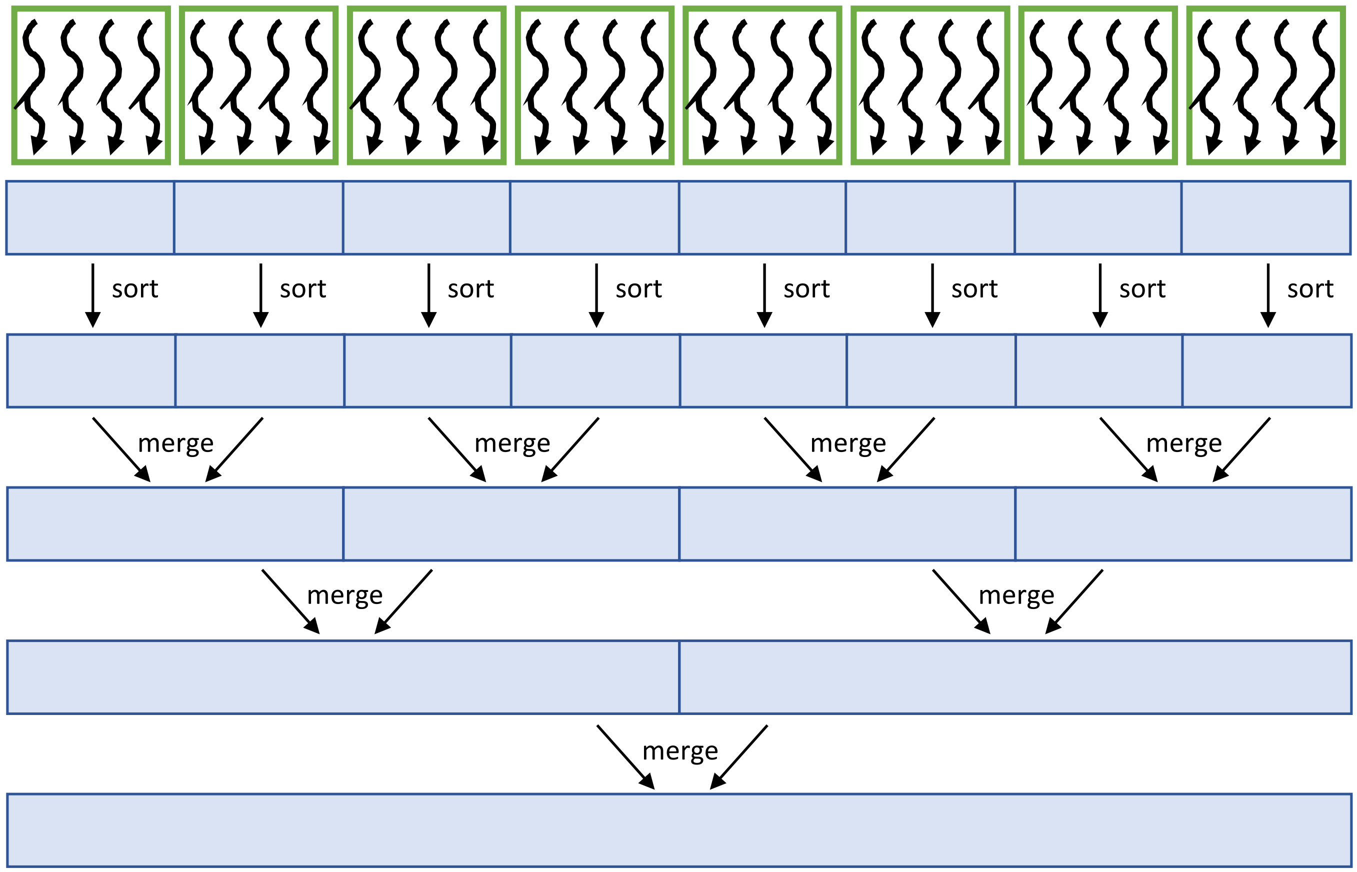
- Better complexity can be achieved by merge sort

Merge Sort

- **Merge sort** divides a list into sub-lists, sorts them, and then repeatedly performs an ordered merge on pairs of sorted sub-lists to get the final list



Parallel Merge Sort



<u>Parallelization approach:</u> Each step performs different ordered merge operations in parallel, and also parallelizes each merge operation (we have already seen how to parallelize ordered merge)

Earlier steps rely more on parallelism across merge operations

Later steps rely more on parallelism within merge operations

Merge Sort Complexity

- Merge sort has a complexity of $O(n \cdot \log n)$ which is the best that can be achieved for a **comparison-based** sorting algorithm
- Better complexity can be achieved by a **non-comparison-based** sorting algorithm such as radix sort

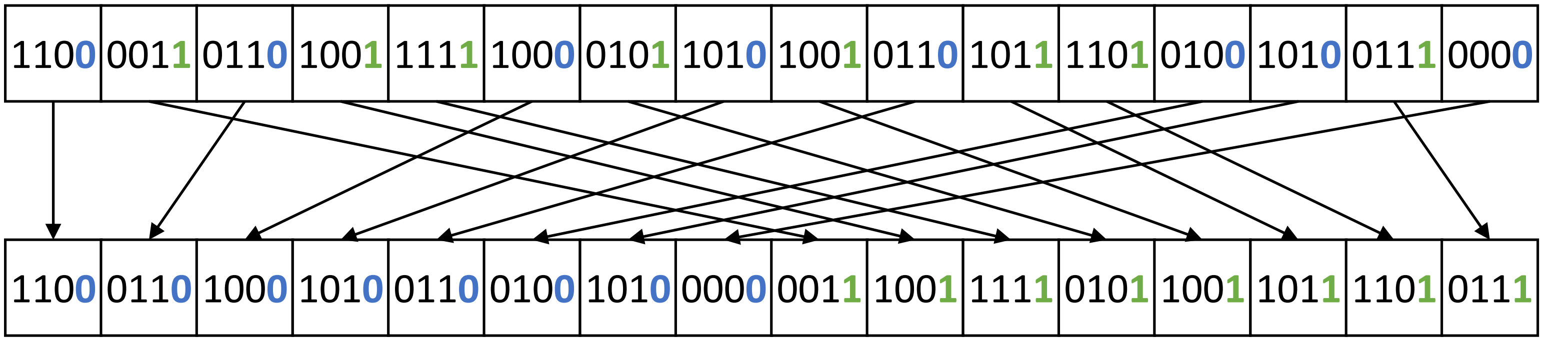
Radix Sort

- **Radix sort** is a sorting algorithm that distributes keys into **buckets** based on a radix (or base)
- Distributing keys into buckets is repeated for each digit, while preserving the order from previous iterations within each bucket
- Using a radix that is a power of two simplifies processing binary numbers
  - Each iteration handles a fixed slice of bits from the key
  - We will start with a radix of two (1 bit) then extend

Radix Sort Example (1-bit Radix)

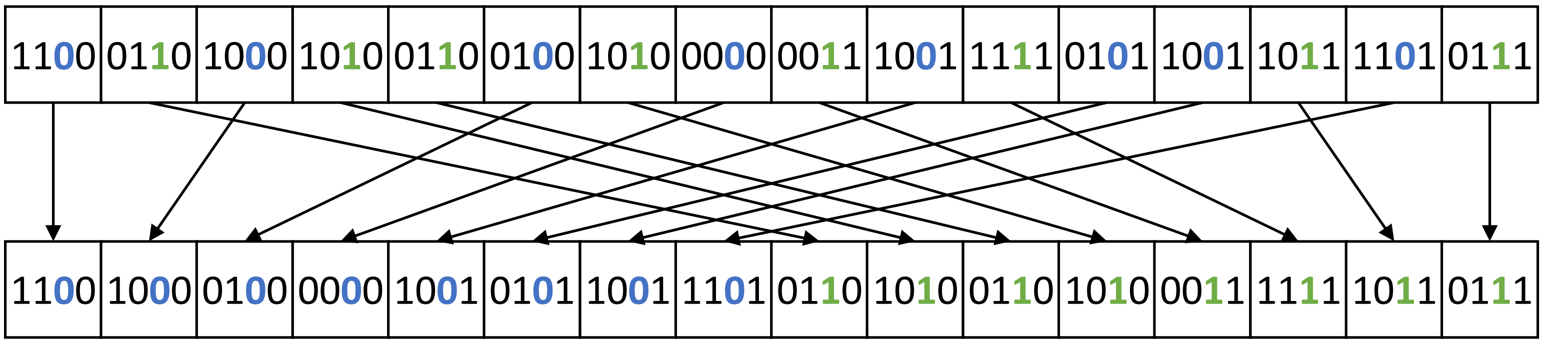
1100	0011	0110	1001	1111	1000	0101	1010	1001	0110	1011	1101	0100	1010	0111	0000

Radix Sort Example (1-bit Radix)



Separate the keys into two buckets based on the first bit

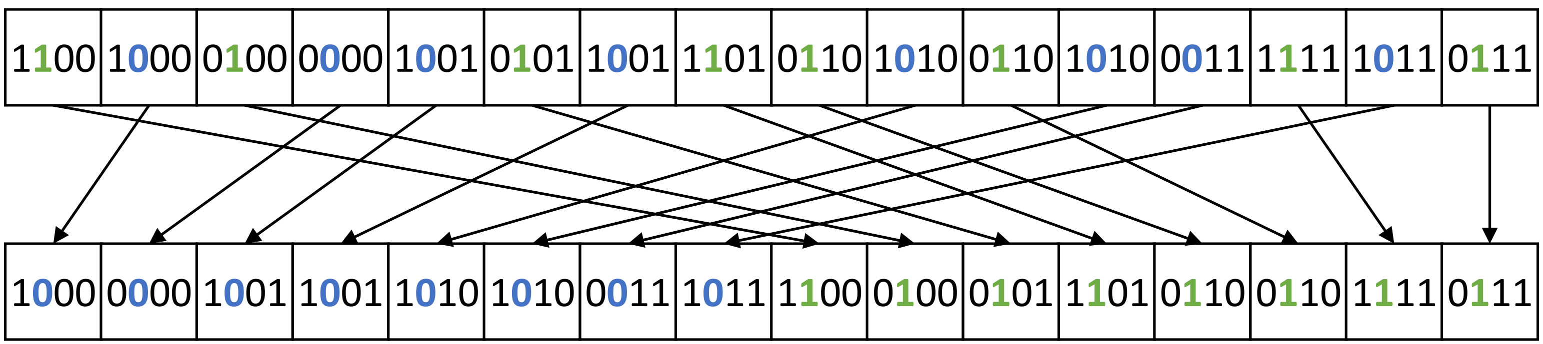
Radix Sort Example (1-bit Radix)



Next, separate the keys based on the second bit

Preserving the order from previous iterations within each bucket ensures that keys are now sorted by the lower two bits

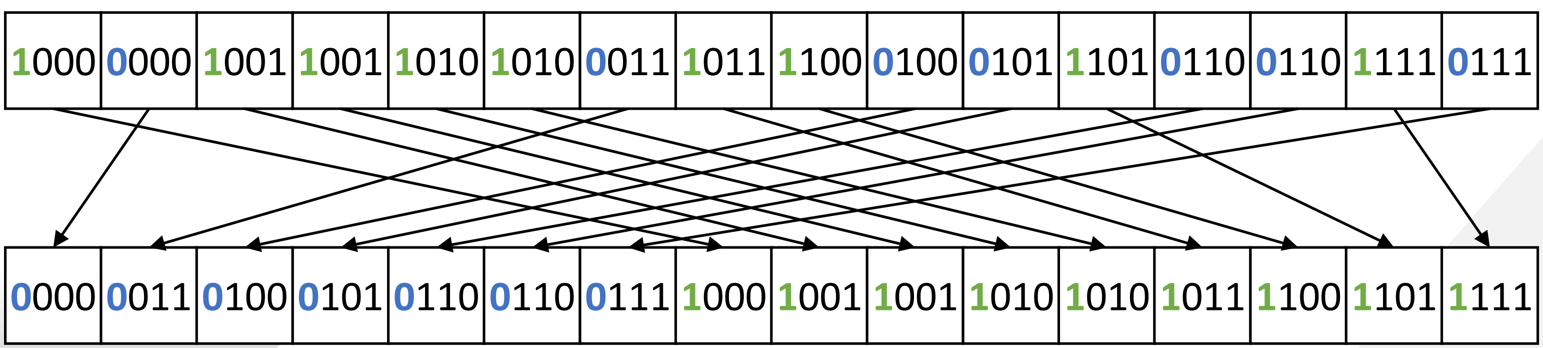
Radix Sort Example (1-bit Radix)



Next, separate the keys based on the third bit

Keys are now sorted by the lower three bits

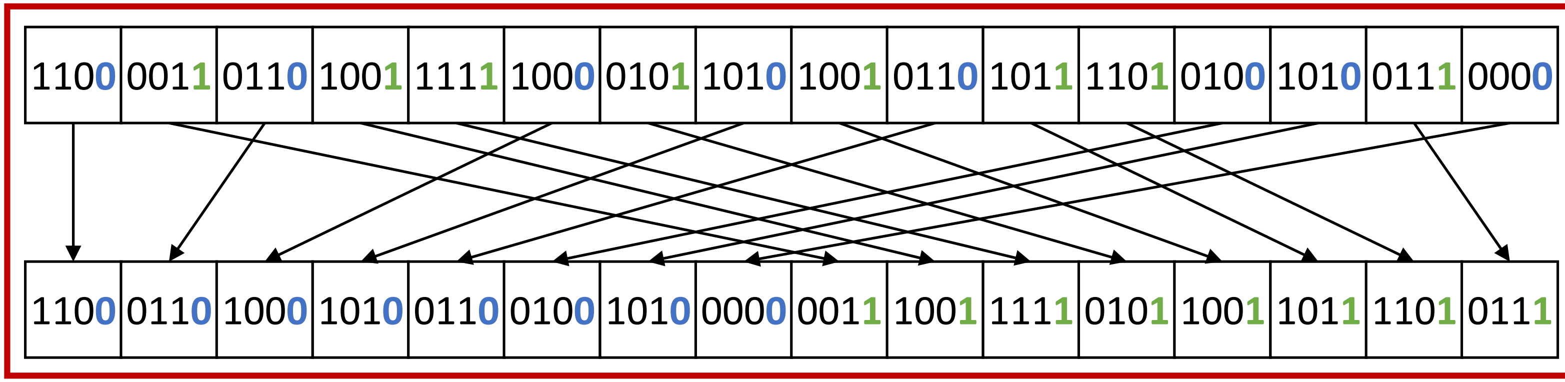
Radix Sort Example (1-bit Radix)



Finally, separate the keys based on the last bit (keys are now sorted by all bits)

Radix Sort Iteration (1-bit Radix)

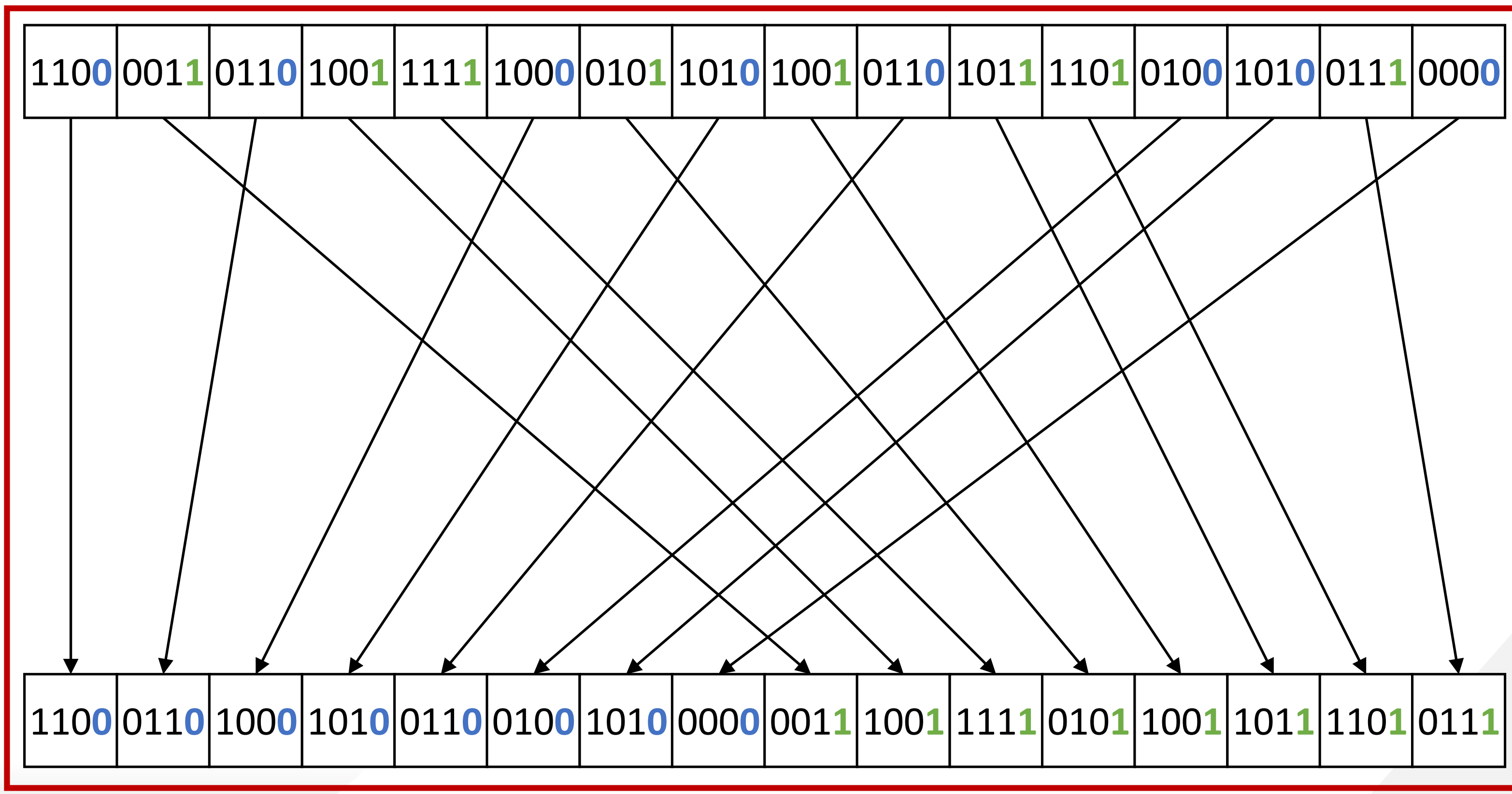
How to separate keys based on one bit?



Radix Sort Iteration (1-bit Radix)

How to find the destination index of each element?

How to separate keys based on one bit?



Finding Destination Index

How to find the destination index of each element?

1100	0011	0110	1001	1111	1000	0101	1010	1001	0110	1011	1101	0100	1010	0111	0000

destination of a **zero** = # **zeros** to the **left**
= # **elements** to the **left** - # **ones** to the **left**
= element index - # **ones** to the **left**

destination of a **one** = # **zeros** in **total** + # **ones** to the **left**
= (# **elements** in **total** - # **ones** in **total**) + # **ones** to the **left**
= input size - # **ones** in **total** + # **ones** to the **left**

Need to find: # **ones** to the **left** of each element
=> use exclusive scan

Finding Destination Index

How to find the destination index of each element?

	1100	0011	0110	1001	1111	1000	0101	1010	1001	0110	1011	1101	0100	1010	0111	0000
is a one?	0	1	0	1	1	0	1	0	1	0	1	1	0	0	1	0
# ones to the left	0	0	1	1	2	3	3	4	4	5	5	6	7	7	7	8
destination	0	8	1	9	10	2	11	3	12	4	13	14	5	6	15	7

extract bit

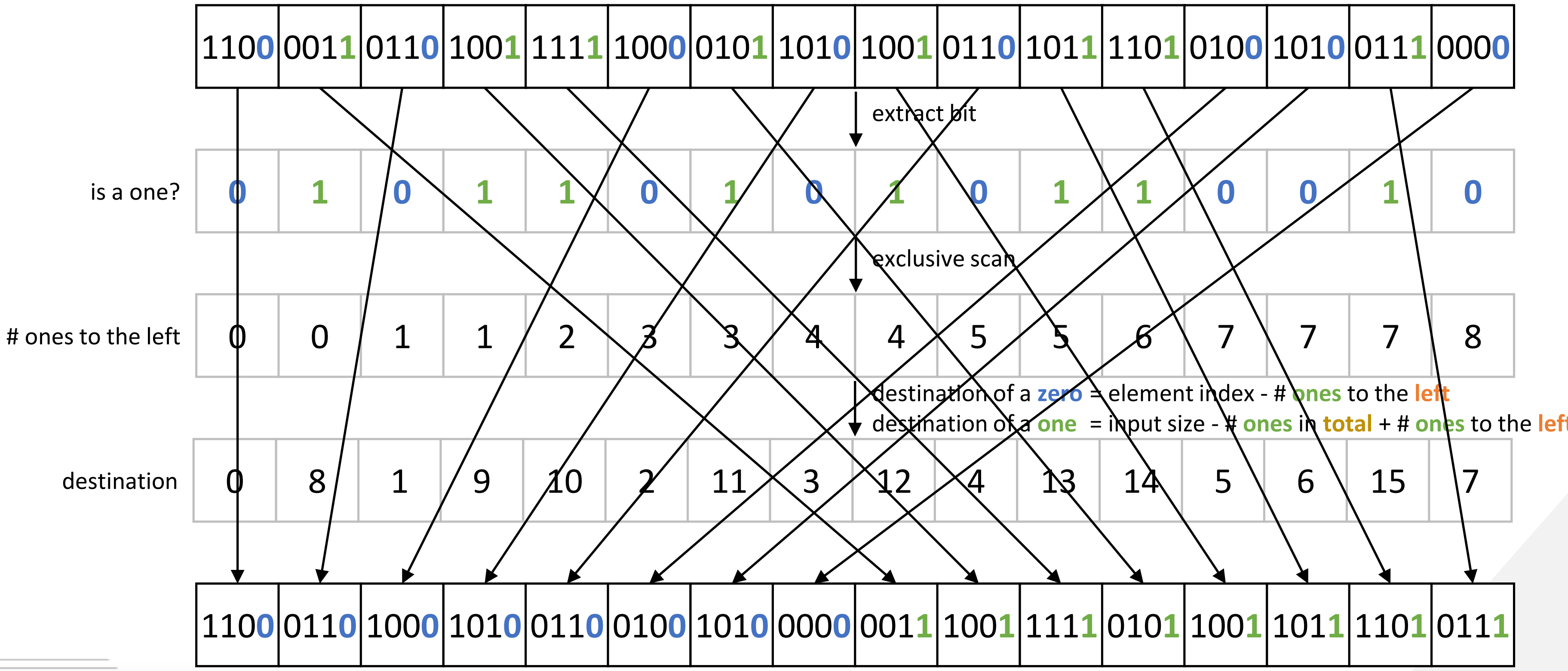
exclusive scan

destination of a zero = element index - # ones to the left

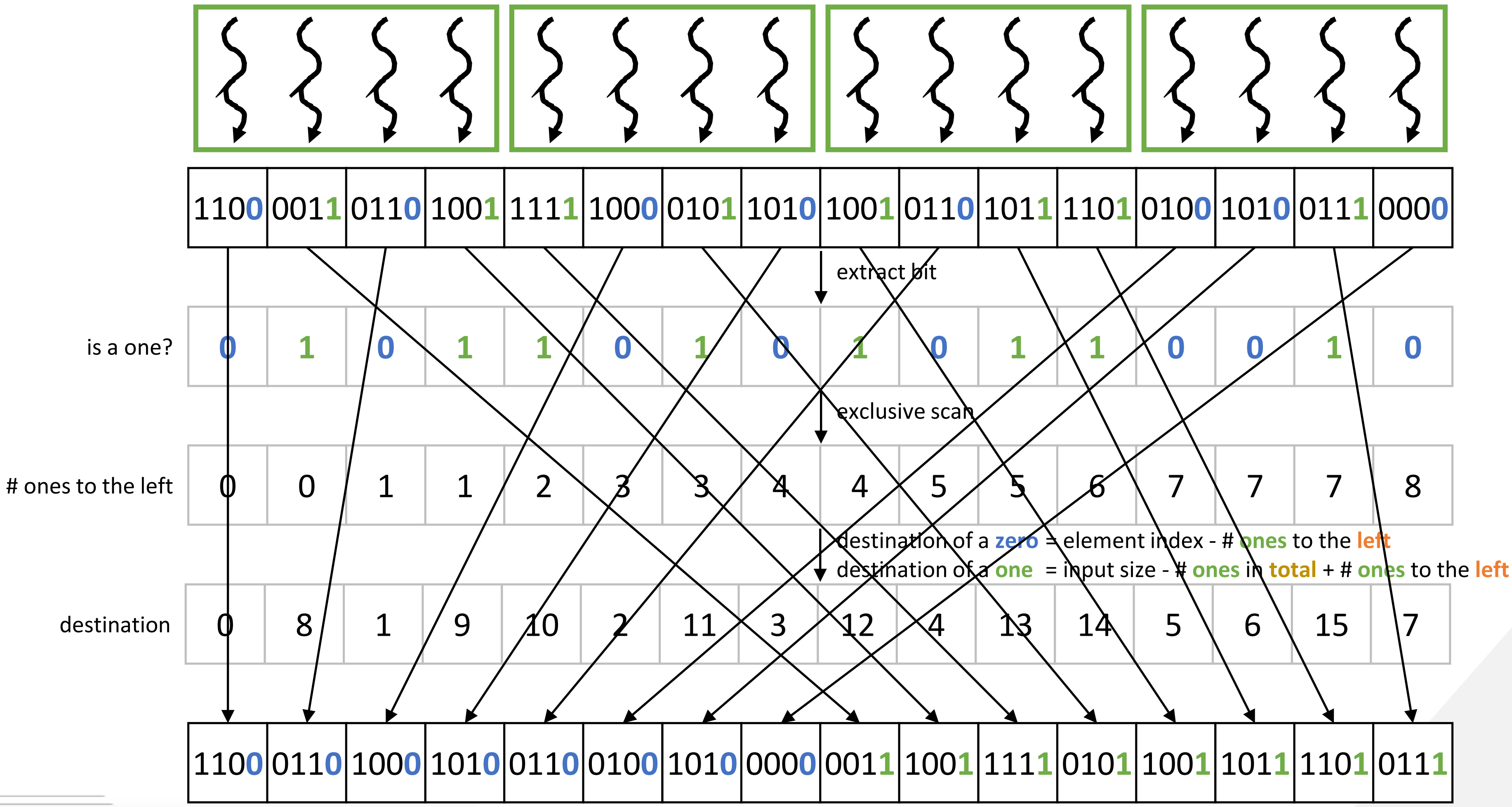
destination of a one = input size - # ones in total + # ones to the left

Finding Destination Index

How to find the destination index of each element?

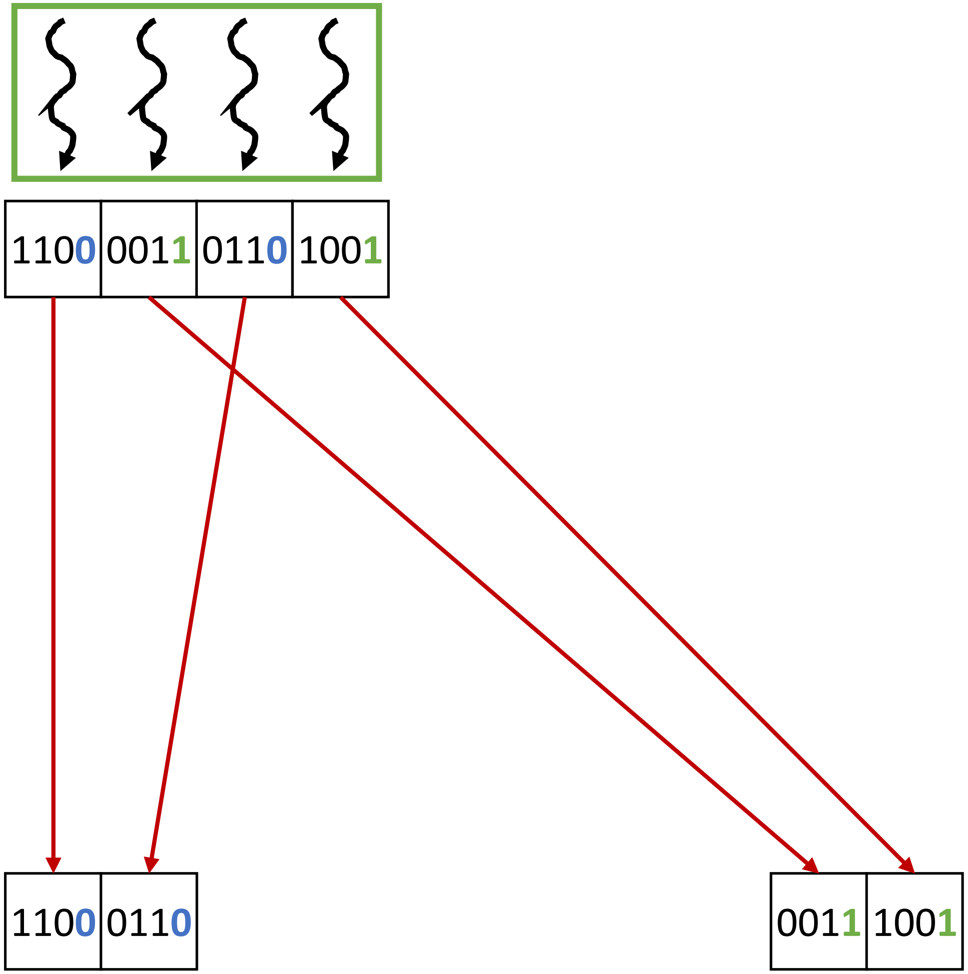


Parallelizing Radix Sort



Parallelization approach: Assign one thread to each element
(we already know how to parallelize scan, the rest is trivial)

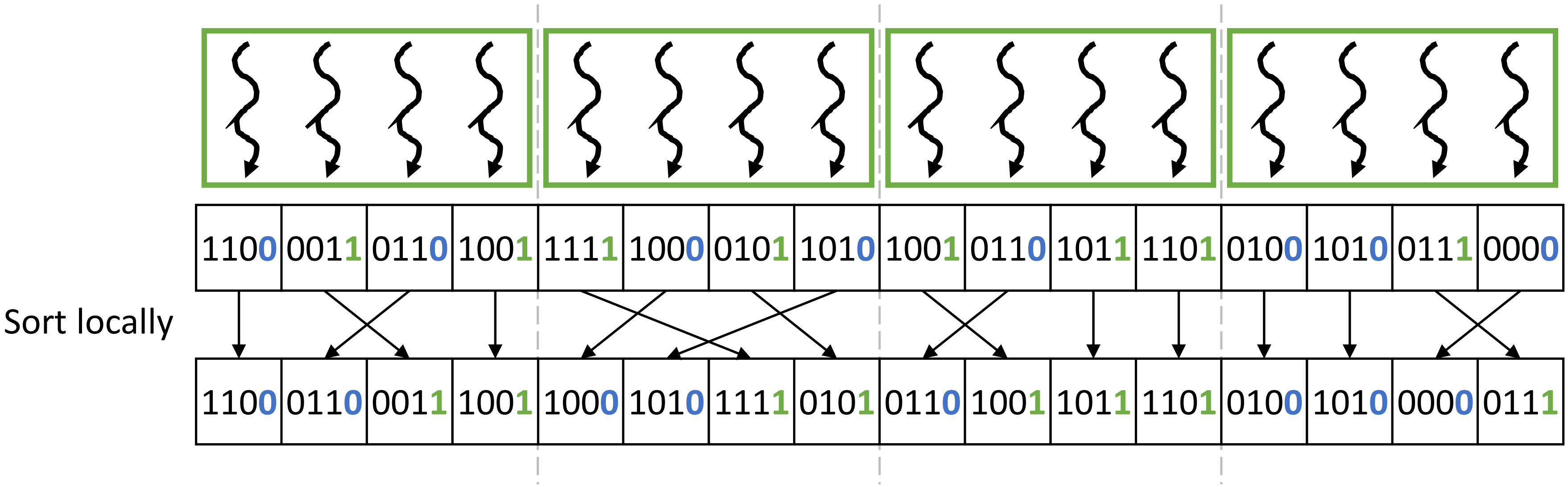
Poor Memory Coalescing



Challenge: Stores are not coalesced (nearby threads write to distant locations in global memory)

Optimization: Sort locally in shared memory, then write each bucket to global memory in a coalesced manner

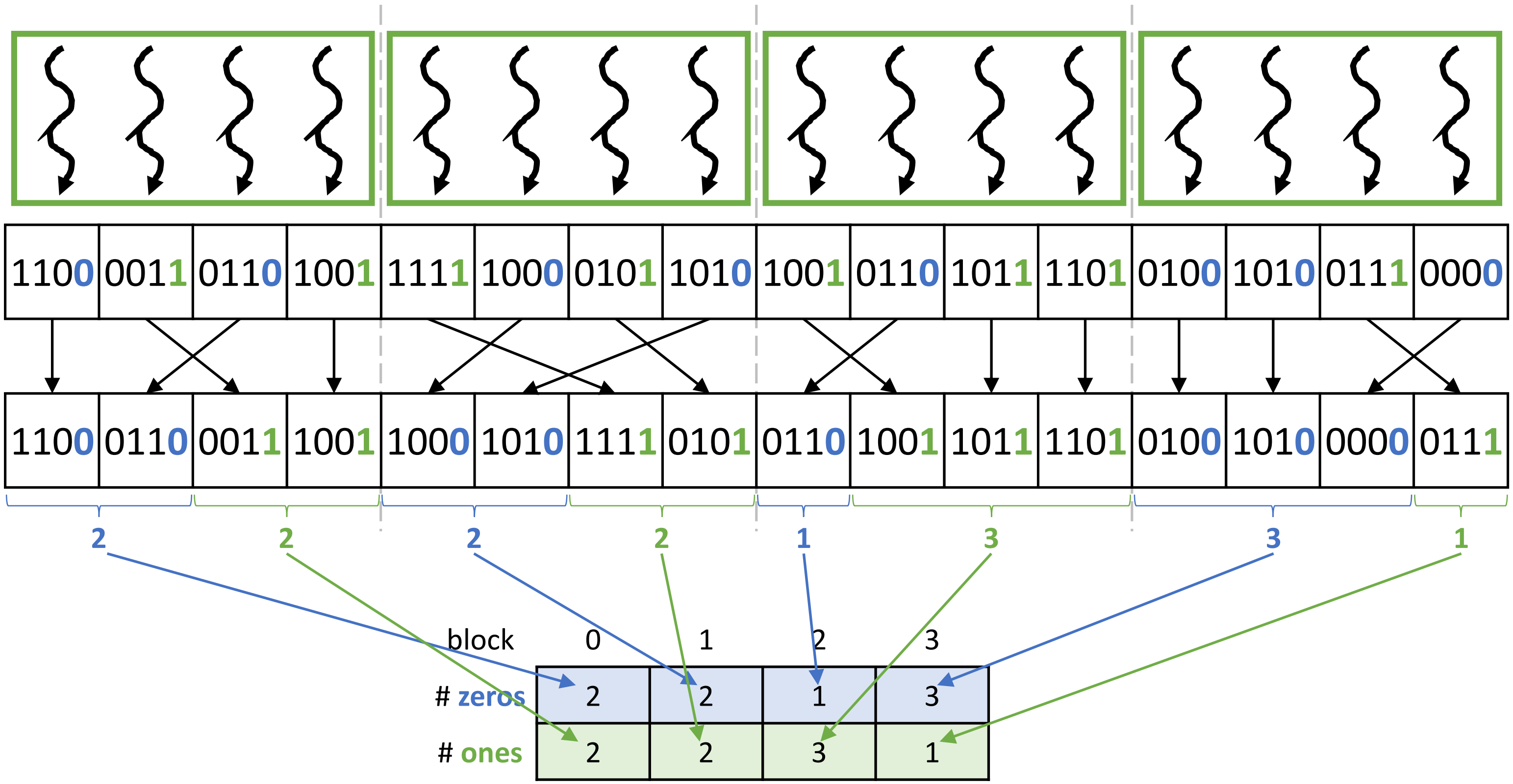
Optimizing Memory Coalescing



Where should each block write each bucket?

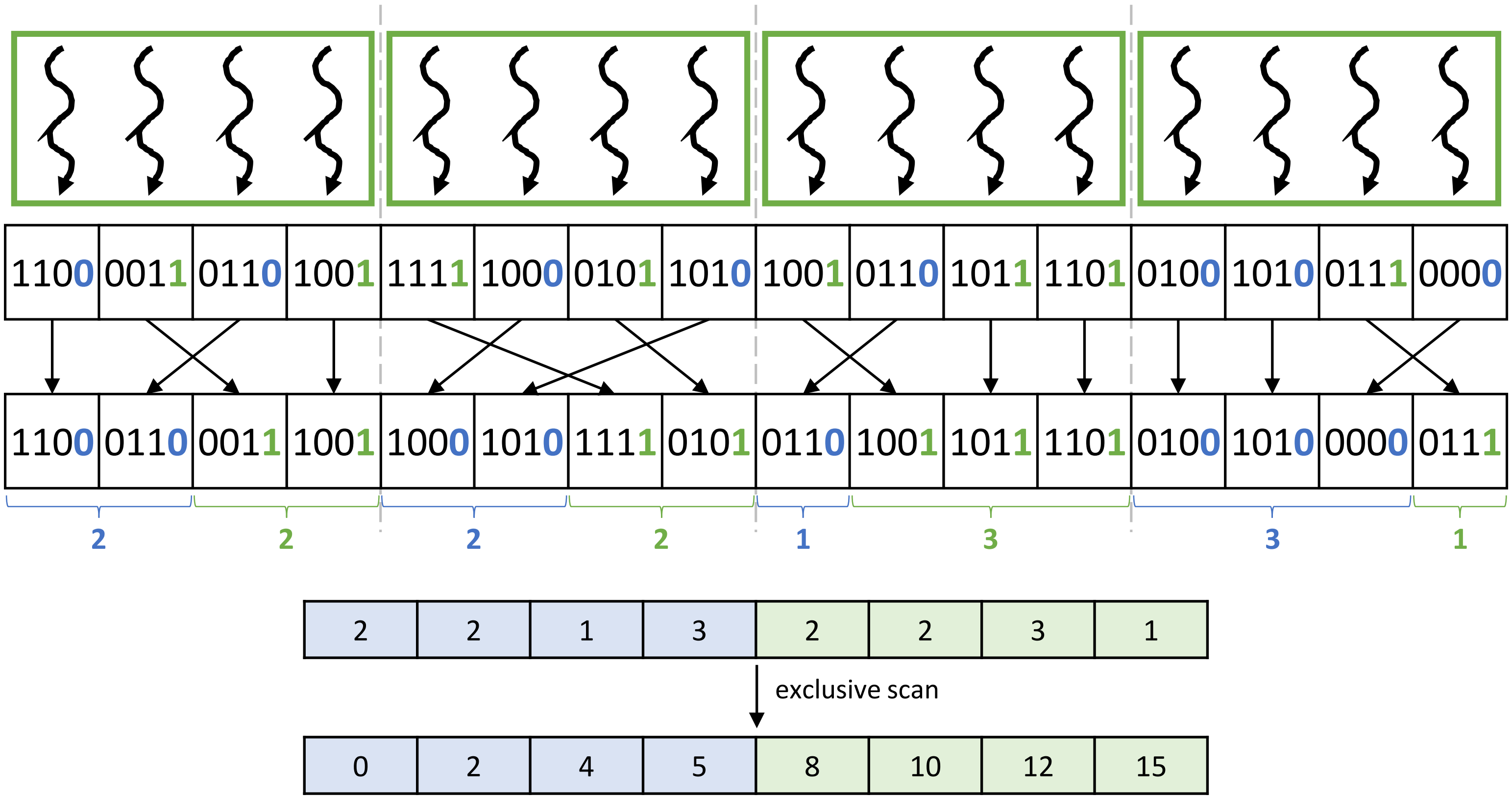
Optimization: Sort locally in shared memory, then write each bucket to global memory in a coalesced manner

Optimizing Memory Coalescing



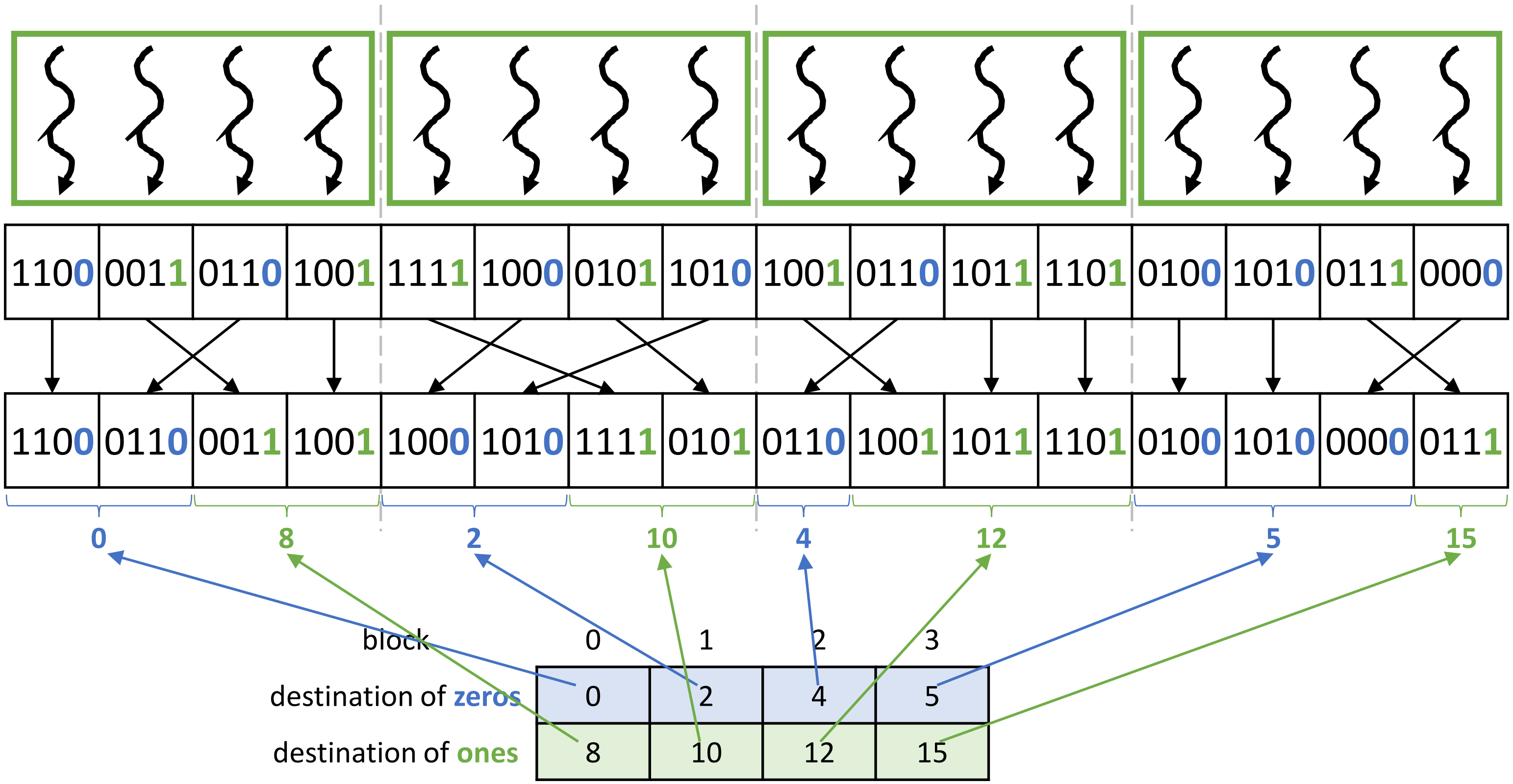
Optimization: Sort locally in shared memory, then write each bucket to global memory in a coalesced manner

Optimizing Memory Coalescing



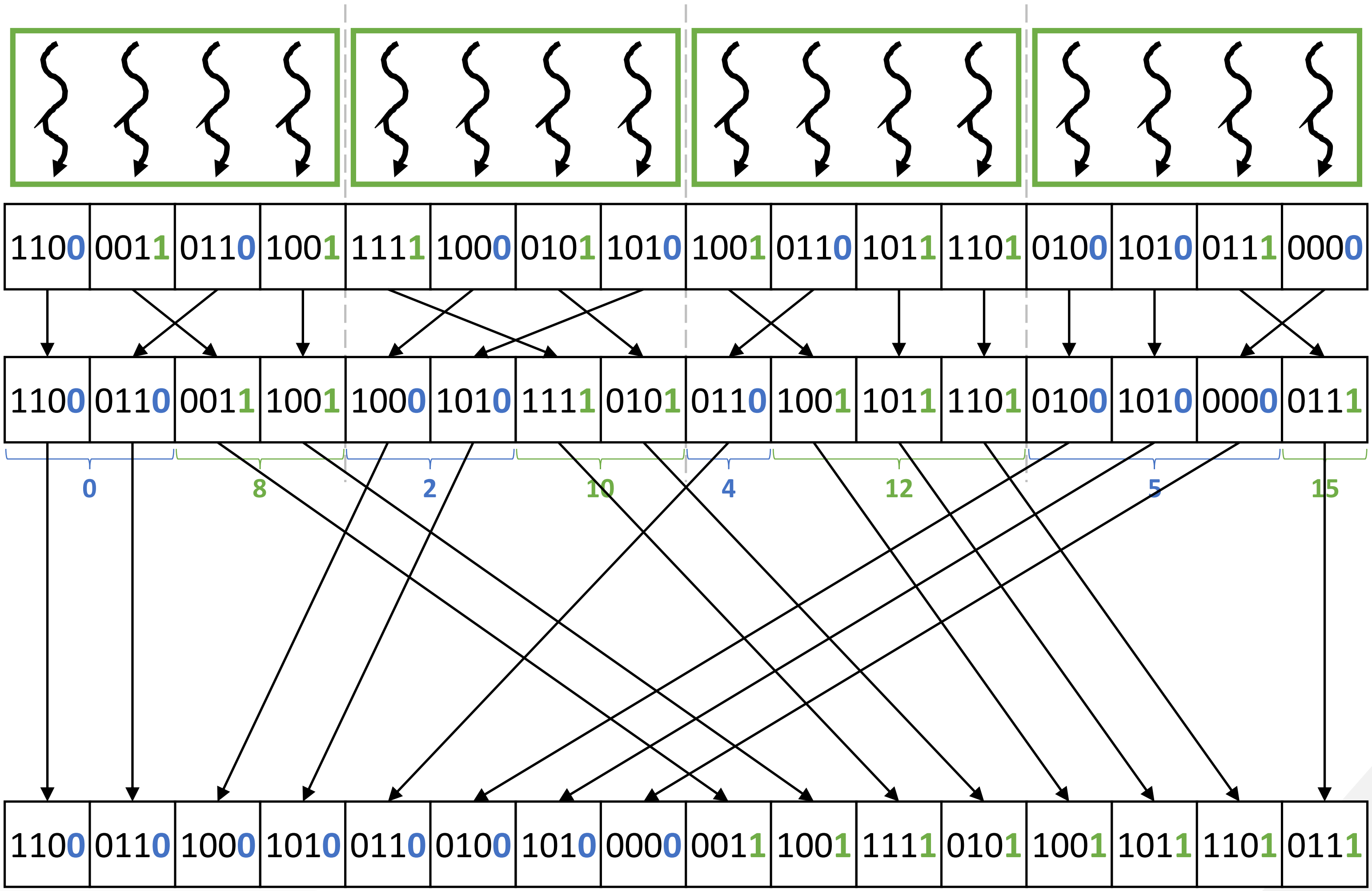
Optimization: Sort locally in shared memory, then write each bucket to global memory in a coalesced manner

Optimizing Memory Coalescing



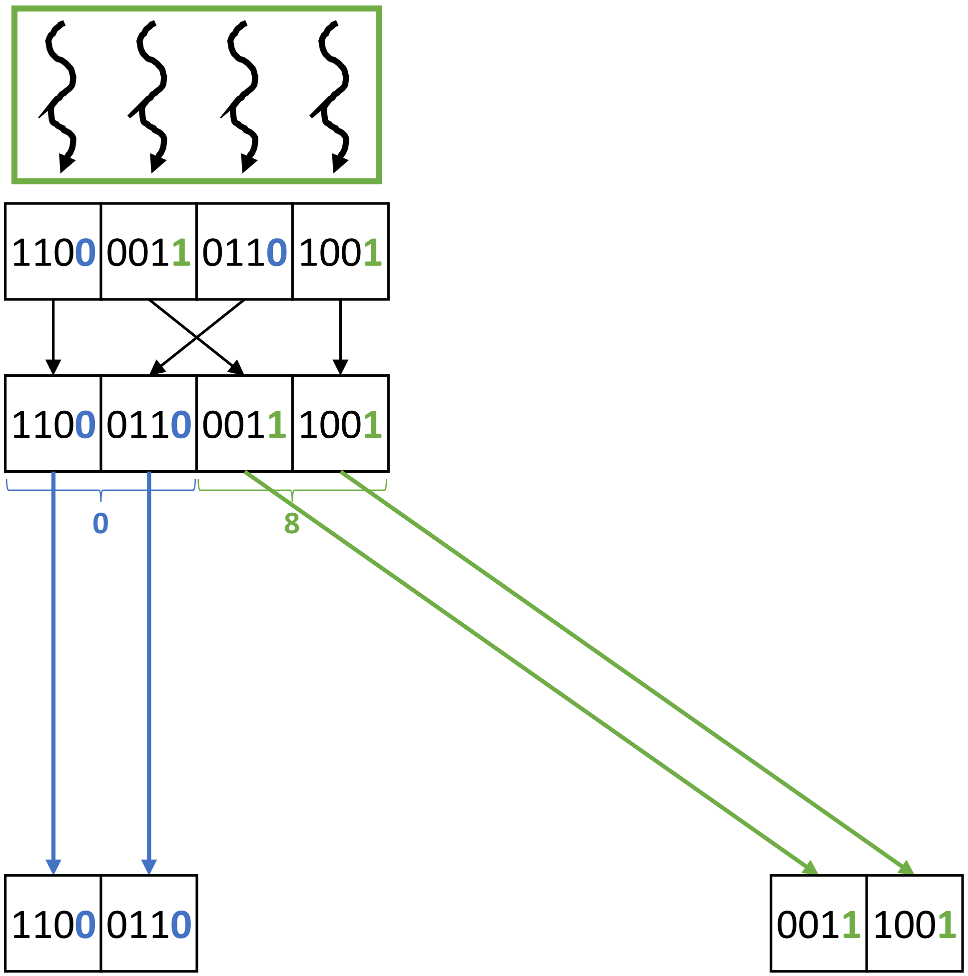
Optimization: Sort locally in shared memory, then write each bucket to global memory in a coalesced manner

Optimizing Memory Coalescing



<u>Optimization:</u> Sort locally in shared memory, then write each bucket to global memory in a coalesced manner

Optimizing Memory Coalescing



Stores are coalesced (nearby threads write to nearby locations in global memory)

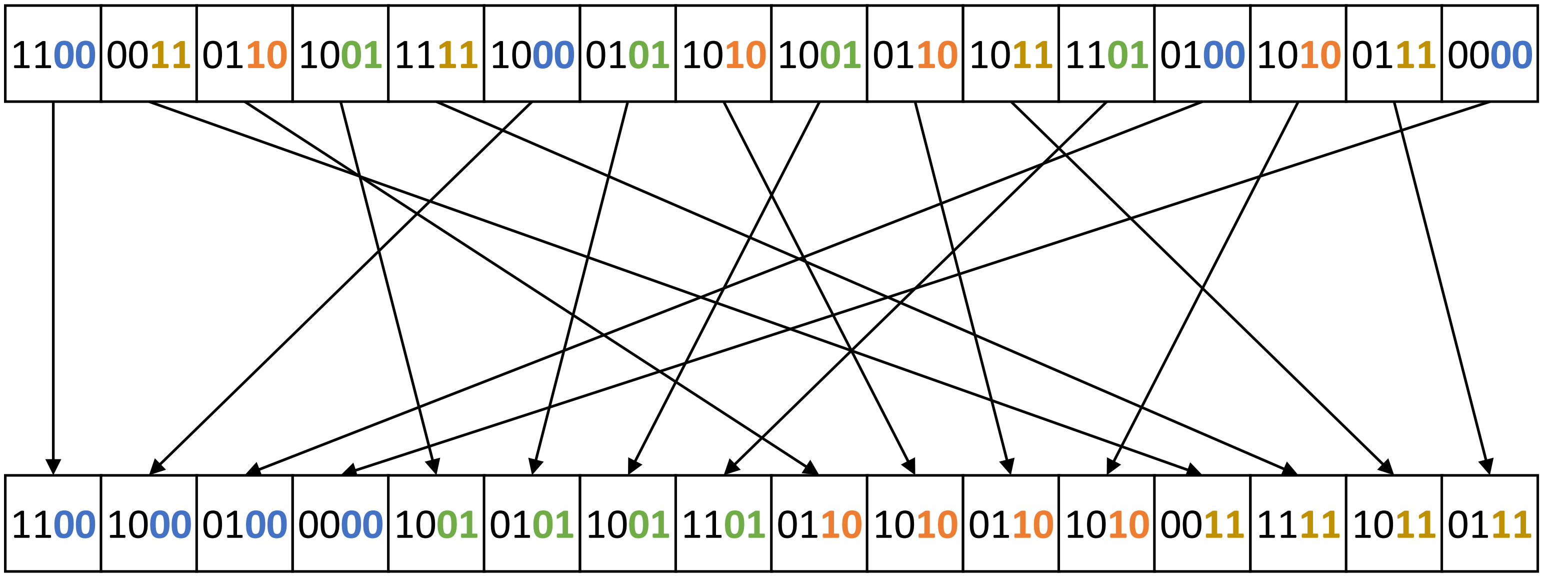
Choice of Radix Value

- So far, a 1-bit radix was used
  - N iterations are needed for keys that are N bits long
- A larger radix can also be used
  - Advantage: fewer iterations
  - Disadvantage: more buckets
    - Results in poorer coalescing (shown later)
- Choice of radix value must balance between the number of iterations and the coalescing behavior

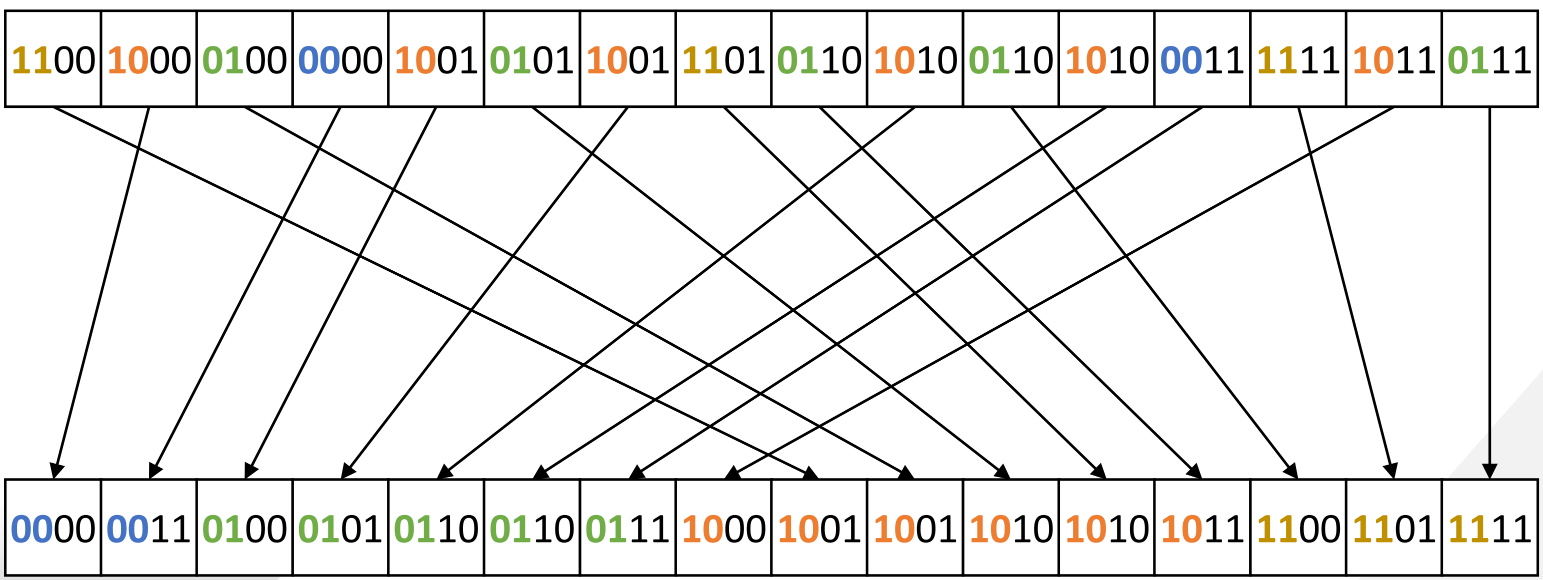
Radix Sort Example (2-bit Radix)

1100	0011	0110	1001	1111	1000	0101	1010	1001	0110	1011	1101	0100	1010	0111	0000

Radix Sort Example (2-bit Radix)

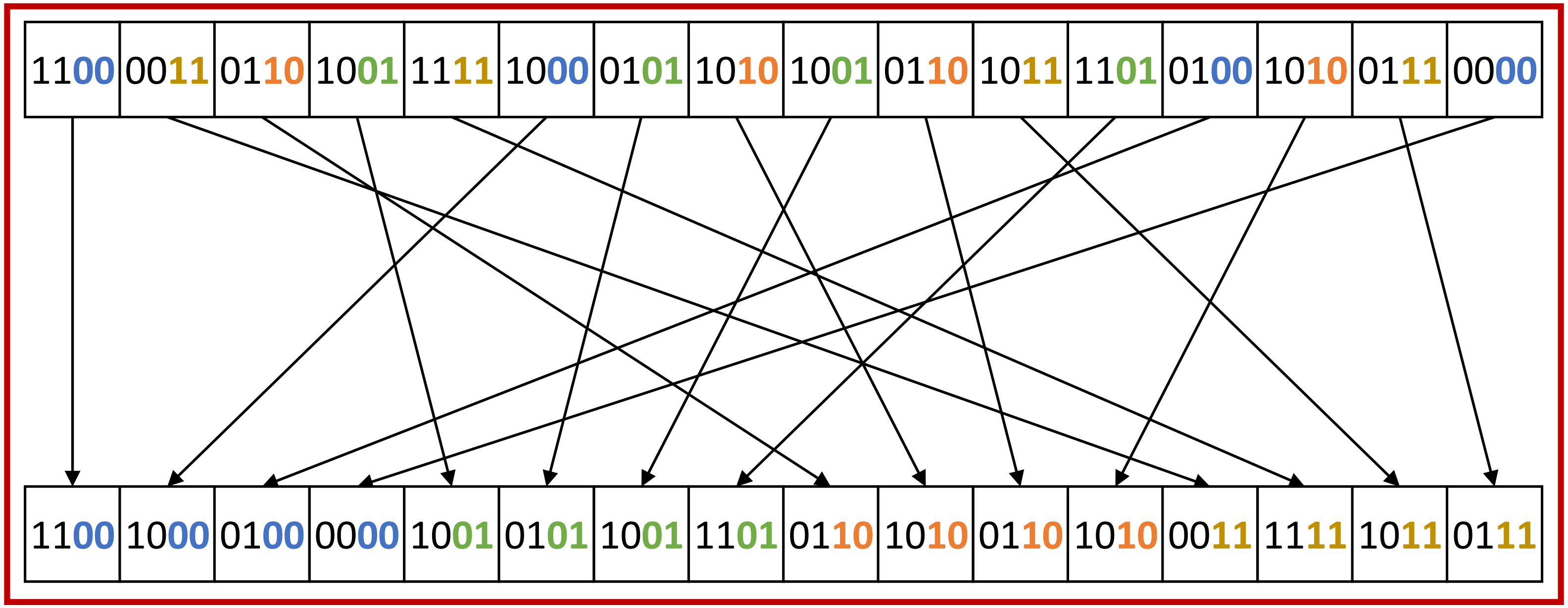


Radix Sort Example (2-bit Radix)



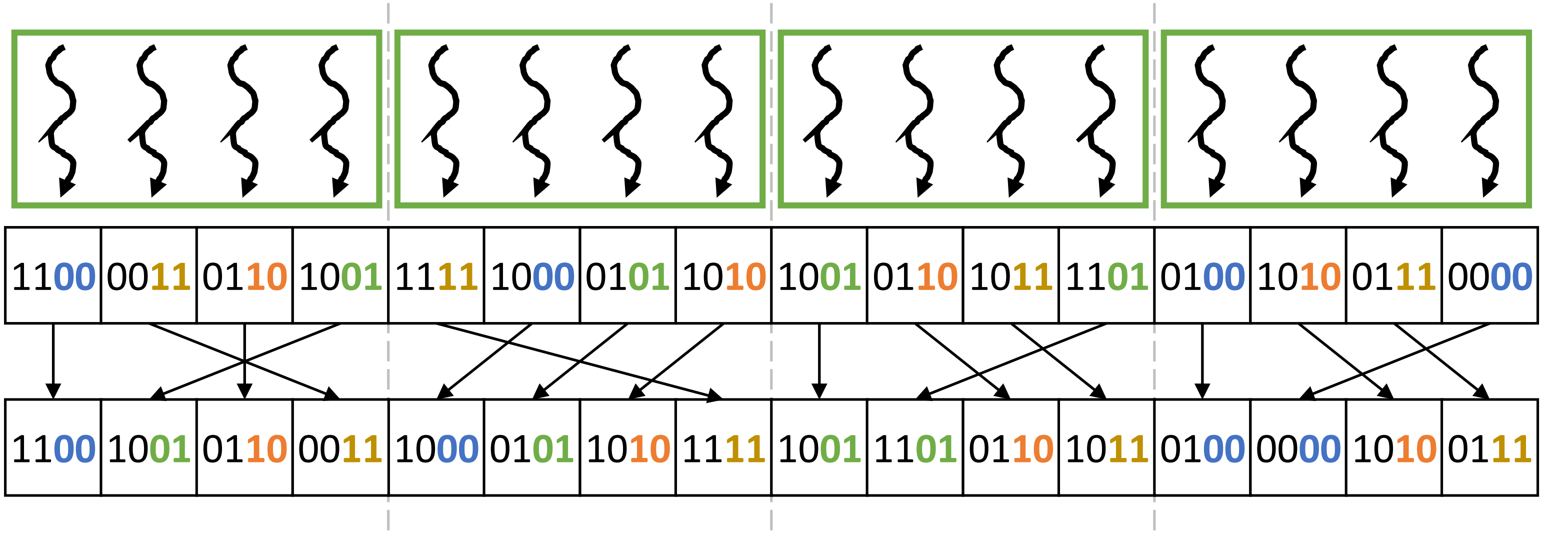
Finish in fewer iterations (for 2-bit radix, need N/2 iterations for N-bit keys)

Radix Sort Iteration (2-bit Radix)



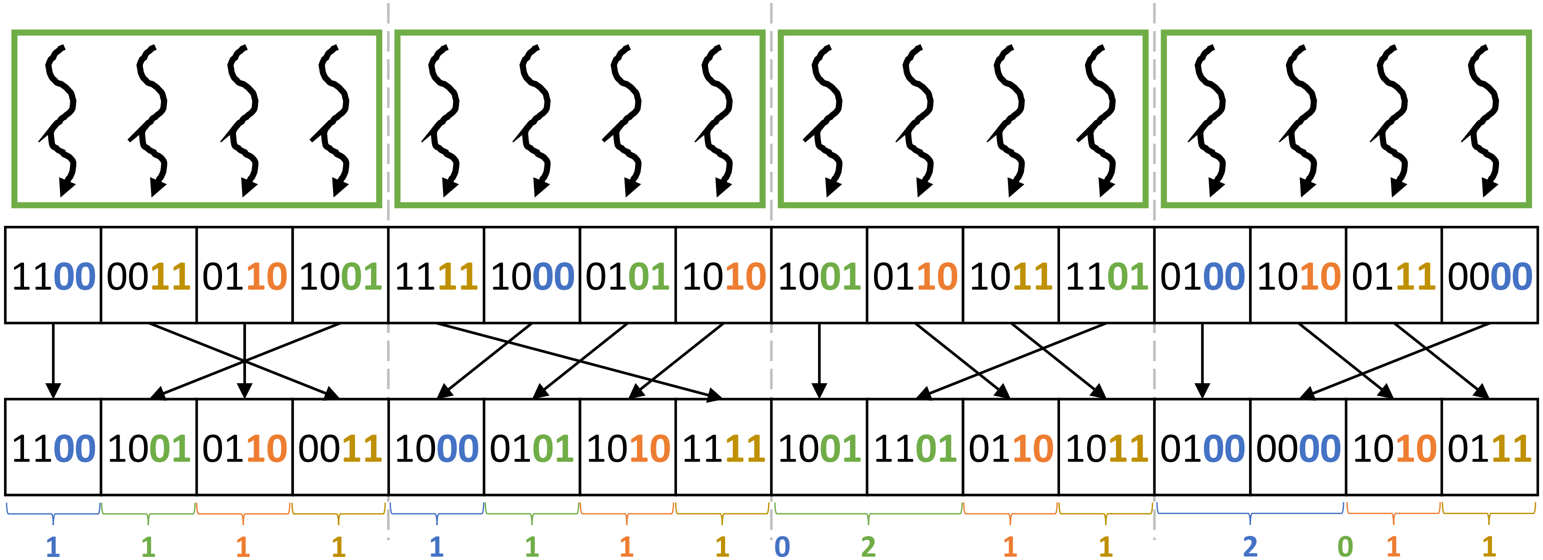
Radix Sort Iteration (2-bit Radix)

Sort locally
(use multiple
1-bit steps)



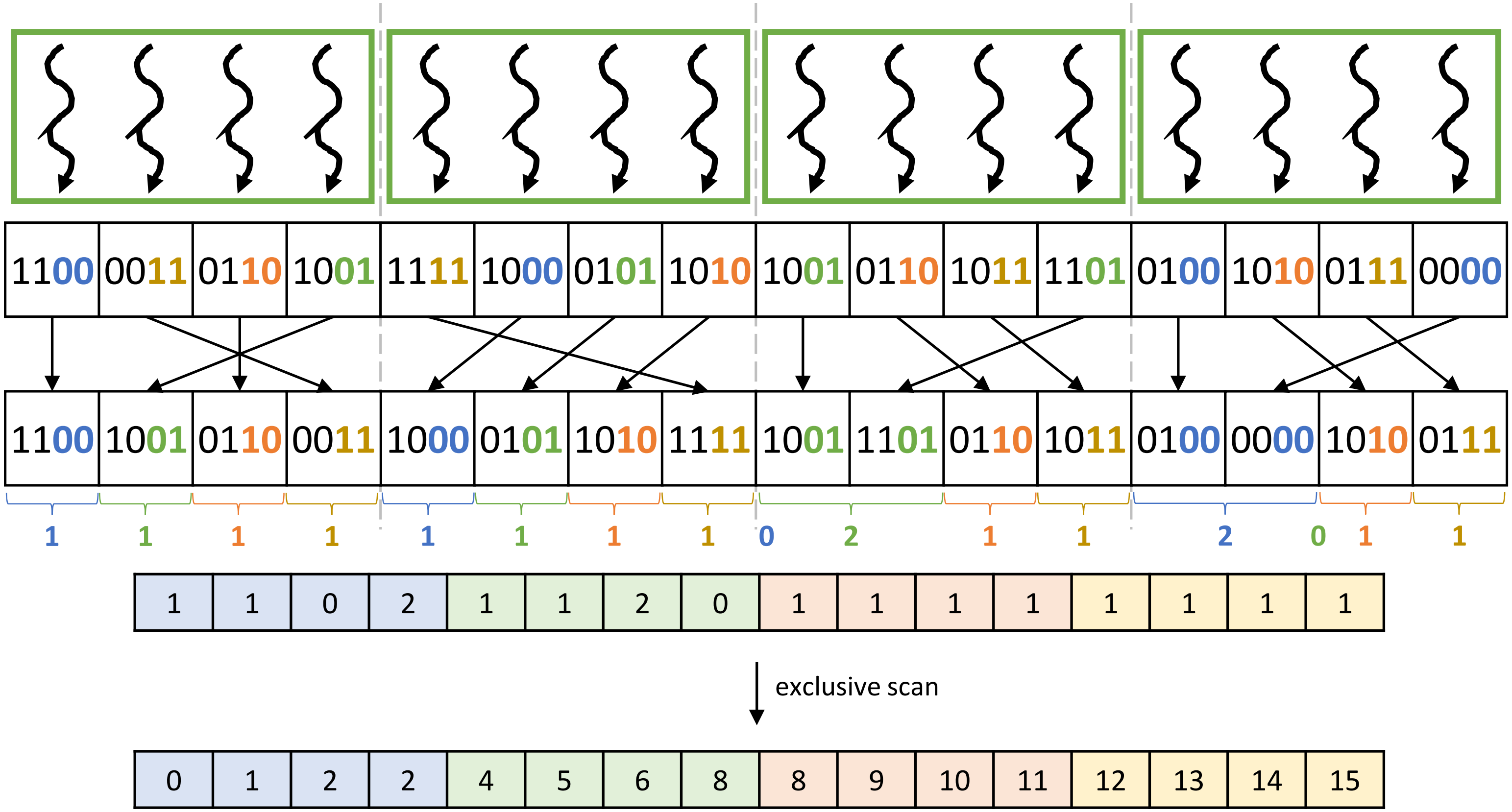
Where should each block write each bucket?

Radix Sort Iteration (2-bit Radix)

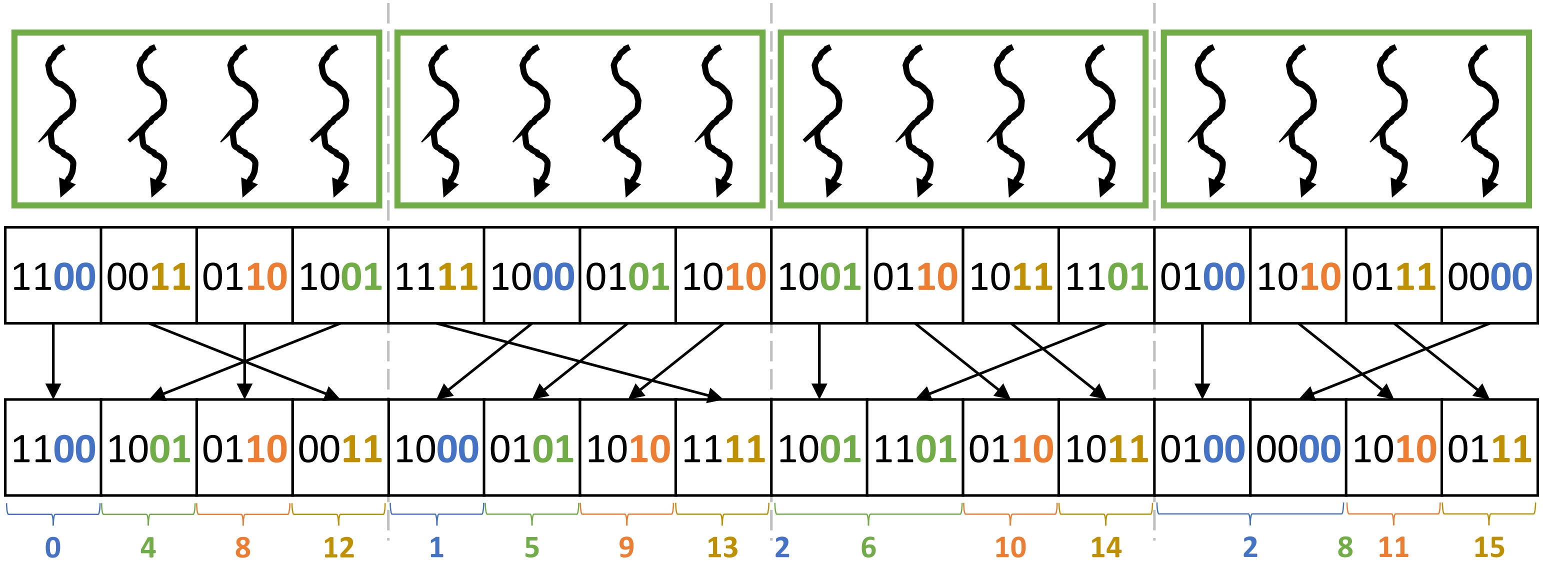


block	0	1	2	3
# 00	1	1	0	2
# 01	1	1	2	0
# 10	1	1	1	1
# 11	1	1	1	1

Radix Sort Iteration (2-bit Radix)

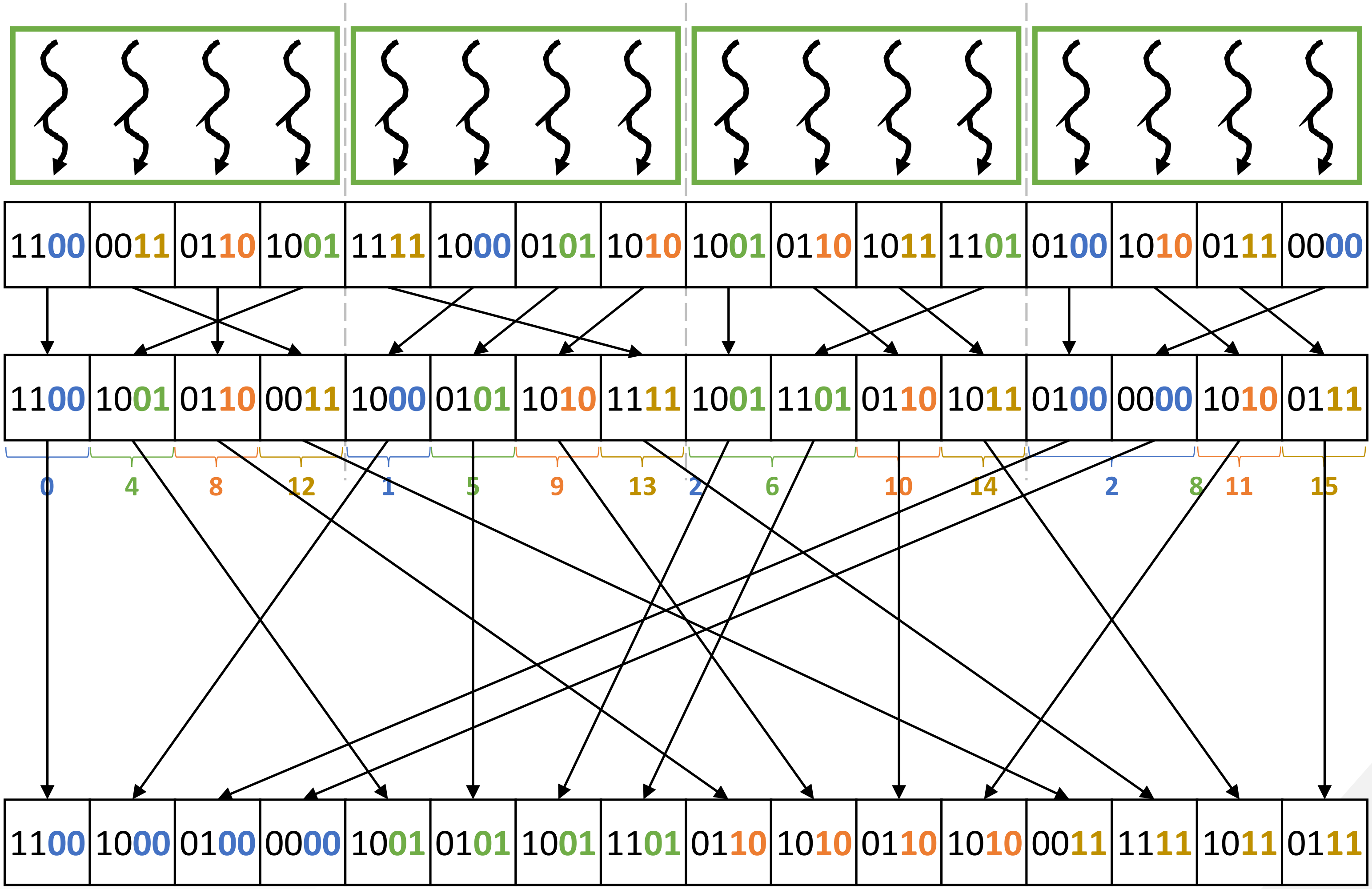


Radix Sort Iteration (2-bit Radix)

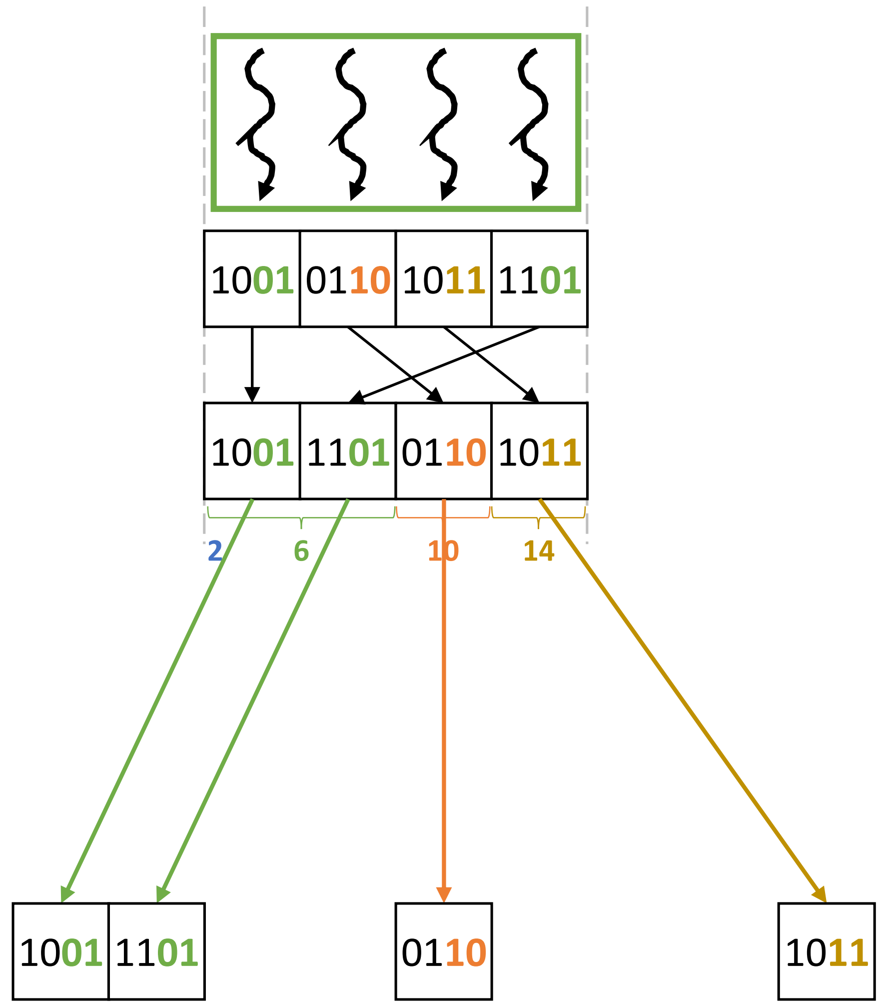


block	0	1	2	3
destination of 00's	0	1	2	2
destination of 01's	4	5	6	8
destination of 10's	8	9	10	11
destination of 11's	12	13	14	15

Radix Sort Iteration (2-bit Radix)



Radix Sort Iteration (2-bit Radix)

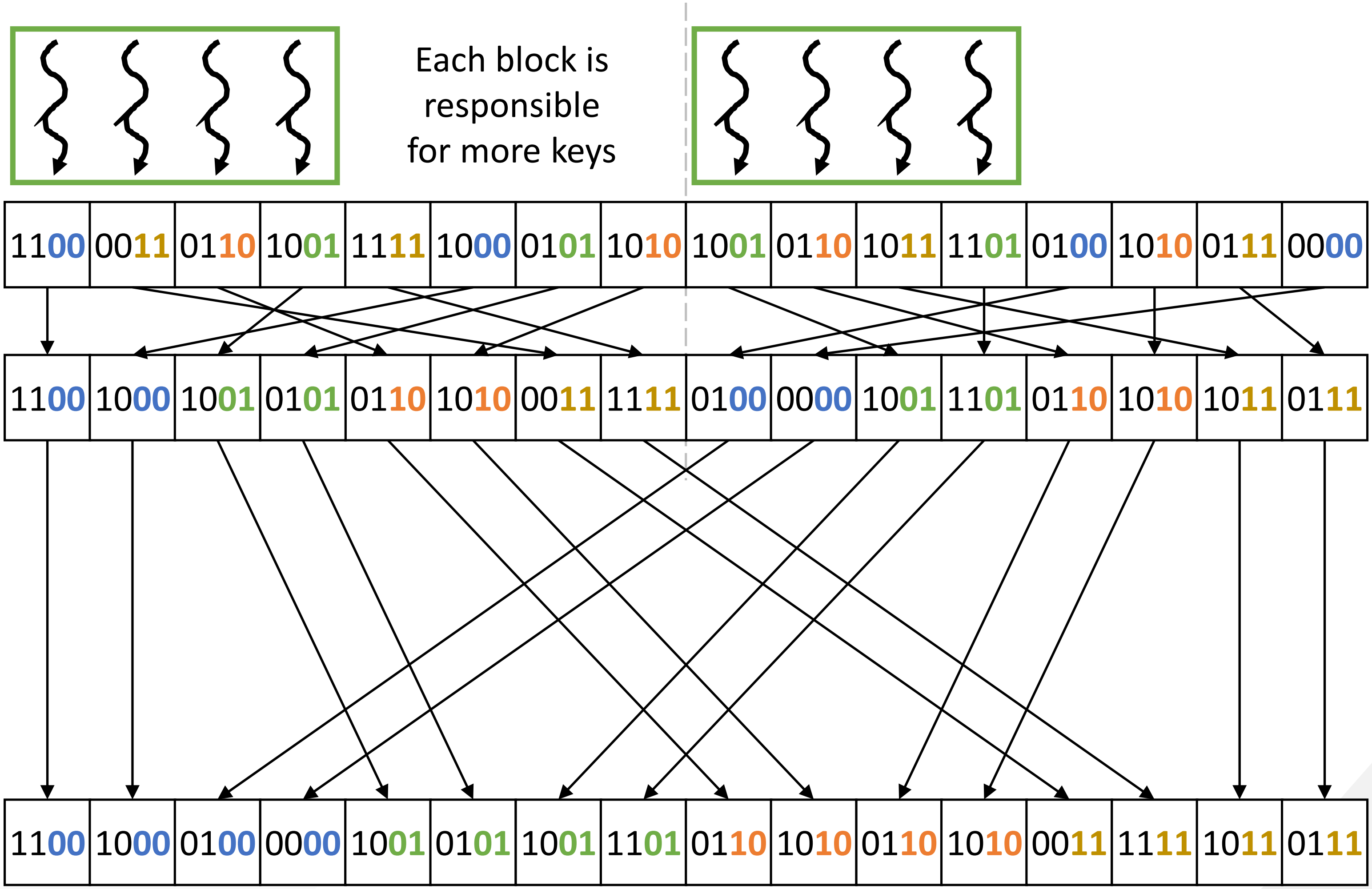


Storing each bucket is coalesced, but there are more buckets, hence poorer coalescing

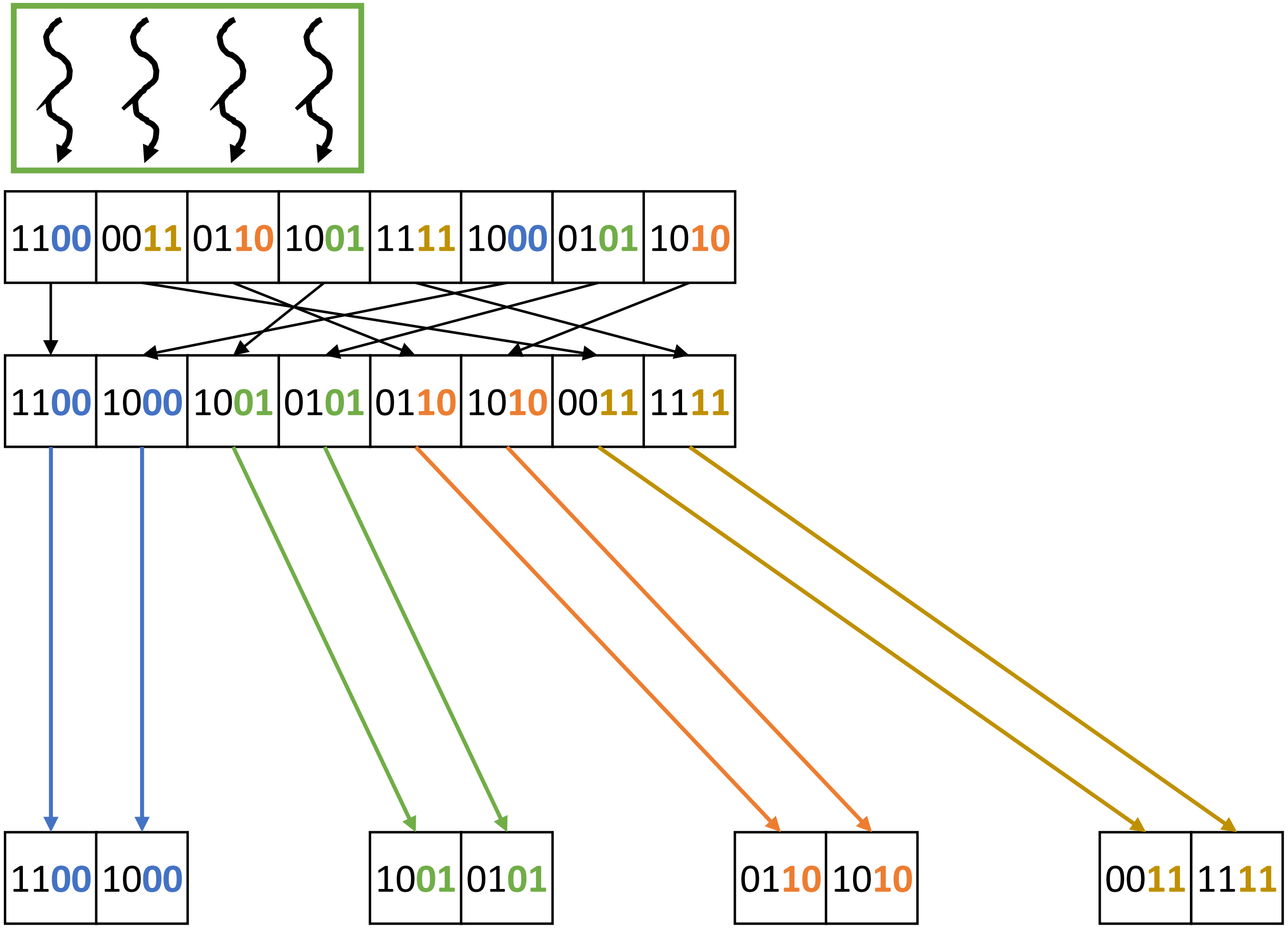
Thread Coarsening

- The price of parallelizing across more blocks is having smaller buckets per block, hence fewer opportunities for coalescing
- Processing more elements per block results in larger buckets per block, hence better coalescing

Radix Sort with Thread Coarsening



Radix Sort with Thread Coarsening



Having larger buckets per block exposes more coalescing opportunities

References

- Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.