Origami: A High-Performance Mergesort Framework

Arif Arman and Dmitri Loguinov
Texas A&M University

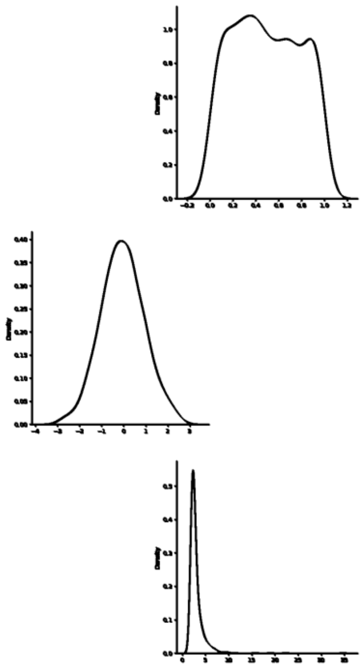
Agenda

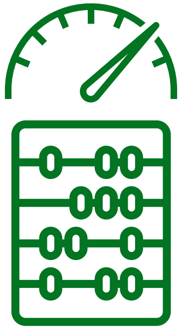
- **Introduction**
- Pipeline Overview
- Tiny Sorters
- In-cache Merge
- Out-of-cache Merge
- Experiments

Motivation

» Mergesort is highly appealing in real-world sorting tasks for several reasons

- Distribution insensitive





MSB Radixsort

Poor unless uniform

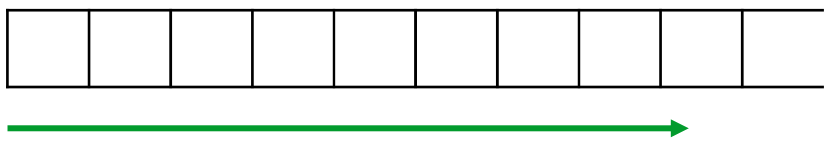
Quicksort
Samplesort
Combsort

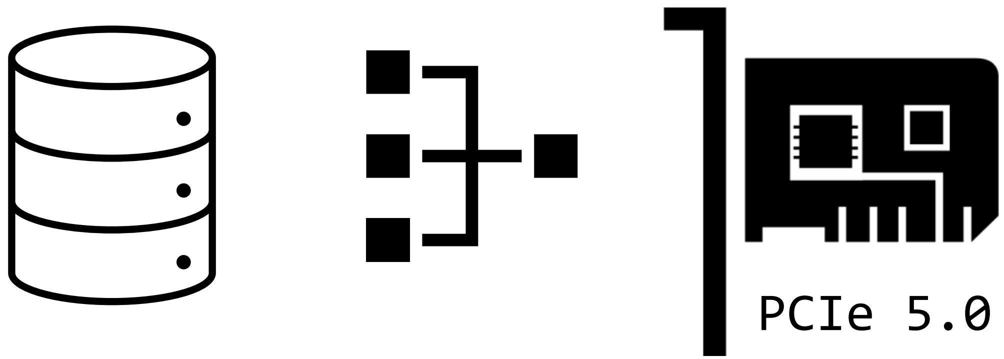
Certain worst-case inputs

Motivation

» Mergesort is highly appealing in real-world sorting tasks for several reasons

- Sequential processing of input/output

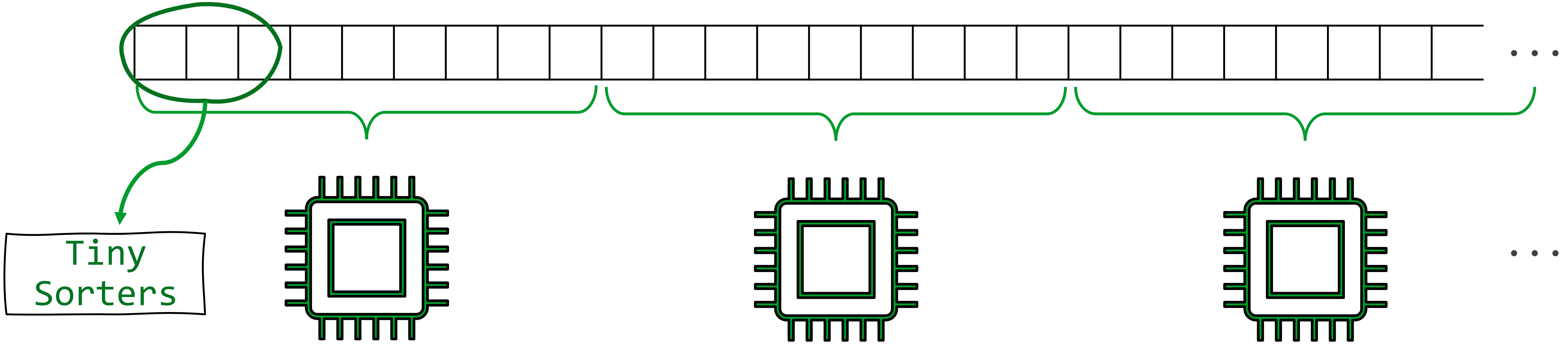




Motivation

» Mergesort is highly appealing in real-world sorting tasks for several reasons

- Well-suited for multi-core parallelization



- Yields new optimized kernels for small inputs

Motivation

» Many mergesort variants have been proposed, however ...

- None examine how to optimize individual phases of the sort pipeline
- Majority single threaded or, if parallel, bottlenecks on memory bandwidth
- Do not offer a unifying solution simultaneously optimized for scalar, SSE, AVX2 and AVX-512 architectures

Contribution

» Introduce Origami, a highly optimized, distribution-insensitive, parallel mergesort framework

» Formalize a four-phase computational model

- Examine how to achieve maximum speed at each phase

» Develop end-to-end sort by efficiently connecting the optimized components

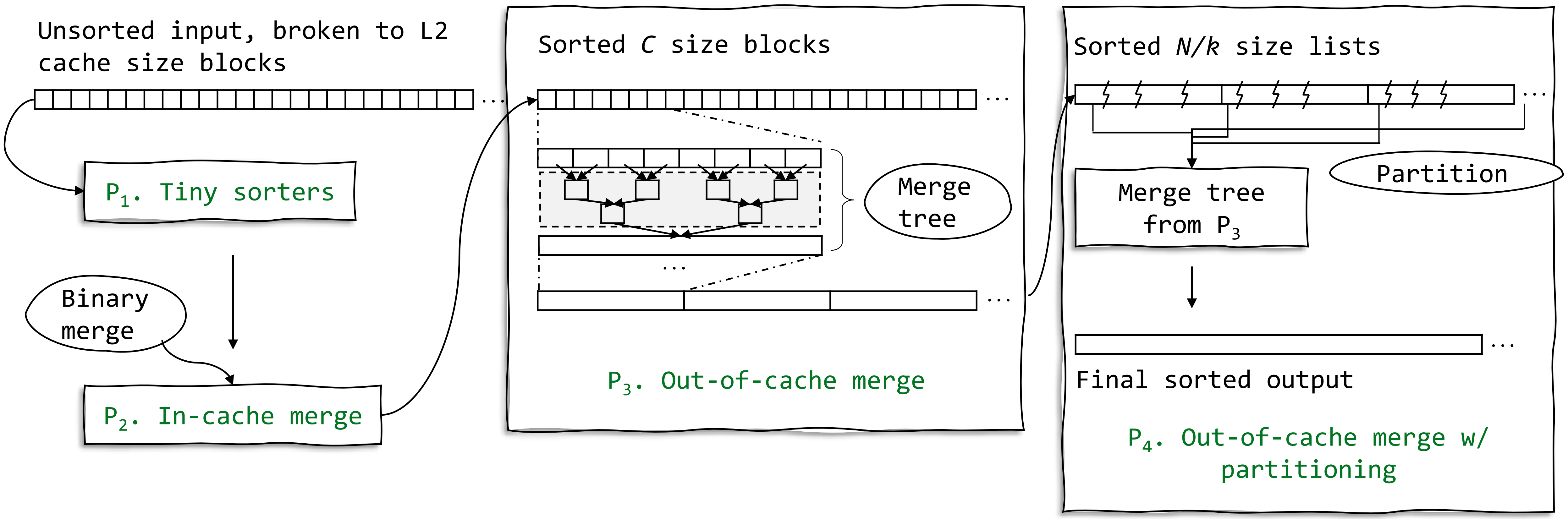
» Generalize the algorithms for Scalar, SSE, AVX2 and AVX-512

» Fastest mergesort (1.5-2x speedup) with near perfect scaling

Agenda

- » Introduction
- » **Pipeline Overview**
- » Tiny Sorters
- » In-cache Merge
- » Out-of-cache Merge
- » Experiments

Pipeline Overview



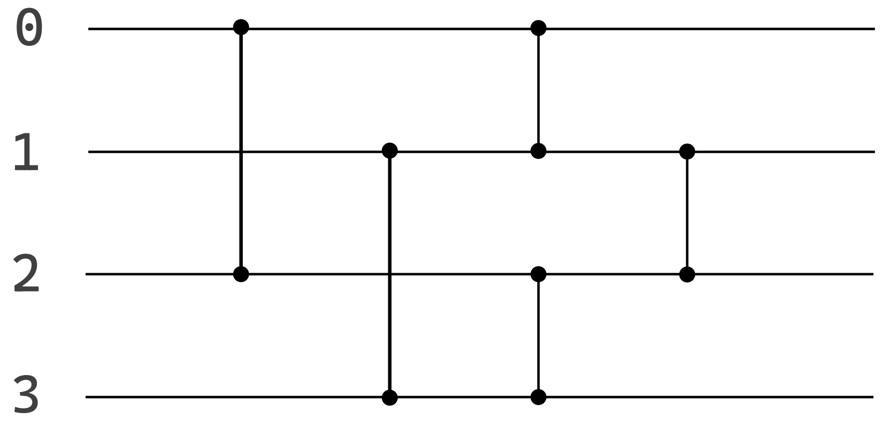
Agenda

- » Introduction
- » Pipeline Overview
- » **Tiny Sorters**
- » In-cache Merge
- » Out-of-cache Merge
- » Experiments

Sorting Networks

» In practice, presort every *m* items with a different algorithm

» Sorting networks have proven to be the fastest option for such small sorts

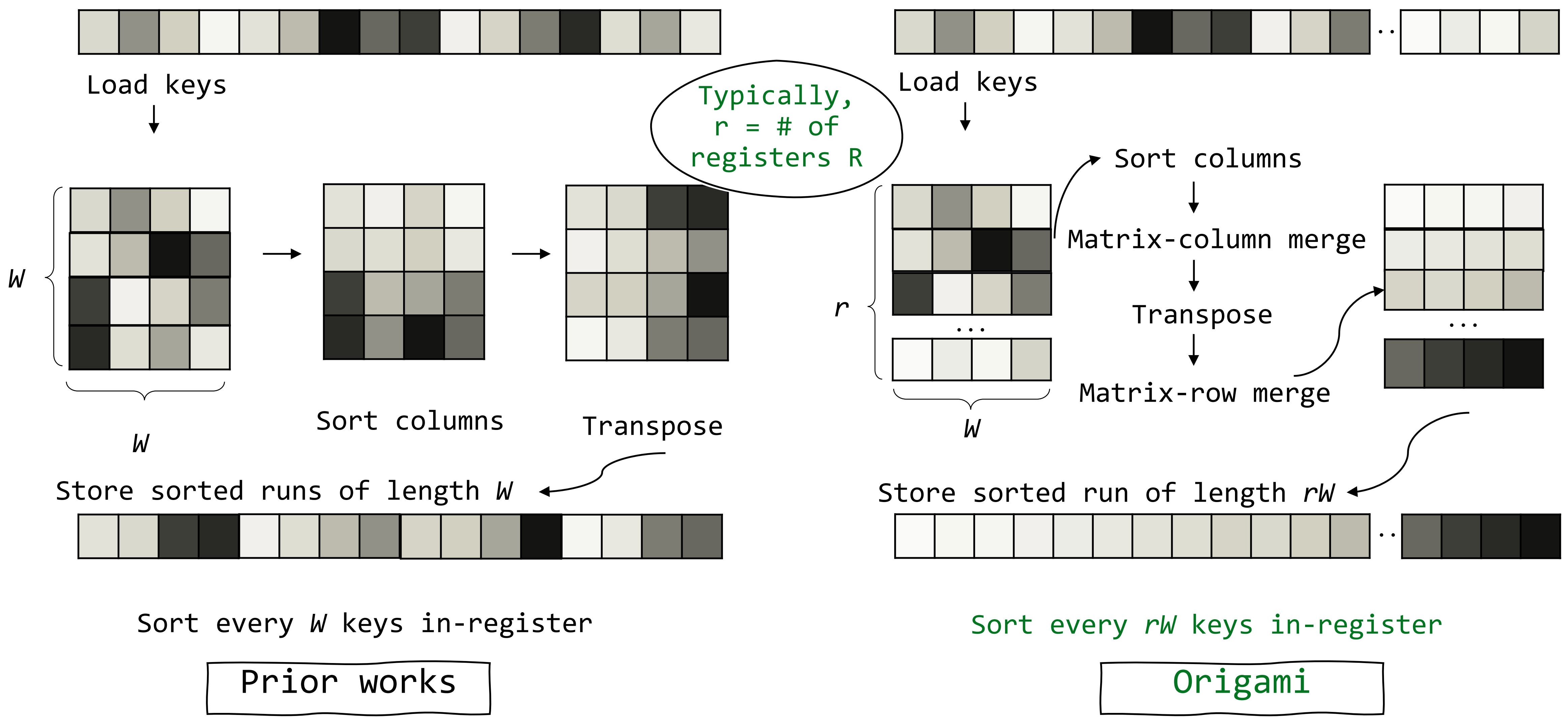


Sorting Network for 4 items

```
swap(x, y):
        tmp = min(x, y)
        y = max(x, y)
        x = tmp
```

» SIMD (single-instruction multiple-data) allows *W* (SIMD_WIDTH) scalar swaps with a pair of _mm_min, _mm_max intrinsics

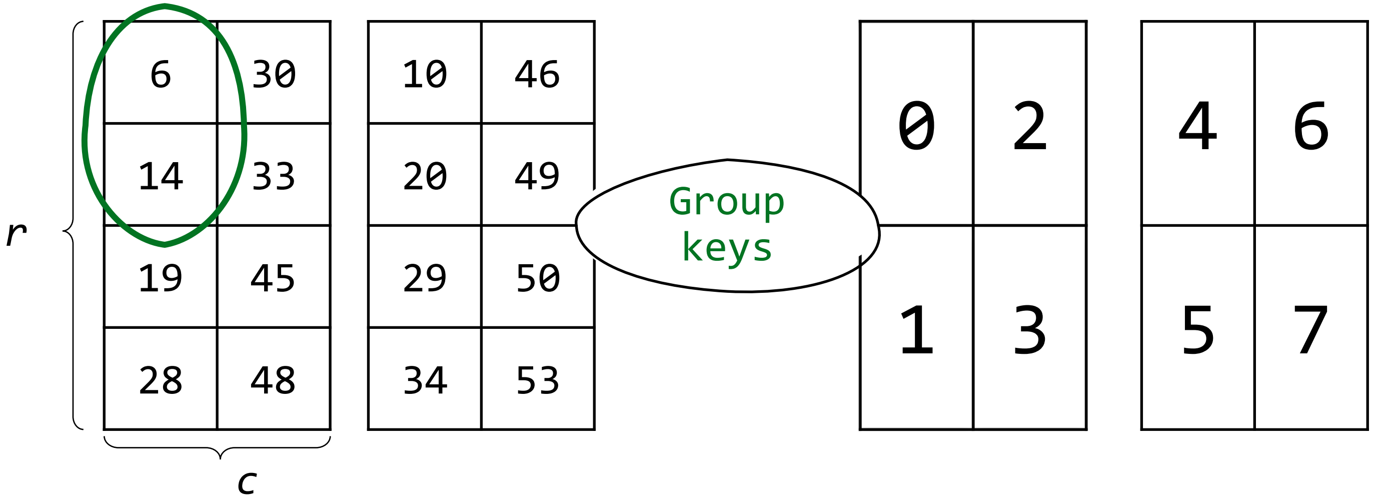
Tiny Sorters: Outline



Matrix-Column Merge (mcmerge)

» Goal: sort matrix in column-major order

- Use merge networks (reduced from sorting networks)
- Group items of matrix in partial columns of *r*/2 x 1
- Run swaps of corresponding merge network

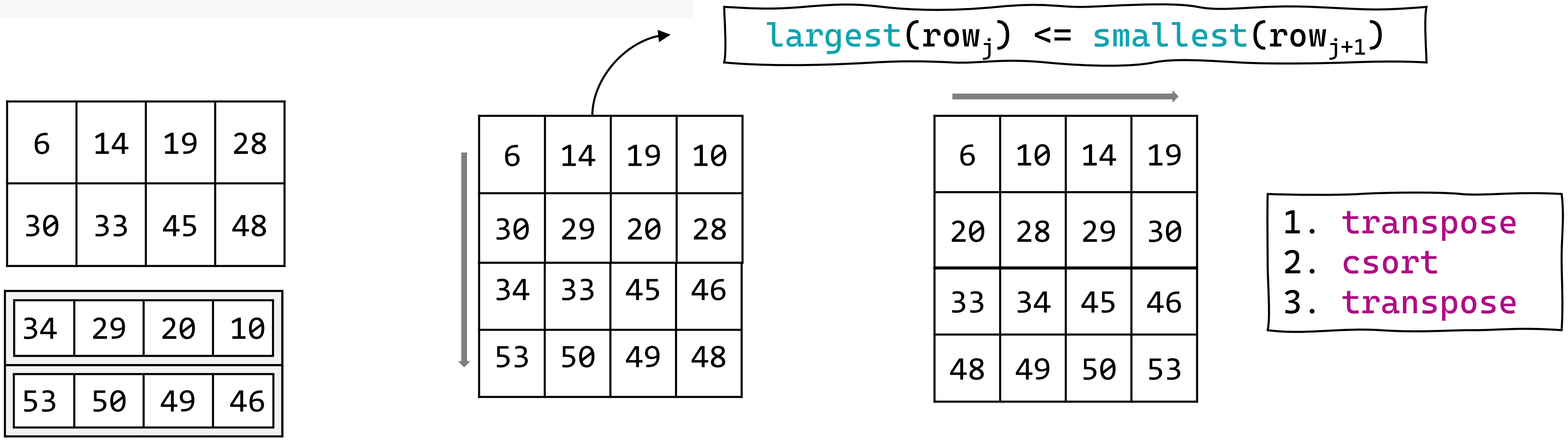


MergeNetwork8 swaps

(0,4), (1,5), (2,6), (3,7)
(2,4), (3,5)
(1,2), (3,4), (5,6)

- With len(keygroup) > 1, replace min/max for a swap with MergeNetwork*r* -- term this **cswap**
- Drawback: With growing *depth* of merge network, shuffles become costlier for large *c*

Matrix-Row Merge (mrmerge)



(a) reverse bottom rows (b) cswap (c) sort rows

» Not significantly affected by increasing complexity of merge networks -- excellent for large matrix sizes

» However, has non-negligible minimum cost (e.g., two transposes)

- Makes it inefficient for short sequences -- in contrast to mcmerge

Agenda

- » Introduction
- » Pipeline Overview
- » Tiny Sorters
- » **In-cache Merge**
- » Out-of-cache Merge
- » Experiments

Advancing Pointers

```
bmerge(Item *A, *endA, *B, *endB, *C):

 load registers r0, ..., rk-1 from A; A += kW
 load registers rk, ..., r2k-1 from B; B += kW

 while A != endA and B != endB:
     rswaps for MergeNetwork2k
     store r0, ..., rk-1 to C; C += kW
     reload r0, ..., rk-1 from A or B
     move A or B forward by kW

 merge keys left in registers and the
 unfinished list
```

- Present works mostly use branching comparisons
  - bmerge_v0

```
if (A[0] < B[0]):
   reload from A; A += kW
else:
   reload from B; B += kW
```

- Some attempts at branchless but still room for improvement
- Origami provides the fastest, purely branchless solution

Advancing Pointers

```
bmerge_v3(Item *A, *endA, *B, *endB, *C):
 load registers r_0, ..., r_{k-1} from A; A += kW
 load registers r_k, ..., r_{2k-1} from B; B += kW
 loadFrom = A; opposite = B;
 while loadFrom != endA and loadFrom != endB:
     rswaps for MergeNetwork2k
     store r_0, ..., r_{k-1} to C; C += kW
     flag = loadFrom[0] < opposite[0]
     tmp = flag ? loadFrom : opposite
     opposite = flag ? opposite : loadFrom
     loadFrom = tmp
     load r_0, ..., r_{k-1} from loadFrom
     loadFrom += kW
 merge keys left in registers and the
 unfinished list
```

- » Solution: bmerge_v3
  - Use two pointers: *loadFrom*, *opposite*
  - Update pointers based on *flag*
  - Always use *loadFrom* for next group of keys and end-of-buffer checks
- » Up to 86% faster than v0
- » Removes speculation from control flow and makes it distribution insensitive
- » Additional boost with multiple simultaneous merges

Agenda

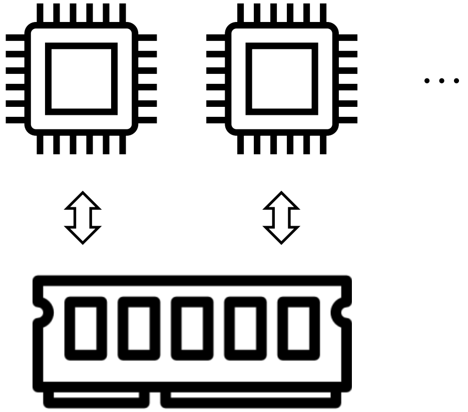
- » Introduction
- » Pipeline Overview
- » Tiny Sorters
- » In-cache Merge
- » **Out-of-cache Merge**
- » Experiments

Independent Merge (P_3)

» P_2 finishes when threads are done sorting lists of L2-cache-size C

» In P_3

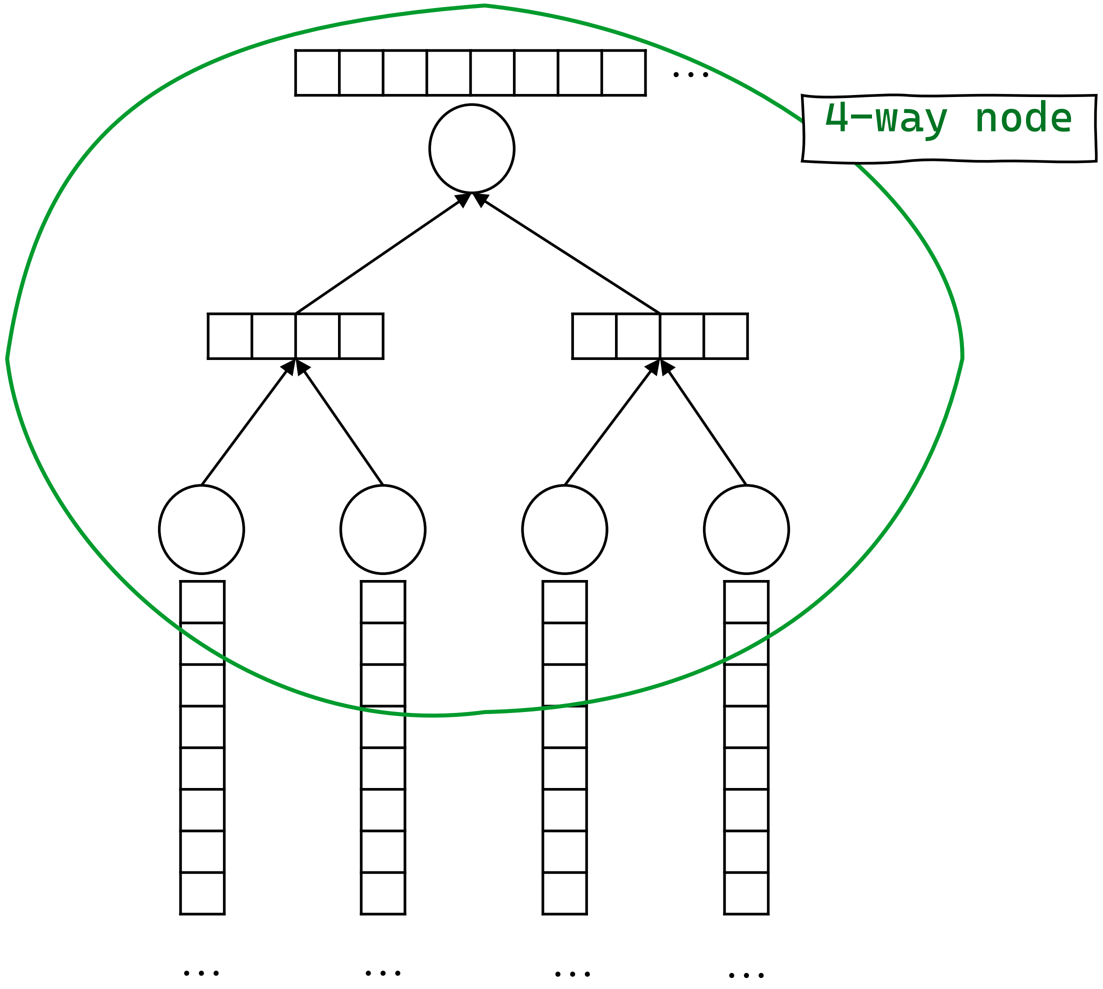
- Threads continue independent merges, but out-of-cache
- Maximum achievable speed is that of memcpy
  - Skylake-X i7 CPUs with DDR4-3200 quad channel memory max: 37 GB/s
  - Vectorized bmerge_v3 exhausts this with just 3 threads
  - One thread may be enough for older CPUs and dual channel memory



» Majority of existing works ignore and continue with binary merges

- A few use desired k-way merges but with limitations
  - L3 residing shared merge tree with circular queue internal buffers ...
  - L2 residing dedicated tree with fixed buffer, fixed k, and encoding-decoding keys with insertion sort tie-breaker ...

Merge Tree



» Origami comes with L2-cache residing *k*-way merge trees (**mtree**)

» Each node performs 4-way merge
- Binary merges internally
- Tiny intermediate buffers (64-128 B)
- Root and leaves remain large

» *k* can be tuned
- Optimal choice depends on number of threads running, memory bandwidth, and L2 cache size

Cooperative Merge (P_4)

» Origami P_4 avoids bottleneck on memory bandwidth

- Merge must utilize >= *k* sequences
- *k* selected optimally by mtree in P_3

» Avoid stragglers by creating many small jobs

- Reduce wait time for the fastest thread
- Leader thread performs initial partition
- All threads parallelly partition further
- Add *k*-way merge jobs to shared queue
  - Threads draw their workload in parallel

Agenda

Setup

S_1

8-core Intel i7-7820X (Skylake-X)

L2 cache: 1 MB

Clock: 4.7 GHz (fixed)

SIMD Support: SSE, AVX2, AVX-512

32 GB DDR4-3200

Quad-channel

S_2

16-core dual socket Intel Xeon E5-2690

L2 cache: 256 KB

Clock: 3.3 GHz

SIMD Support: SSE, AVX

256 GB DDR3-1333

Quad-channel

Visual Studio 2019

Windows Server 2016

intel

Intel VTune

Tiny Sorters

Table 2: Merge speed (B keys/s) in a $32 \times \mathcal{W}$ matrix

$\mathcal{B}$	$\mathcal{K}$	SSE			AVX2			AVX-512		
		$\mathcal{X}$	mc	mr	$\mathcal{X}$	mc	mr	$\mathcal{X}$	mc	mr
32	8	8	10.39	3.75	16	24.11	5.19	32	21.98	-
	16	4	6.26	3.52	8	13.82	5.21	16	16.92	7.63
	32	2	2.81	3.24	4	6.24	5.02	8	7.53	7.23
	64	1	1.58	2.83	2	3.98	4.74	4	5.04	6.71
64	8	4	3.51	1.96	8	4.66	2.36	16	10.98	3.22
	16	2	2.45	1.71	4	3.21	1.99	8	8.46	3.07
	32	1	1.06	1.41	2	1.41	1.83	4	3.53	2.88
	64	–	–	–	1	0.93	1.49	2	2.33	2.68
64+64	8	2	1.44		4	2.08	1.23	8	3.61	1.26
	16	1	1.06		2	1.43	1.08	4	3.06	1.13
	32	–	–	–	1	0.66	0.92	2	1.32	1.03
	64	–	–	–	–	–	–	1	0.89	1.01

Chunked-sort (Out-of-cache)

Table 11: Chunked speed in C_3 (M/s); $\mathcal{N}$ = 256M, $\mathcal{B}$ = 32

Define Checkpoint

C_i = execution of phases P_1 through P_i

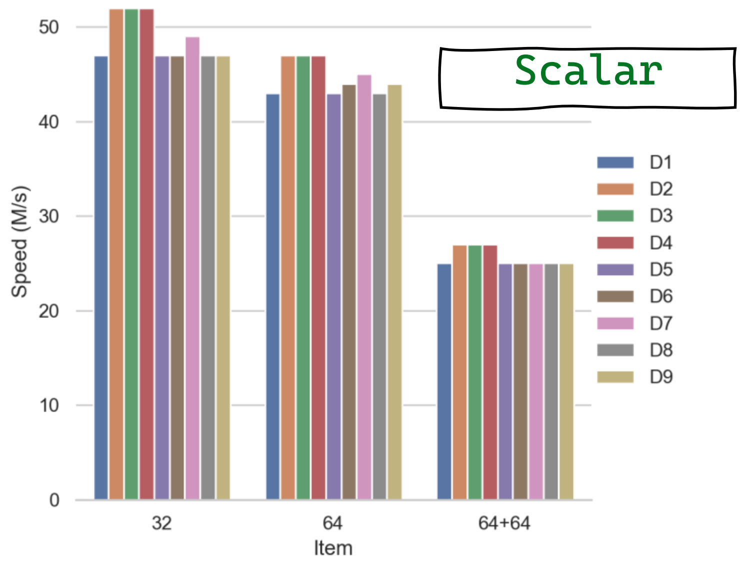
chunk size c	SSE		AVX2			AVX-512			
	[15]	C_3	[14]	[26]	C_3	[30]	[32]	[33]	C_3
128 K	63	176	53	139	228	40	198	140	295
256 K	61	147	47	128	210	33	184	130	269
512 K	59	138	44	120	195	30	172	113	249
1 M	57	131	41	109	183	28	160	102	232
2 M	55	124	39	92	174	25	150	95	216
4 M	54	118	37	81	168	23	140	88	203
8 M	52	112	35	77	162	21	131	83	191
16 M	50	107	33	73	153	20	122	78	181
32 M	48	102	32	70	145	19	115	72	172
64 M	47	98	30	67	138	18	109	69	163
128 M	45	95	29	65	132	17	103	66	156
256 M	44	91	28	63	126	17	97	64	149

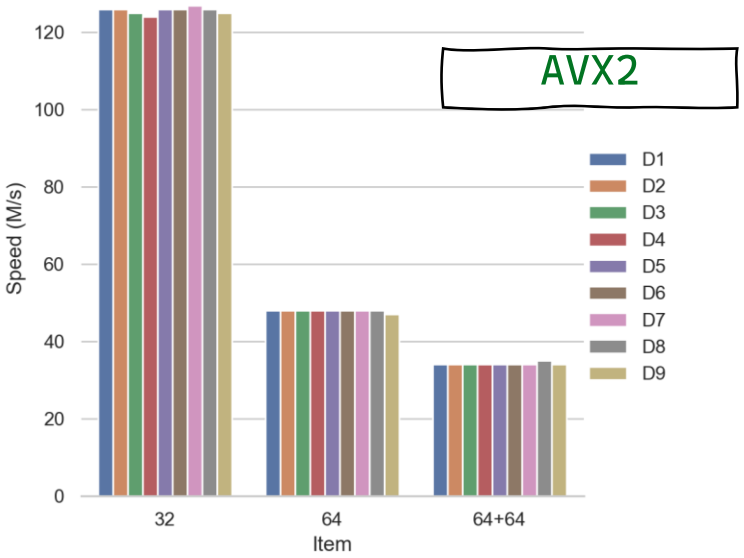
SSE: 110%

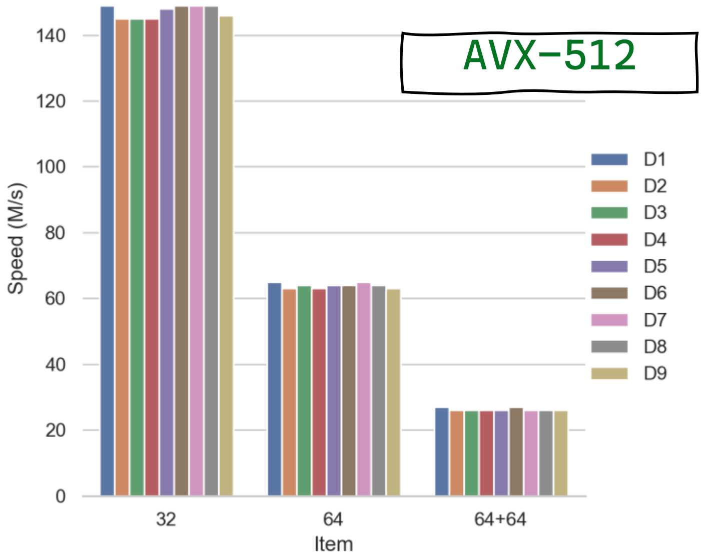
AVX2: 100%

AVX-512: 53%

Distribution Insensitivity



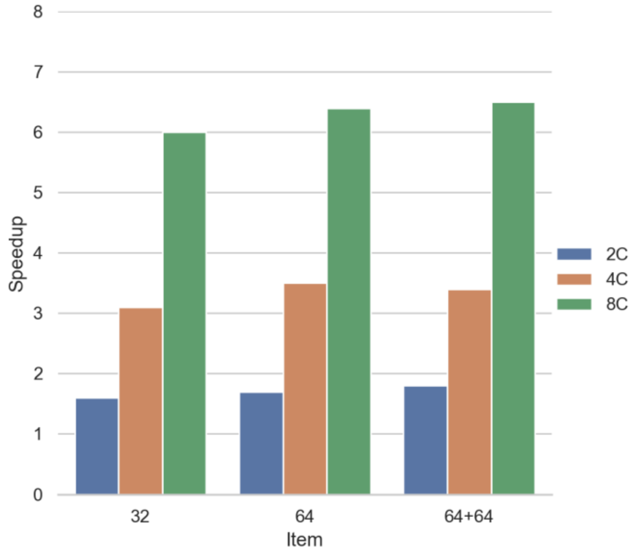




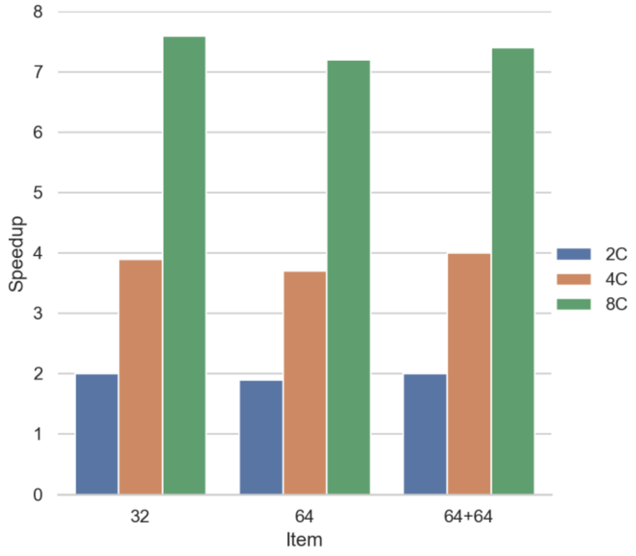
D1: Uniform
D2: All same
D3: Sorted
D4: Reverse sorted
D5: Almost sorted (7th = MAX)
D6: Pareto
D7: Bursts of same keys (length from D6, key from D1)
D8: Random shuffle of D7
D9: Fibonacci

Multi-core Speedup

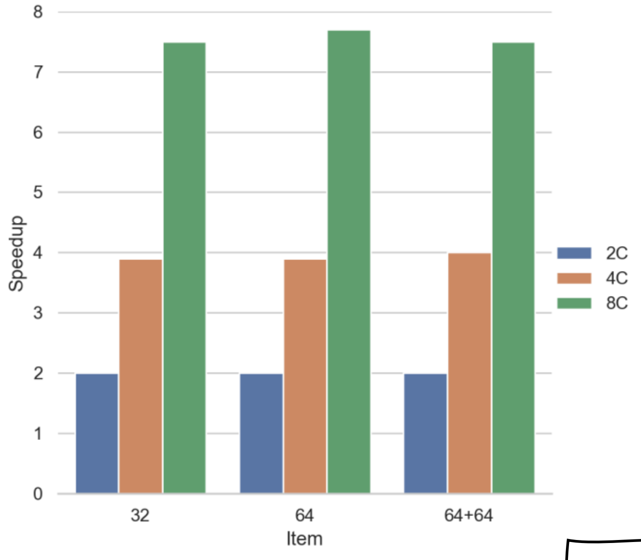
Scalar



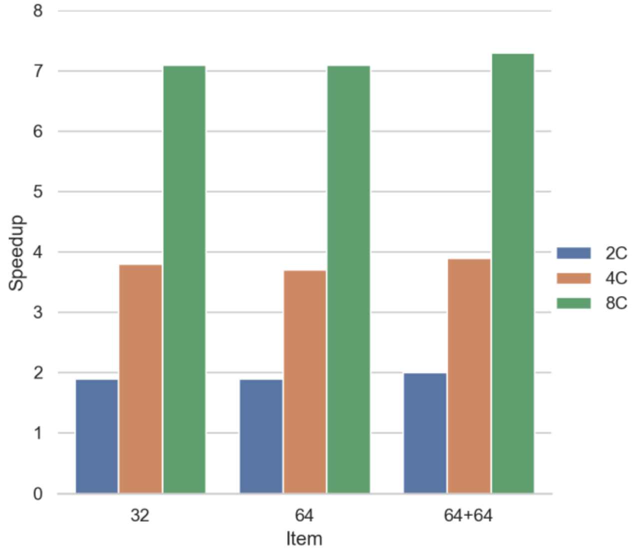
SSE



AVX2



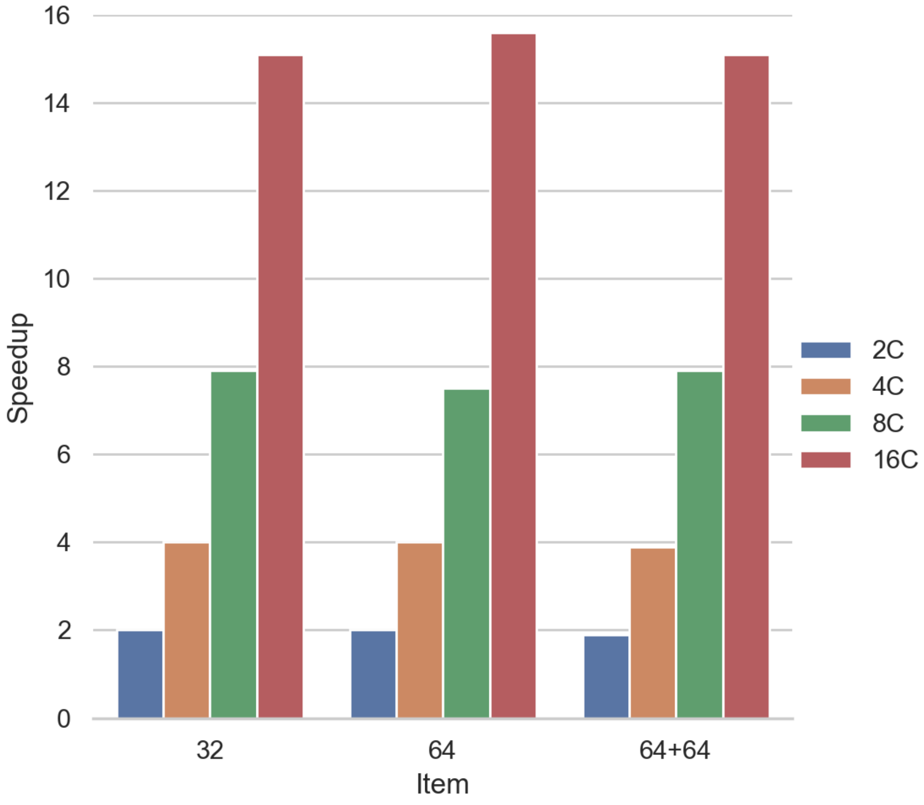
AVX-512



1 GB

Multi-core Speedup (Xeons)

SSE



64 GB

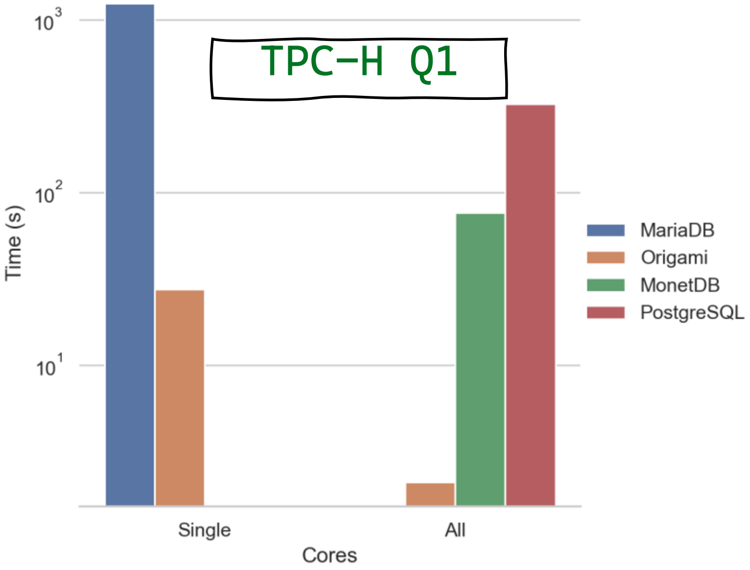
Database Queries (Xeons)

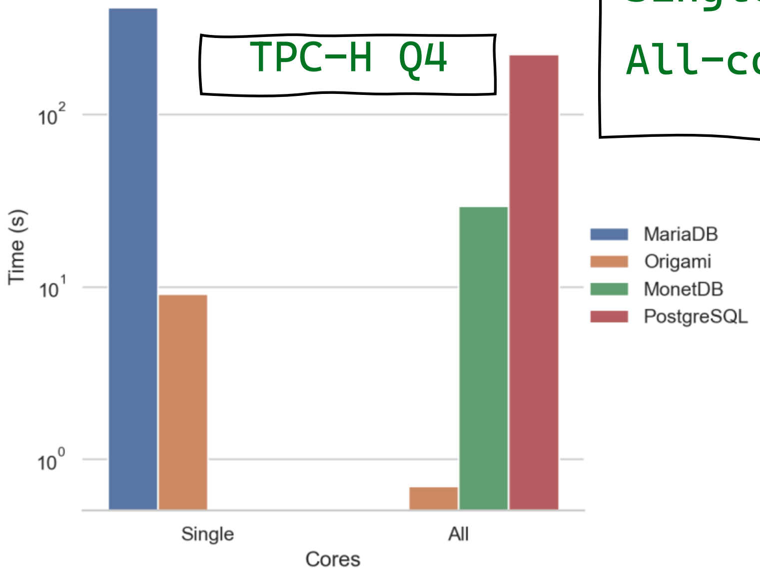
» IRLbot query

```
SELECT dst, COUNT(*) as cnt
FROM A INNER JOIN B ON A.src=B.src
WHERE A.outdeg < 1000000
GROUP BY dst
ORDER BY cnt DESC
```

Single-core: 37-60x
All-cores: 30-113x

» TPC-H queries
» Scaling factor: 100





Concluding Remarks

» Origami offers a highly optimized mergesort framework
  - Runs in a fast, constant speed for different data distributions
  - Gains a nearly linear speed-up in multi-core environments

» The proposed components are flexible to accommodate future SIMD extension sets
  - Programmer only needs to write a few arch-specific intrinsics

» Future work will examine
  - External memory sorting
  - Longer key/value pairs
  - Incorporation into existing DBMS

Thank You

Arif Arman

arman@tamu.edu

https://arif-arman.github.io