Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited

Presenter: Haonan Wang

Slides Credit: CMU 15-721 Spring 2018

haononw@mit.edu

March 19, 2019

Overview

Section 1

Background

Subsection 1

Sort vs. Hash

Sort vs. Hash

There are two main approaches for the PARALLEL JOIN ALGORITHMS:
→ Hash Join
→ Sort-Merge Join

History of Hash VS. Sort

- 1970s Sorting
- 1980s Hashing
- 1990s Equivalent
- 2000s Hashing
- 2010s Hashing (Partitioned vs. Non-Partitioned)
- 2020s ???

What Is Merge-Sort Join

Sort-merge join algorithm explained

Customers table

Id	Login
2	User#2
1	User#1
4	User#4
3	User#3

Orders table

Id	User id
1001	2
1002	4
1003	4
1004	1

Sorting

Id	Login
1	User#1
2	User#2
3	User#3
4	User#4

Id	User id
1004	1
1001	2
1002	4
1003	4

Merging

1. Takes the first row in the left - (1, User#1). Does it have a match in the right ? Yes

SIMD?

What is SIMD?
A class of CPU instructions that allow the processor to perform the same operation on multiple data points simultaneously.

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

Both current AMD and Intel CPUs have ISA and microarchitecture support SIMD operations.
$\rightarrow$ MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, AVX

SIMD Makes Sorting Better Than Hashing?

SORT VS. HASH REVISITED: FAST JOIN IMPLEMENTATION ON MODERN MULTI-CORE CPUS
VLDB 2009

ORACLE intel

→ Hashing is faster than Sort-Merge.
→ Sort-Merge is faster w/ wider SIMD.

DESIGN AND EVALUATION OF MAIN MEMORY HASH JOIN ALGORITHMS FOR MULTI-CORE CPUS
SIGMOD 2011

WISCONSIN UNIVERSITY OF WISCONSIN–MADISON

→ Trade-offs between partitioning & non-partitioning Hash-Join.

MASSIVELY PARALLEL SORT-MERGE JOINS IN MAIN MEMORY MULTI-CORE DATABASE SYSTEMS
VLDB 2012

HyPer

→ Sort-Merge is already faster than Hashing, even without SIMD.

MASSIVELY PARALLEL NUMA-AWARE HASH JOINS
IMDM 2013

HyPer

→ Ignore what we said last year.
→ You really want to use Hashing!

MAIN-MEMORY HASH JOINS ON MULTI-CORE CPUS: TUNING TO THE UNDERLYING HARDWARE
ICDE 2013

Systems@ETH zürich

→ New optimizations and results for Radix Hash Join.

AN EXPERIMENTAL COMPARISON OF THIRTEEN RELATIONAL EQUI-JOINS IN MAIN MEMORY
SIGMOD 2016

UNIVERSITÄT DES SAARLANDES

→ Hold up everyone! Let's look at everything for real!

Section 2

Merge - Sort Join

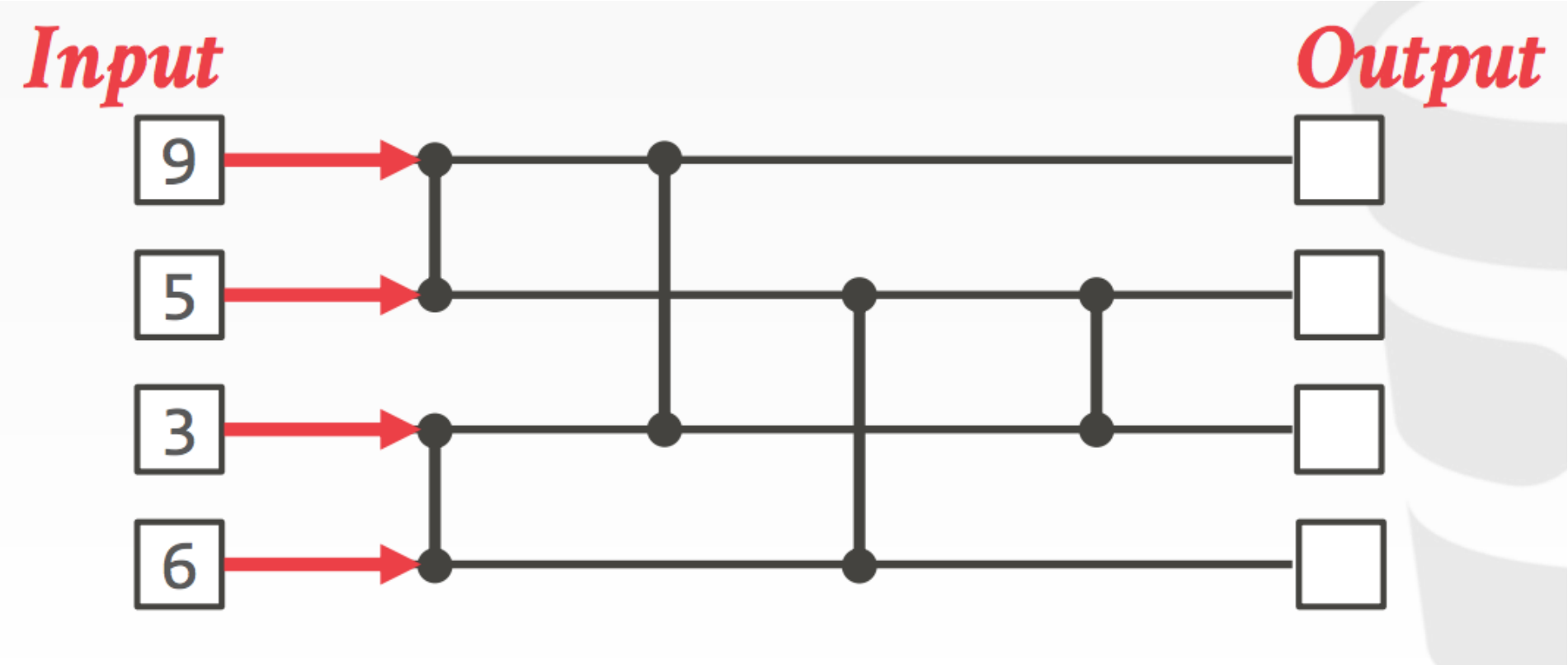
The basic idea for the designing

- Partition Phase(Optional)
  $\rightarrow$ Partition R and assign them to workers / cores.
- Sort Phase
  $\rightarrow$ Sort the tuples of R and S based on the join key.
- Merge Phase
  $\rightarrow$ Scan the sorted relations and compare tuples.
  $\rightarrow$ The outer relation R only needs to be scanned once.

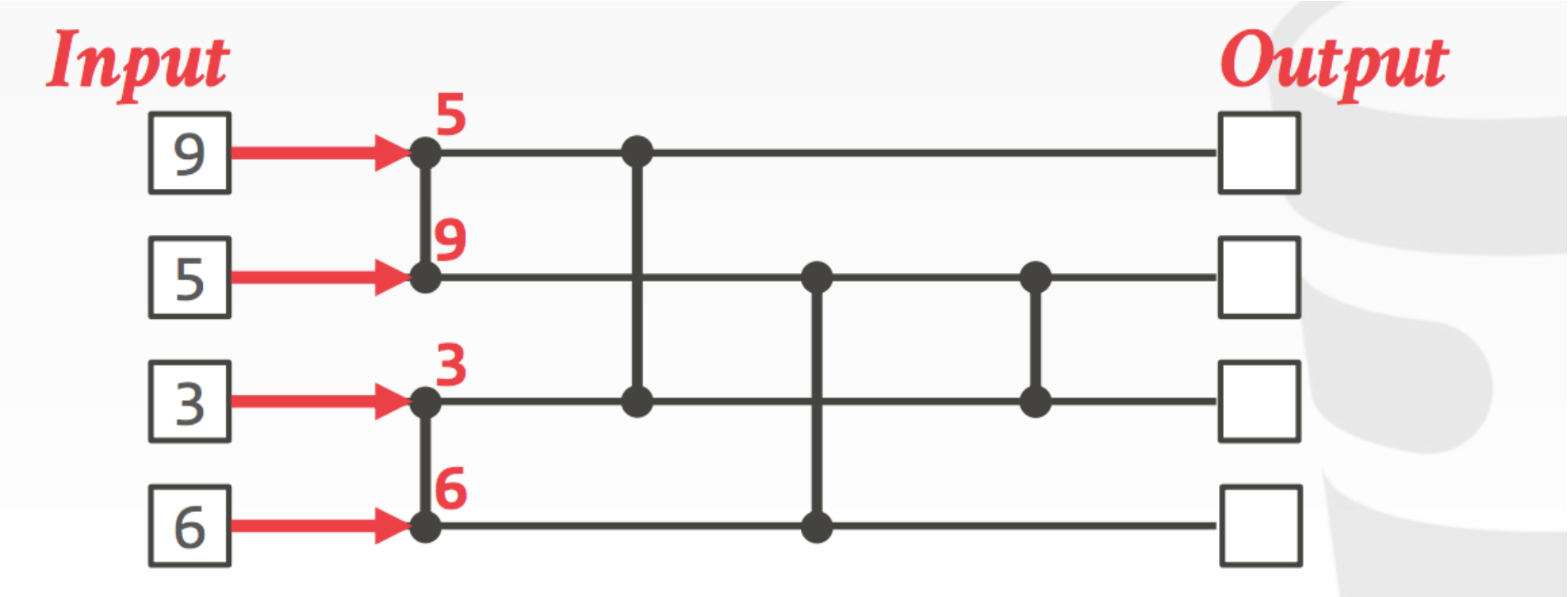
Subsection 2

Sort Phase

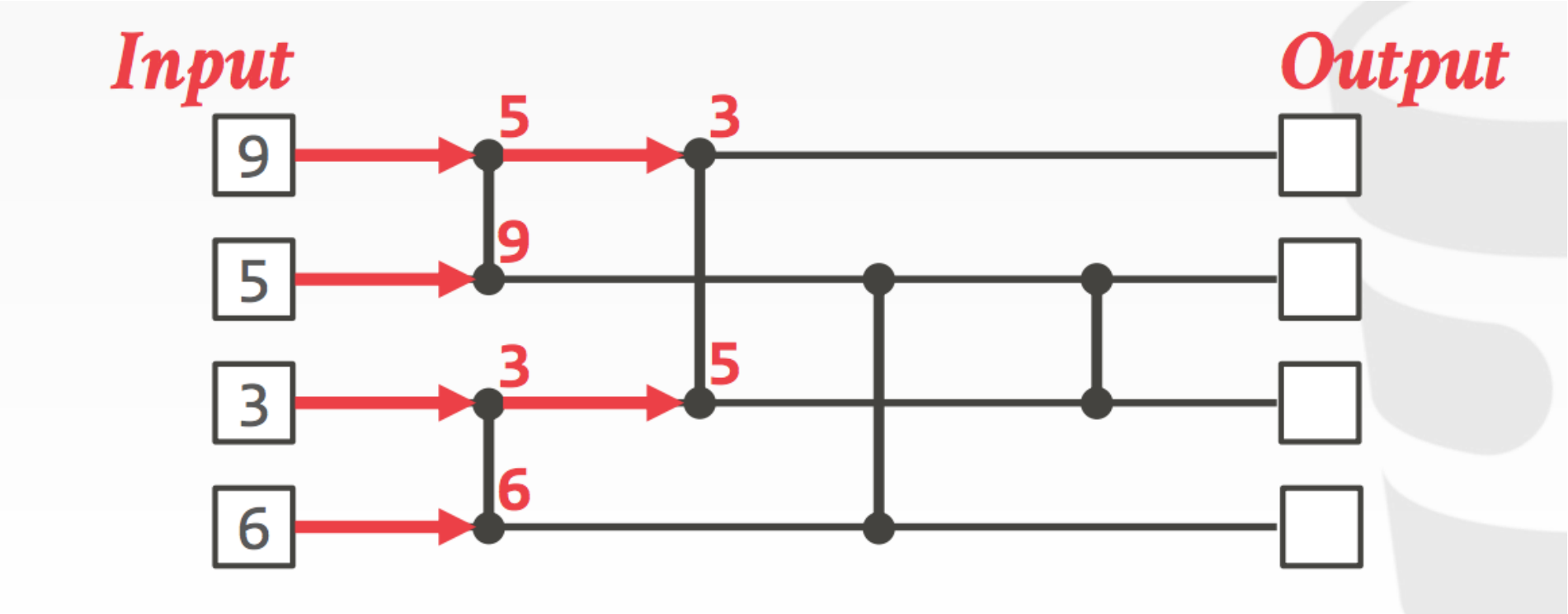
Sorting Networks(1)



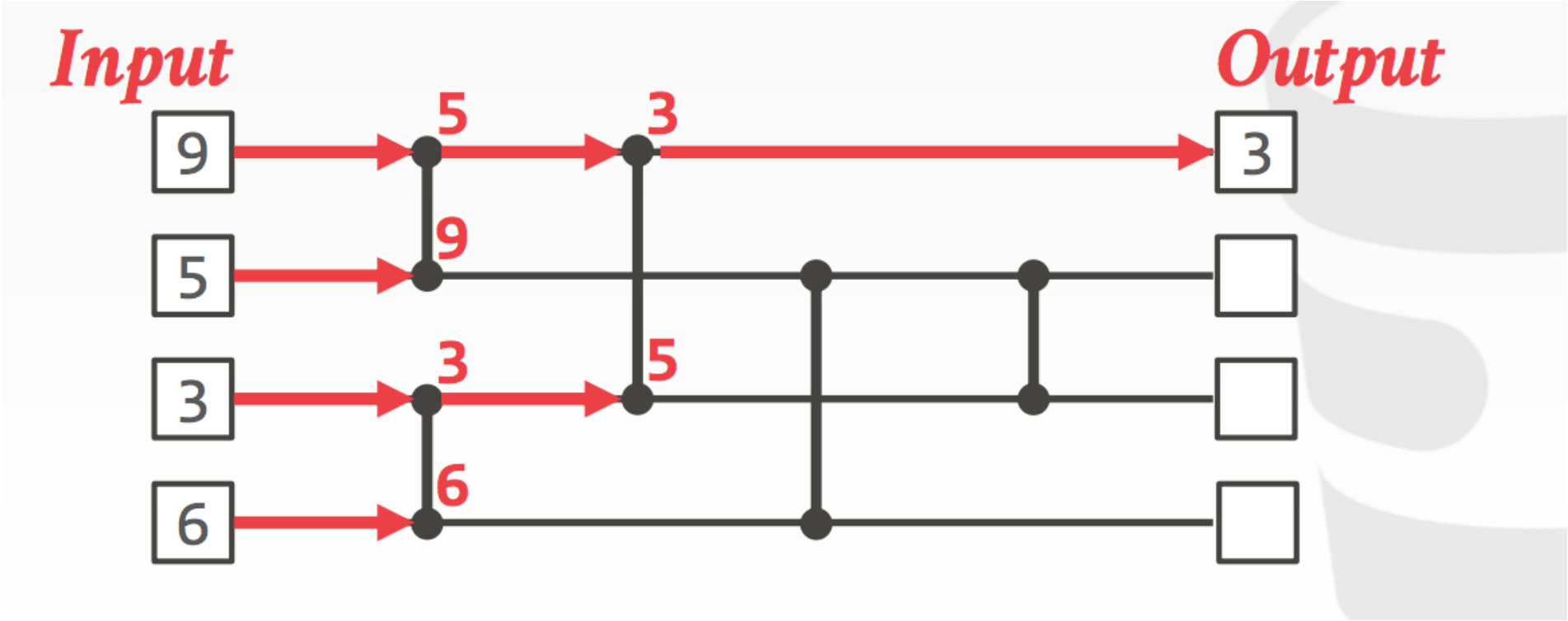
Sorting Networks(2)



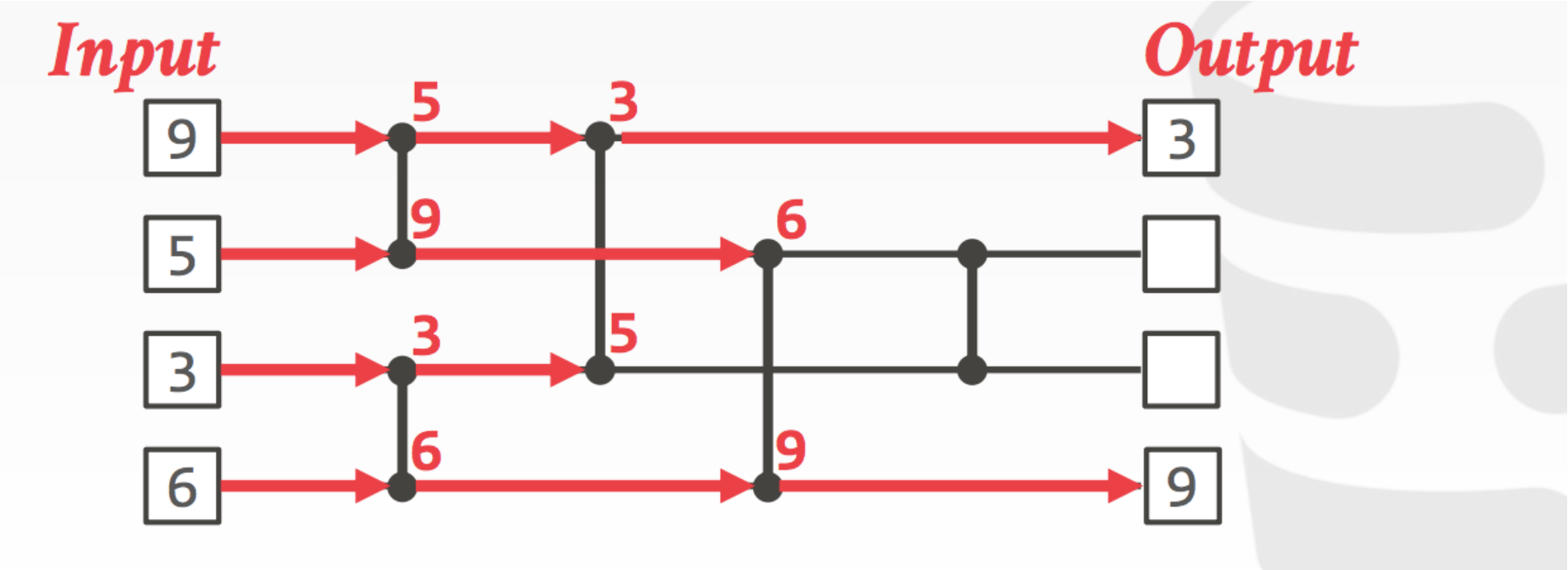
Sorting Networks(3)



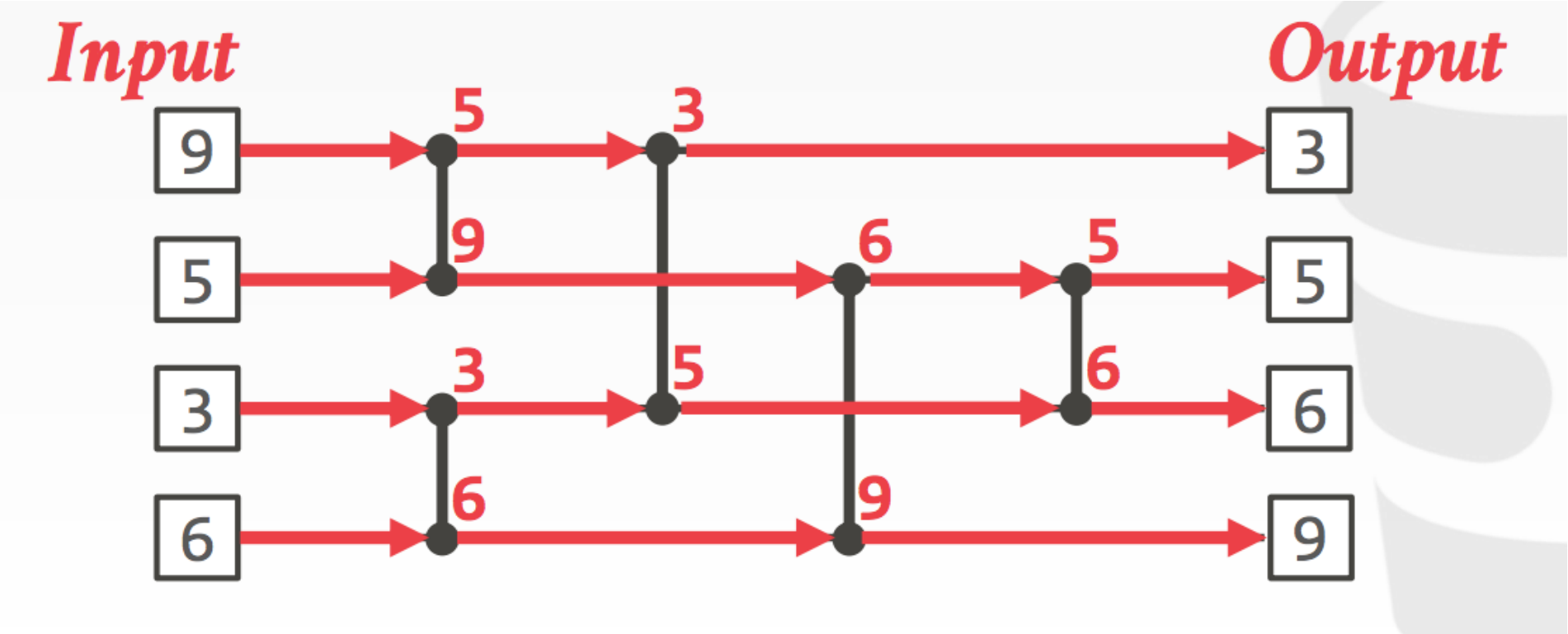
Sorting Networks(4)



Sorting Networks(5)



Sorting Networks(6)



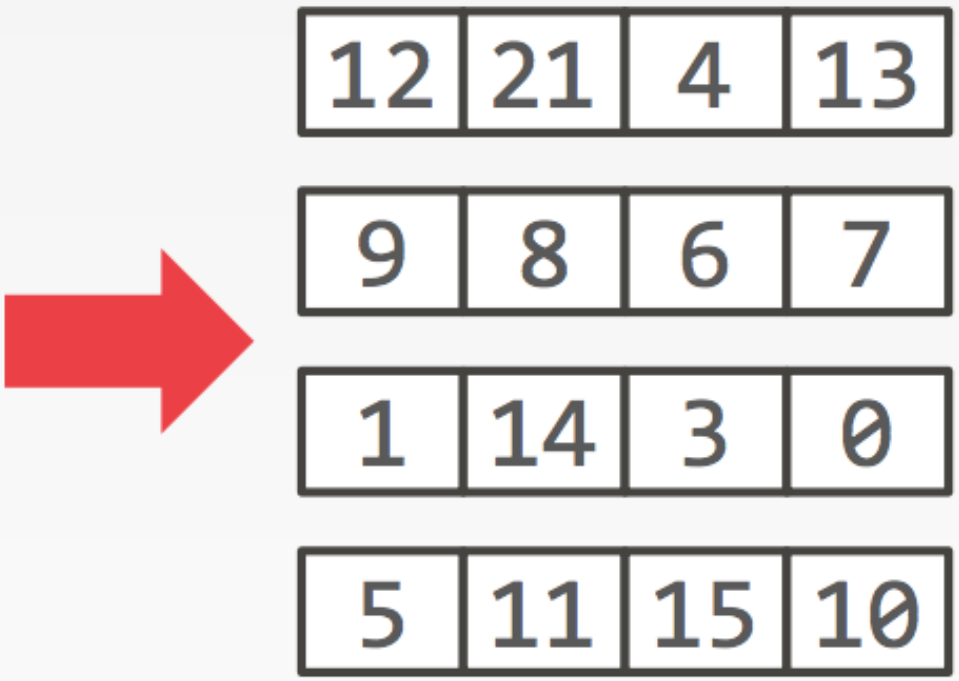
Sorting Networks Summary(1)

```
e=min(a,b)
f=max(a,b)
g=min(c,d)
h=max(c,d)
i=max(e,g)
j=min(f,h)
w=min(e,g)
x=min(i,j)
y=max(i,j)
z=max(f,h)
```

Sorting Networks Summary(2)

- Always has fixed wiring paths for lists with the same number of elements.
- Efficient to execute on modern CPUs because of limited data dependencies and no branches.

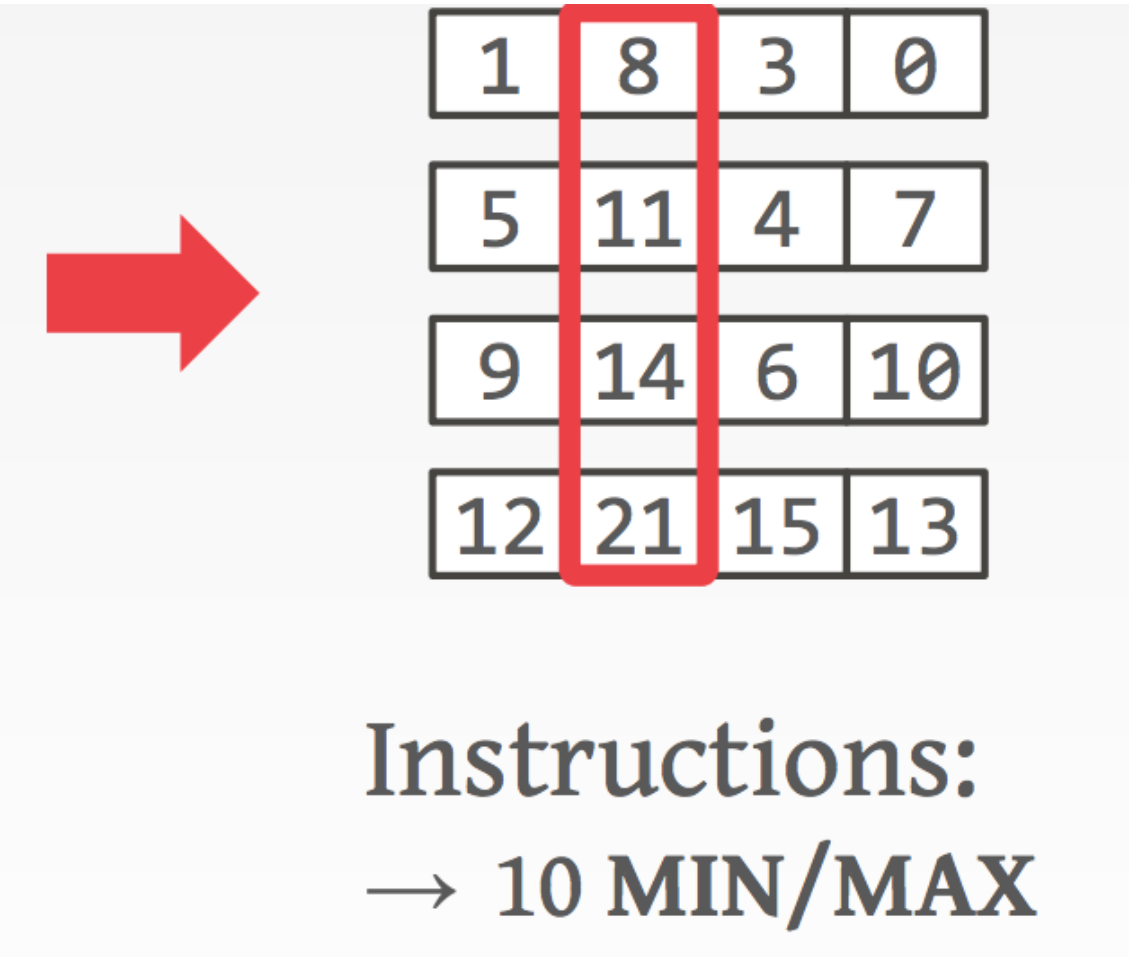
Sorting Network Speed Up With SIMD(1)



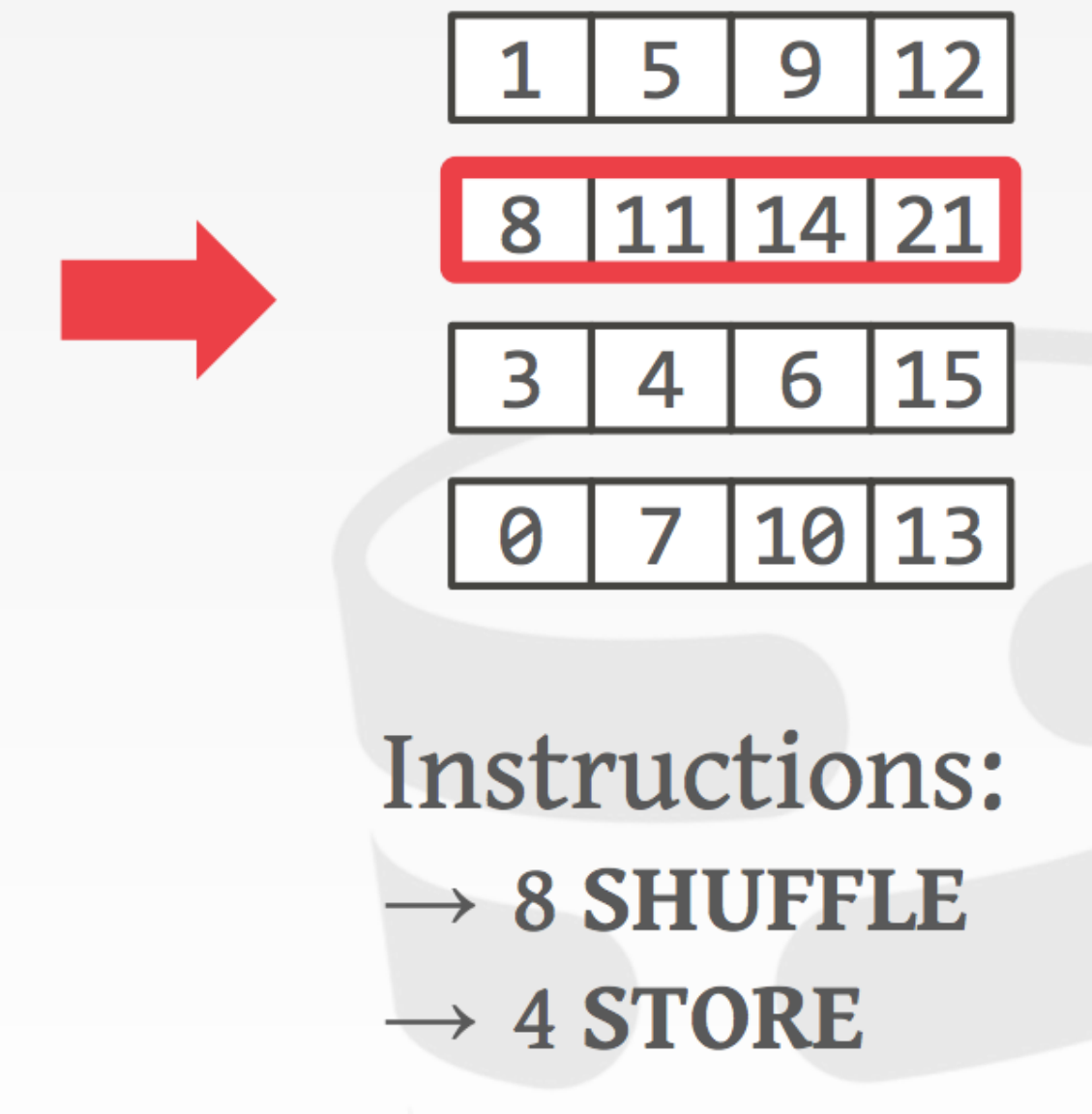
Instructions:

→ 4 **LOAD**

Sorting Network Speed Up With SIMD(2)



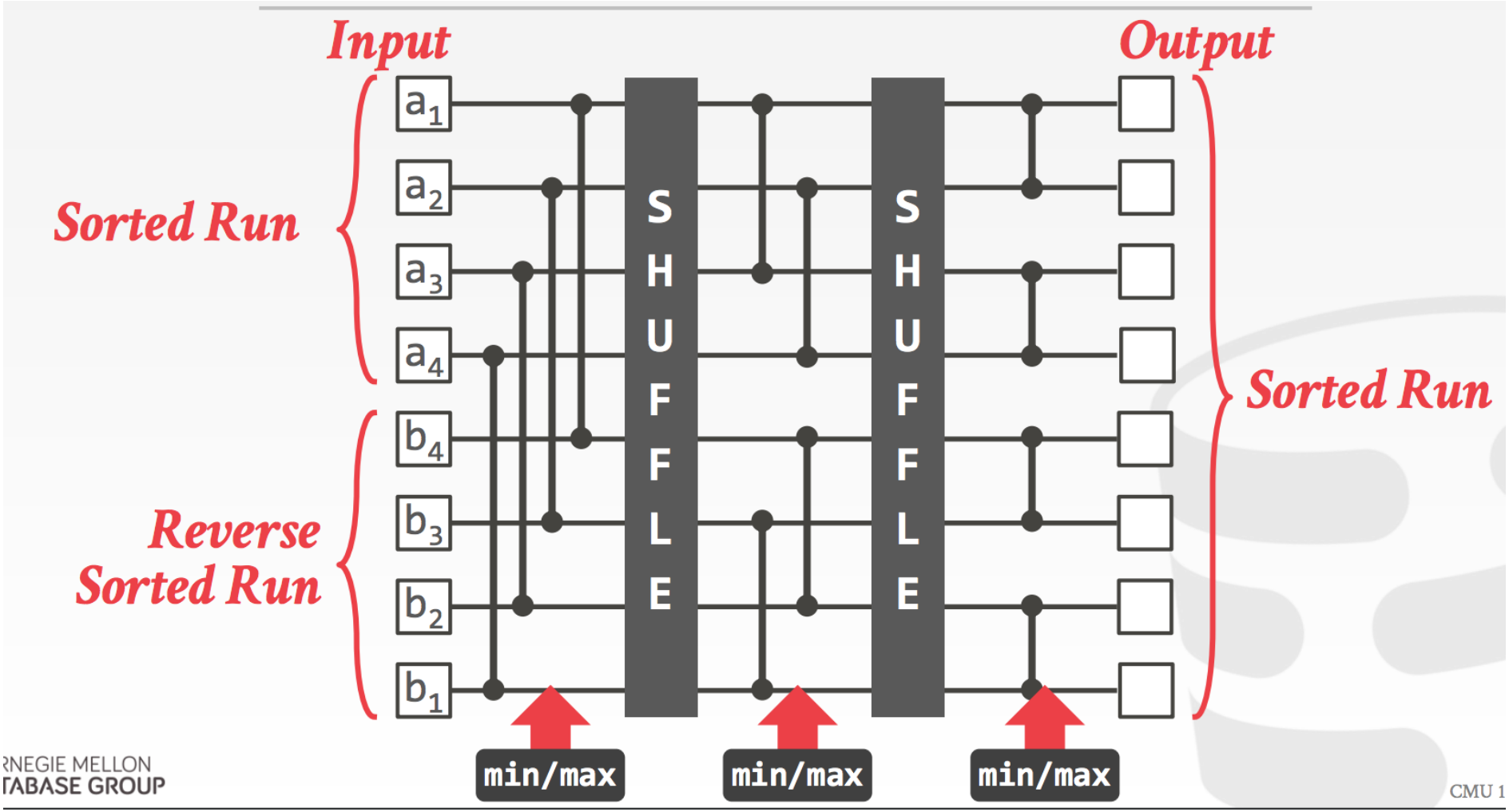
Sorting Network Speed Up With SIMD(3)



Subsection 3

Merge Phase

Bitonic Merge Networks



Merging Larger Lists using Bitonic Merge

Algorithm 1: Merging larger lists with help of bitonic merge kernel `bitonic_merge4()` ($k = 4$).

```
a ← fetch4(in_1); b ← fetch4(in_2);
repeat
    ⟨a, b⟩ ← bitonic_merge4(a, b);
    emit a to output;
    if head(in_1) < head(in_2) then
        a ← fetch4(in_1);
    else
        a ← fetch4(in_2);
until eof(in_1) or eof(in_2);
⟨a, b⟩ ← bitonic_merge4(a, b);
emit4(a); emit4(b);
if eof(in_1) then
    emit rest of in_2 to output;
else
    emit rest of in_1 to output;
```

Merging-Sort Tree

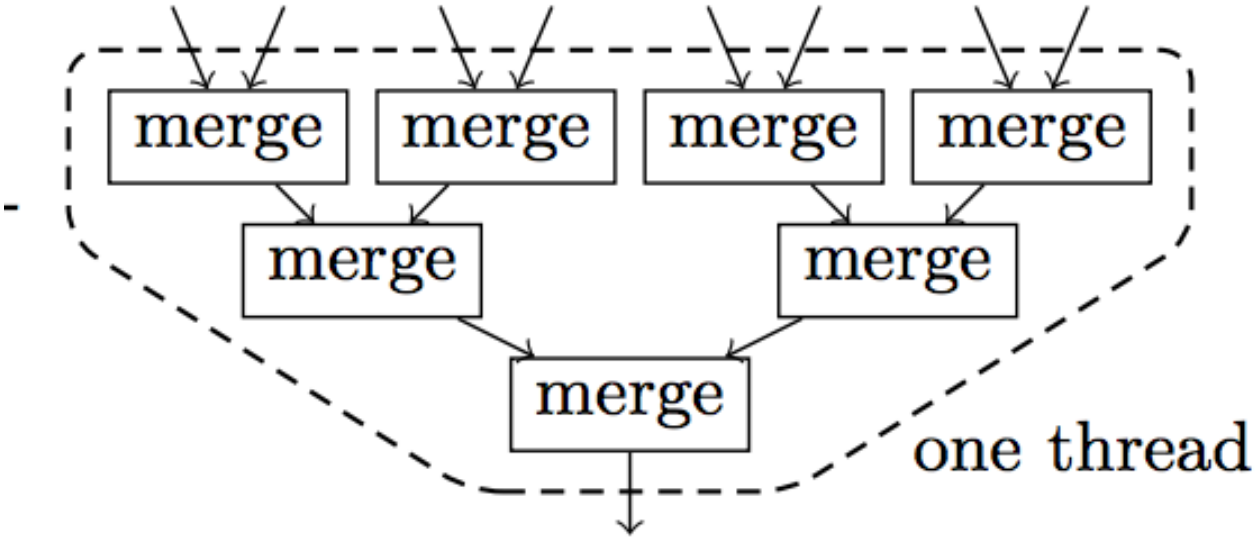


Figure 3: Multi-way merging.

Merging-Sort Hierarchy(Summary)

- *in-register sorting*, with runs that fit into (SIMD) CPU registers;
- *in-cache sorting*, where runs can still be held in a CPU-local cache;
- *out-of-cache sorting*, once runs exceed cache sizes.

Subsection 4

Multi-Way Merge

Impact Of Numa

- In practice, at least some merging passes will inevitably cross NUMA boundaries.
- multisocket systems show an increasing asymmetry, where the NUMA interconnect bandwidth stays further and further behind the aggregate memory bandwidth that the individual memory controllers could provide.

m-way

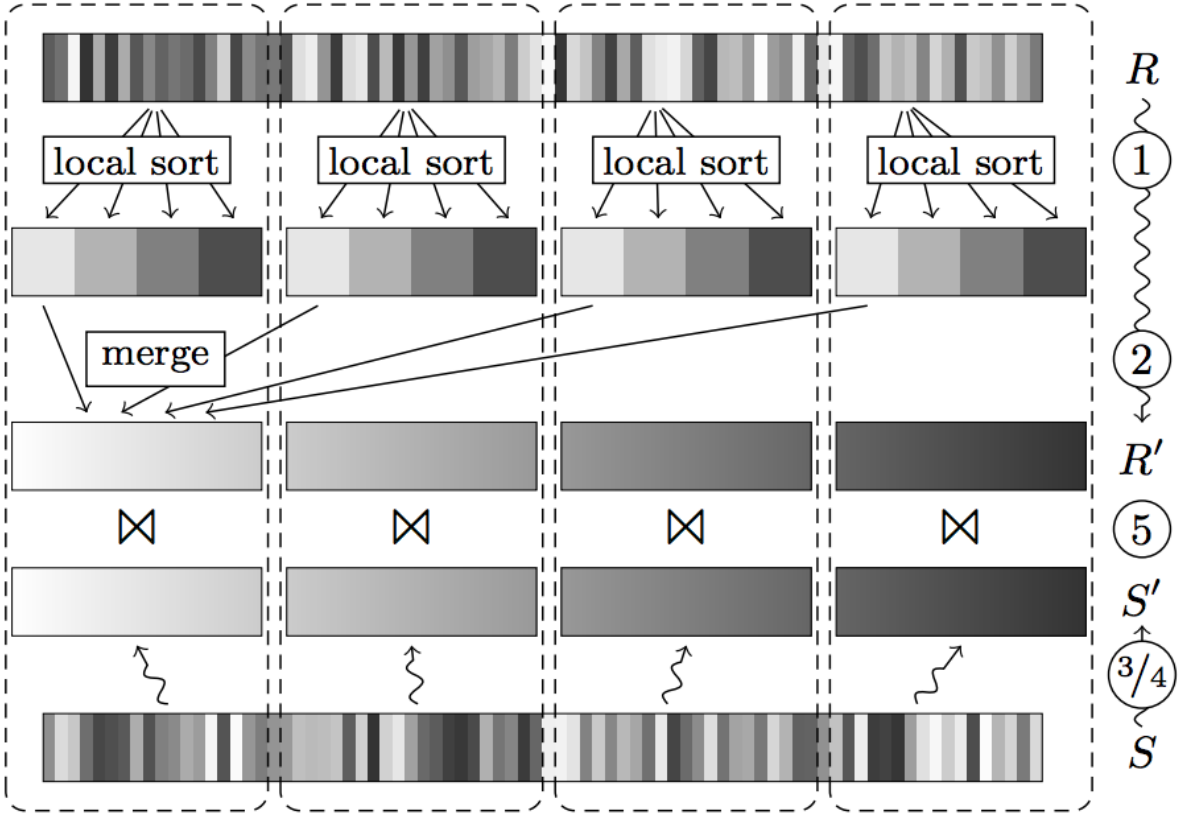


Figure 4: *m-way*: NUMA-aware sort-merge join with multi-way merge and SIMD.

Section 3

Experiment

Settings

- Intel Sandy Bridge with a 256-bit AVX instruction set.
- Four-socket configuration, with each CPU socket containing 8 physical cores and 16 thread contexts by the help of the hyper-threading.
- Cache sizes are 32 KiB for L1, 256 KiB for L2, and 20 MiB L3 (the latter shared by the 16 threads within the socket).The cache line size of the system is 64 bytes. TLB1 contains 64/32 entries when using 4 KiB/2 MiB pages (respectively) and 512 TLB2 entries (page size 4 KiB). Total memory available is 512 GiB (DDR3 at 1600 MHz).

Scalability

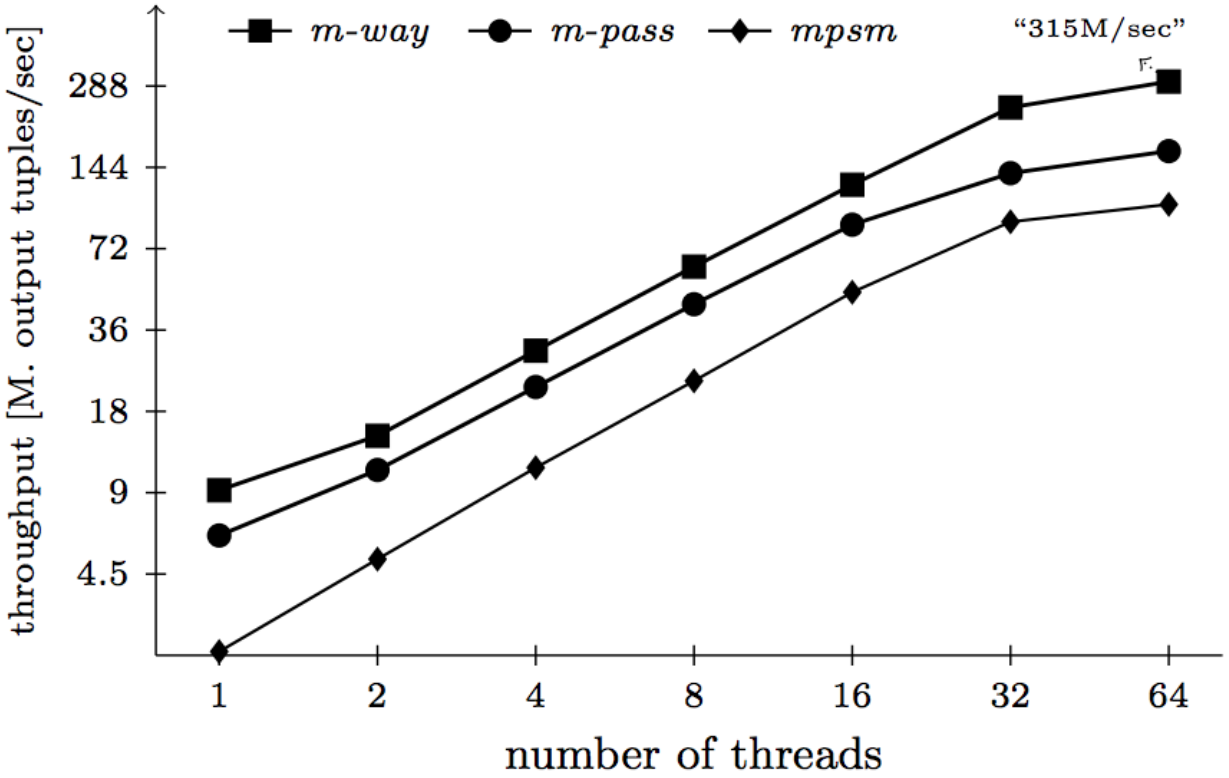
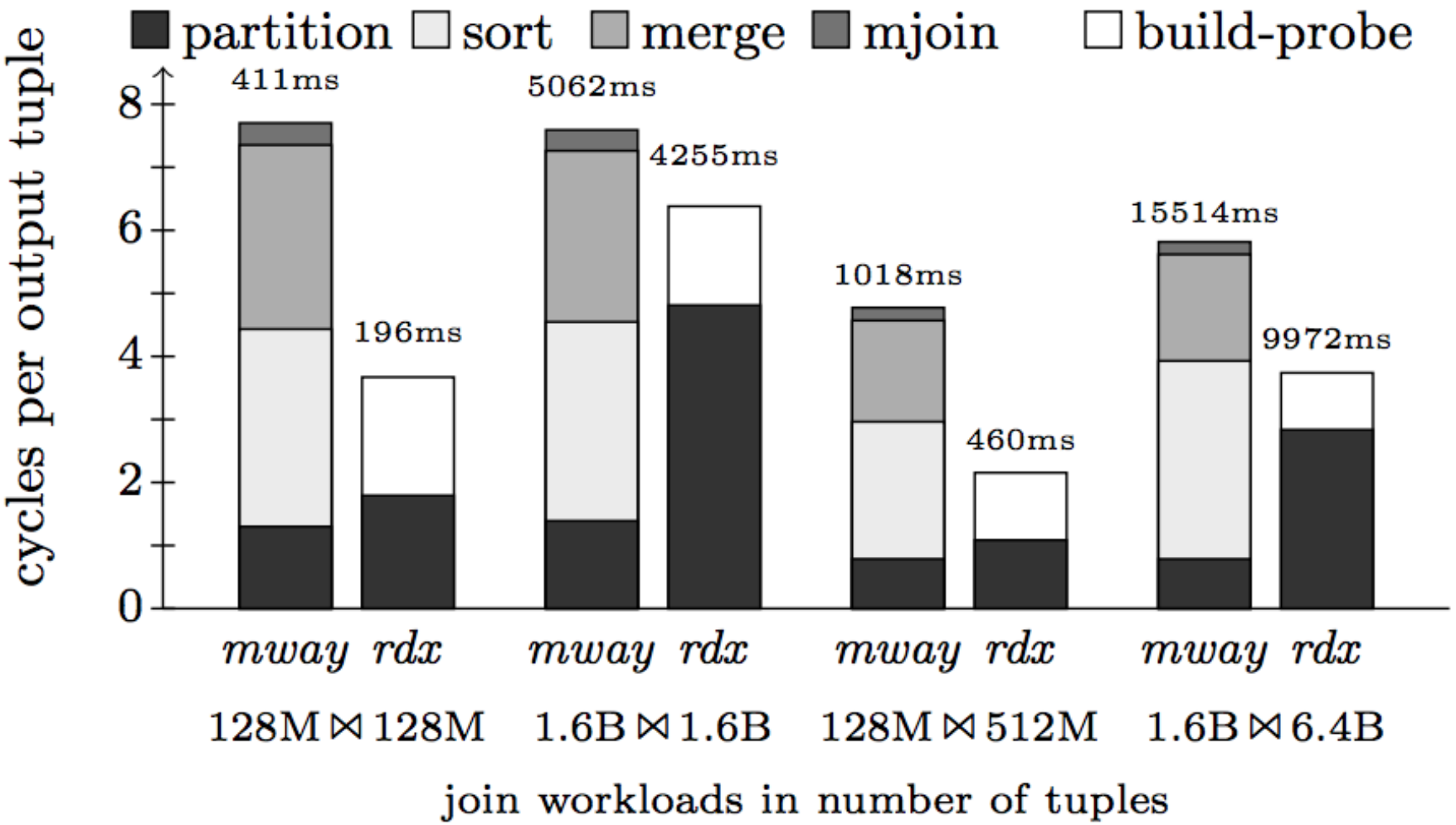


Figure 13: Scalability of sorting-based joins. Workload A, (11.92 GiB ⋈ 11.92 GiB). Throughput metric is output tuples per second, *i.e.* $|S|$/execution time.

Result(1)


partition
sort
merge
mjoin
build-probe
cycles per output tuple
0
2
4
6
8
411ms
196ms
5062ms
4255ms
1018ms
460ms
15514ms
9972ms
mway rdx
mway rdx
mway rdx
mway rdx
128M ⋈ 128M
1.6B ⋈ 1.6B
128M ⋈ 512M
1.6B ⋈ 6.4B
join workloads in number of tuples

Result(2)

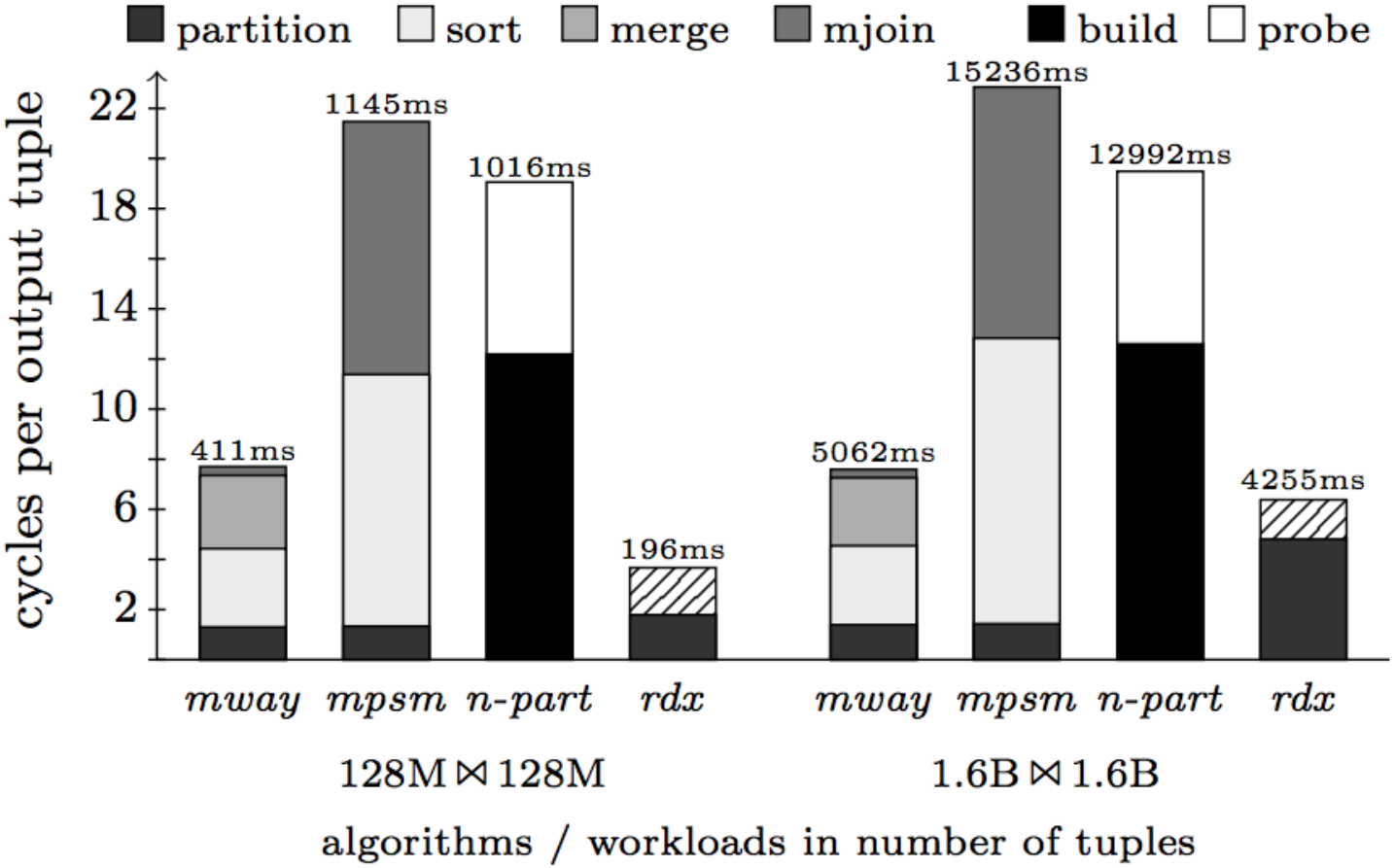


Figure 18: Sort vs. hash join comparison with extended set of algorithms. All using 64 threads.

The End