IBM Research – Tokyo

THE UNIVERSITY OF TOKYO

# SIMD- and Cache-Friendly Algorithm for Sorting an Array of Structures

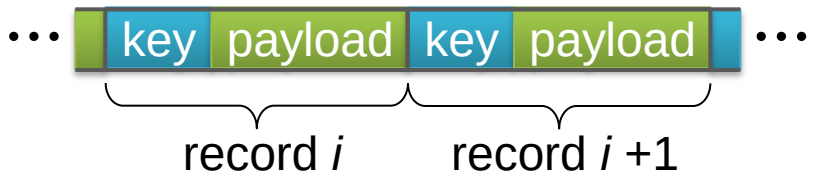

Hiroshi Inoue[†‡] and Kenjiro Taura[‡]
[†] IBM Research – Tokyo
[‡] University of Tokyo

September 3, 2015

# Our Goal

- Develop a fast algorithm for sorting an array of structures

```
struct Record {
    int32_t key;
    int32_t dataX;
    int32_t dataY;      } payload
    ...
};
struct Record array[N];
```




**Our approach:**

- To use SIMD-based multiway mergesort as the basis
- To exploit SIMD instructions to accelerate mergesort kernel
- To avoid random memory accesses that cause excessive cache misses

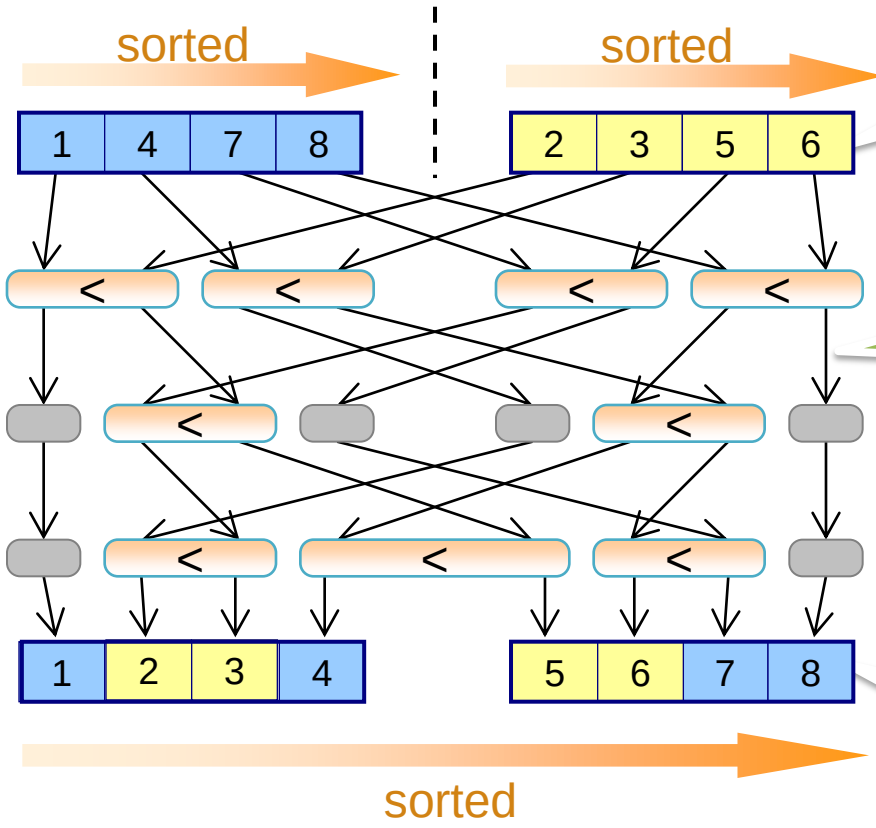
# Outline

- ✓Goal
- Background: SIMD multiway mergesort for integers
- Performance problems in existing approaches
- Our approach
- Performance results

# Background: Mergesort with SIMD for sorting integers [Inoue *et al.* PACT 2007, Chhugani *et al.* PVLDB 2008 etc.]

- Merge operation for 32-bit or 64-bit integers can be efficiently implemented with SIMD by:
  - merging multiple values in vector registers using SIMD min and max instructions (i.e. without conditional branches)
  - integrating the in-register vector merge into usual comparison-based merge operation
- SIMD min and max instructions can accelerate sorting by
  - parallelizing comparisons
  - avoiding unpredictable conditional branches

# Background: SIMD-based merge for values in two vector registers



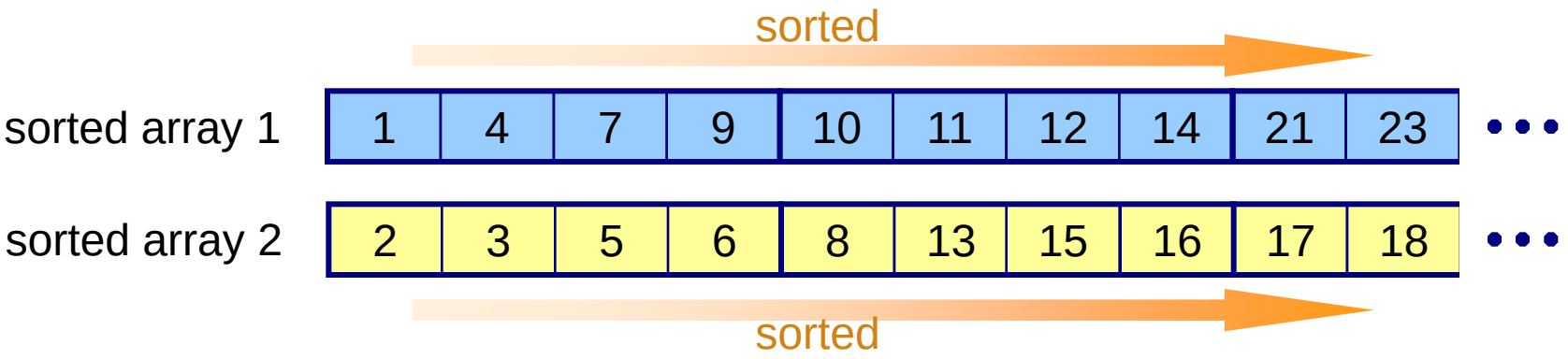

**Input**
two vector registers contain four presorted values in each

**SIMD merging**
one SIMD comparison and "shuffle" operations for each stage *without conditional branch*

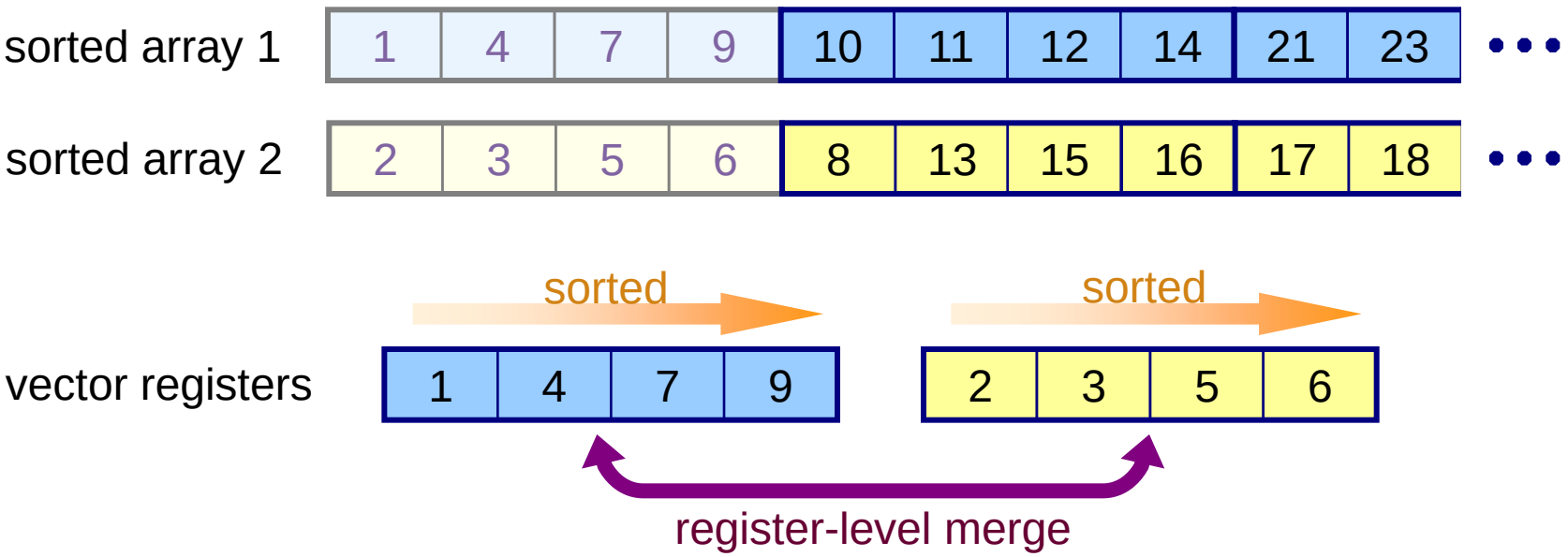
**Output**
eight values in two vector registered are now sorted

(example of odd-even merge)

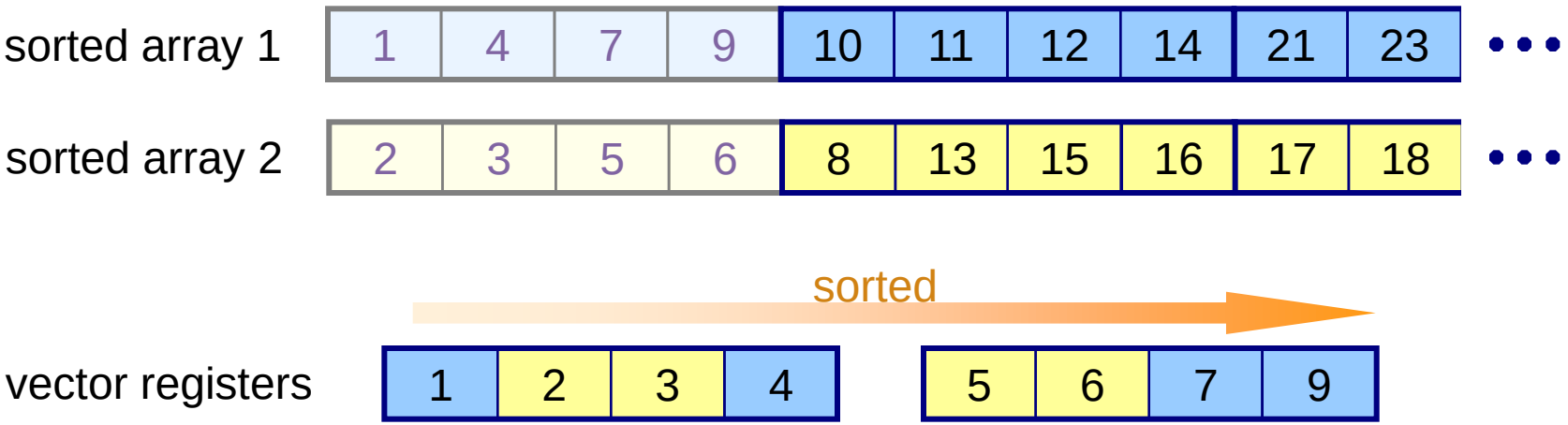
# Background: Integrating Odd-even merge into usual merge



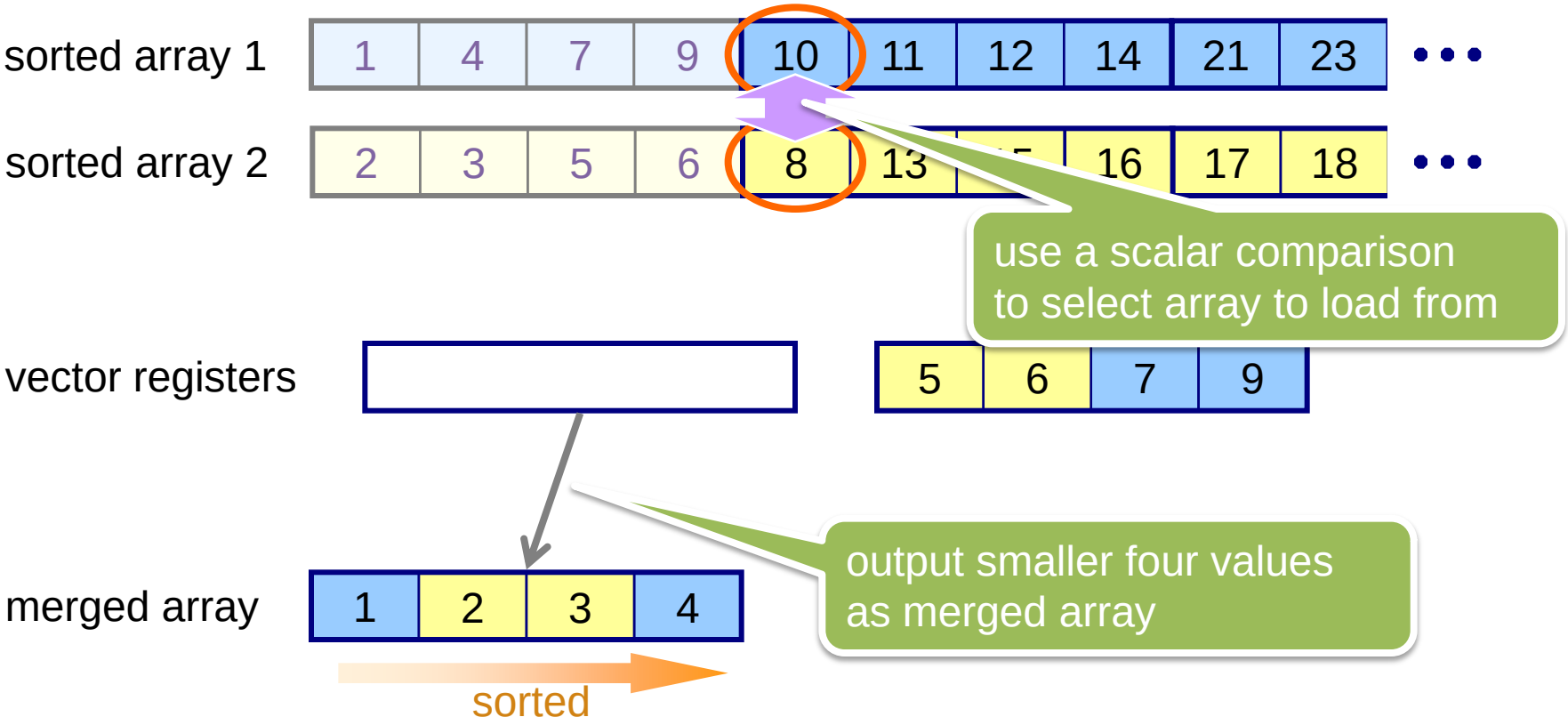
# Background: Integrating Odd-even merge into usual merge



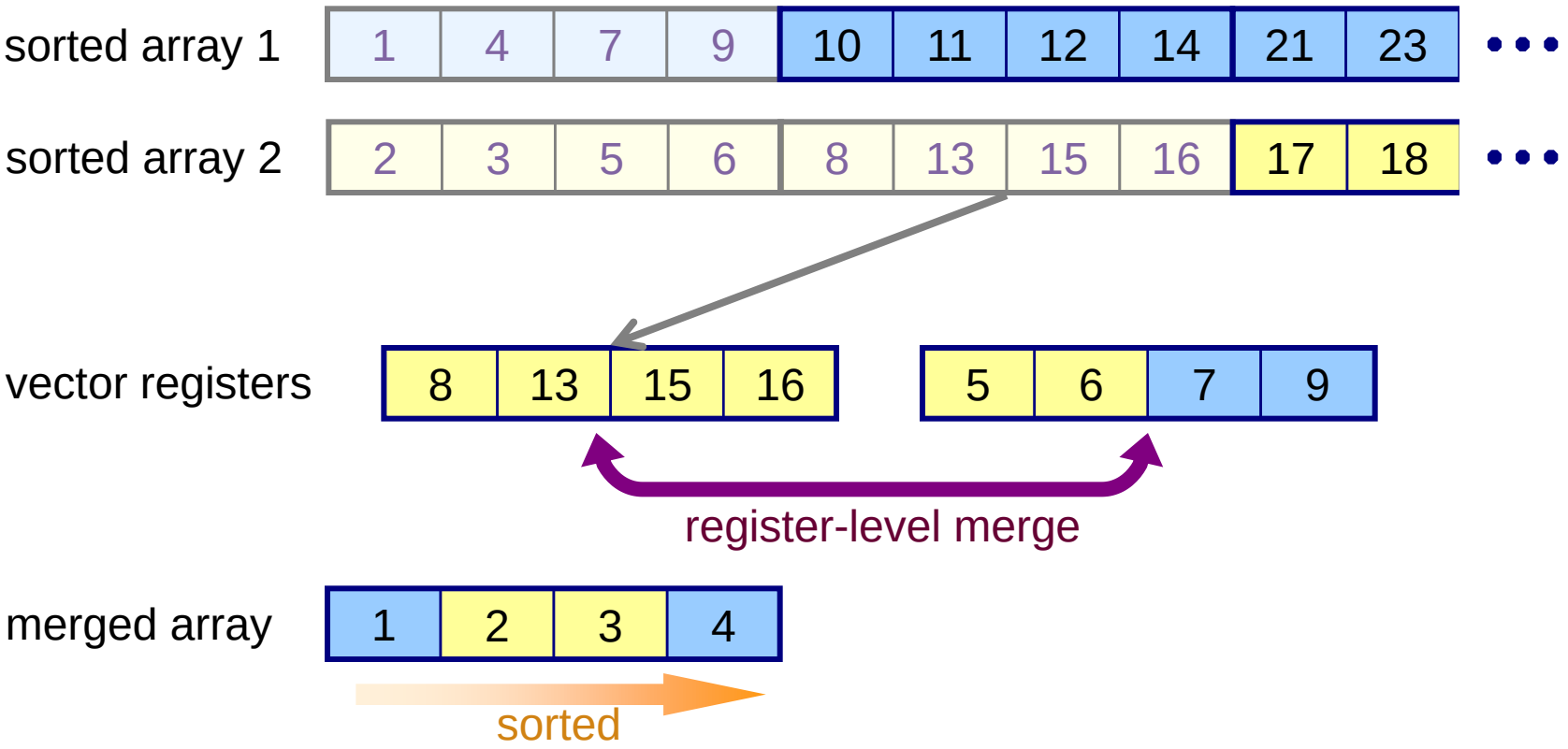
# Background: Integrating Odd-even merge into usual merge



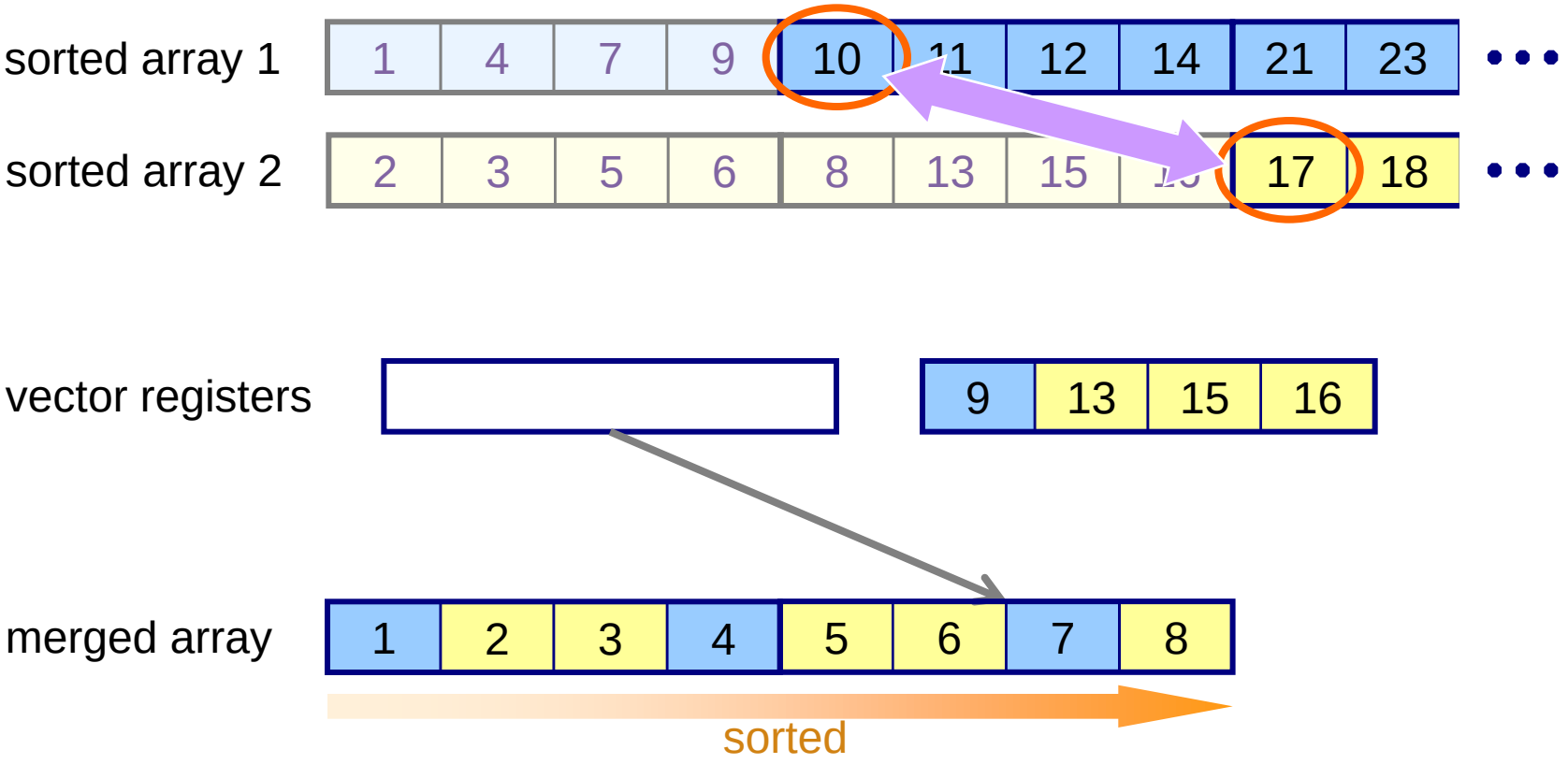
# Background: Integrating Odd-even merge into usual merge



# Background: Integrating Odd-even merge into usual merge

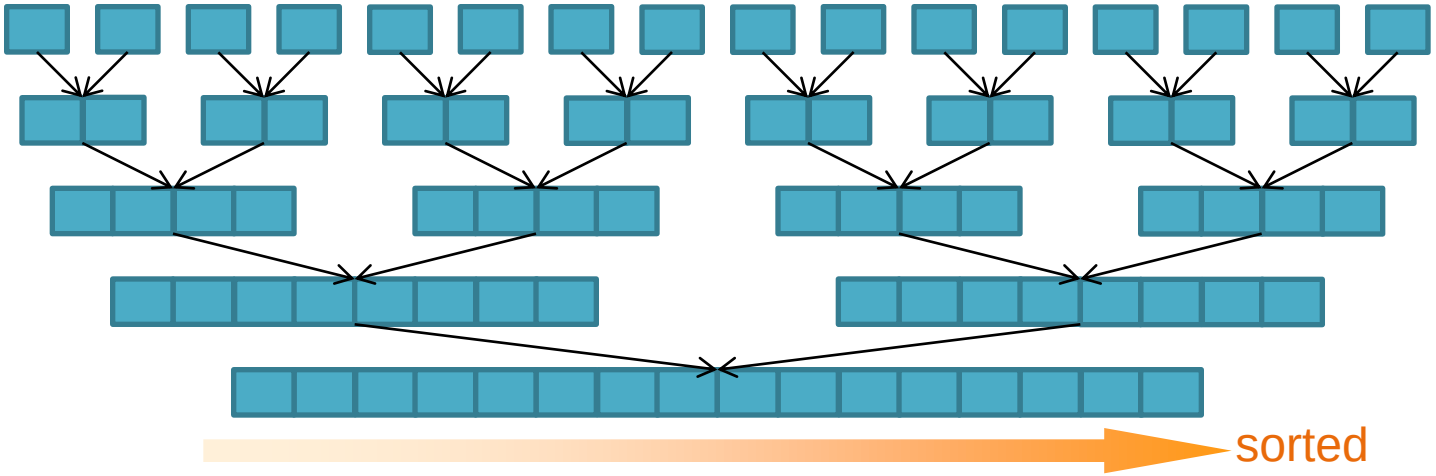


# Background: Integrating Odd-even merge into usual merge



# Background: Multiway mergesort

Standard (2-way) mergesort



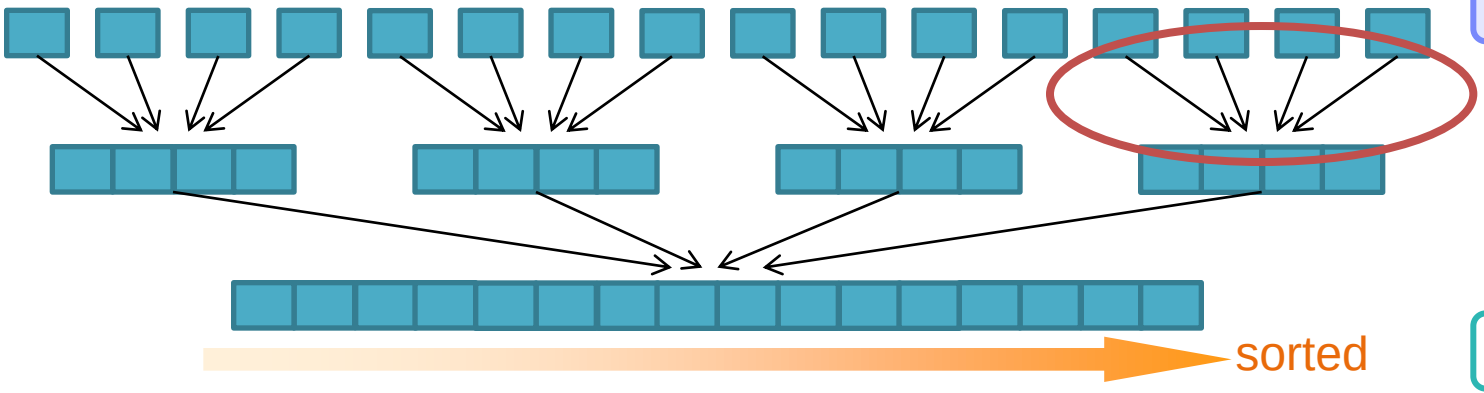

**unsorted** input

2way
2way
2way
2way

repeat 2-way merge for $\log_2(N)$ stages

**sorted** output

Multiway (*k*-way) mergesort, here $k = 4$



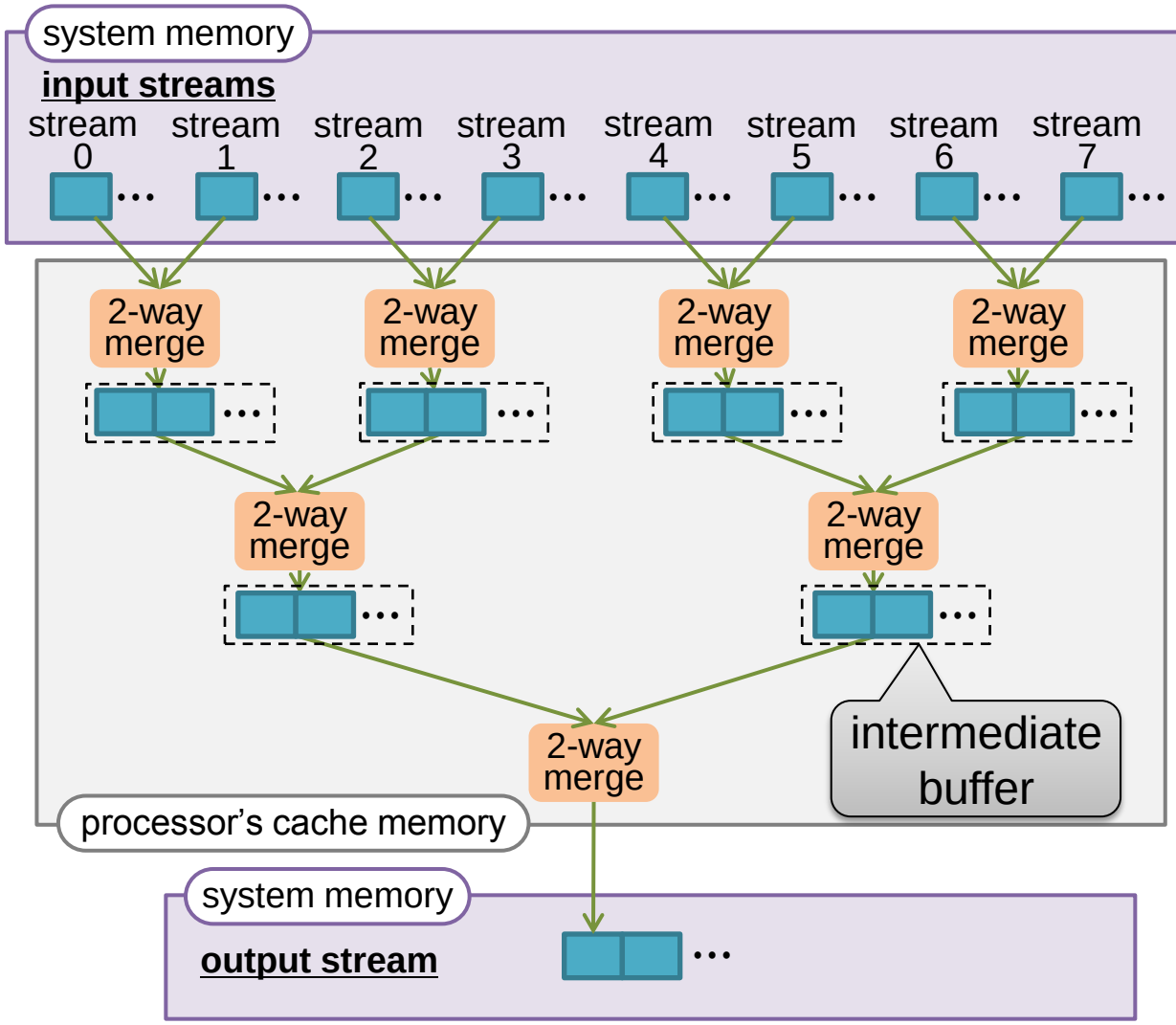

**unsorted** input

4way
4way

*k*-way merge $\log_k(N)$ stages

**sorted** output

# Background: Multiway merge with SIMD




(8-way merge as an example.
we used 32-way merge
in actual implementation)

# Outline

- ✓Goal
- ✓Background: SIMD multiway mergesort for integers
- Performance problems in existing approaches
- Our approach
- Performance results

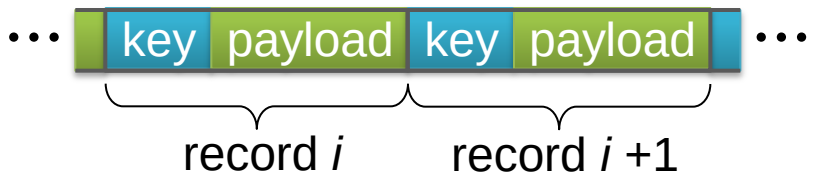
# Existing approach for sorting structures with SIMD

- For sorting structures with SIMD mergesort, a frequently used approach (*key-index approach*) is
  1. pack key and index for each record into an integer.
  2. sort the key-index pairs with SIMD, and then
  3. **rearrange** the records based on the sorted key-index pairs

☹ **Costly due to random accesses for main memory**

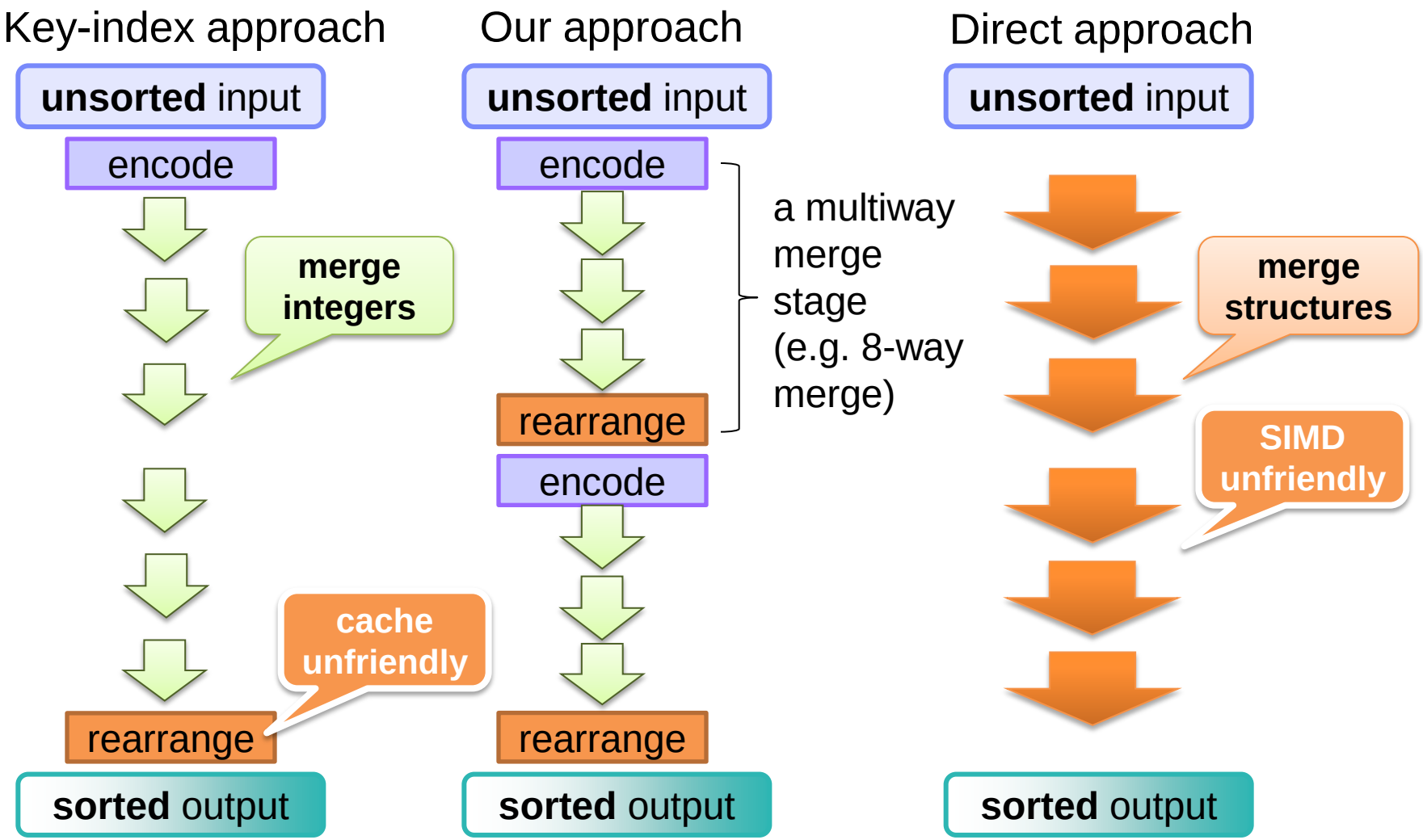
# Performance problems in existing approaches

- Existing approaches for sorting structures with SIMD instructions have performance problems
  - Key-index approach: SIMD friendly but not cache friendly due to random memory accesses
  - Direct approach: cache friendly but not SIMD friendly because the keys are not stored contiguously in memory

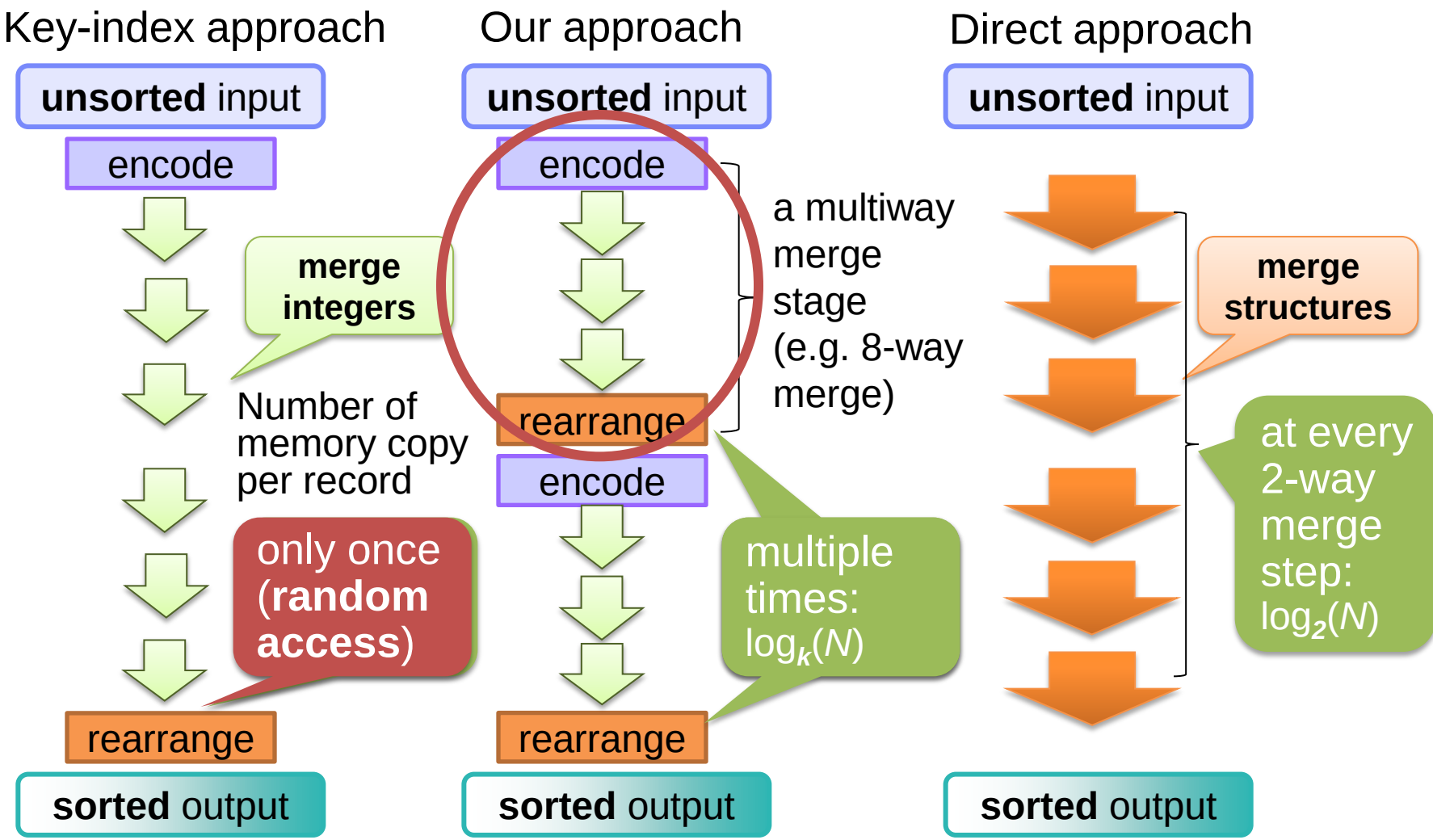



- Our approach takes the benefits of both approaches

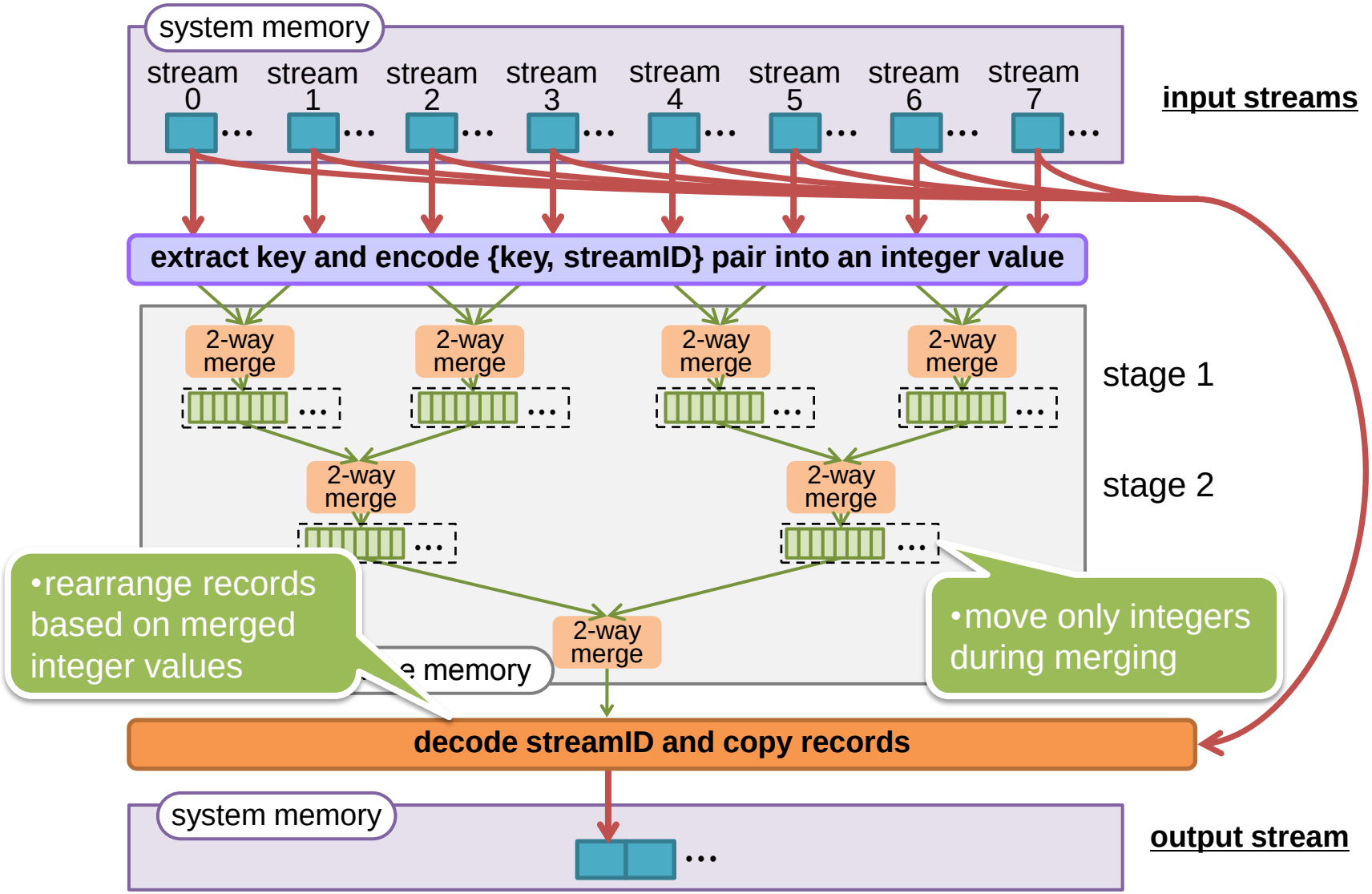
# Our approach: Rearranging at each multiway merge step



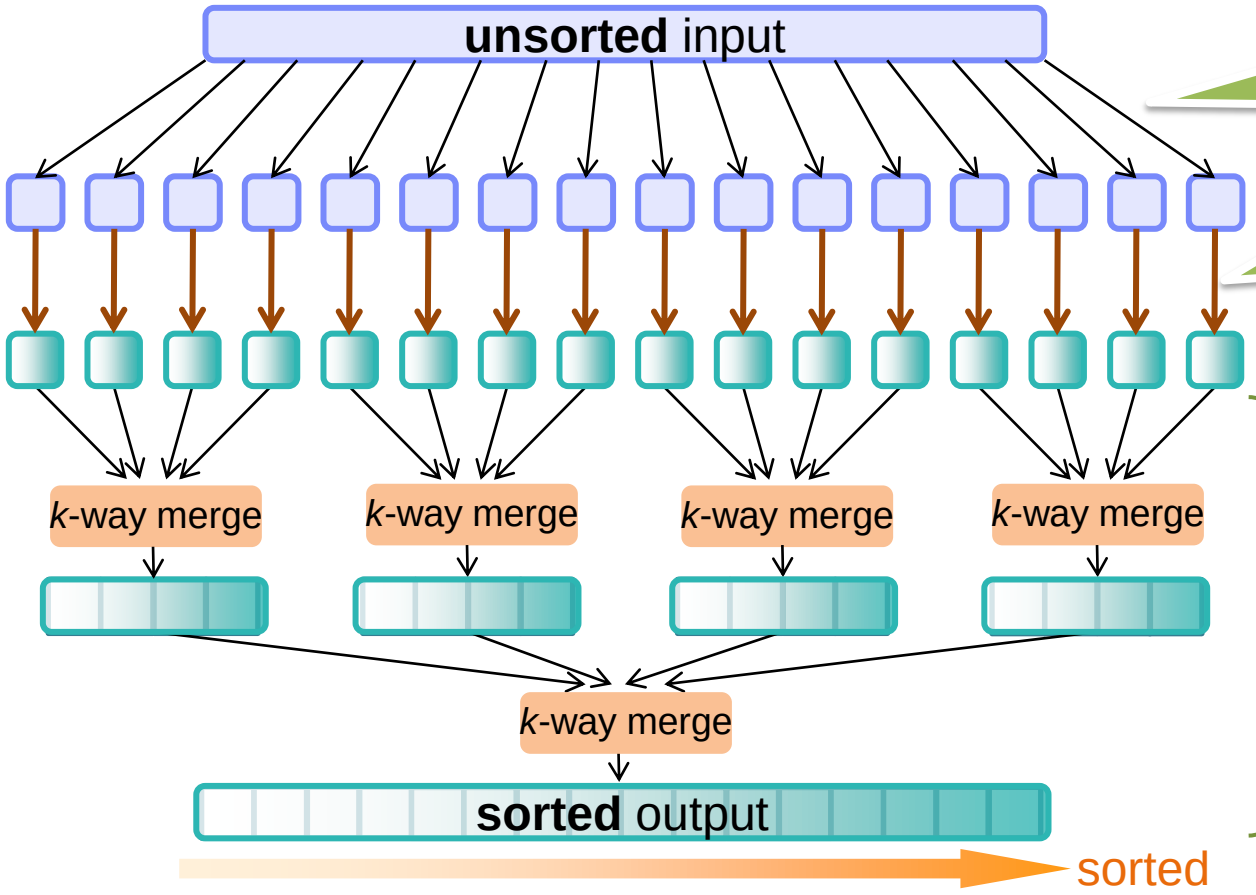
# Our approach: Rearranging at each multiway merge step



# Our approach: Multiway Merge with SIMD



# Our approach: Overall sorting scheme




Step 1: divide into small blocks that can fir in (L2) cache

Step 2: sort each block by vectorized combsort

Step 3: repeatedly execute multiway merge to merge all blocks

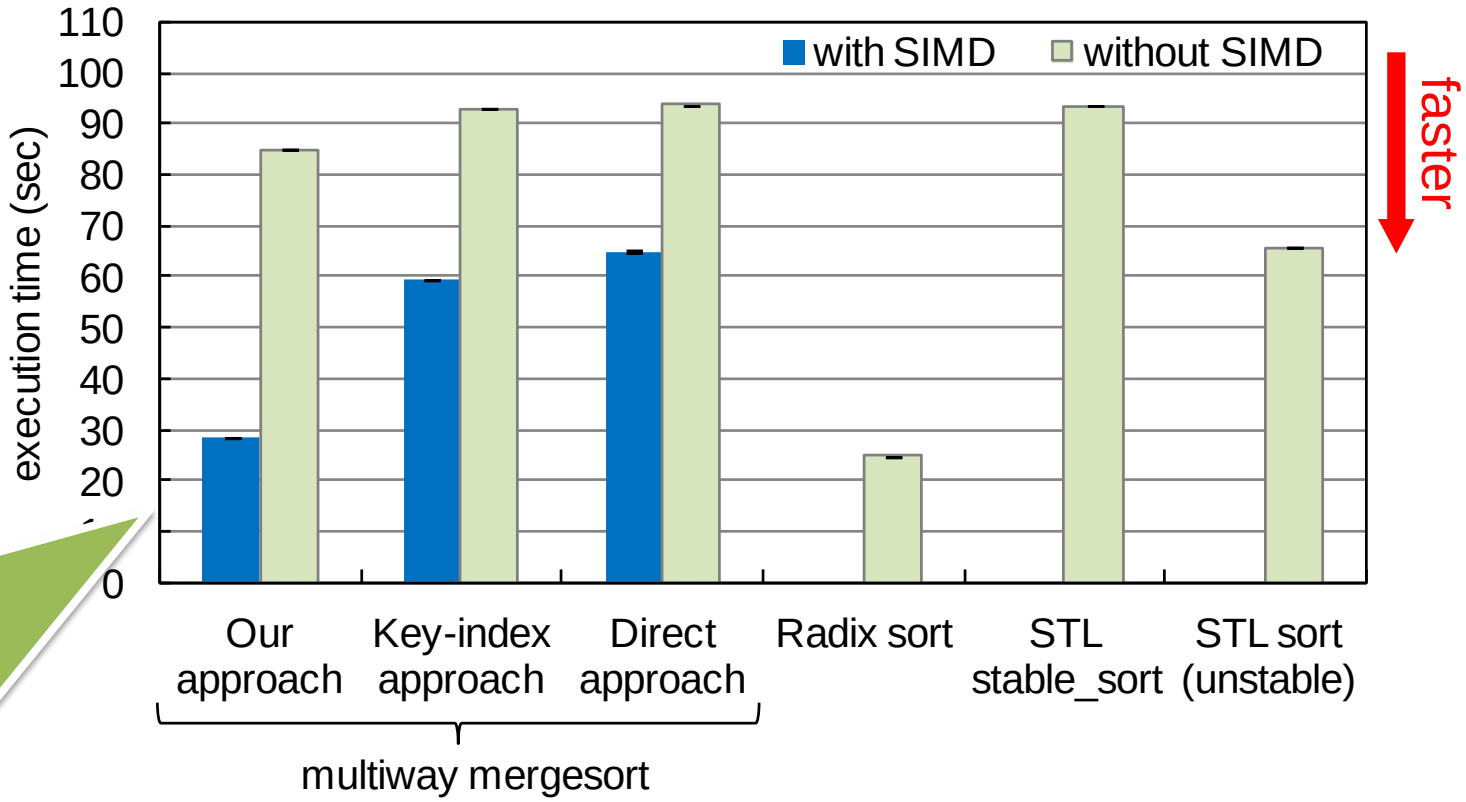
(read the paper on vectorized combsort)

# Performance Evaluations

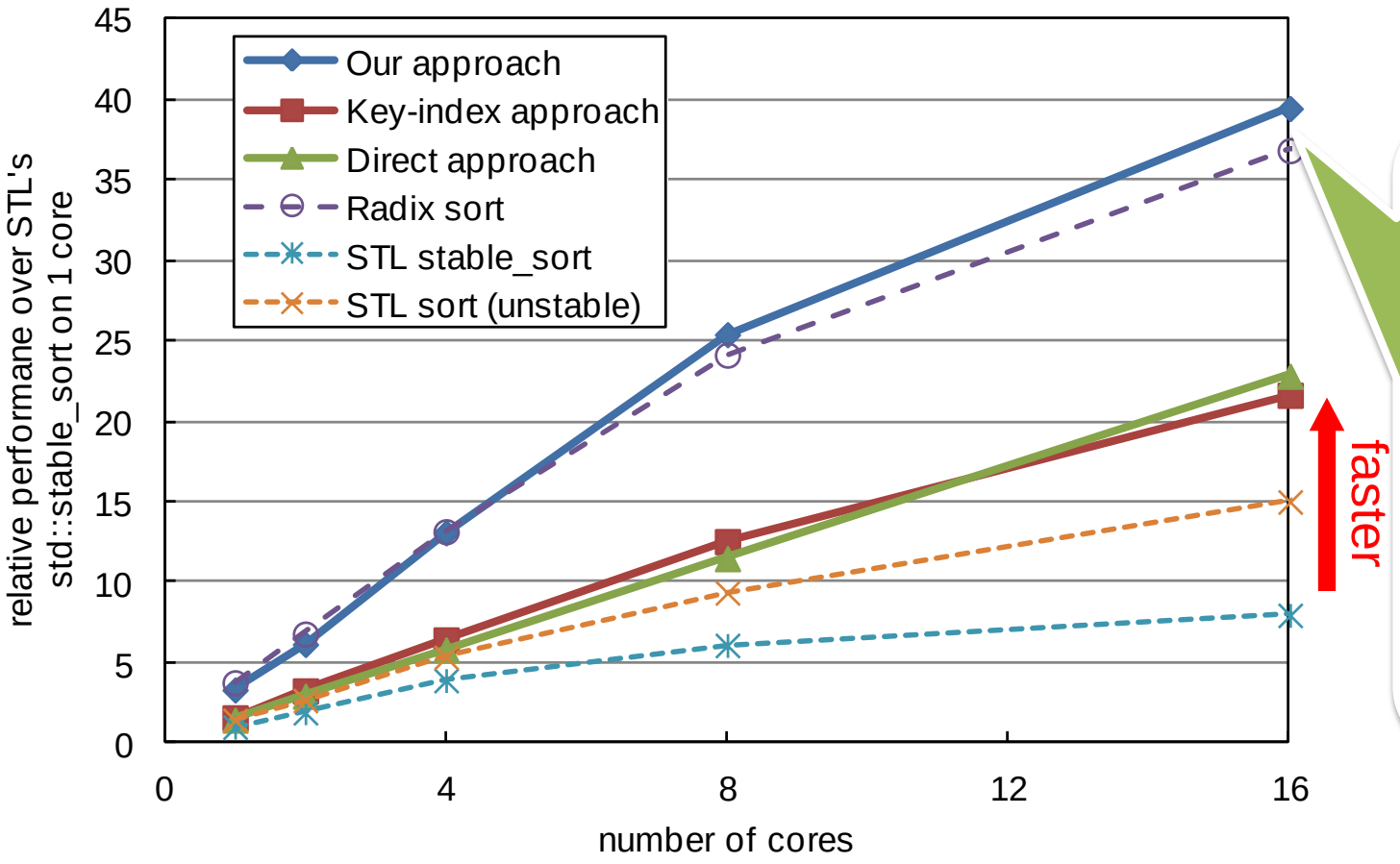
- System
  - 2.9-GHz Xeon E5-2690 (SandyBridge-EP) processors
    - 2 sockets x 8 cores = 16 cores
    - using SSE instructions (128-bit SIMD)
  - 96 GB of system memory
  - Redhat Enterprise Linux 6.4, gcc-4.8.2
- We compared the performance of following algorithms
  - Multiway mergesort ($k$ = 32 way)
    - Our approach, key-index approach, direct approach
  - Radix sort, STL std::stable_sort, STL (unstable) std::sort

# Performance for Sorting 512M 16-byte records with and without SIMD instructions

- Our approach gained largest speed up from SIMD among three approaches
- It achieved comparable performance to radix sort



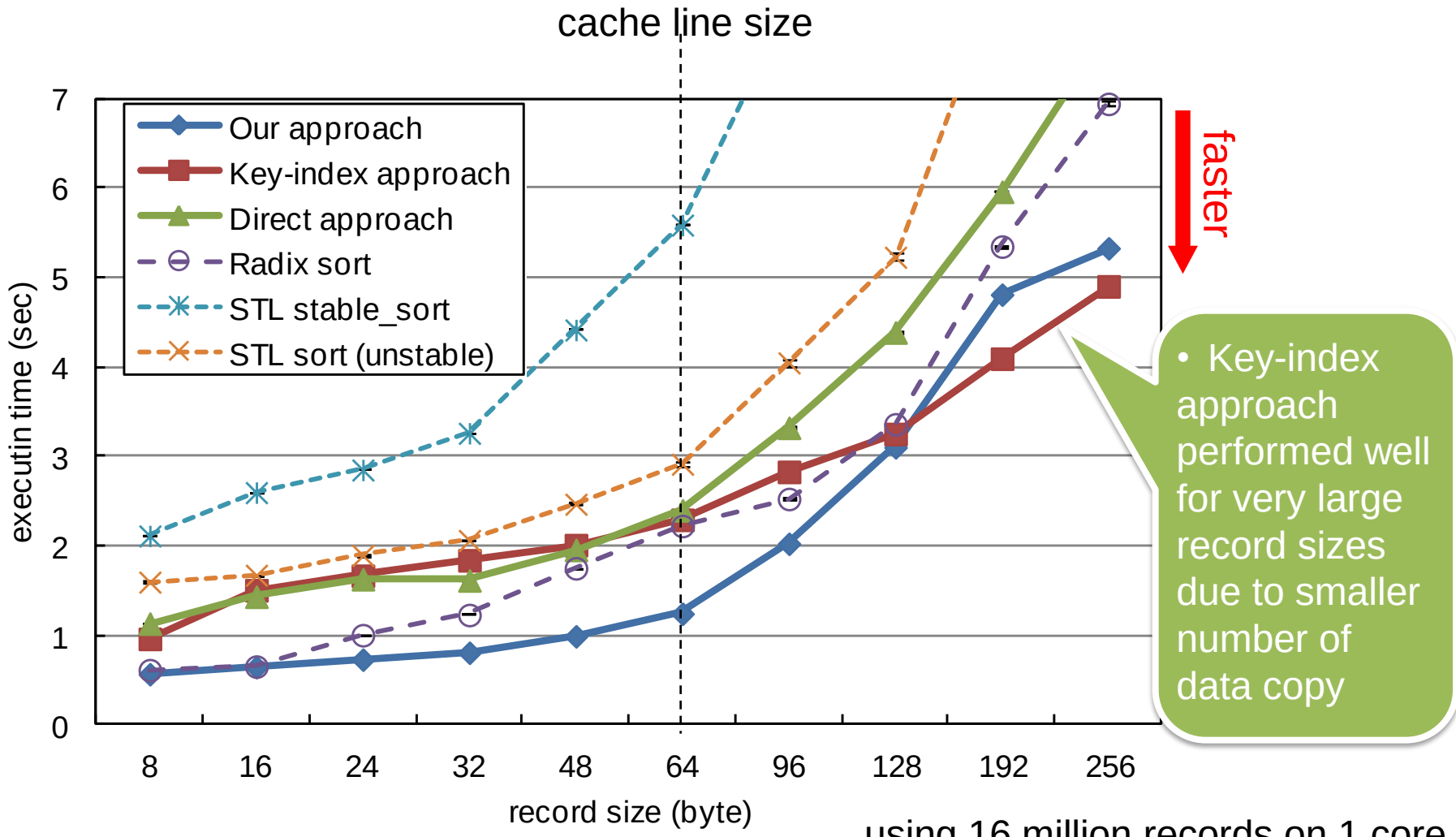
# Performance Scalability with multiple cores




- Our approach achieved the best performance with multiple cores up to 16 cores
- It showed better scalability than radix sort

using 512M 16-byte records

# Performance with various sizes of a record




using 16 million records on 1 core

# Summary

- We developed a new algorithm for sorting an array of structures, that enables
  - efficient use of SIMD instructions
  - efficient use of cache memory (no random accesses)
- Our new algorithm outperformed the key-index approaches in multiway mergesort especially when each record is smaller than a cache line
- It outperformed the radix sort for records larger than 16 byte and showed better scalability with multiple cores

➔Read the paper for more detail:
efficient 4-wide SIMD exploitation, results with 10-byte ASCII key, sorting for variable-length strings

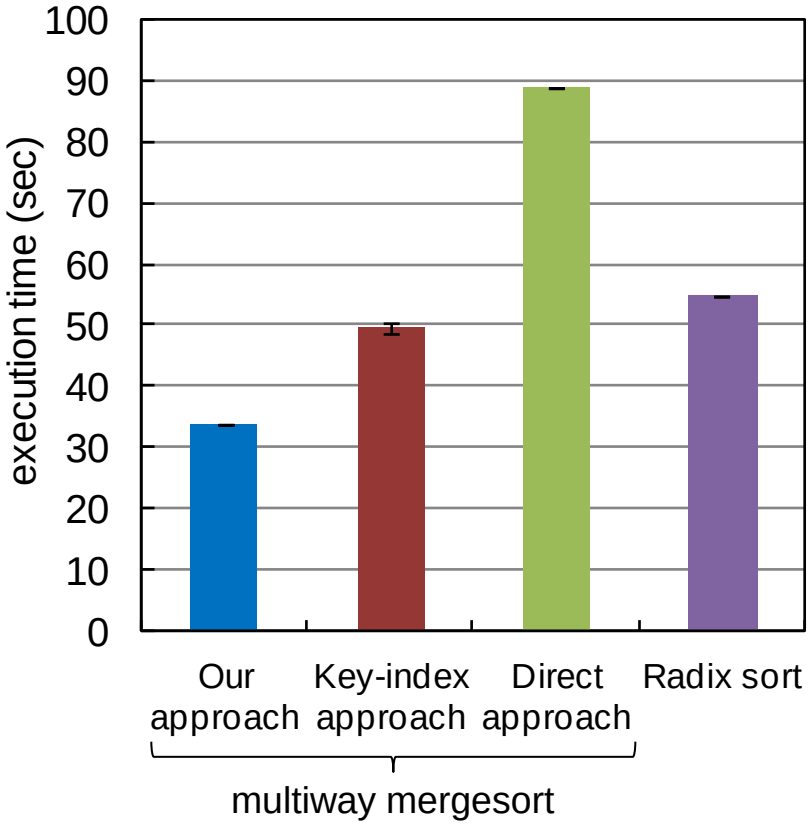
# backup

# Optimization techniques: partial key comparison

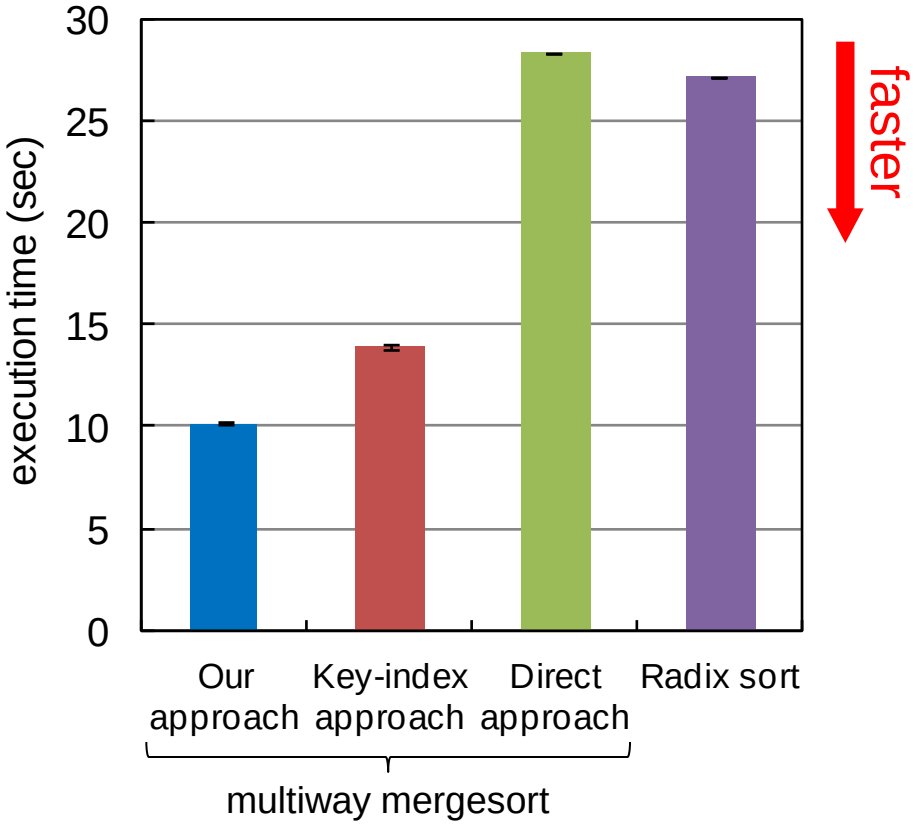
- To use 4-wide SIMD (32 bit x 4) instead of 2-wide SIMD (64 bit x 2)
  - Increasing data parallelism and giving the higher performance
  - Sorting based on a partial key
    - We confirm that the sorted order is correct using the entire key when rearranging records using scalar comparison

# Sorting for variable-length strings

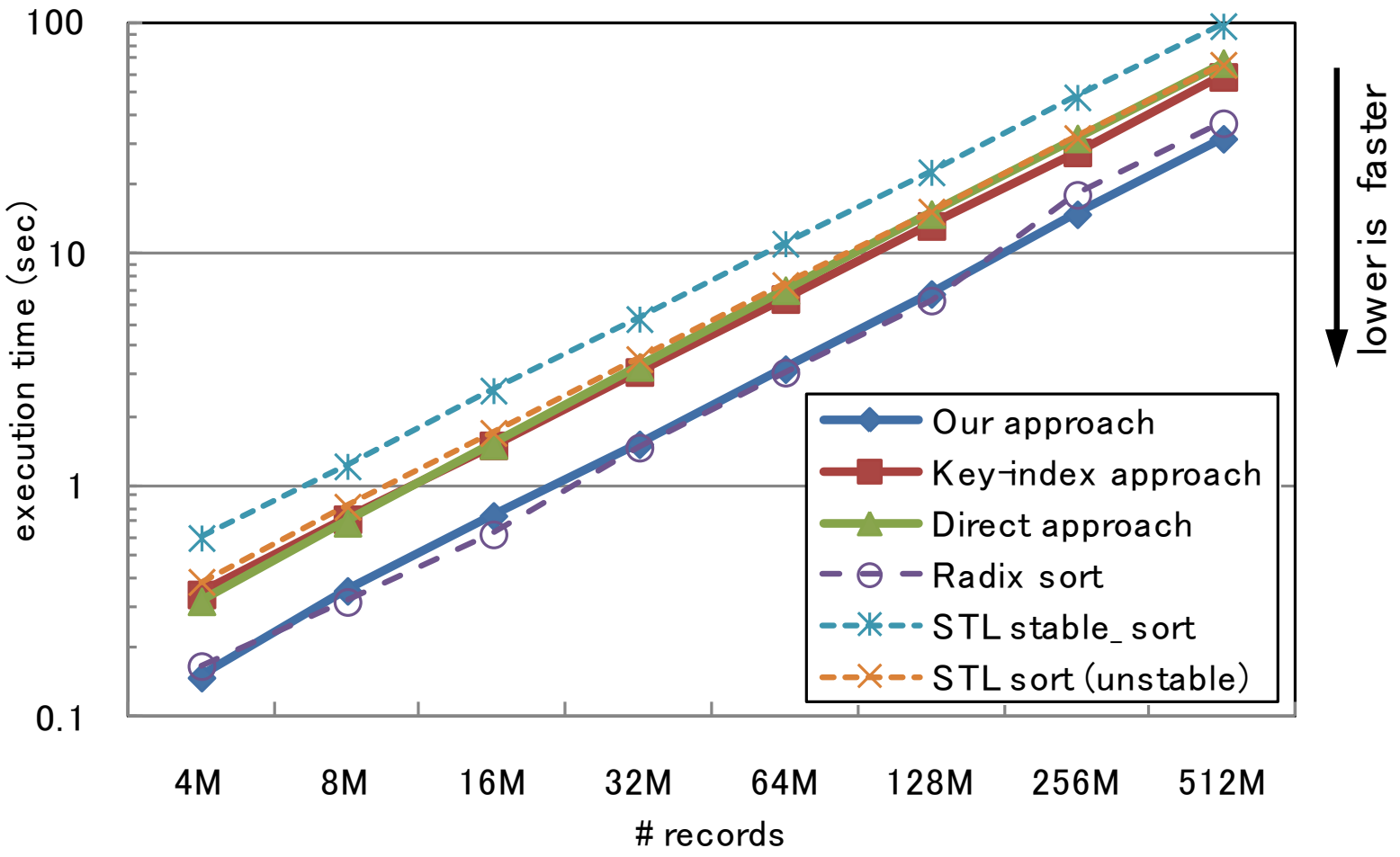
256M 12-byte to 20-byte records
(16 bytes on average)




64M 12-byte to 84-byte records
(48 bytes on average)



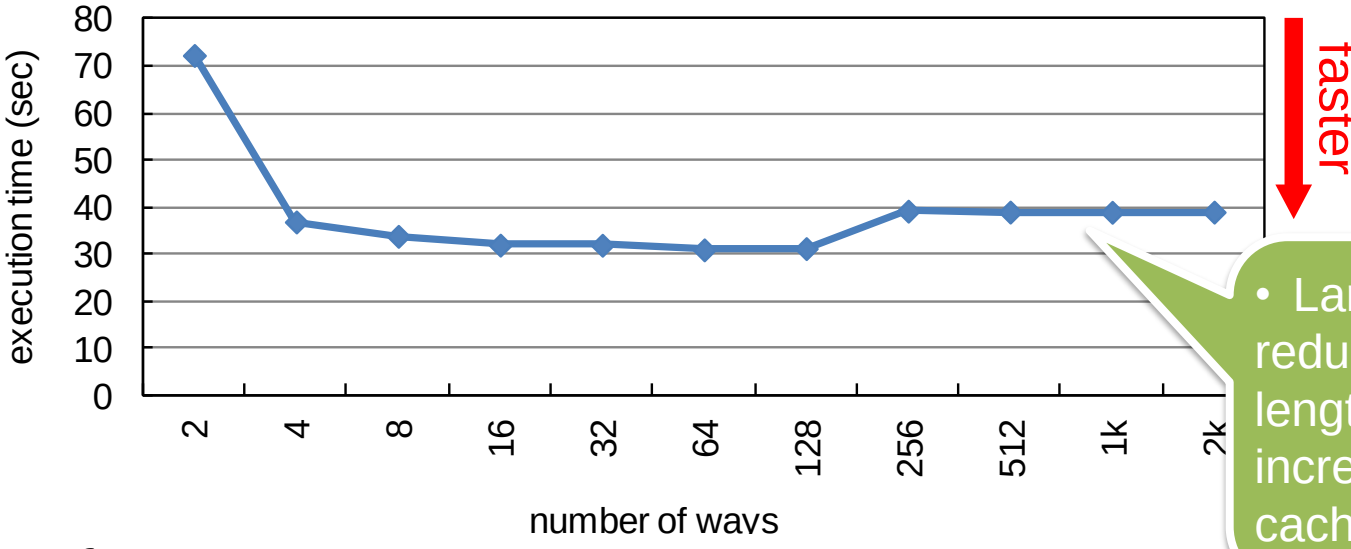
# Performance with various numbers of records




using 16-byte records on 1 core

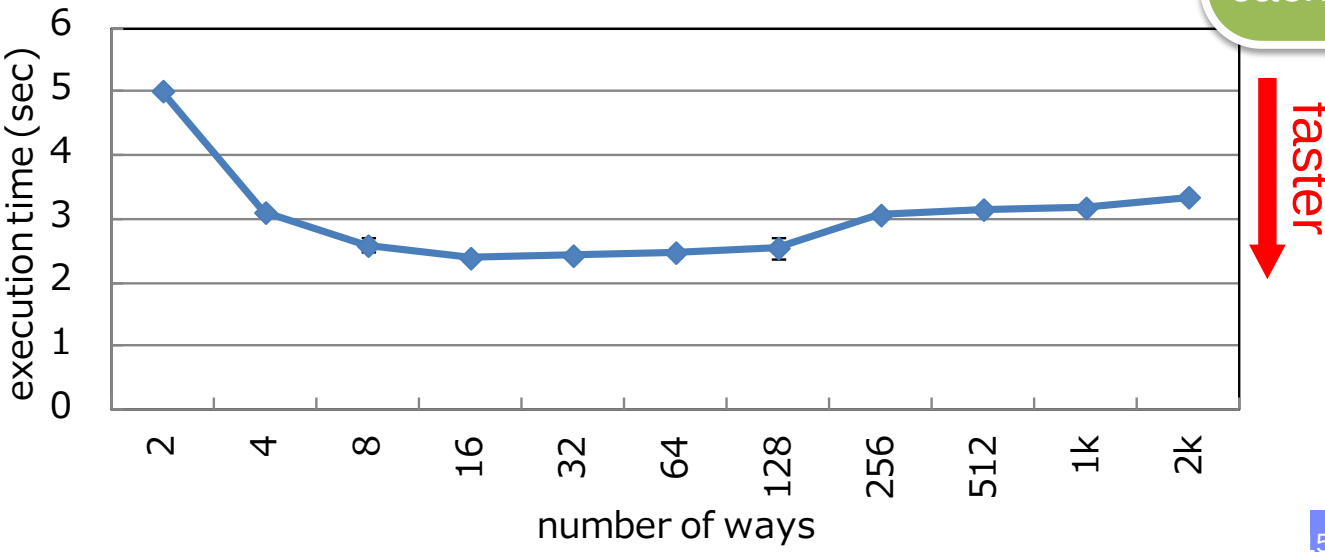
# Effect of number of ways in multi-way merge

on 1 core




- Larger ways reduced path length but increased cache misses

on 16 cores




using 512M 16-byte records