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

Hiroshi Inoue (inouehrs@jp.ibm.com)[†] [‡] Kenjiro Taura[‡]
[†]IBM Research – Tokyo [‡]University of Tokyo

Introduction

- SIMD-based multiway mergesort has been used for in-memory sorting of integers
- We extend this for sorting structures (key + payload)

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

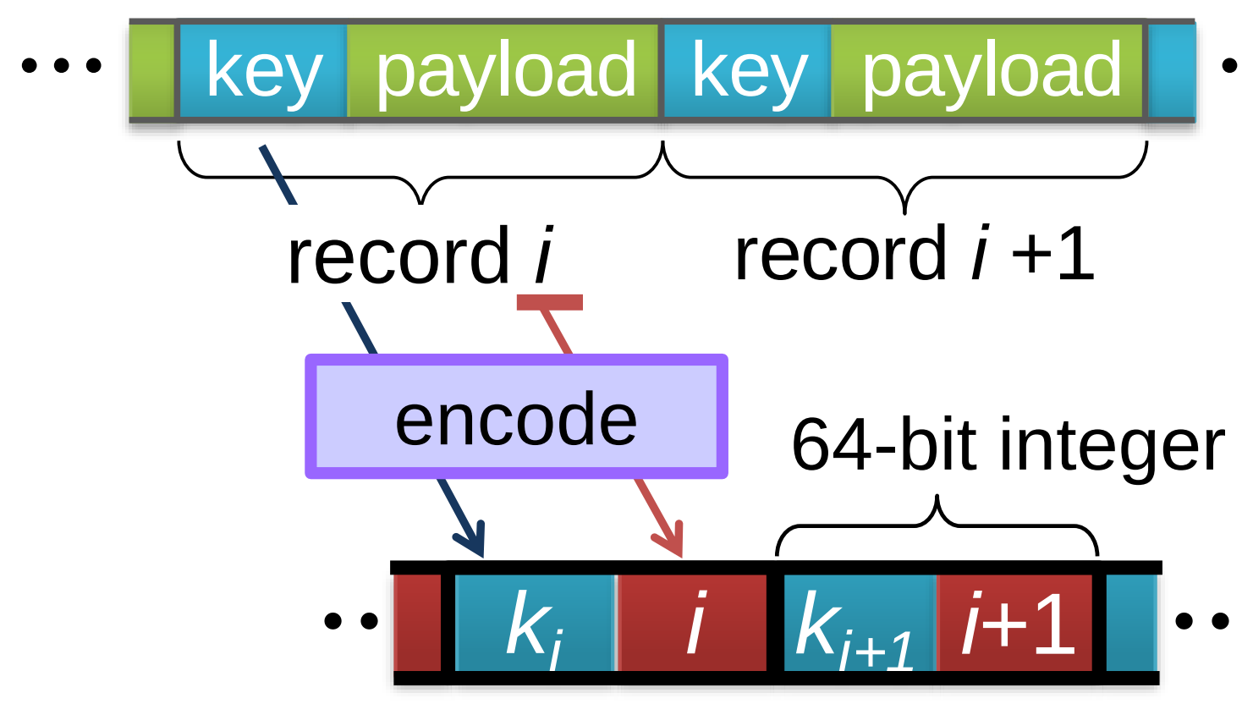
Existing approaches

- **Key-index approach:**

1. encode ***key*** and ***index*** for each record into an integer,
2. sort the key-index pairs with SIMD, and
3. rearrange the records based on the sorted key-index pairs

☹ Costly due to random accesses for memory

➔SIMD friendly but NOT cache friendly



- **Direct approach:**

1. sort records directly without encoding into an integer

☹ Inefficient with SIMD due to gather for keys

➔Cache friendly but NOT SIMD friendly

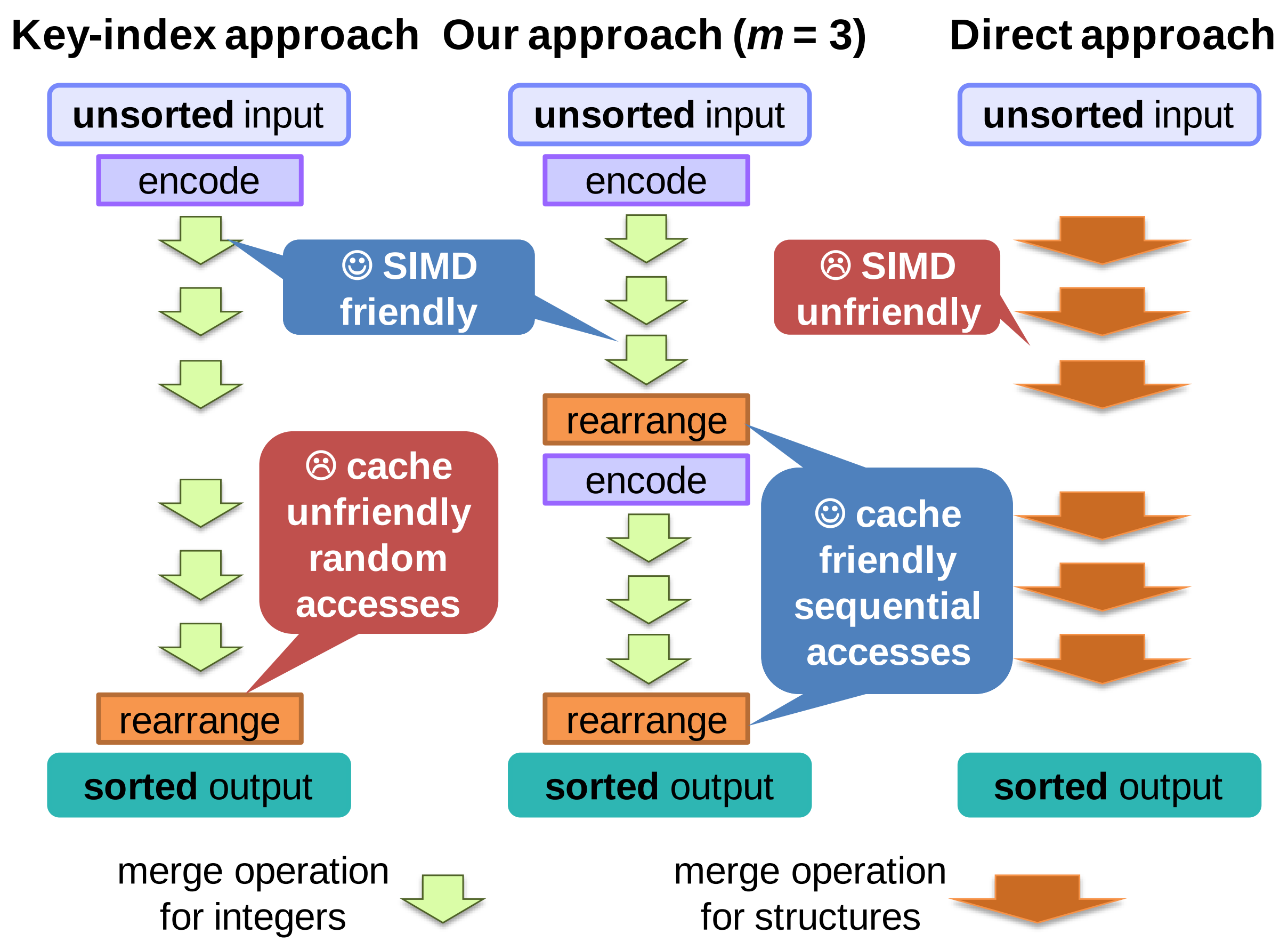
Our approach

Key idea:

- to execute rearranging of records more frequently, e.g. once per *m* merge stages (*m* > 1), instead of only once at the last in key-index approach

Benefits

- **Cache friendly:** the rearrange operation reads from $k = 2^m$ input streams and write to one output stream; hence the memory accesses are sequential unless *m* is too large
- **SIMD friendly:** most of the merge operations are done for integers; reading keys from records, which is costly with SIMD, only once per *m* stages



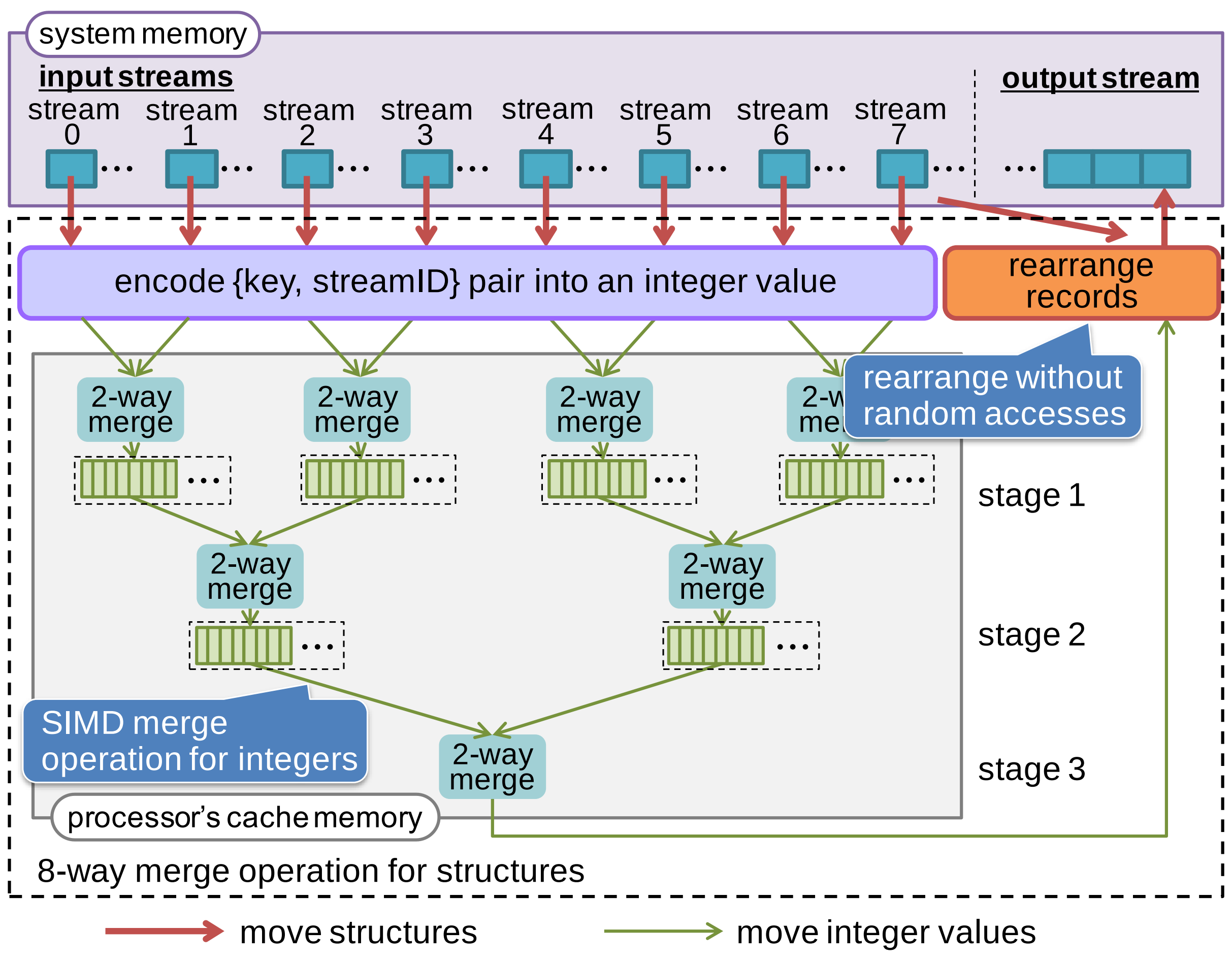
Implementation in multiway merge

We integrate key encoding and rearrange into multiway merge operation

- multiway merge, which merges *k* (*k* > 2) input streams into one output stream, is a common technique to reduce memory bandwidth in mergesort

Steps of our multiway merge operation

1. at the first stage, read records from system memory and encode *key* and *streamID* into an integer
2. merge integer values using SIMD
3. at the last stage, rearrange records based on the encoded *streamID*



Optimizations and overall scheme

Optimizations:

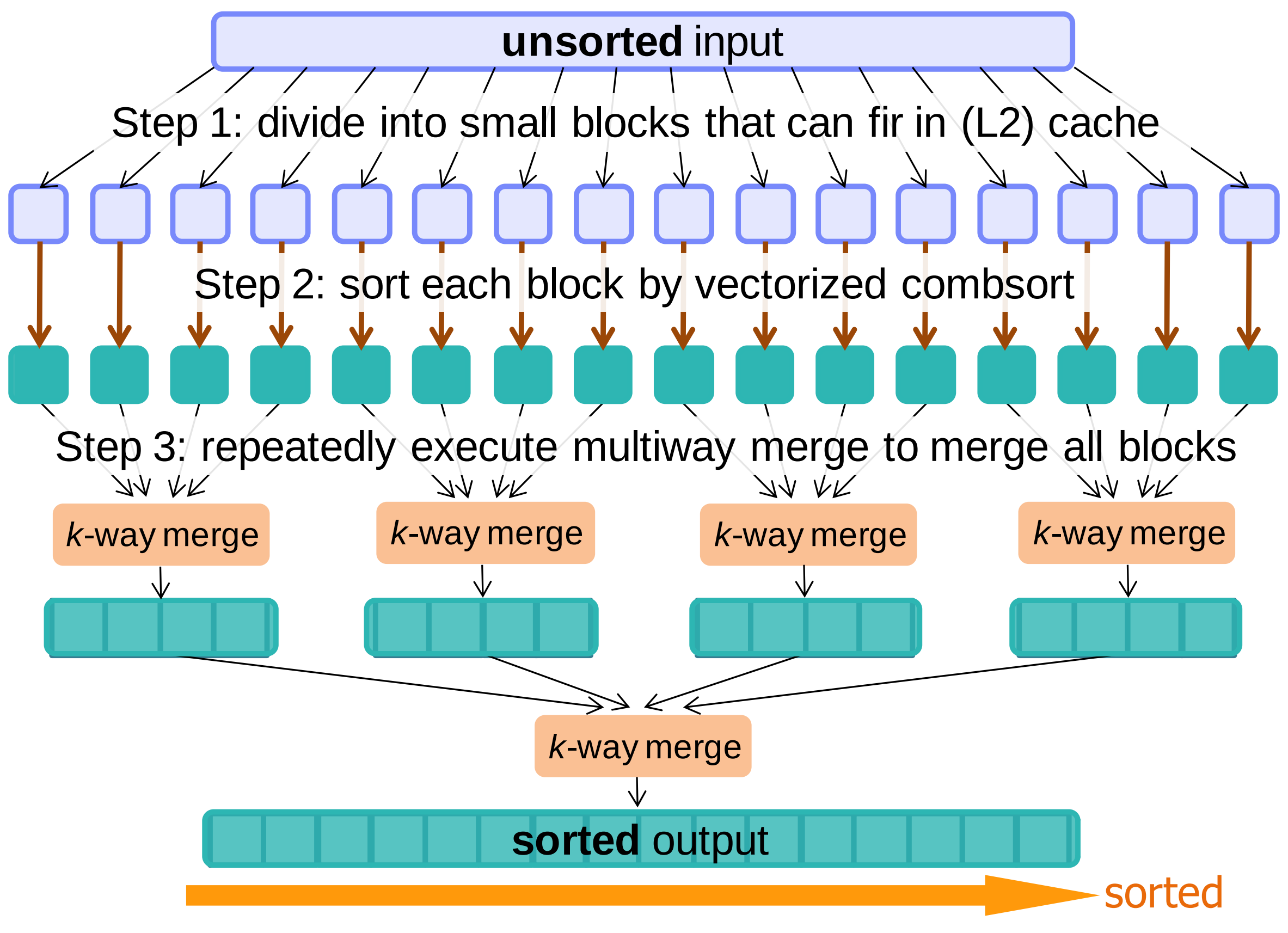
exploiting 4-wide SIMD by encoding a {key, id} pair into 32-bit integer

- we encode *streamID* (up to *k*) accompanied with its key into an intermediate integer; the streamID is much smaller than the index (up to *N*) ➔ we can use more bits for the key
- we use a 32-bit integer instead of a 64-bit integer to encode (a part of) key and streamID to use higher data parallelism when the number of elements to merge is smaller than a threshold
- if we use only a part of keys for merging, we check the order by using the entire key when we rearrange records (without using SIMD)

vectorized combsort for initial sorting

- because mergesort is not efficient for small amount of data, we switch to vectorized combsort if a block to sort is small enough to fit into L2 cache
- combsort is efficient with SIMD but shows very poor memory access locality ➔ good for initial sorting of small blocks

Overall sorting scheme:

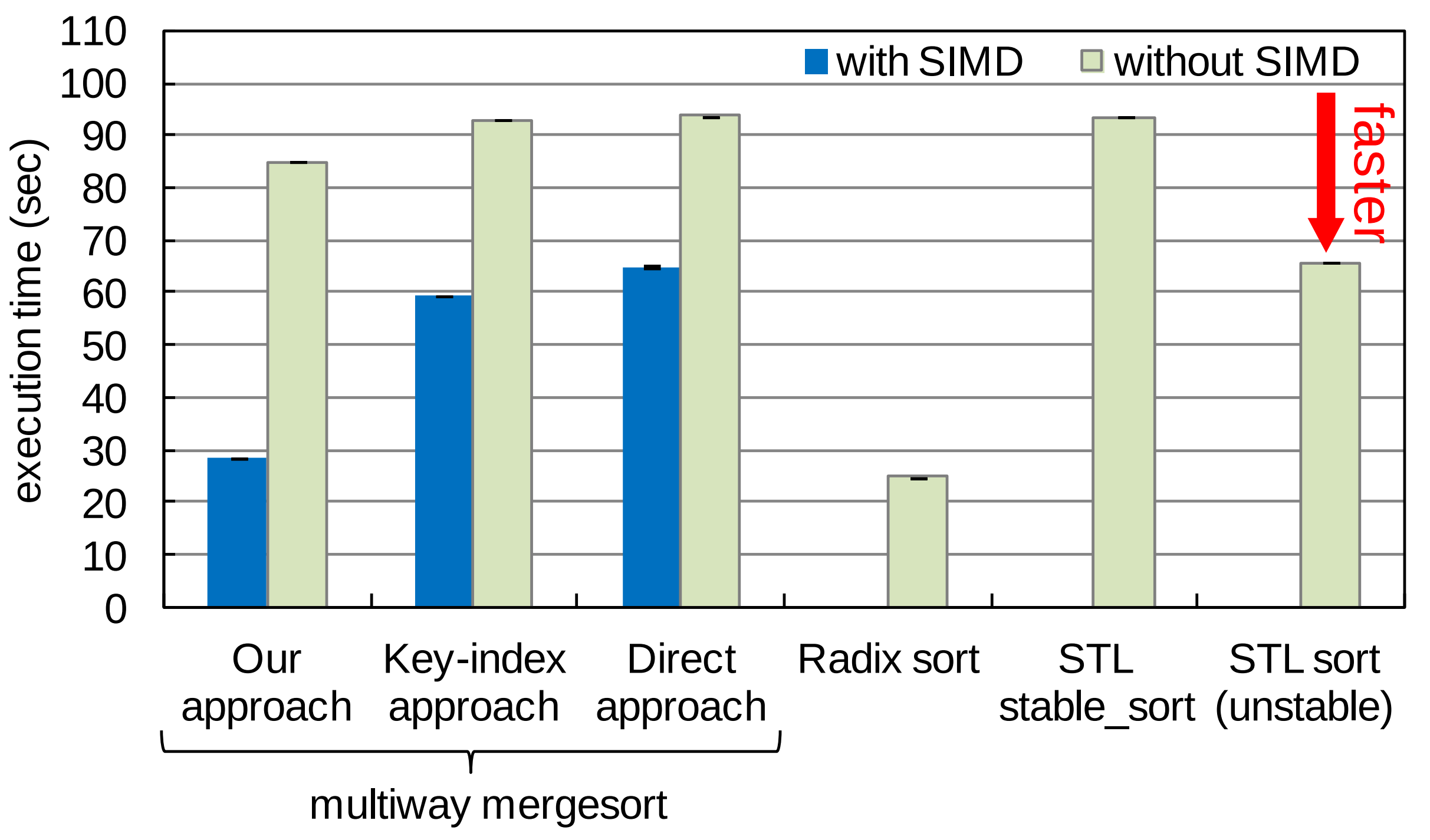

unsorted input
Step 1: divide into small blocks that can fir in (L2) cache
Step 2: sort each block by vectorized combsort
Step 3: repeatedly execute multiway merge to merge all blocks
k-way merge
k-way merge
k-way merge
k-way merge
k-way merge
sorted output
sorted

Performance results

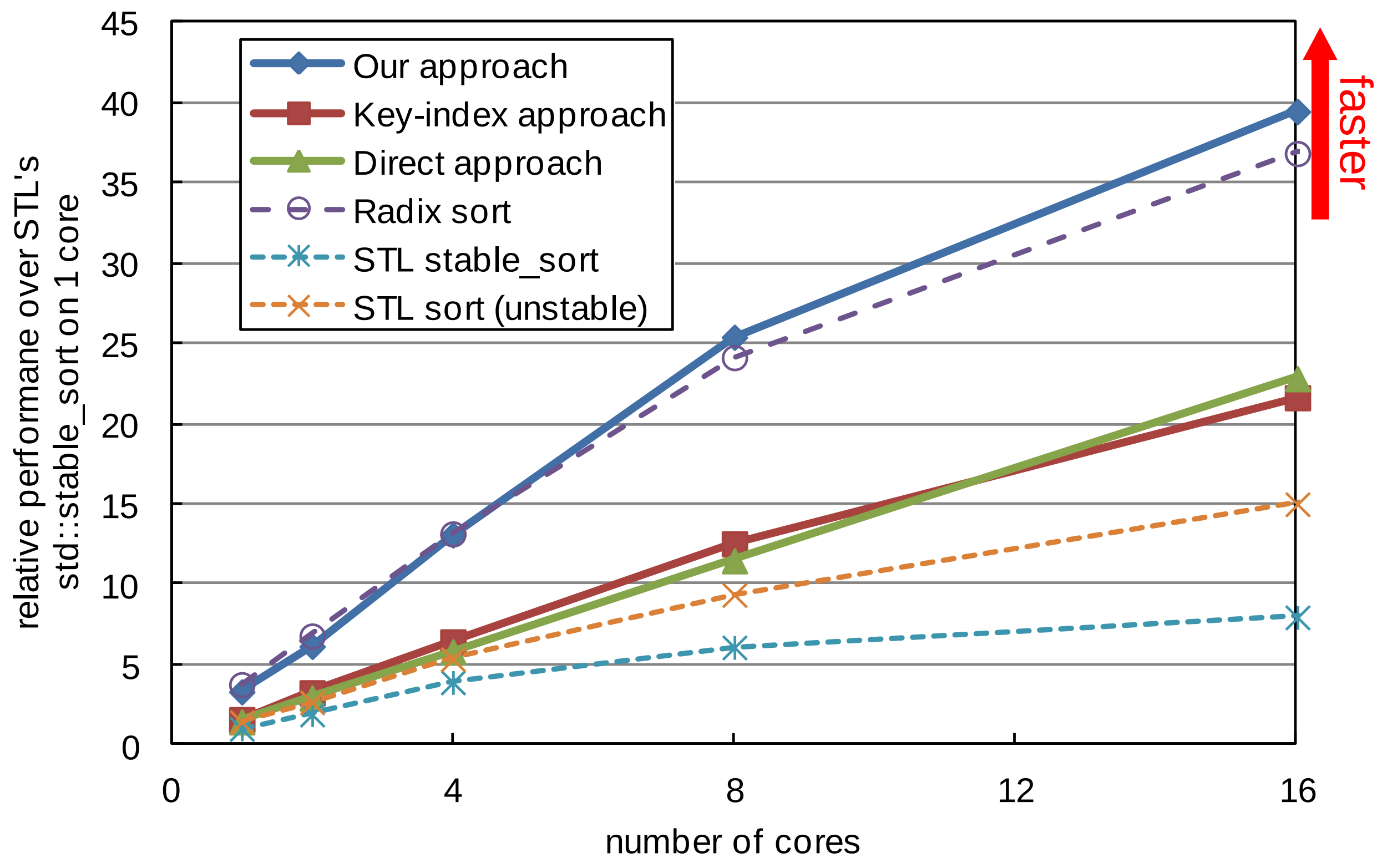
System

- 2.9-GHz Xeon (SandyBridge) / RHEL 6.4 / gcc-4.8 / using SSE (128-bit SIMD)

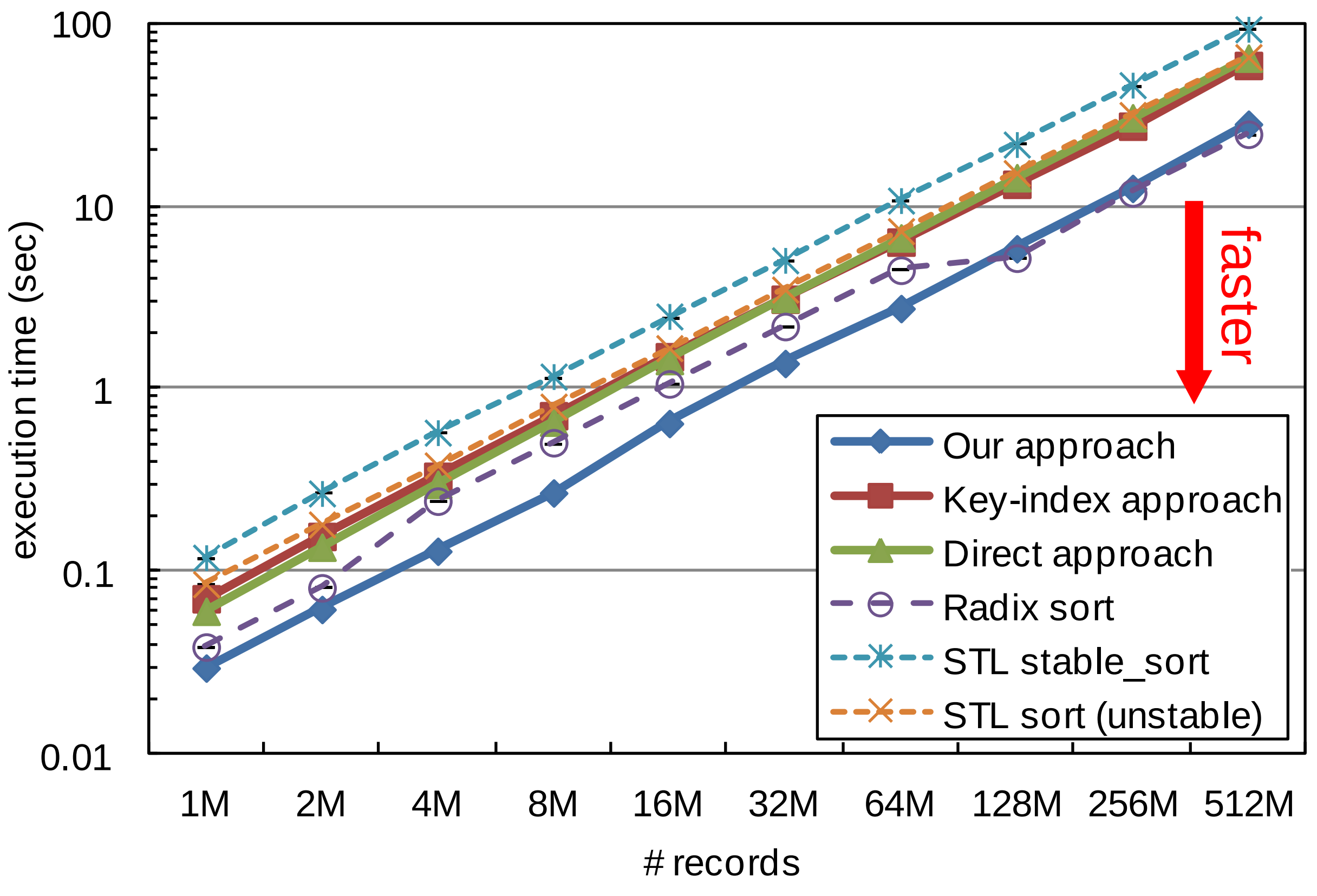
Sorting 512M records of 16 byte



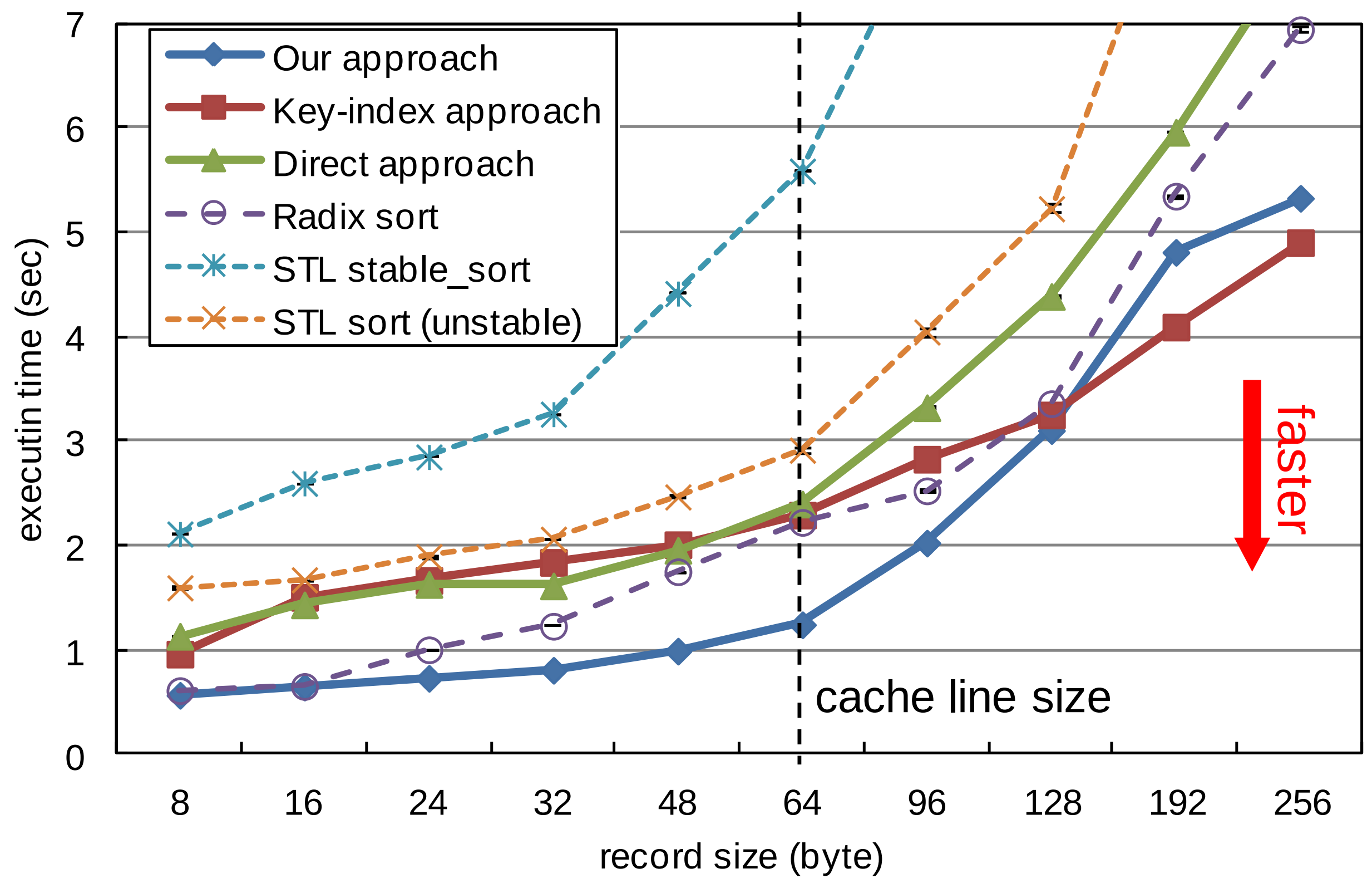
Scalability with multiple cores



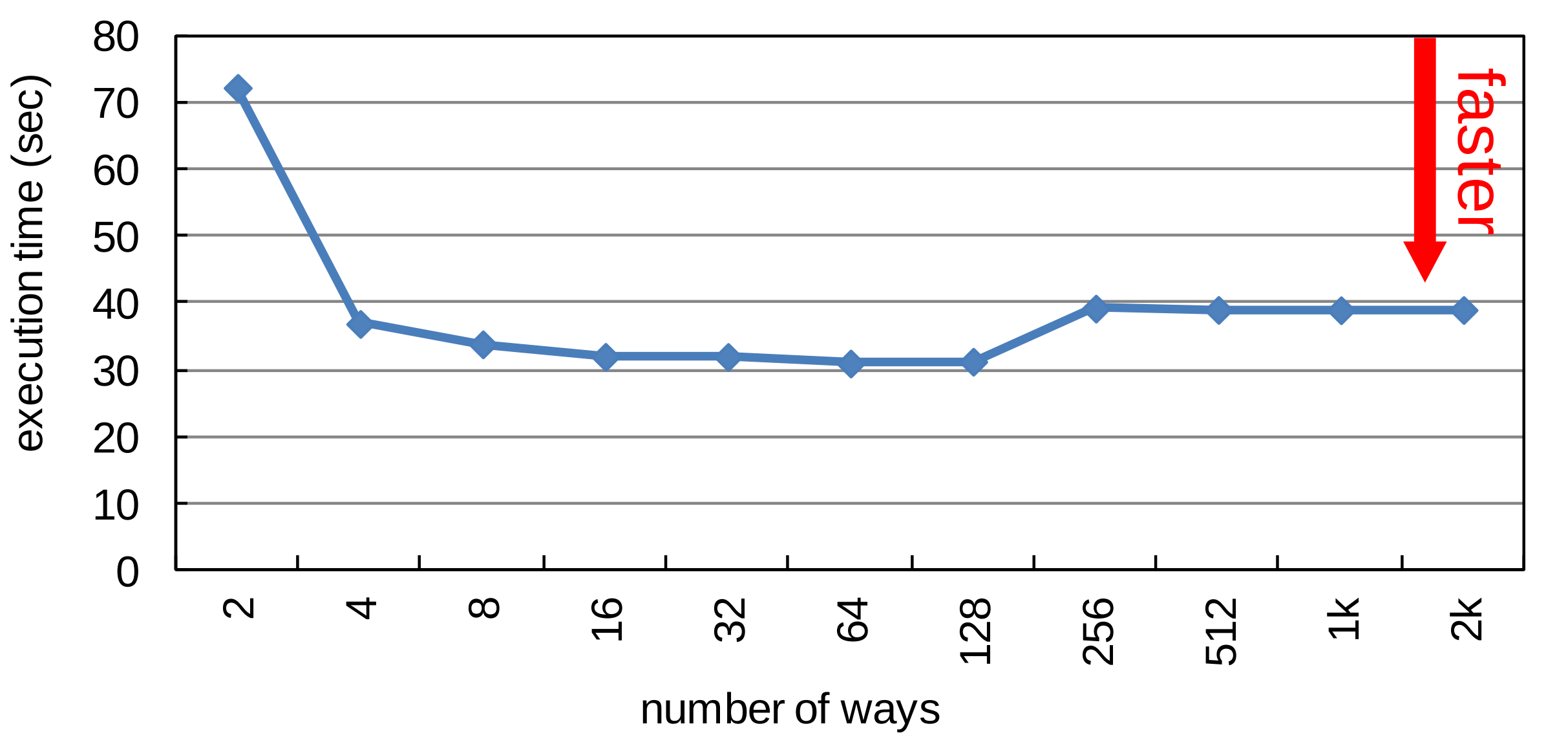
Sorting various numbers of 16-byte records



Sorting 16M records of various record sizes



Effect of number of ways (*k*)


execution time (sec)
1k
2k
number of ways
faster