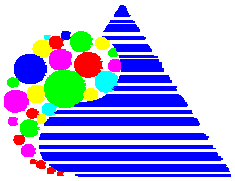
IBM Tokyo Research Laboratory

AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors



Hiroshi Inoue, Takao Moriyama,
Hideaki Komatsu and Toshio Nakatani
IBM Tokyo Research Laboratory

September 18th, 2007 PACT2007 at Brasov, Romania

Goal

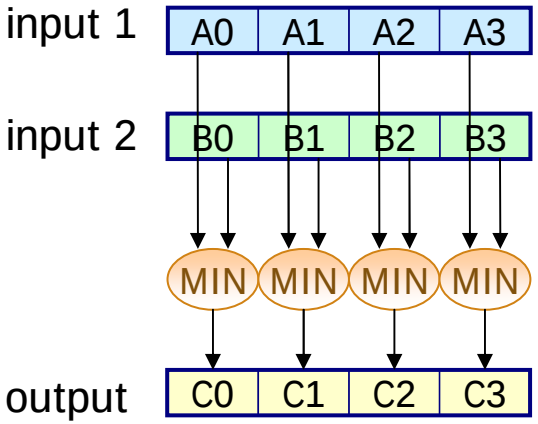
Develop a fast sorting algorithm by exploiting the following features of today's processors

- *SIMD instructions*
- *Multiple Cores (Thread-level parallelism)*

- Outperform existing algorithms, such as Quick sort, by exploiting SIMD instructions on a single thread
- Scale well by using thread-level parallelism of multiple cores

Benefit and limitation of SIMD instructions

- Benefits: SIMD selection instructions (e.g. select minimum instruction) can accelerate sorting by
  - parallelizing comparisons
  - avoiding unpredictable conditional branches

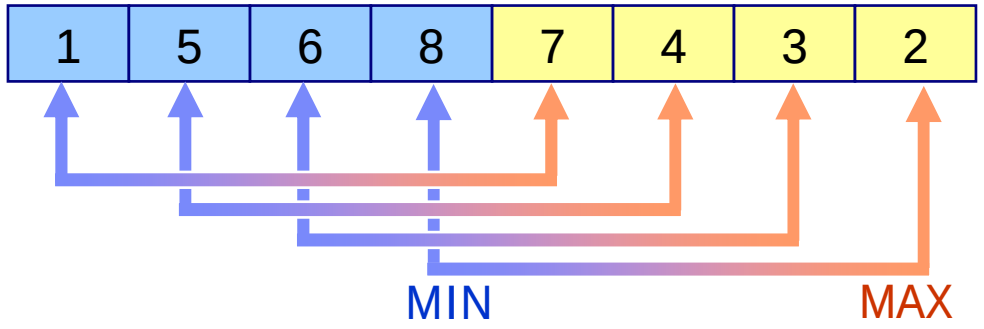


- Limitation: SIMD load / store instructions can be effective only when they access contiguous 128 bits of data (e.g. four 32-bit values) aligned on 128-bit boundary

SIMD instructions are effective for some existing sorting algorithms but slower than Quick sort

- Such as Bitonic merge sort, Odd-even merge sort, and GPUTeraSort [Govindaraju '05]
- They are slower than Quick sort for sorting a large number (N) of elements
  - Their computational complexity is $O(N (\log (N))^2)$

A step of Bitonic merge sort

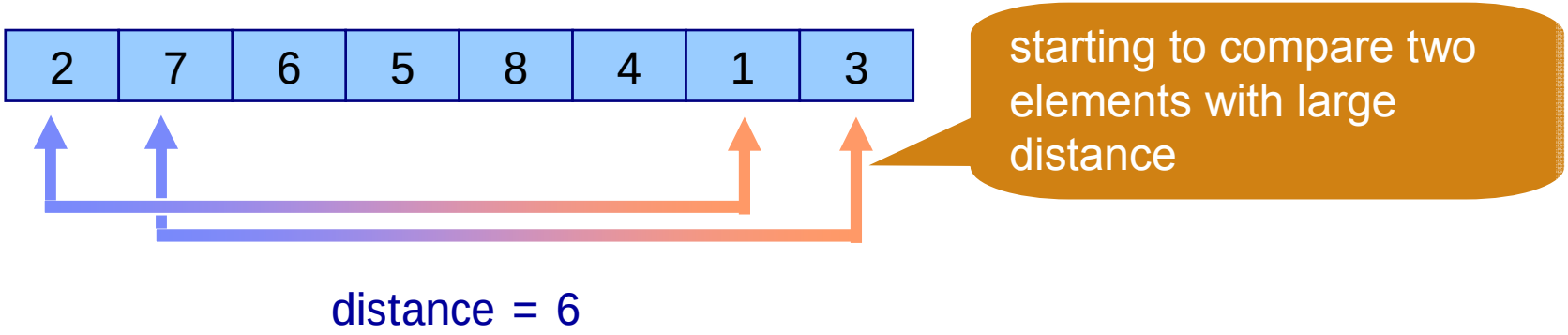


Presentation Outline

- Motivation
- AA-sort: Our new sorting algorithm
- Experimental results
- Summary

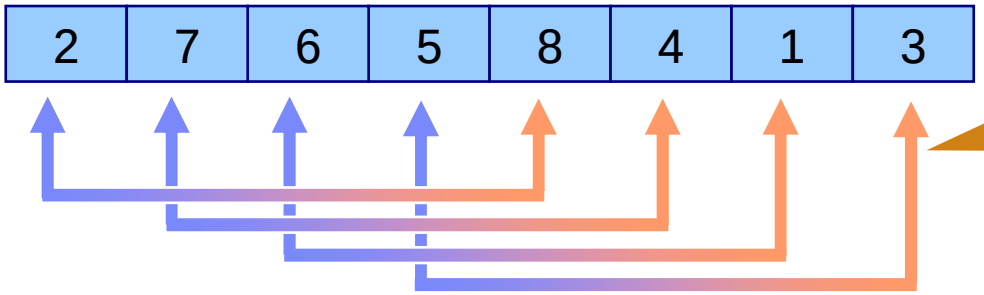
What is Comb sort?

- Comb sort [Lacey '91] is an extension to Bubble sort
- It compares two *non-adjacent* elements



What is Comb sort?

- Comb sort [Lacey '91] is an extension to Bubble sort
- It compares two *non-adjacent* elements



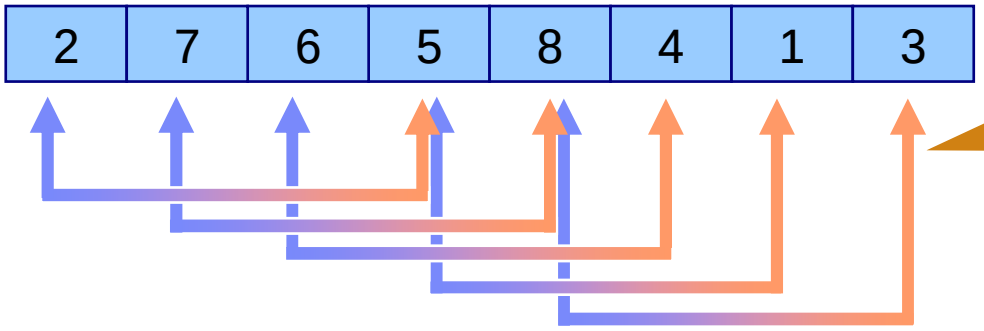
starting to compare two elements with large distance

distance = distance / 1.3 = 4

decreasing distance by dividing a constant called shrink factor (1.3) for each iteration

What is Comb sort?

- Comb sort [Lacey '91] is an extension to Bubble sort
- It compares two *non-adjacent* elements



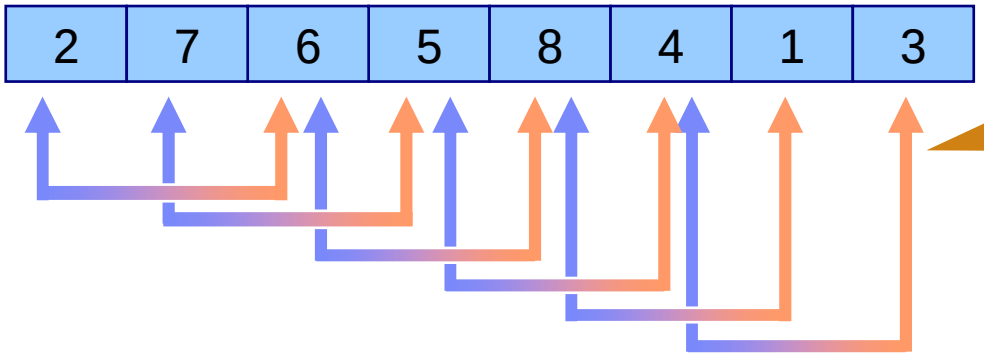
starting to compare two elements with large distance

distance = distance / 1.3 = 3

decreasing distance by dividing a constant called shrink factor (1.3) for each iteration

What is Comb sort?

- Comb sort [Lacey '91] is an extension to Bubble sort
- It compares two *non-adjacent* elements



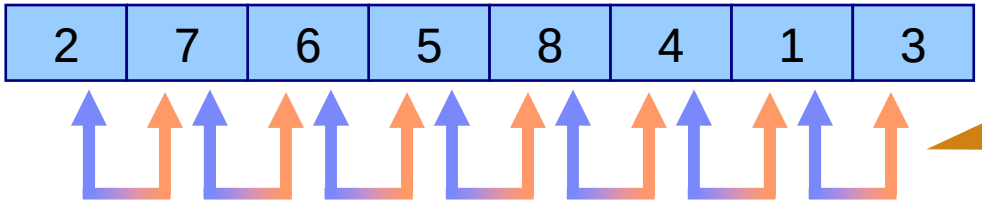
starting to compare two elements with large distance

distance = distance / 1.3 = 2

decreasing distance by dividing a constant called shrink factor (1.3) for each iteration

What is Comb sort?

- Comb sort [Lacey '91] is an extension to Bubble sort
- It compares two *non-adjacent* elements



starting to compare two elements with large distance

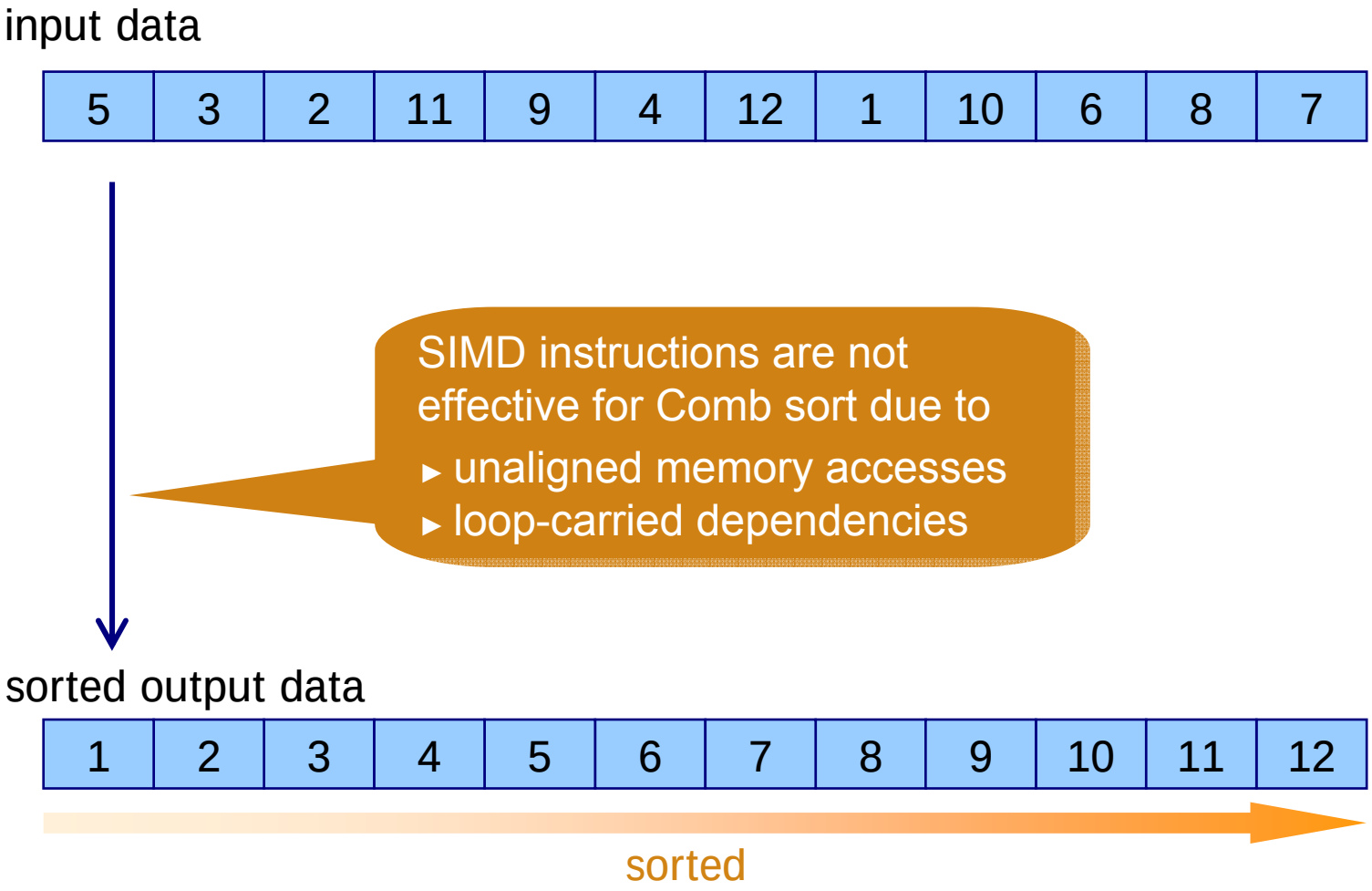
distance = distance / 1.3 = 1

decreasing distance by dividing a constant called shrink factor (1.3) for each iteration

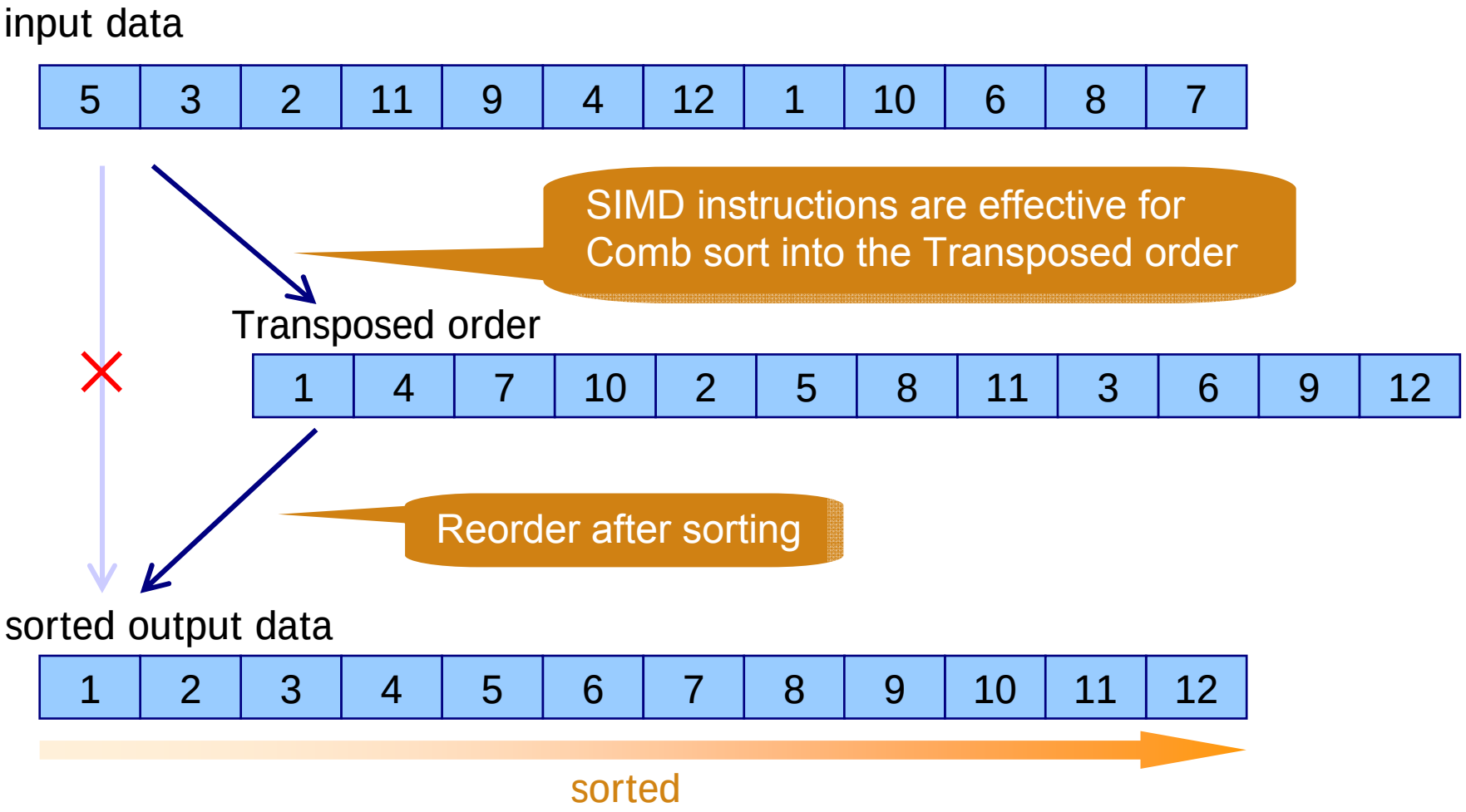
repeat until all data are sorted with distance = 1

average complexity
N* log *(N)

Our technique to SIMDize the Comb sort



Our technique to SIMDize the Comb sort



Our technique to SIMDize the Comb sort

input data

5	3	2	11	9	4	12	1	10	6	8	7

Transposed order

1	4	7	10	2	5	8	11	3	6	9	12

assume four elements in one vector

sorted output data

1	2	3	4	5	6	7	8	9	10	11	12

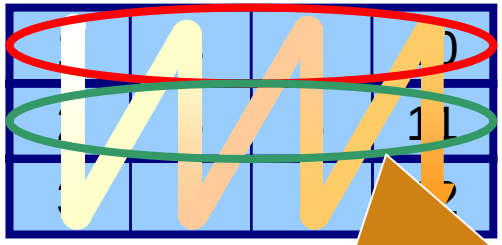
sorted

Our technique to SIMDize the Comb sort

input data

5	3	2	11	9	4	12	1	10	6	8	7

Transposed order



- unaligned access
- loop-carried dependency

- ***no*** unaligned access
- ***no*** loop-carried dependency

sorted output data

1	2	3	4	5	6	7	8	9	10	11	12

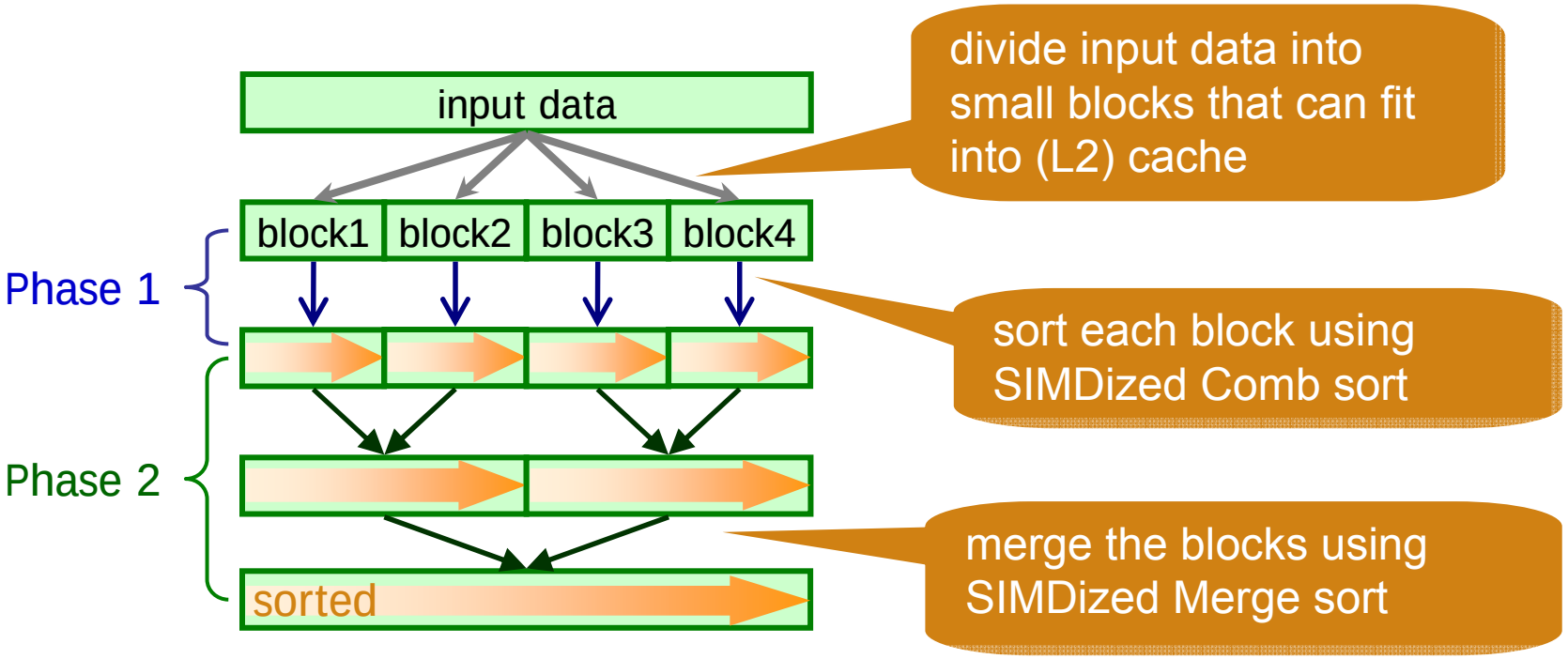
sorted

Analysis of our SIMDized Comb sort

	our SIMDized Comb sort	naively SIMDized Comb sort	original (scalar) Comb sort
Number of unpredictable conditional branches	almost 0	almost 0	$O(N \log(N))$
Number of unaligned memory accesses	almost 0	$O(N \log(N))$	N/A
Computational complexity	$O(N \log(N))$ reordering: $O(N)$	$O(N \log(N))$	$O(N \log(N))$

Overview of the AA-Sort

- It consists of two phases:
  - Phase 1: SIMDized Comb sort
  - Phase 2: SIMDized Merge sort

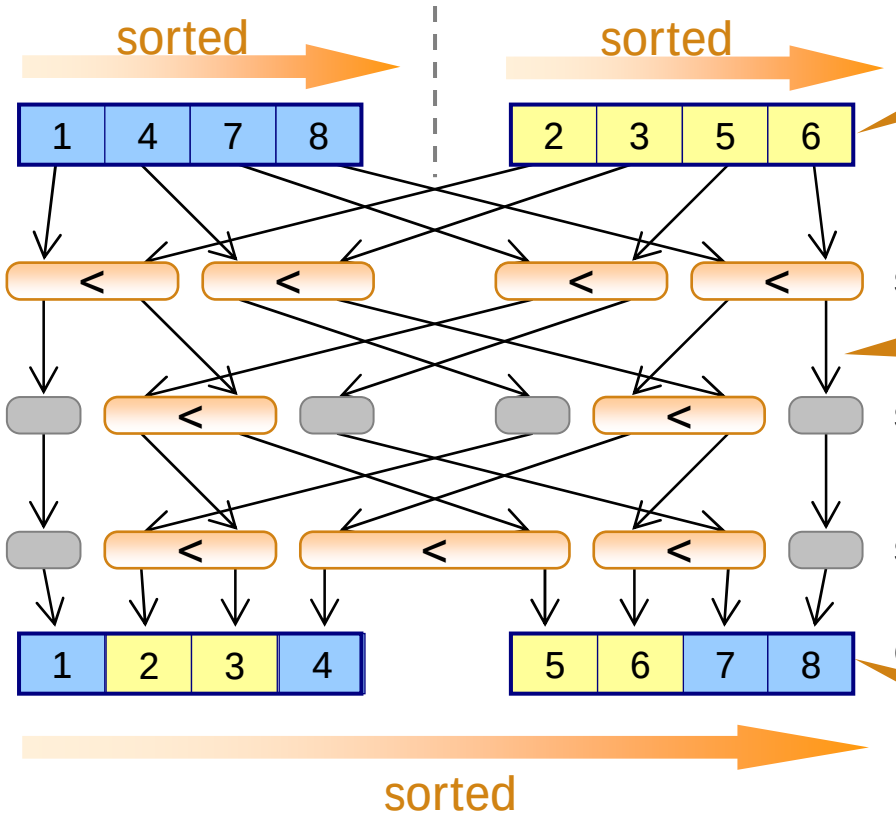


Our approach to SIMDize merge operations

- SIMD instructions are effective for Bitonic merge or Odd-even merge
- Their computational complexity are higher than usual merge operations

- *Our solution*
  - Integrate Odd-even merge into the usual merge operation to take advantage of SIMD instructions while keeping the computational complexity of usual merge operation

Odd-even merge for values in two vector registers



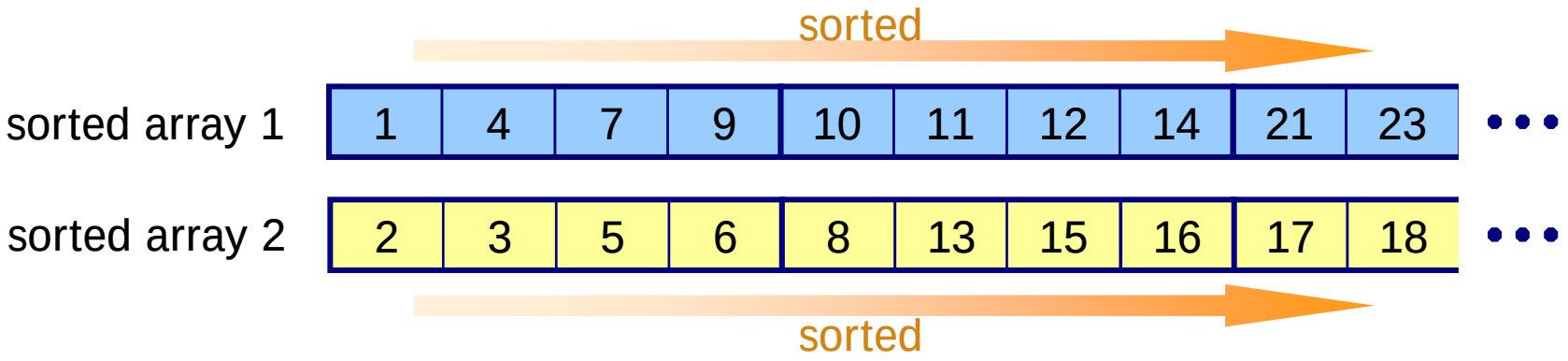
Input
two vector registers contain four presorted values in each

Odd-even Merge
one SIMD comparison and “shuffle” operations for each stage

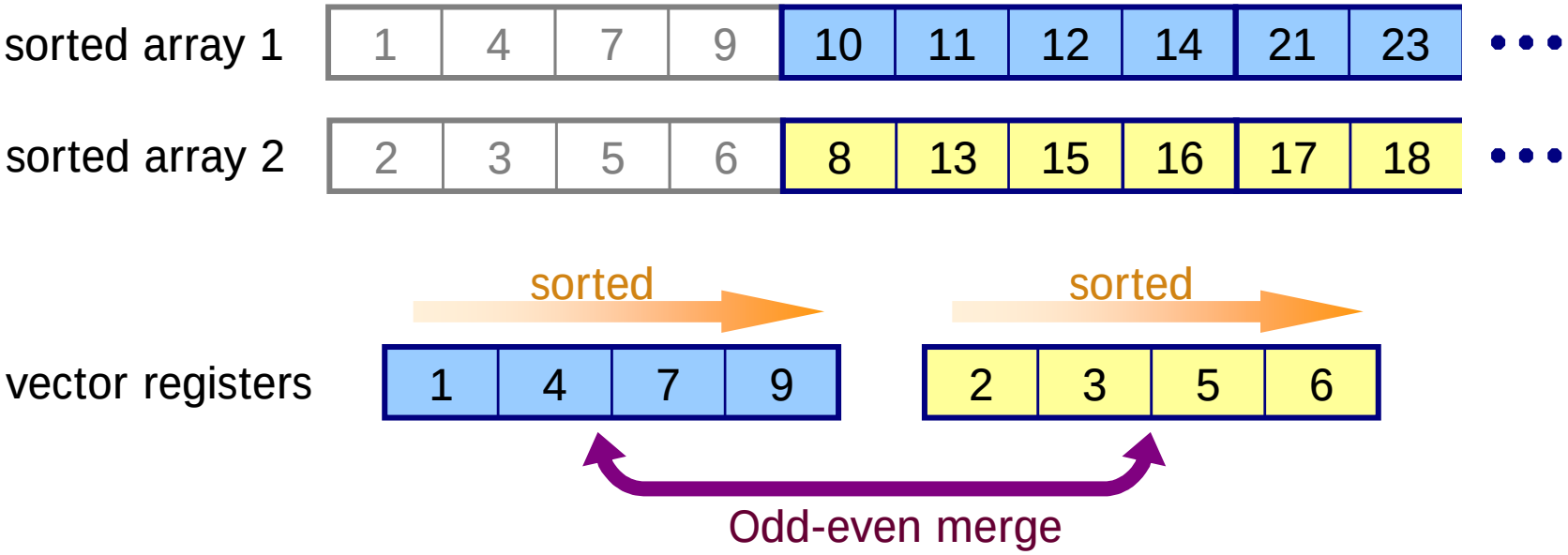
No conditional branches!

Output
eight values in two vector registered are now sorted

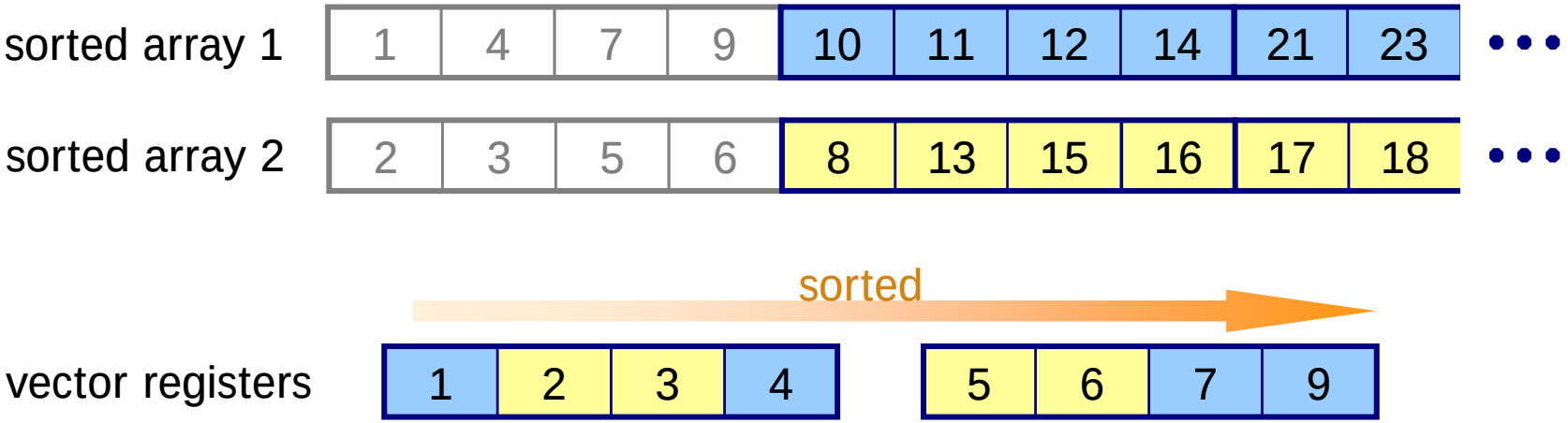
Our technique to integrate Odd-even merge into usual merge



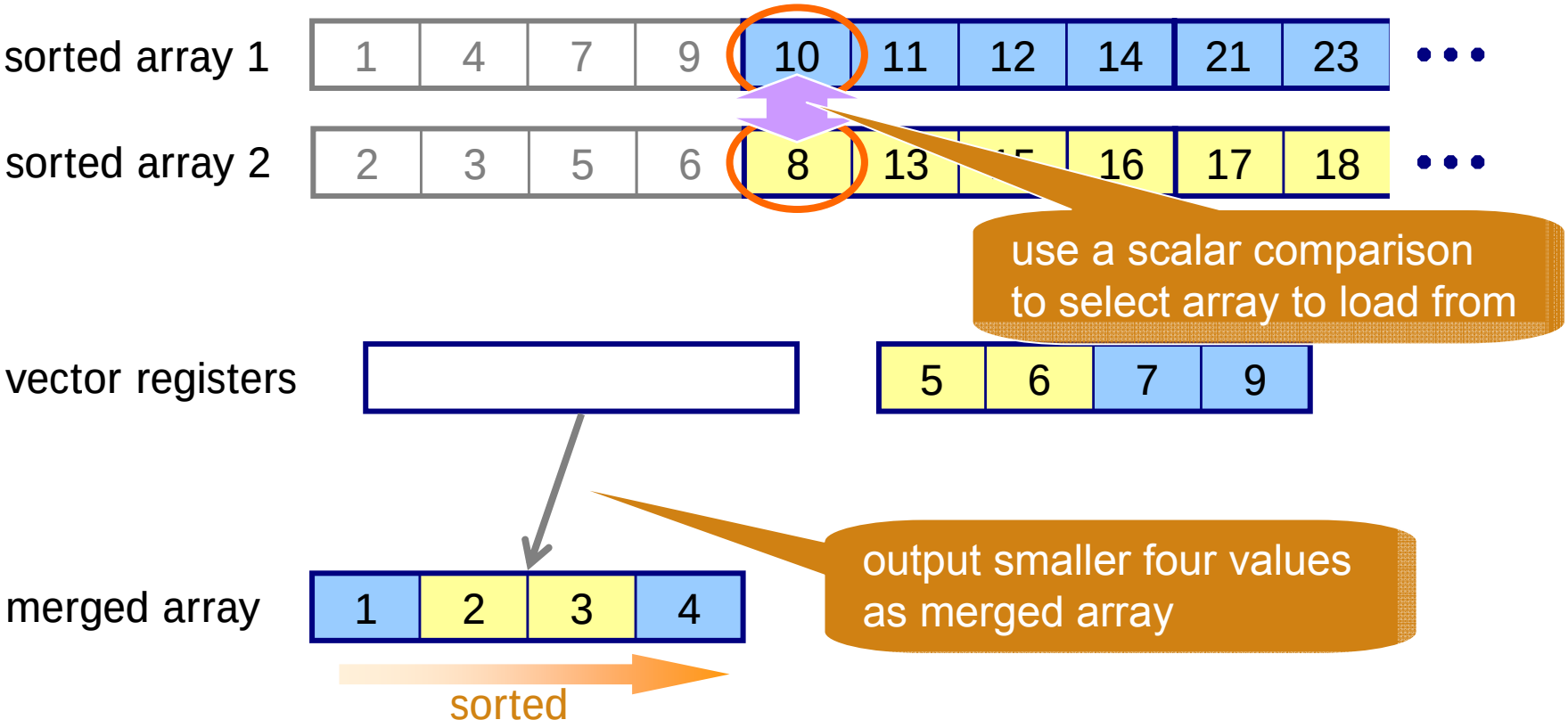
Our technique to integrate Odd-even merge into usual merge



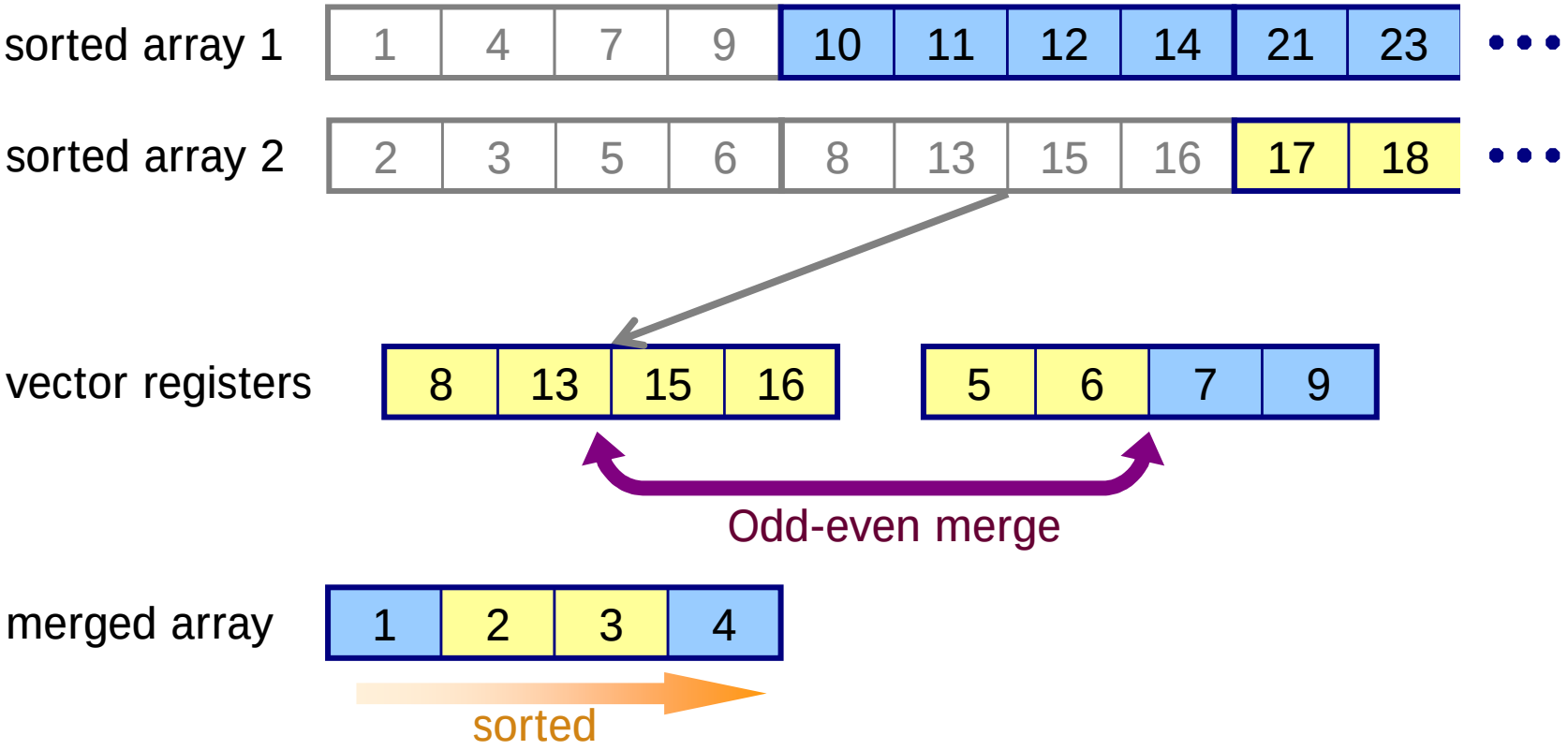
Our technique to integrate Odd-even merge into usual merge



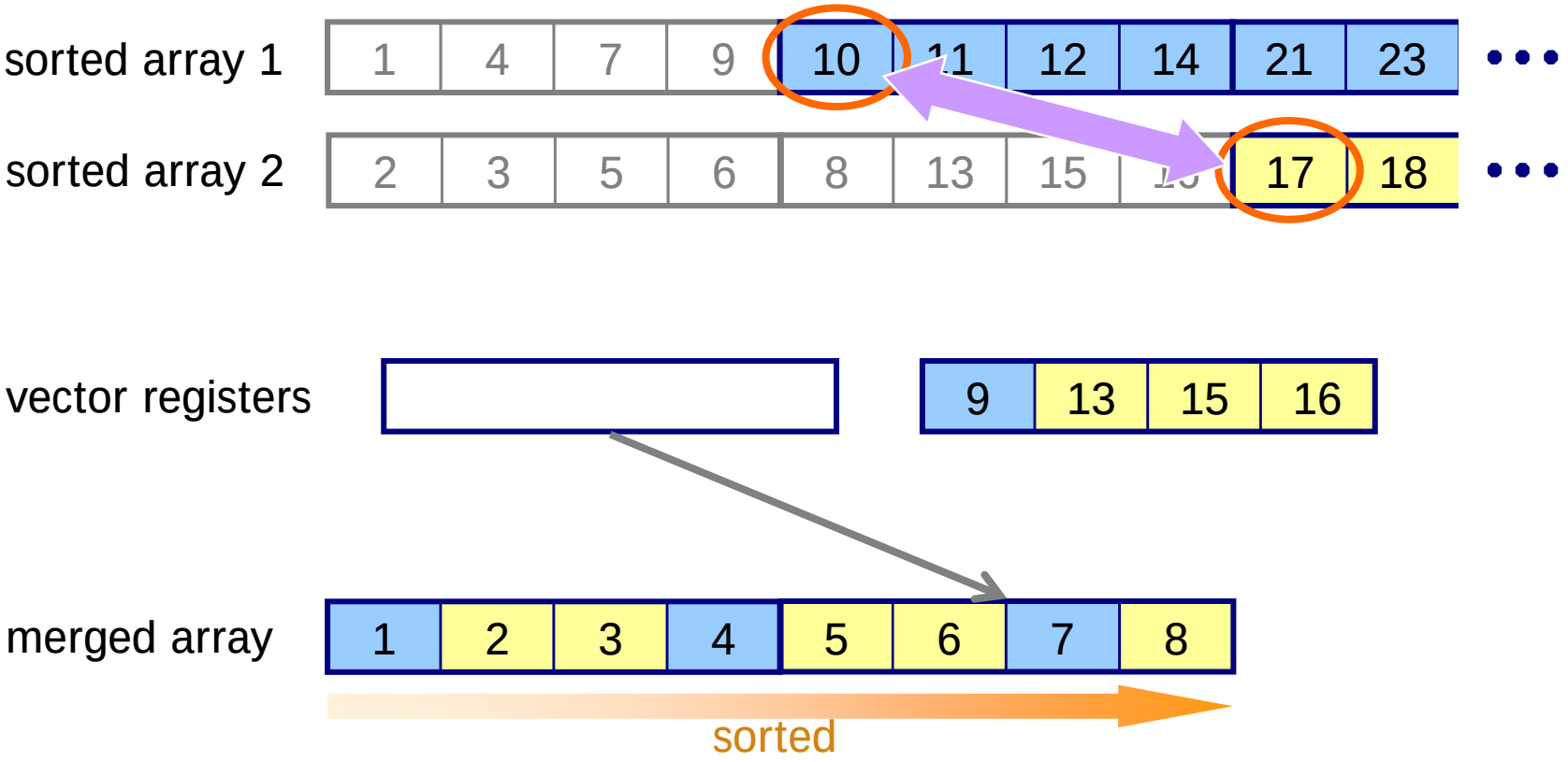
Our technique to integrate Odd-even merge into usual merge



Our technique to integrate Odd-even merge into usual merge



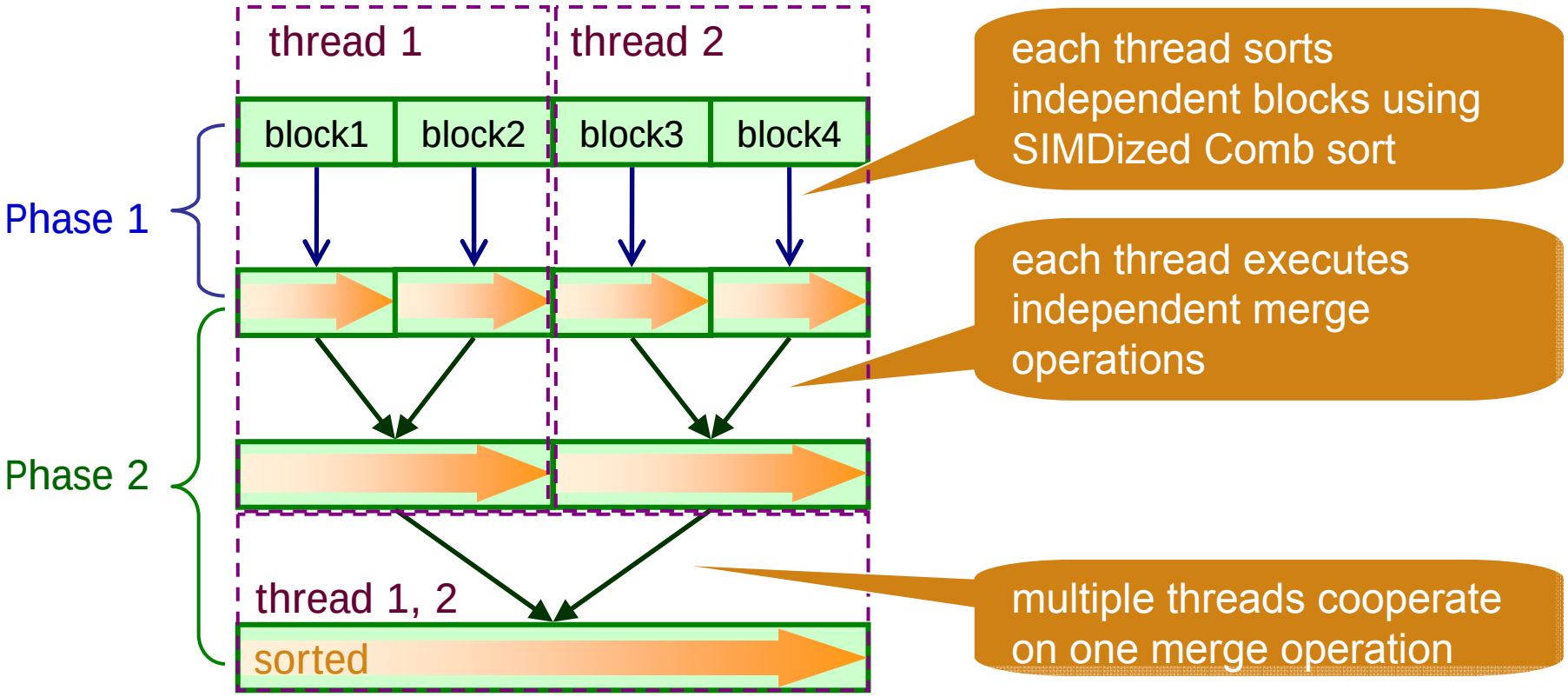
Our technique to integrate Odd-even merge into usual merge



Comparing merge operations

	our integrated merge operation	odd-even merge implemented with SIMD	usual (scalar) merge operation
Number of unpredictable conditional branches	1 for every output *vector*	0	1 for every output *element*
Computational complexity	$O(N)$	$O(N \log(N))$	$O(N)$

Parallelizing AA-sort among multiple threads



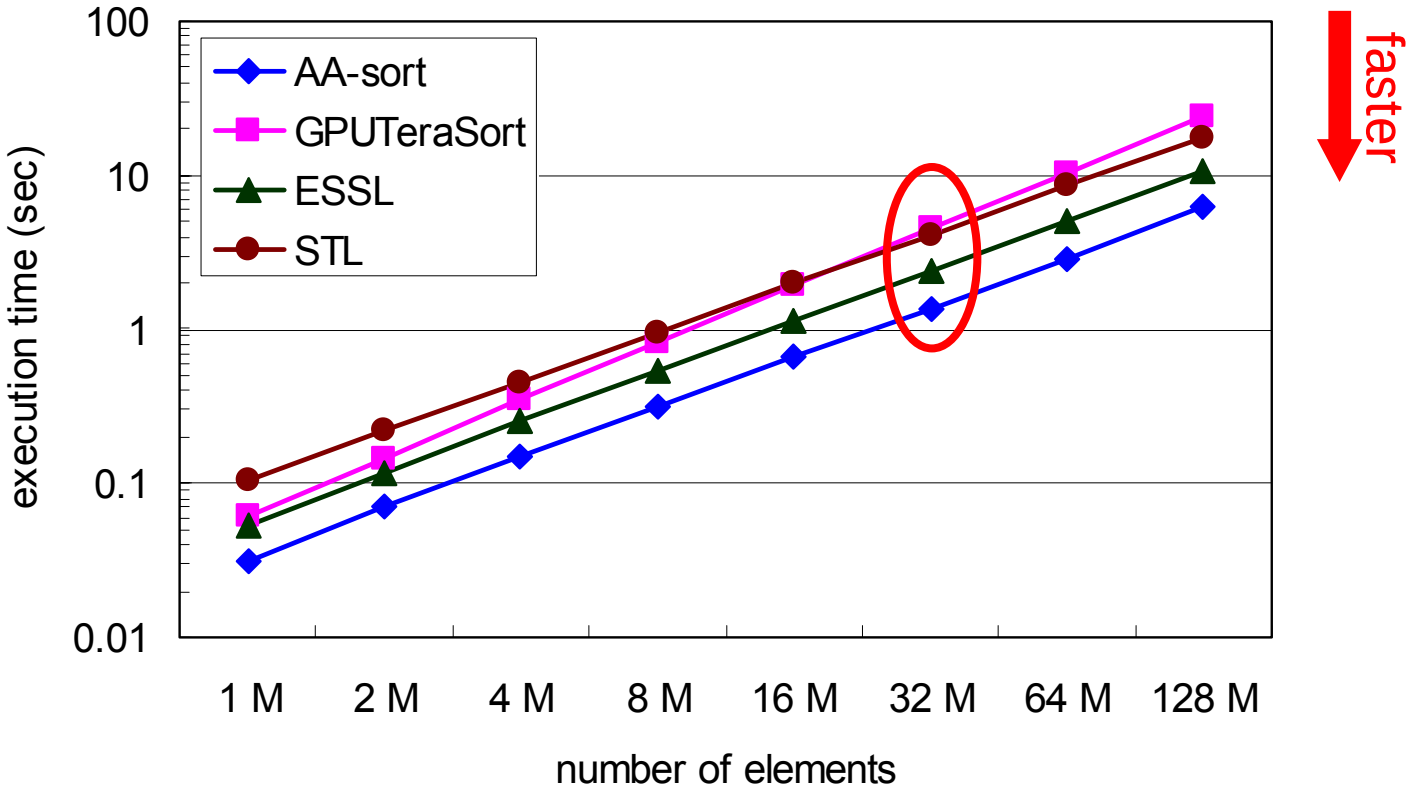
Presentation Outline

- Motivation
- AA-sort: Our new sorting algorithm
- Experimental results
- Summary

Environment for Performance Evaluation

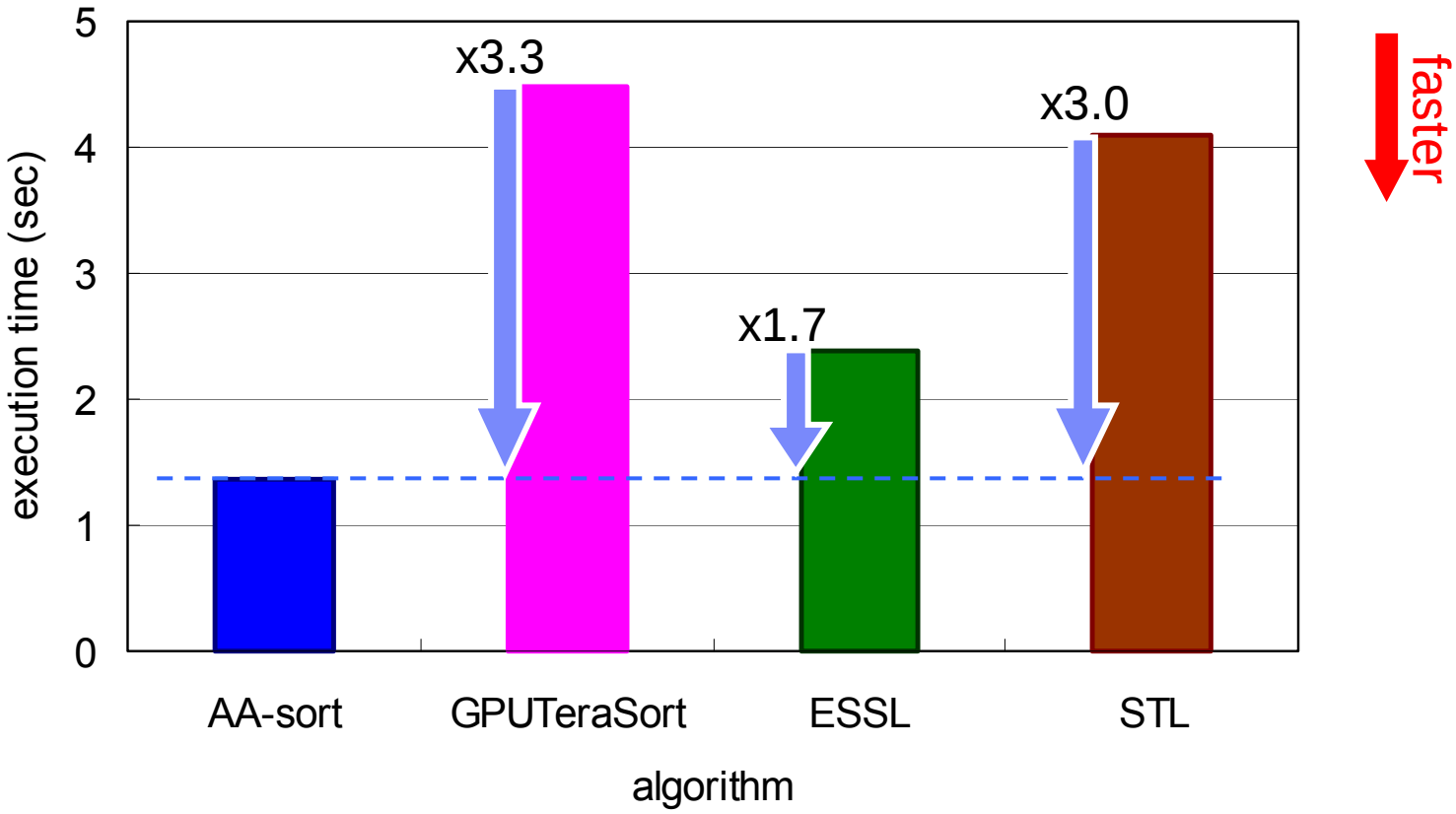
- We used two processors for performance evaluation
  - PowerPC 970 using VMX instructions (up to 4 cores)
  - Cell BE using SPE cores (up to 16 SPE cores)
- We compared the performance of four algorithms
  - AA-sort
  - GPUTeraSort [Govindaraju '05]
  - ESSL (IBM's optimized library)
  - STL delivered with GCC (open source library)

Single-thread performance on PowerPC 970 for various input size



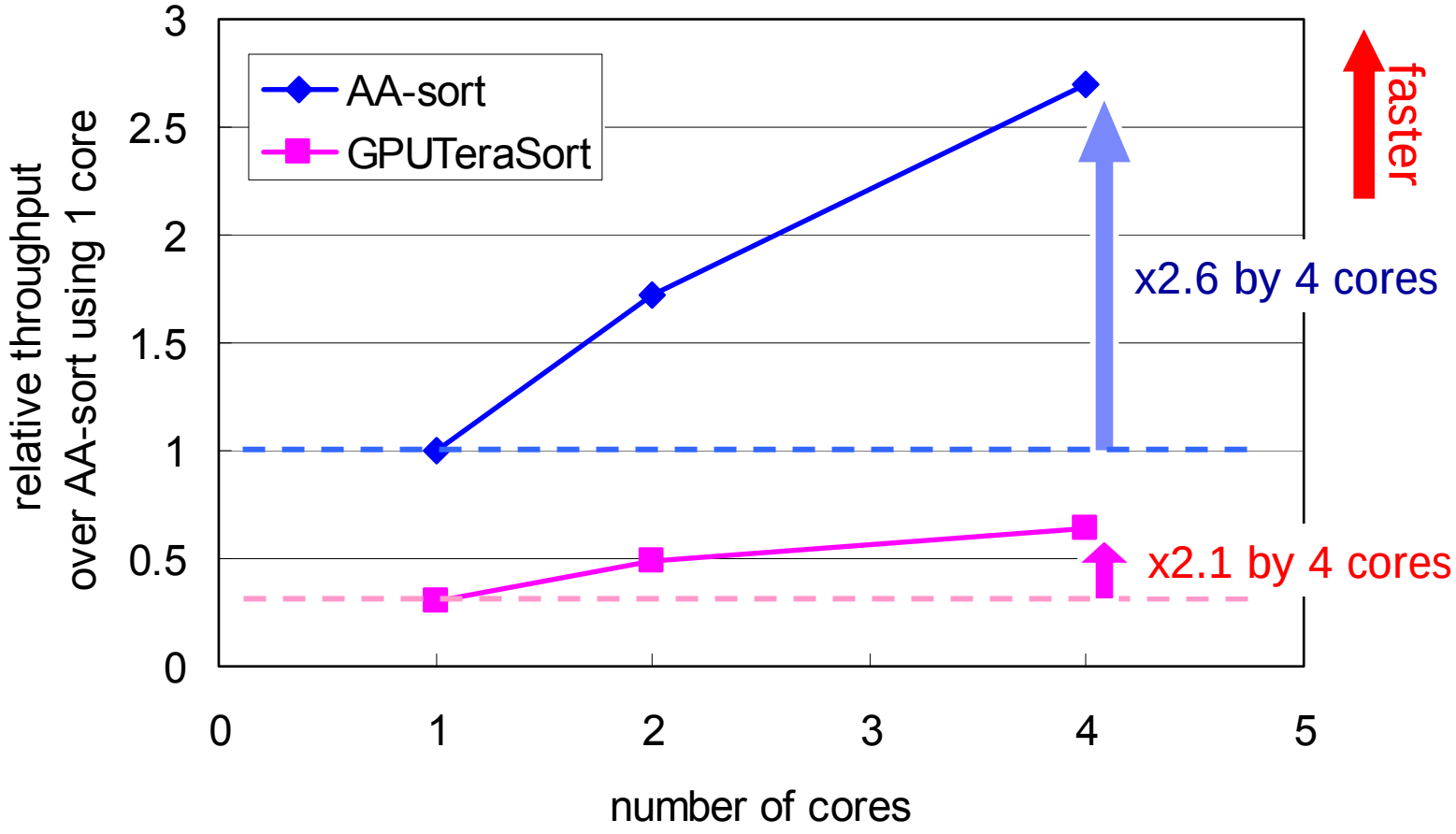
sorting 32-bit random integers on one core of PowerPC 970

Single-thread performance on PowerPC 970



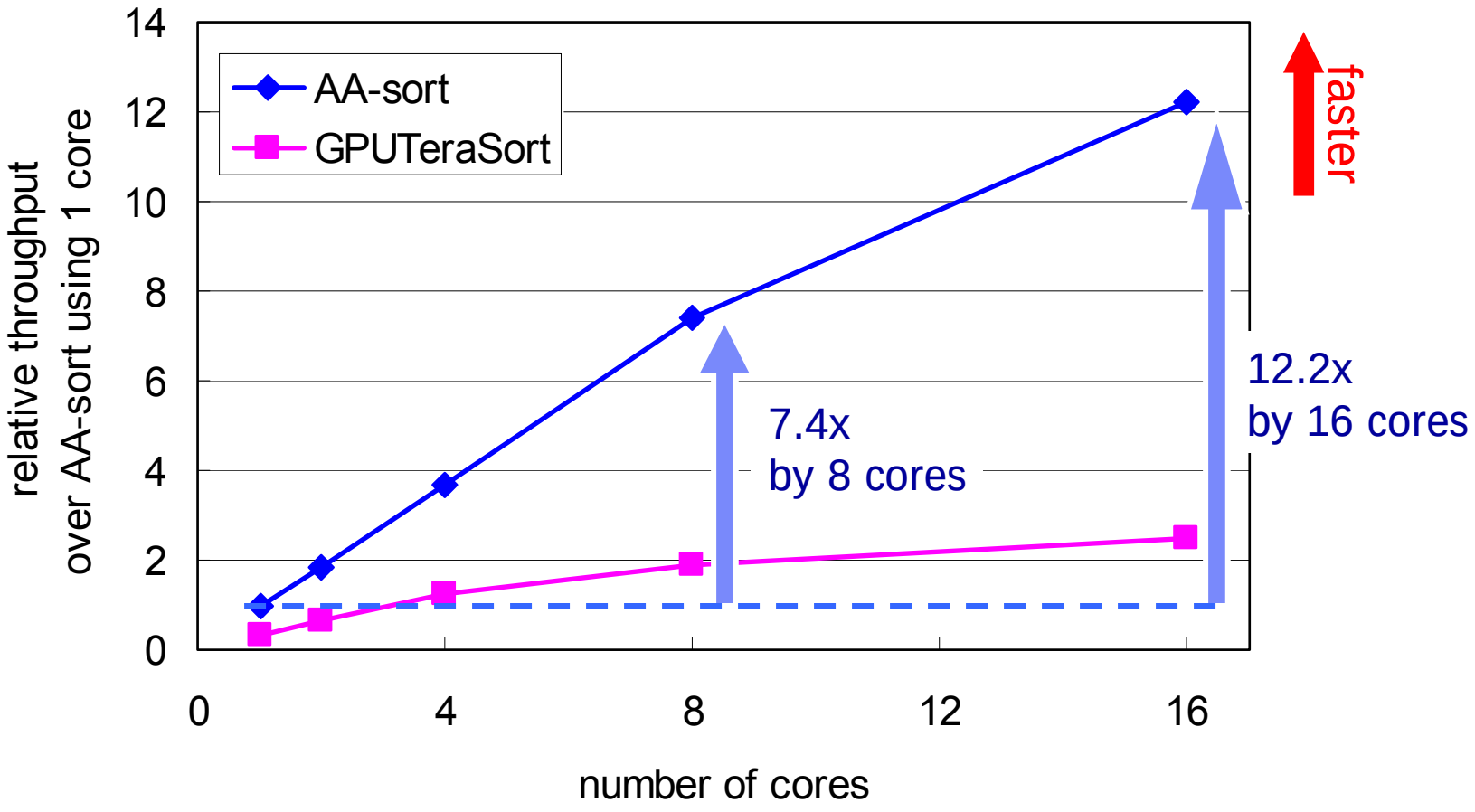
sorting 32 M elements of random 32-bit integers on PowerPC 970

Scalability with multiple cores on PowerPC 970



sorting 32 M elements of random 32-bit integers on PowerPC 970

Scalability with multiple cores on Cell BE



sorting 32 M elements of random 32-bit integers on Cell BE

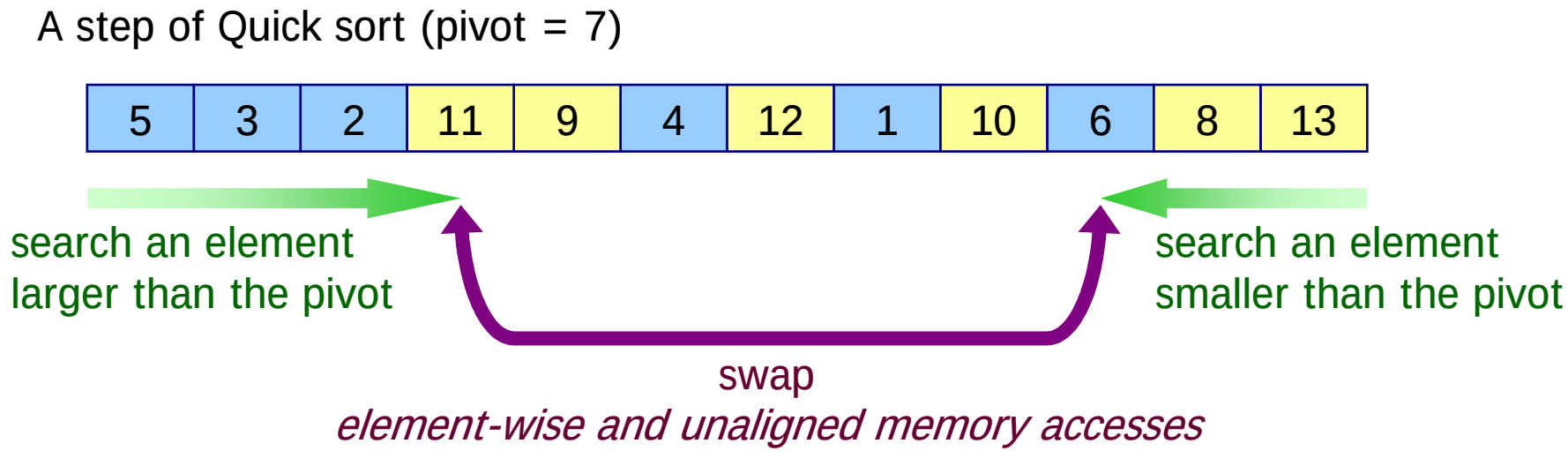
Summary

- We proposed a new sorting algorithm called AA-sort, which can take advantage of
  - SIMD instructions
  - Multiple Cores (thread-level parallelism)
- We evaluated the AA-sort on PowerPC 970 and Cell BE
  - Using only 1 core of PowerPC 970, the AA-sort outperformed
    - IBM's ESSL by 1.7x
    - GPUTeraSort by 3.3x
  - On Cell BE, the AA-sort showed good scalability
    - 8 cores 7.4x
    - 16 cores 12.2x

Thank you for your attention!

SIMD instructions are not effective for Quick sort

- SIMD instructions are ***NOT*** effective for Quick sort, which requires element-wise and unaligned memory accesses



Transposed order

- To see the values to sort as a two dimensional array