Origami: A High-Performance Mergesort Framework

Arif Arman and Dmitri Loguinov
Texas A&M University

About Me

I am a Ph.D. student at Texas A&M University. My research focus is on high-performance computing and sorting, algorithm optimization for the underlying hardware, and information retrieval at a large scale. These require a deeper understanding of Computer Systems and Architectures -- which are my favorite topics.

Agenda

» **Introduction**

» Pipeline Overview

» Tiny Sorters

» In-cache Merge

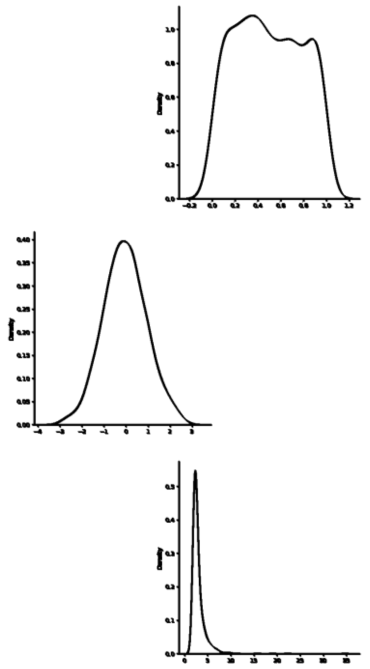
» Out-of-cache Merge

» Experiments

Motivation

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

- Distribution insensitive



MSB Radixsort

Poor unless uniform

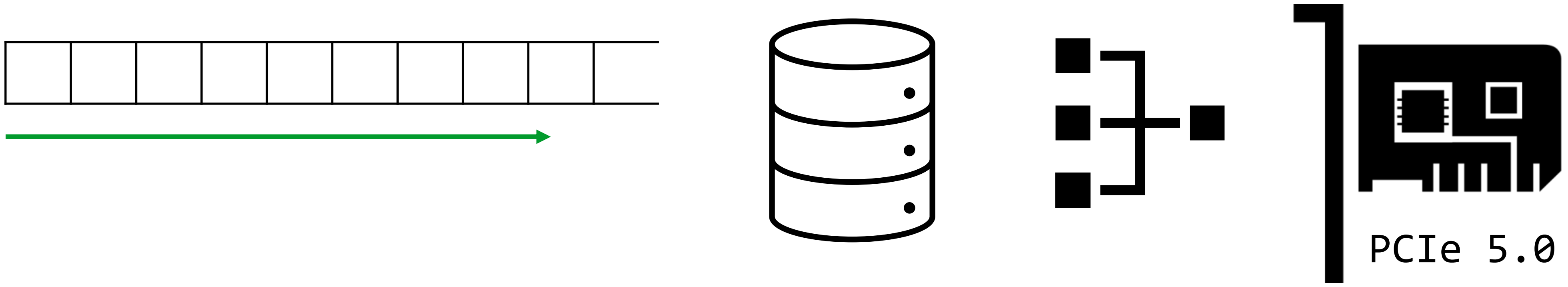
Quicksort
Samplesort
Combsort

Certain worst-case inputs

Motivation

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

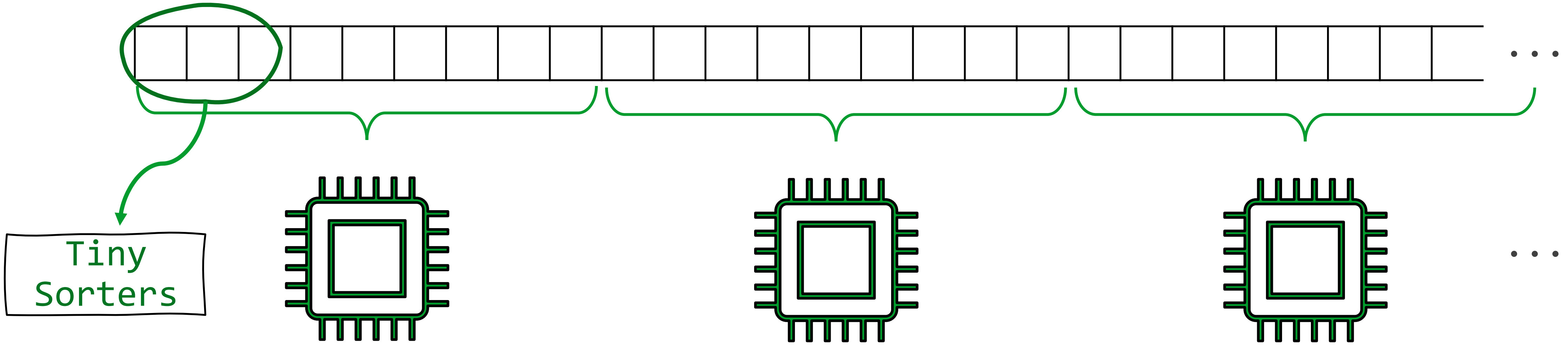
- Sequential processing of input/output



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 ...

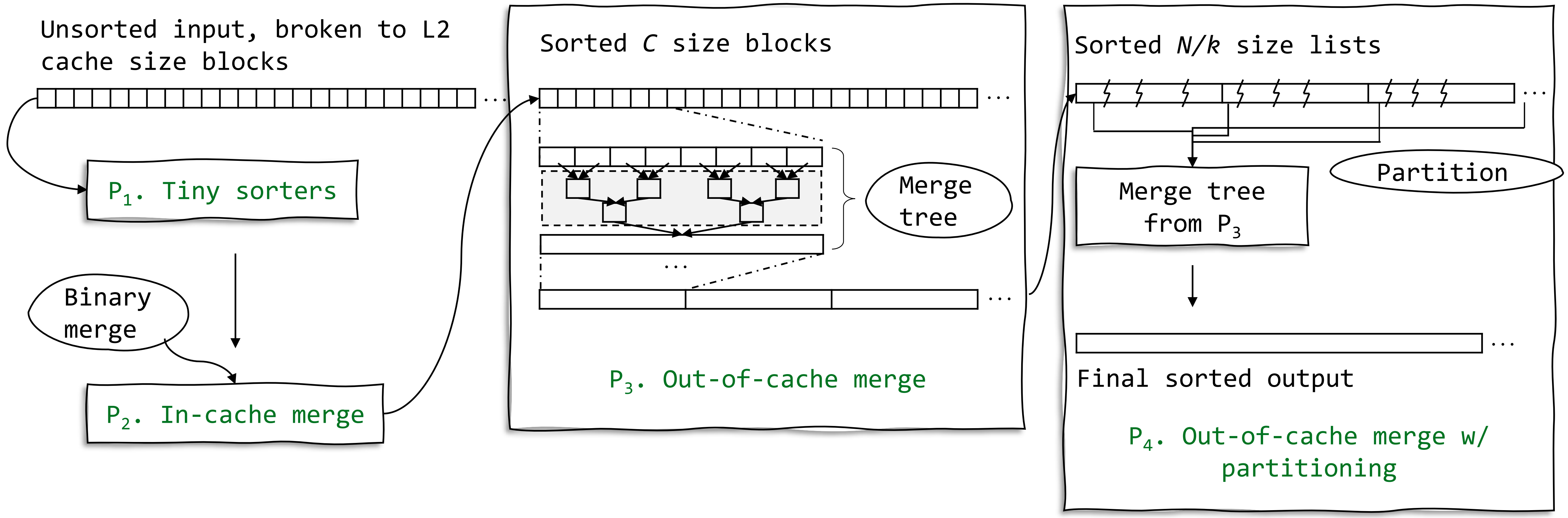
- None examine how to optimize individual phases of the sort pipeline
- Majority single threaded or, if parallel, bottlenecks on memory bandwidth
- Do not offer a unifying solution simultaneously optimized for scalar, SSE, AVX2 and AVX-512 architectures

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

Pipeline Overview

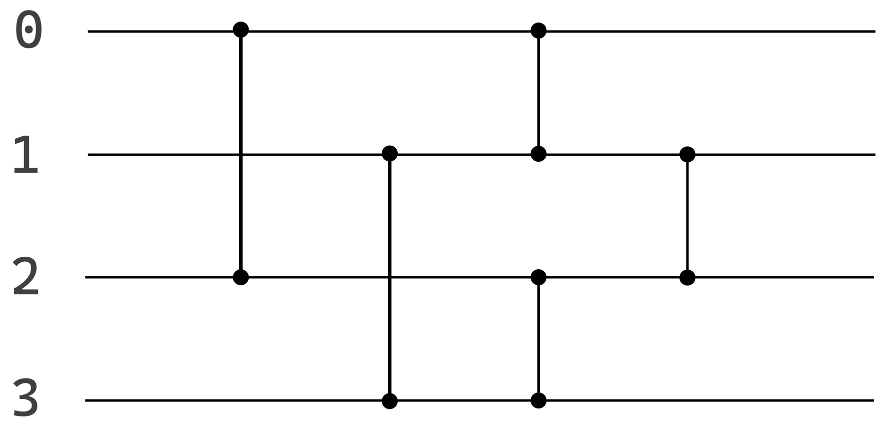


Agenda

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

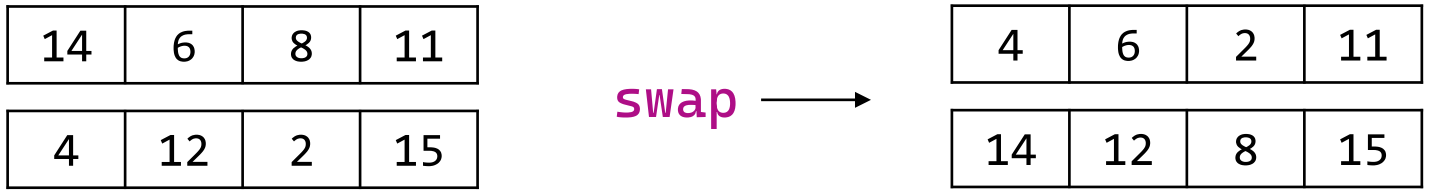


Sorting Network for 4 items

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

SIMD

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



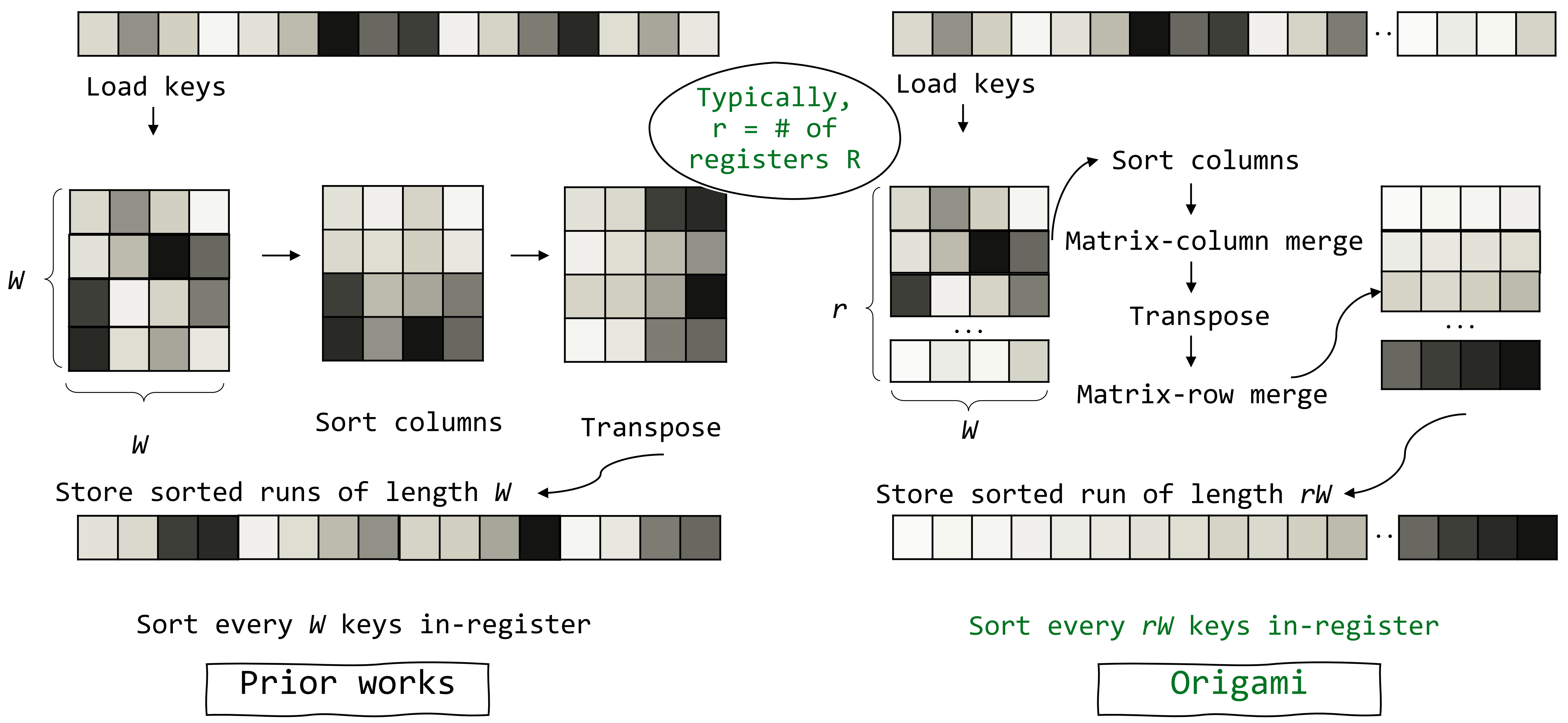
» Min/max must be ***branchless*** for maximum speed

- Vectorized min/max intrinsics by design branchless
- For Scalar, use cmov (conditional move) instruction
  - Use ? in C/C++ (e.g., tmp = x < y ? x : y; for min(x, y))

distribution insensitive

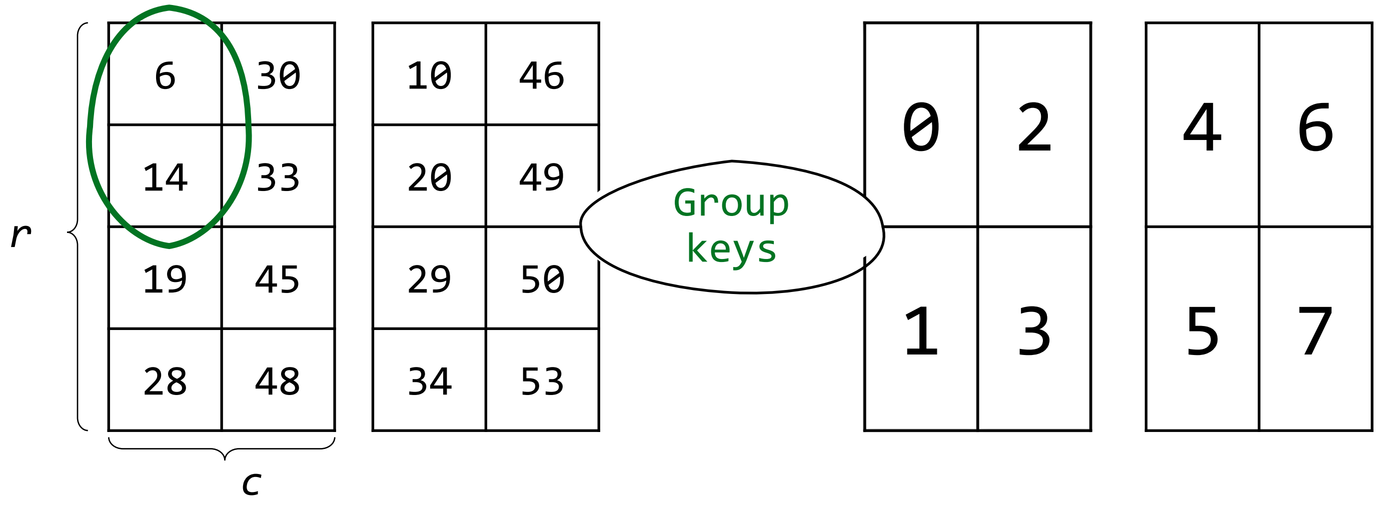
» Stack multiple registers and vertically sort *W* columns in parallel -- term this technique **csort**

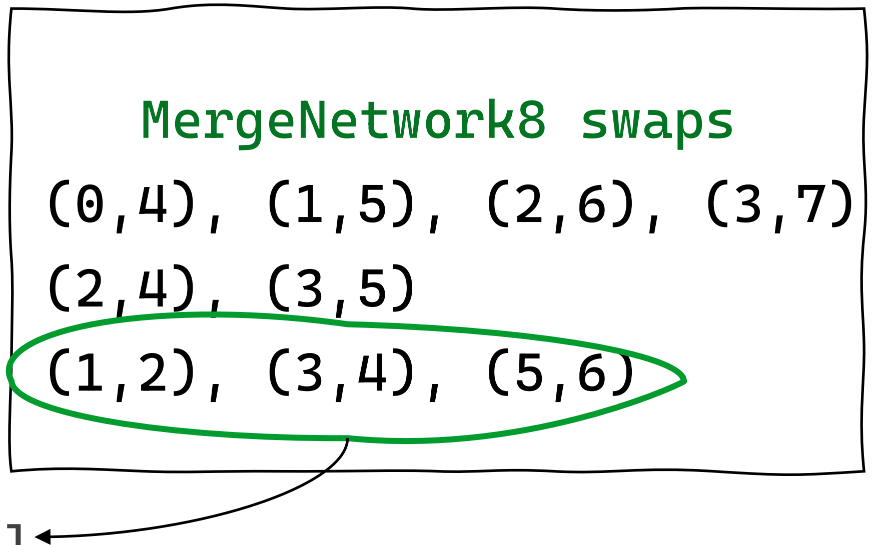
Tiny Sorters: Outline


Load keys
W
W
Sort columns
Transpose
Store sorted runs of length W
Sort every W keys in-register
Prior works
Typically, r = # of registers R
Load keys
Sort columns
Matrix-column merge
Transpose
Matrix-row merge
r
W
...
Store sorted run of length rW
Sort every rW keys in-register
Origami

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

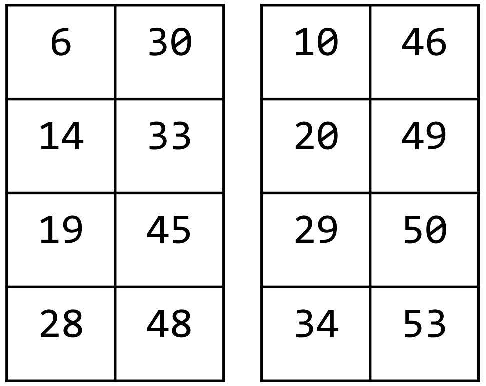



MergeNetwork8 swaps
(0,4), (1,5), (2,6), (3,7)
(2,4), (3,5)
(1,2), (3,4), (5,6)

- With SIMD, swaps of a layer runs in parallel
- With len(keygroup) > 1, replace min/max for a swap with MergeNetwork*r* -- term this **cswap**

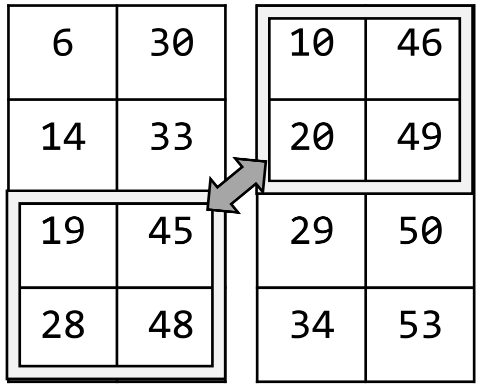
Matrix-Column Merge: Example

```
MergeNetwork8
(0,4), (1,5), (2,6), (3,7)
(2,4), (3,5)
(1,2), (3,4), (5,6)
```



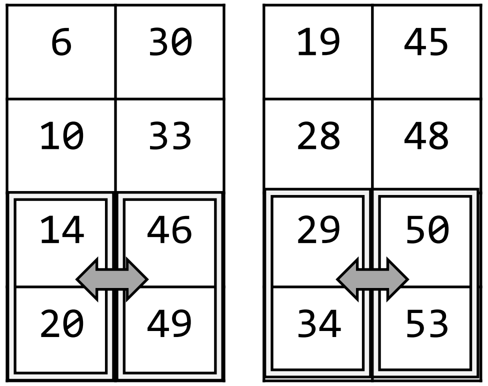
6	30	10	46
14	33	20	49
19	45	29	50
28	48	34	53

(a) initial



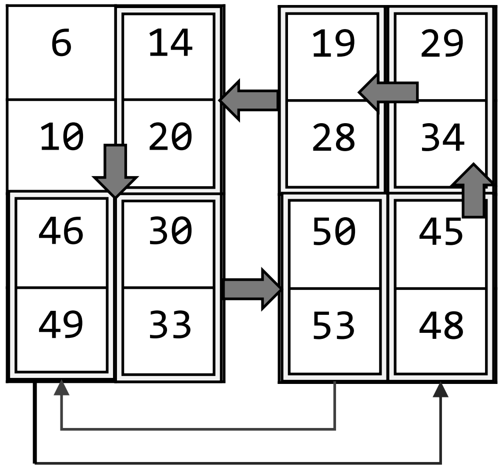
6	30	10	46
14	33	20	49
19	45	29	50
28	48	34	53

(b) shufffle #1



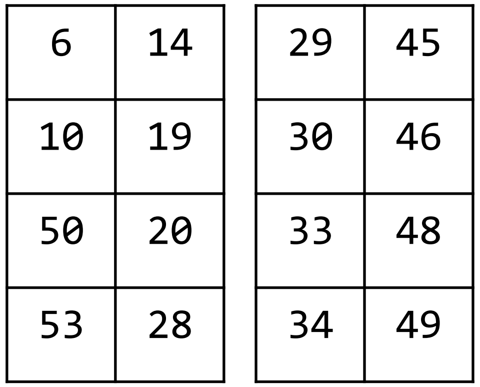
6	30	19	45
10	33	28	48
14	46	29	50
20	49	34	53

(c) cswap #1, shuffle #2



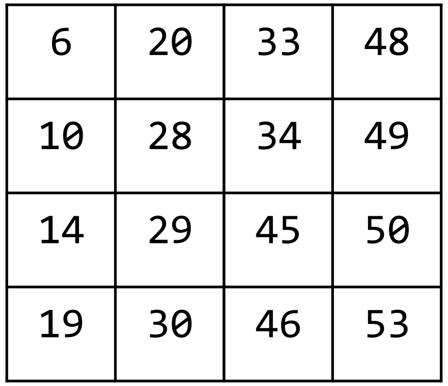
6	14	19	29
10	20	28	34
46	30	50	45
49	33	53	48

(d) cswap #2, shuffle #3



6	14	29	45
10	19	30	46
50	20	33	48
53	28	34	49

(e) cswap #3



6	20	33	48
10	28	34	49
14	29	45	50
19	30	46	53

(f) final

Matrix-Column Merge: Summary

» Advantages

- Maximum utilization of data parallelism -- allows simultaneous operations on all $W/2c$ pairs of matrices at no extra cost

- Number of steps is the *depth* of merge network, which is proved optimal for networks of <= 17 items

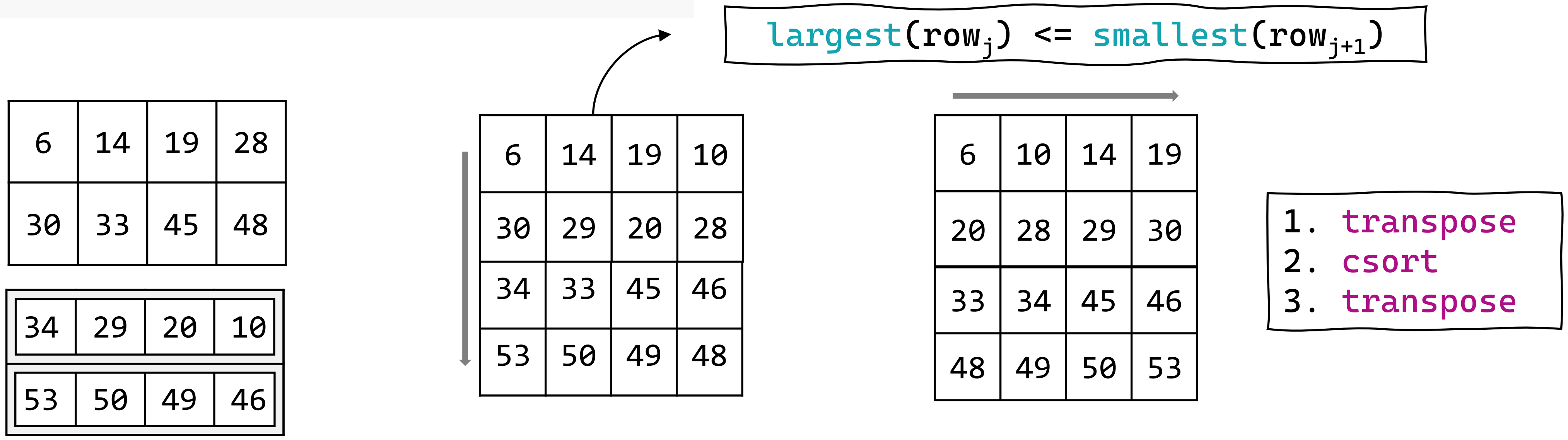
- Final reordering can be omitted for back-to-back merges

» Drawbacks

- With growing *depth* of merge network, shuffles become costlier for large c

» Solution: transpose switch to **Matrix-Row Merge** at one point

Matrix-Row Merge (mrmerge)



6	14	19	28
30	33	45	48
34	29	20	10
53	50	49	46

(a) reverse bottom rows

6	14	19	10
30	29	20	28
34	33	45	46
53	50	49	48

(b) cswap

6	10	14	19
20	28	29	30
33	34	45	46
48	49	50	53

(c) sort rows

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

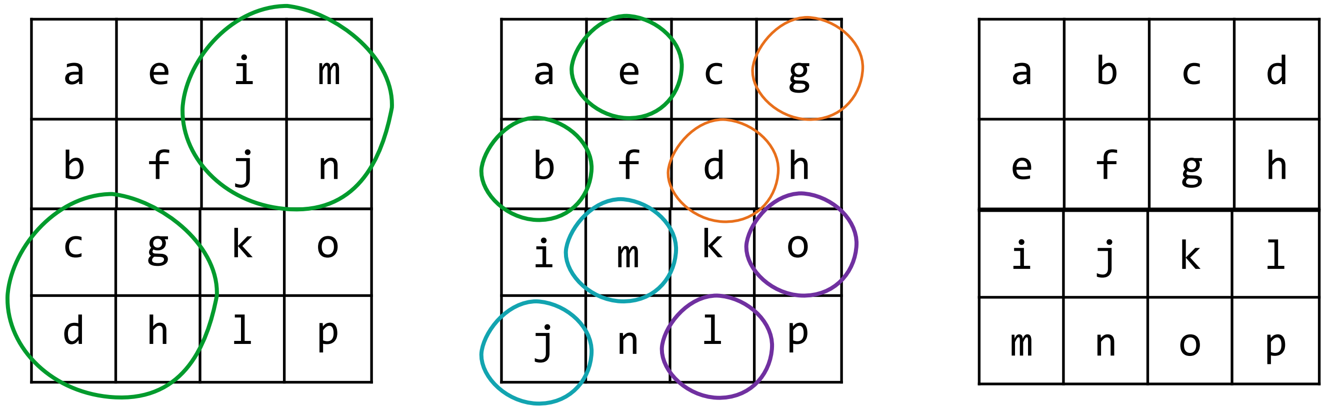
» However, has non-negligible minimum cost (e.g., two transposes)

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

Matrix Transpose

» Transpose performed by a series of diagonal exchanges

» Typically done with two shuffle or permute intrinsics
- Port 5 pressure
- Solution: replace some shuffles with blend (use port 0, 1 and 5)



transpose_v0
```
_v0 = _mm256_shuffle_ps(v0, v1, 0x44)
_v1 = _mm256_shuffle_ps(v0, v1, 0xEE)
```

transpose_v1
```
  v = _mm256_shuffle_ps(v0, v1, 0x4E)
_v0 = _mm256_blend_ps(v0, v, 0xCC)
_v1 = _mm256_blend_ps(v1, v, 0x33)
```

Tiny Sorters: Summary

» Begin with mcmerge and switch to mrmerge

» Use the optimized transpose

» Sort m (in $[W, RW]$) items, completely in register

» Choice of m dependent on
  - $S_1(m)$: P_1 speed to sort m items
  - S_{merge}: In-cache merge speed
  - Optimal m minimizes

$$f(m) = \frac{S_{merge}}{S_{1(m)}} - \log_2 m$$

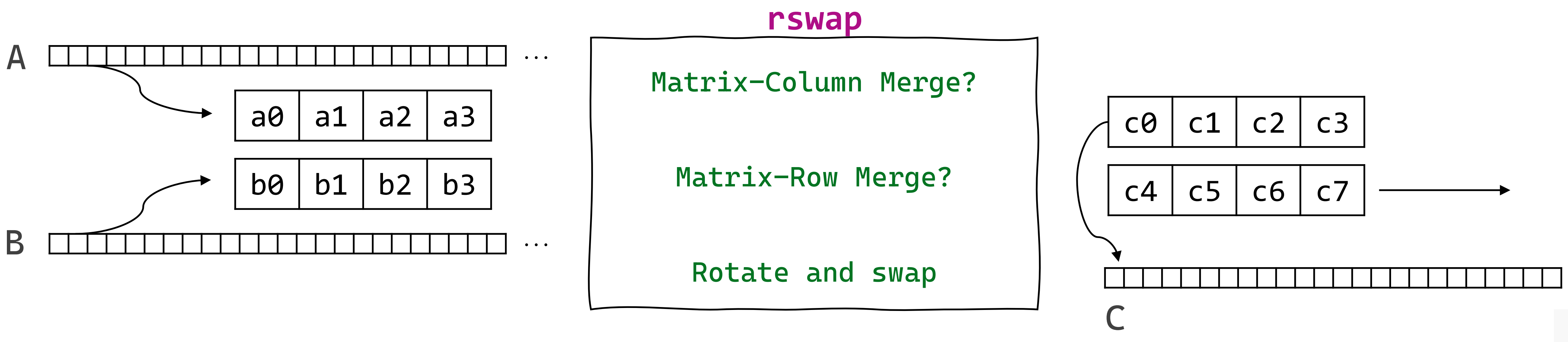
Agenda

Merge Kernel

» Main building block of merge-based sorts: binary merge (**bmerge**)

» Up to $\log_2 n$ passes over the entire data
  - Significant in overall performance

» Require a fast kernel to merge two sorted registers



Advancing Pointers

```
bmerge(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

 while A != endA and B != endB:
     rswaps for MergeNetwork2k
     store r_0, ..., r_{k-1} to C; C += kW
     reload r_0, ..., r_{k-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(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

 while A != endA and B != endB:
     rswaps for MergeNetwork2k
     store r_0, ..., r_{k-1} to C; C += kW
     reload r_0, ..., r_{k-1} from A or B
     move A or B forward by kW

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

» A trivial method is to use cmov instructions
- bmerge_v1

```
flag = A[0] < B[0]
r_i = flag ? load(A + iW) : load(B + iW)
A += flag ? kW : 0
B += flag ? 0 : kW
```

» However, SIMD _mm_load intrinsics do not support conditional moves

Advancing Pointers

```
bmerge(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

 while A != endA and B != endB:
     rswaps for MergeNetwork2k
     store r_0, ..., r_{k-1} to C; C += kW
     reload r_0, ..., r_{k-1} from A or B
     move A or B forward by kW

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

- » Solution: use cmov to compute the pointer to load
  - bmerge_v2

```
src = flag ? A : B
r_i = load(src + iW); i in [0, k-1]
A += flag ? kW : 0
B += flag ? 0 : kW
```

- » Up to 50% faster than v0
- » Checks end-of-buffer for both A and B; fails to keep pointers in register

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

Scalar Merge Optimizations

» Load *k* > 1 keys from each buffer

- Utilize registers
- Decrease loop overhead per sorted item

» Reduce number of swaps needed by the merge network

- Outputs *k* from *2k* sorted items
- Latter half can be kept *partially* sorted – skip the swaps that involve registers r_k, ..., r_{2k-1}
- MergeNetwork8: 9 -> 8; MergeNetwork16: 25 -> 20
- Does not apply to SIMD since *depth* is not reduced

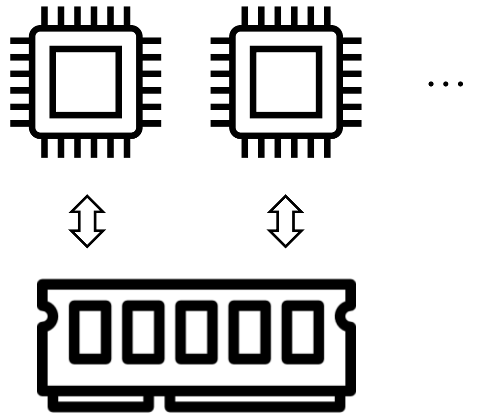
Agenda

Independent Merge (P_3)

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

» In P_3

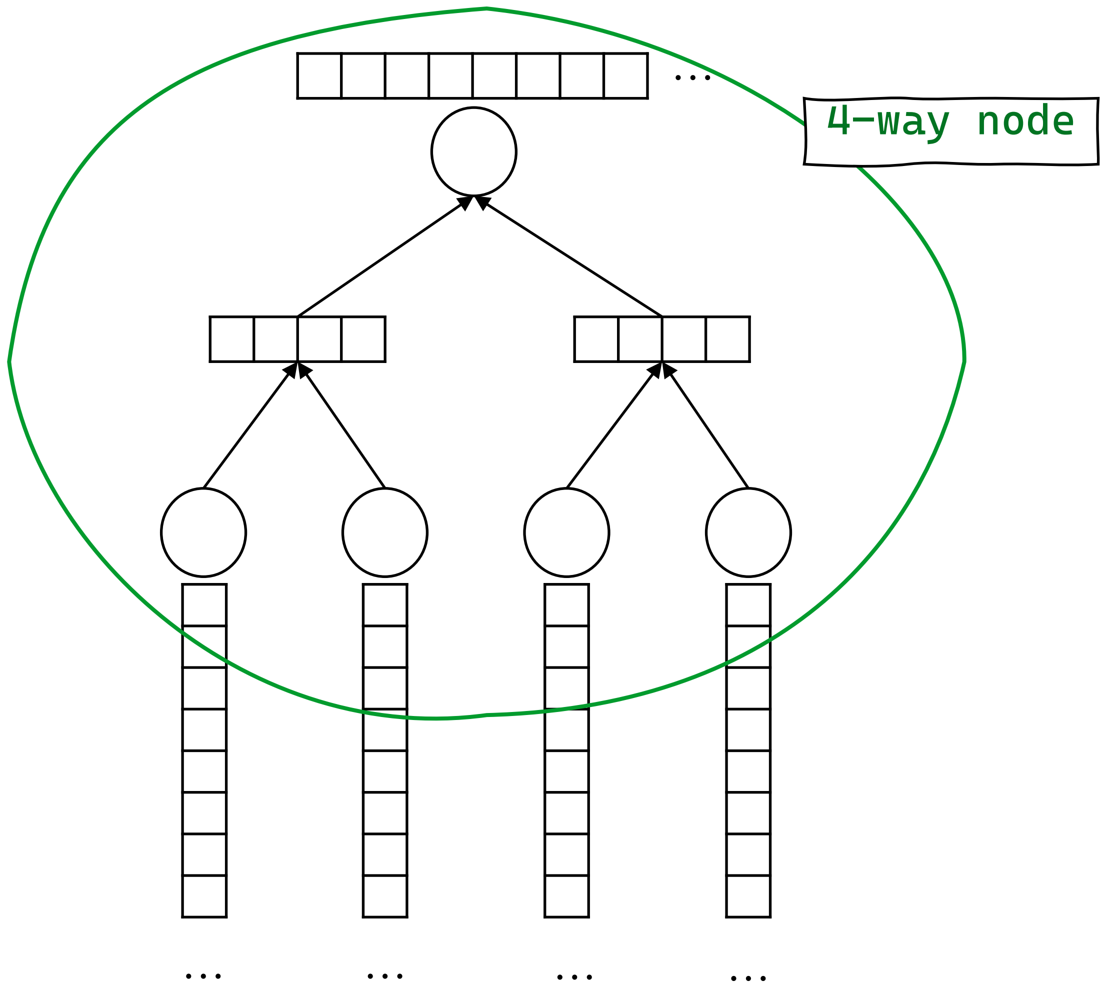
- Threads continue independent merges, but out-of-cache
- Maximum achievable speed is that of memcpy
  - Skylake-X i7 CPUs with DDR4-3200 quad channel memory max: 37 GB/s
  - Vectorized bmerge_v3 exhausts this with just 3 threads
  - One thread may be enough for older CPUs and dual channel memory



» 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

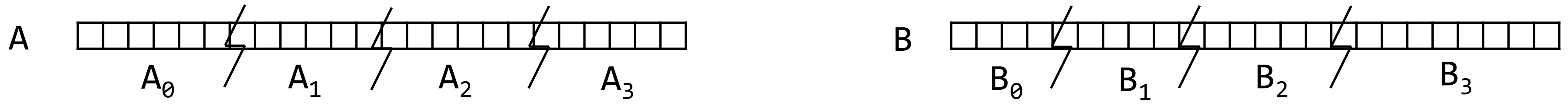
» *k* can be tuned
- Optimal choice depends on number of threads running, memory bandwidth, and L2 cache size

Cooperative Merge (P_4)

» P_4 begins when the number of lists to merge becomes insufficient to continue independent merging

» Most prior work begins this phase with *T* (number of threads) lists

- Use binary-search based partitioning
- Partition the lists into *T* segments such that:
  - Items in (A_i, B_i, ...) <= (A_j, B_j, ...) for i < j
  - $|A_i| + |B_i| + \ldots = n/T$



- Assign thread *i* to run *T*-way merge on (A_i, B_i, ...)
- May bottleneck on memory bandwidth and/or contain stragglers

» Others shrink *T* as merge nears the end, or skip multi-threading

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

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

Setup

$\boxed{S_1}$

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

$\boxed{S_2}$

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

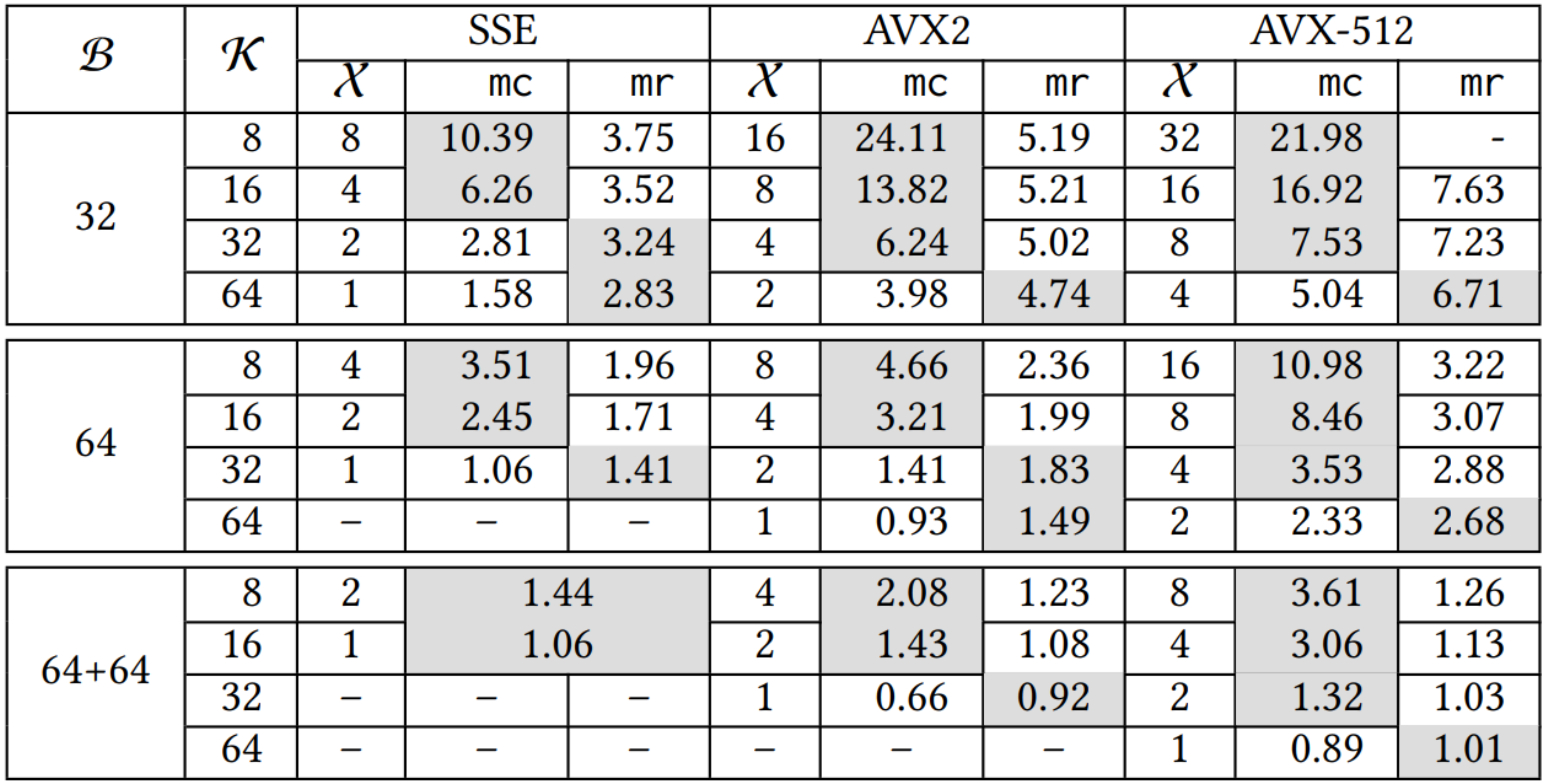
Visual Studio 2019

Windows Server 2016

Intel VTune

Tiny Sorters

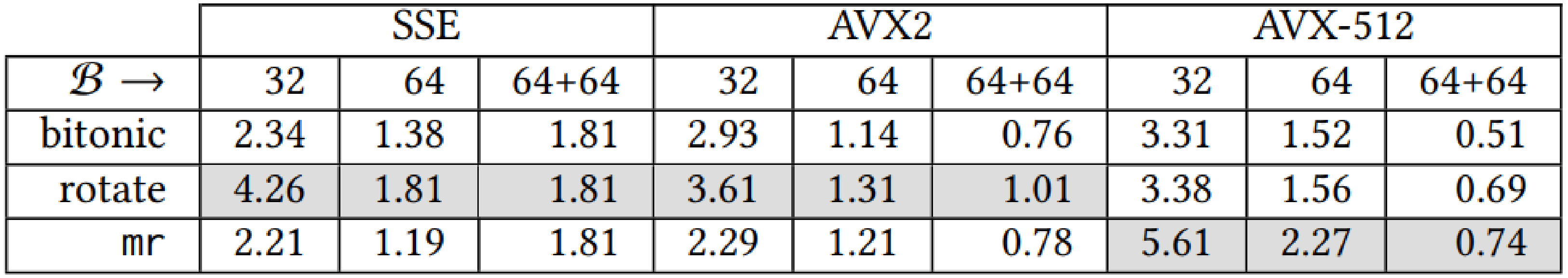
Table 2: Merge speed (B keys/s) in a $32 \times \mathcal{W}$ matrix



$\mathcal{B}$	$\mathcal{K}$	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

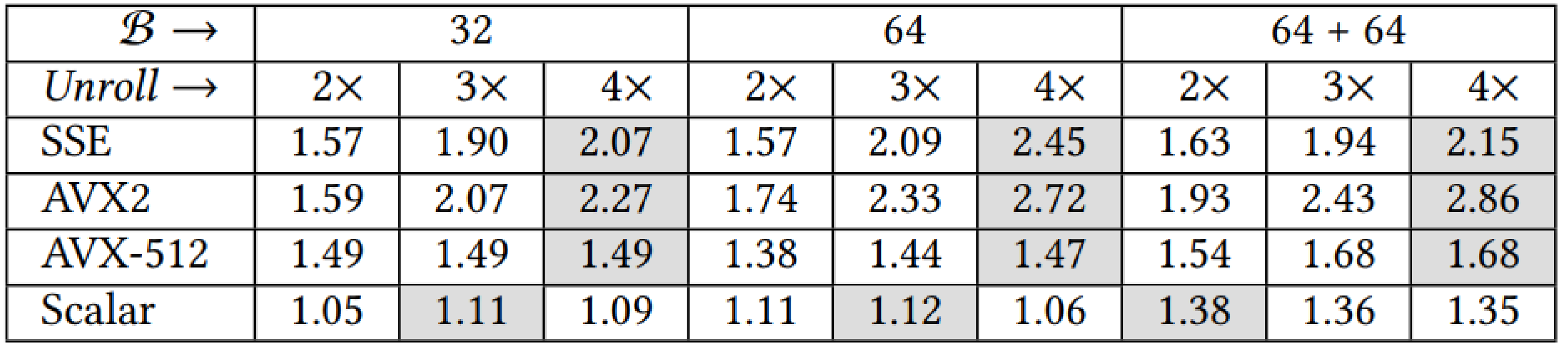
Merge Kernel

Table 3: rswap speed (B keys/s) for a size-$2\mathcal{W}$ merge



	SSE			AVX2			AVX-512		
$\mathcal{B} \rightarrow$	32	64	64+64	32	64	64+64	32	64	64+64
bitonic	2.34	1.38	1.81	2.93	1.14	0.76	3.31	1.52	0.51
rotate	4.26	1.81	1.81	3.61	1.31	1.01	3.38	1.56	0.69
`mr`	2.21	1.19	1.81	2.29	1.21	0.78	5.61	2.27	0.74

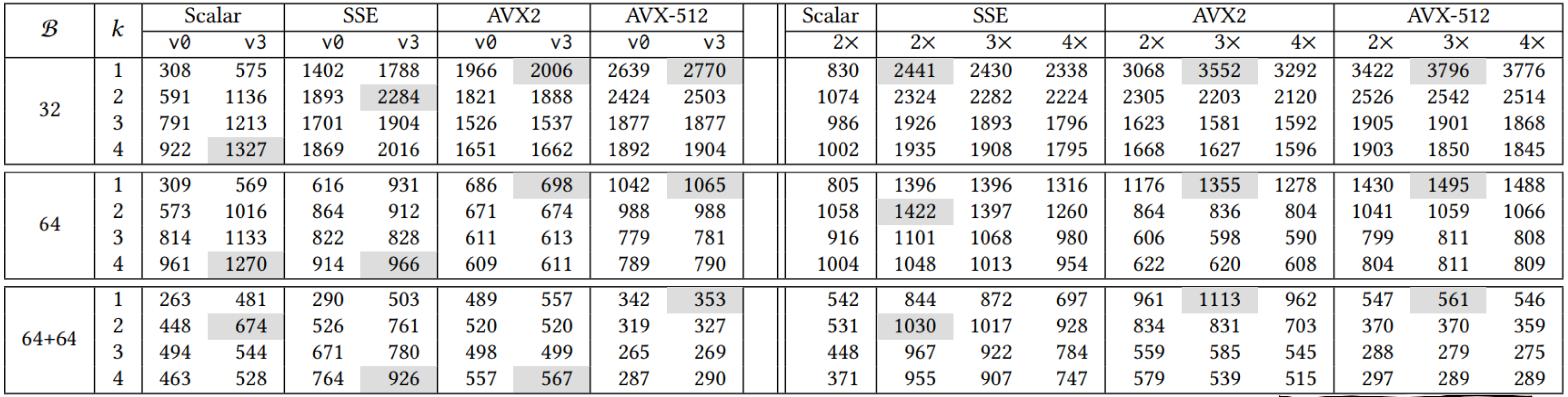
Table 4: Speed-up factors over the best speed from Table 3 for running simultaneous independent rswaps (*unrolling*)



$\mathcal{B} \rightarrow$	32			64			64 + 64		
Unroll $\rightarrow$	2×	3×	4×	2×	3×	4×	2×	3×	4×
SSE	1.57	1.90	2.07	1.57	2.09	2.45	1.63	1.94	2.15
AVX2	1.59	2.07	2.27	1.74	2.33	2.72	1.93	2.43	2.86
AVX-512	1.49	1.49	1.49	1.38	1.44	1.47	1.54	1.68	1.68
Scalar	1.05	1.11	1.09	1.11	1.12	1.06	1.38	1.36	1.35

In-cache Merge

Table 5: In-cache bmerge speed (M/s); the left half of the table compares Origami optimized branchless merge (v3) with naive branched merge (v0); the right half shows further improvement from unrolling v3 to merge multiple sequences



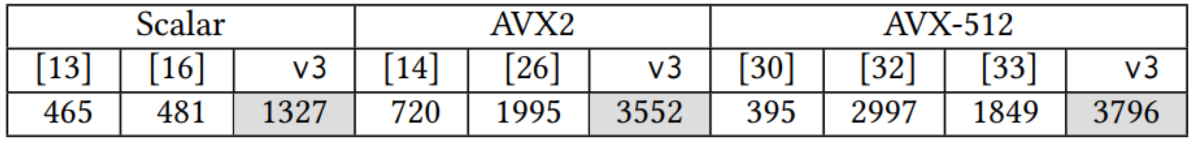
$\mathcal{B}$	k	Scalar		SSE		AVX2		AVX-512		Scalar	SSE			AVX2			AVX-512		
		v0	v3	v0	v3	v0	v3	v0	v3	2×	2×	3×	4×	2×	3×	4×	2×	3×	4×
32	1	308	575	1402	1788	1966	2006	2639	2770	830	2441	2430	2338	3068	3552	3292	3422	3796	3776
	2	591	1136	1893	2284	1821	1888	2424	2503	1074	2324	2282	2224	2305	2203	2120	2526	2542	2514
	3	791	1213	1701	1904	1526	1537	1877	1877	986	1926	1893	1796	1623	1581	1592	1905	1901	1868
	4	922	1327	1869	2016	1651	1662	1892	1904	1002	1935	1908	1795	1668	1627	1596	1903	1850	1845
64	1	309	569	616	931	686	698	1042	1065	805	1396	1396	1316	1176	1355	1278	1430	1495	1488
	2	573	1016	864	912	671	674	988	988	1058	1422	1397	1260	864	836	804	1041	1059	1066
	3	814	1133	822	828	611	613	779	781	916	1101	1068	980	606	598	590	799	811	808
	4	961	1270	914	966	609	611	789	790	1004	1048	1013	954	622	620	608	804	811	809
64+64	1	263	481	290	503	489	557	342	353	542	844	872	697	961	1113	962	547	561	546
	2	448	674	526	761	520	520	319	327	531	1030	1017	928	834	831	703	370	370	359
	3	494	544	671	780	498	499	265	269	448	967	922	784	559	585	545	288	279	275
	4	463	528	764	926	557	567	287	290	371	955	907	747	579	539	515	297	289	289

SSE: 47%

AVX2: 96%

AVX-512: 59%

Table 6: In-cache bmerge speed (M/s); $\mathcal{B} = 32$

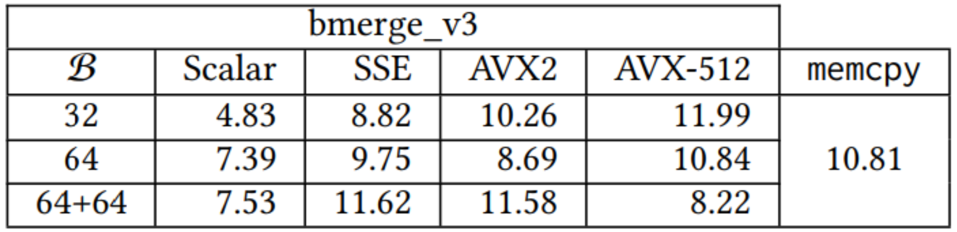


Scalar			AVX2			AVX-512			
[13]	[16]	v3	[14]	[26]	v3	[30]	[32]	[33]	v3
465	481	1327	720	1995	3552	395	2997	1849	3796

Up to 2.85x

Out-of-cache Merge

Table 7: Single-threaded memory throughput (GB/s)



	bmerge_v3				
$\mathcal{B}$	Scalar	SSE	AVX2	AVX-512	memcpy
32	4.83	8.82	10.26	11.99	
64	7.39	9.75	8.69	10.84	10.81
64+64	7.53	11.62	11.58	8.22	

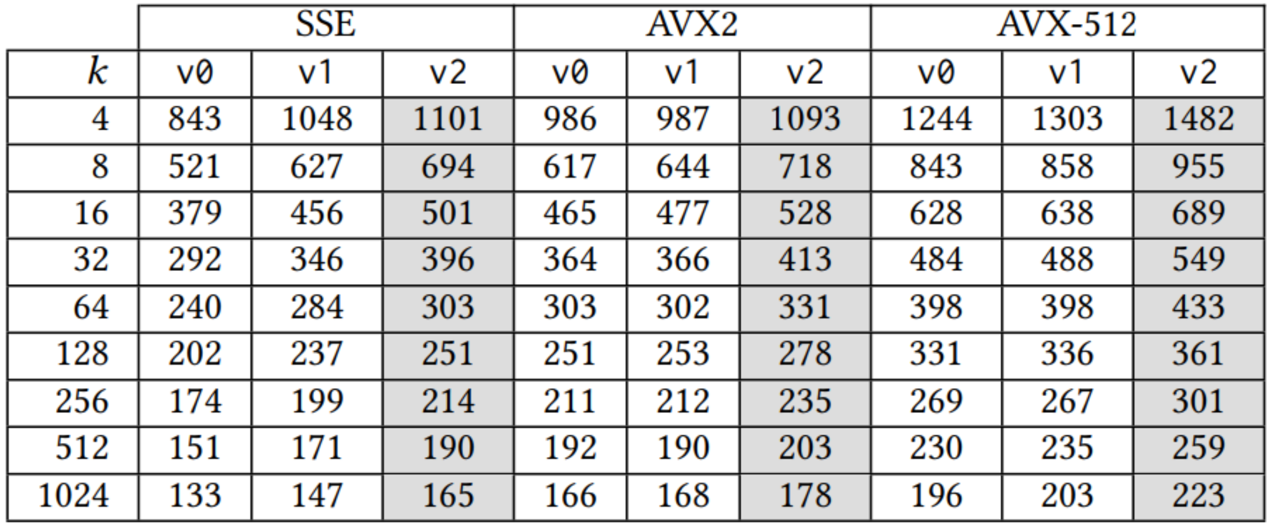
Est. upper bound for single core: 12.6 GB/s

Table 8: Single-threaded mtree speed (M/s); $\mathcal{B} = 32$

v0: bmerge_v0 + binary tree

v1: bmerge_v3 + binary tree

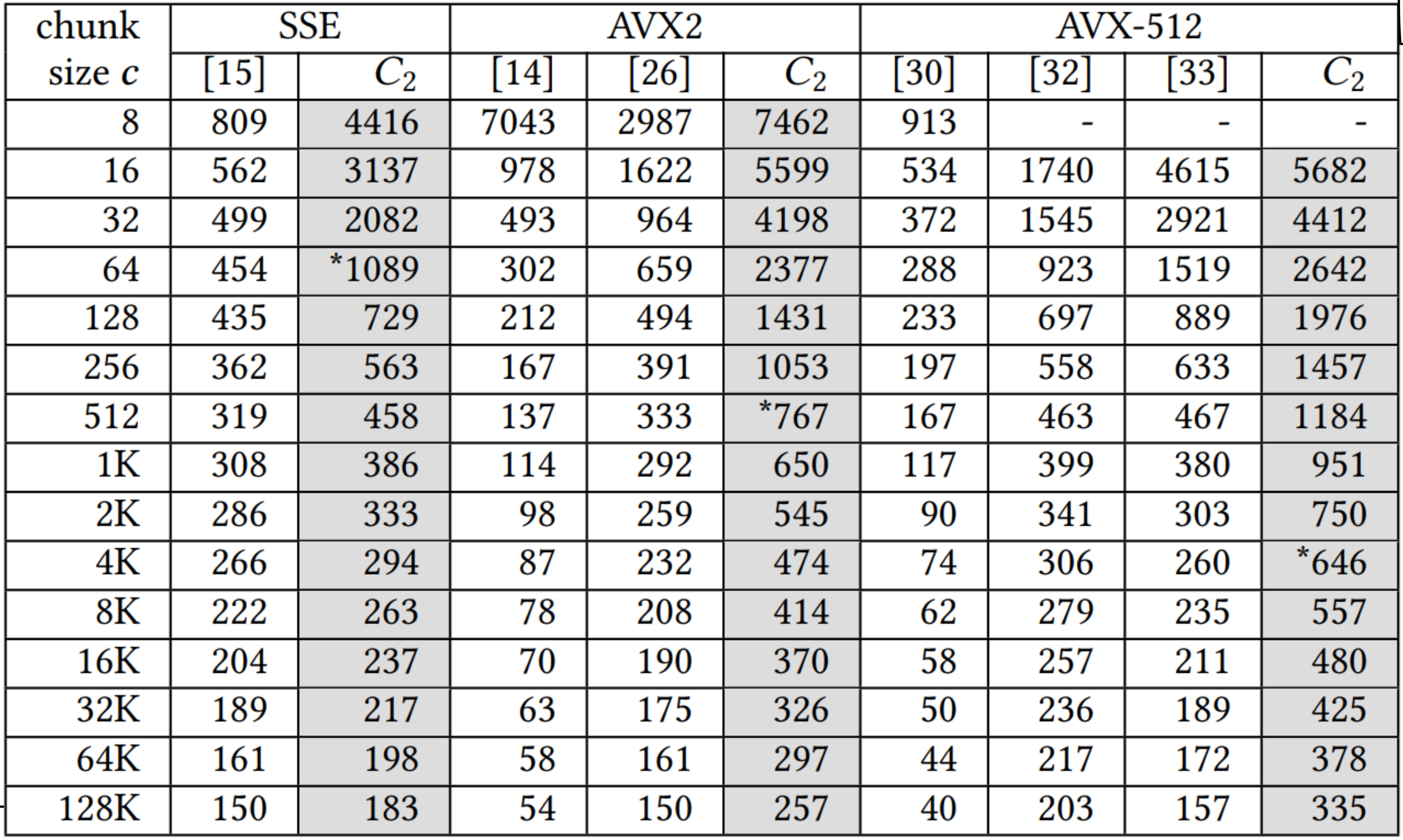
v2: bmerge_v3 + quad node tree



	SSE			AVX2			AVX-512		
k	v0	v1	v2	v0	v1	v2	v0	v1	v2
4	843	1048	1101	986	987	1093	1244	1303	1482
8	521	627	694	617	644	718	843	858	955
16	379	456	501	465	477	528	628	638	689
32	292	346	396	364	366	413	484	488	549
64	240	284	303	303	302	331	398	398	433
128	202	237	251	251	253	278	331	336	361
256	174	199	214	211	212	235	269	267	301
512	151	171	190	192	190	203	230	235	259
1024	133	147	165	166	168	178	196	203	223

Chunked-sort (In-cache)

Table 10: Chunked speed in C_2 (M/s); $\mathcal{N} = 128K$, $\mathcal{B} = 32$



chunk	SSE		AVX2			AVX-512			
size c	[15]	C_2	[14]	[26]	C_2	[30]	[32]	[33]	C_2
8	809	4416	7043	2987	7462	913	-	-	-
16	562	3137	978	1622	5599	534	1740	4615	5682
32	499	2082	493	964	4198	372	1545	2921	4412
64	454	*1089	302	659	2377	288	923	1519	2642
128	435	729	212	494	1431	233	697	889	1976
256	362	563	167	391	1053	197	558	633	1457
512	319	458	137	333	*767	167	463	467	1184
1K	308	386	114	292	650	117	399	380	951
2K	286	333	98	259	545	90	341	303	750
4K	266	294	87	232	474	74	306	260	*646
8K	222	263	78	208	414	62	279	235	557
16K	204	237	70	190	370	58	257	211	480
32K	189	217	63	175	326	50	236	189	425
64K	161	198	58	161	297	44	217	172	378
128K	150	183	54	150	257	40	203	157	335

Define Checkpoint

C_i = execution of phases P_1 through P_i

SSE: 22%

AVX2: 71%

AVX-512: 65%

Chunked-sort (Out-of-cache)

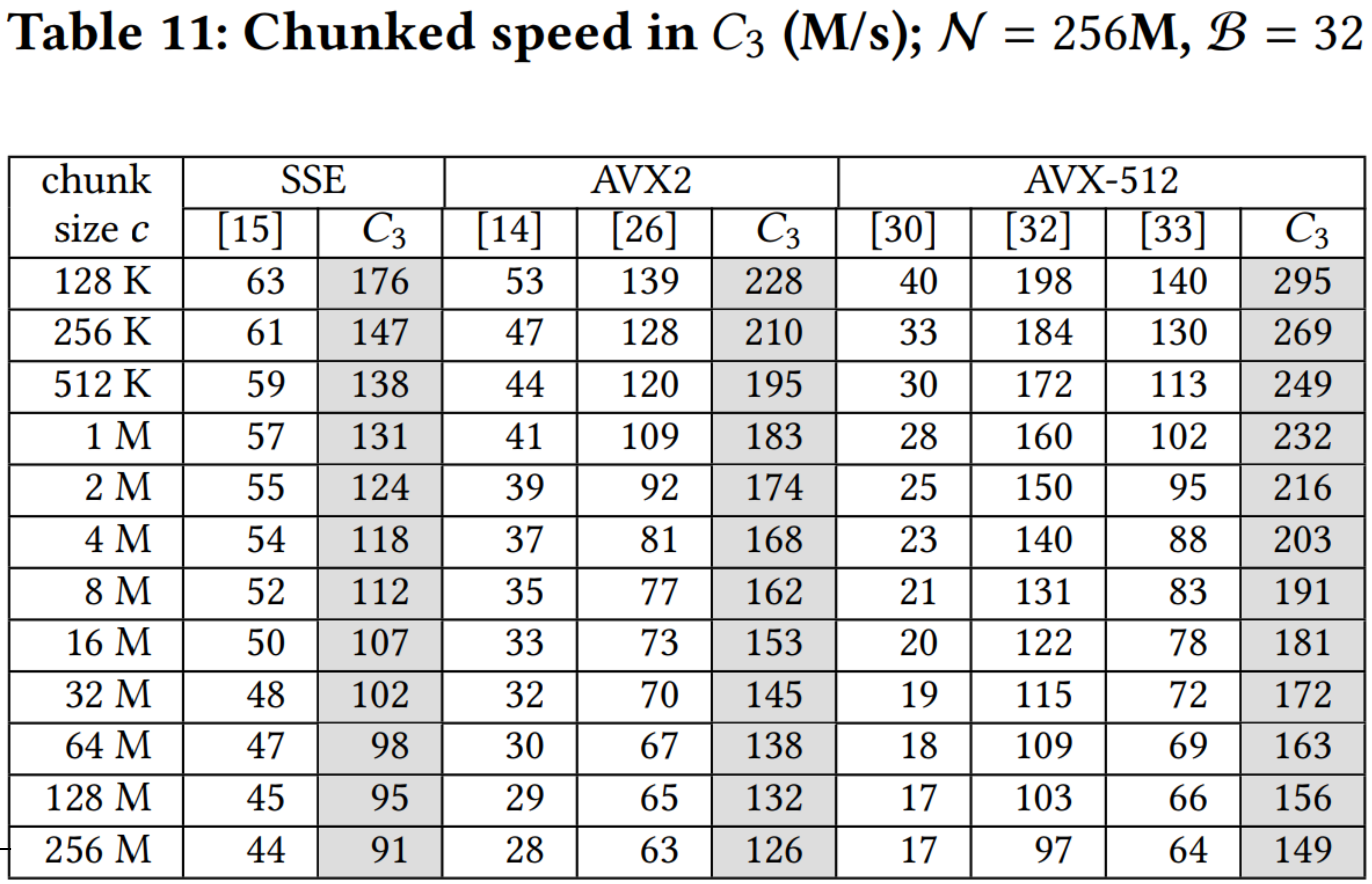


Table 11: Chunked speed in C_3 (M/s); $\mathcal{N}$ = 256M, $\mathcal{B}$ = 32

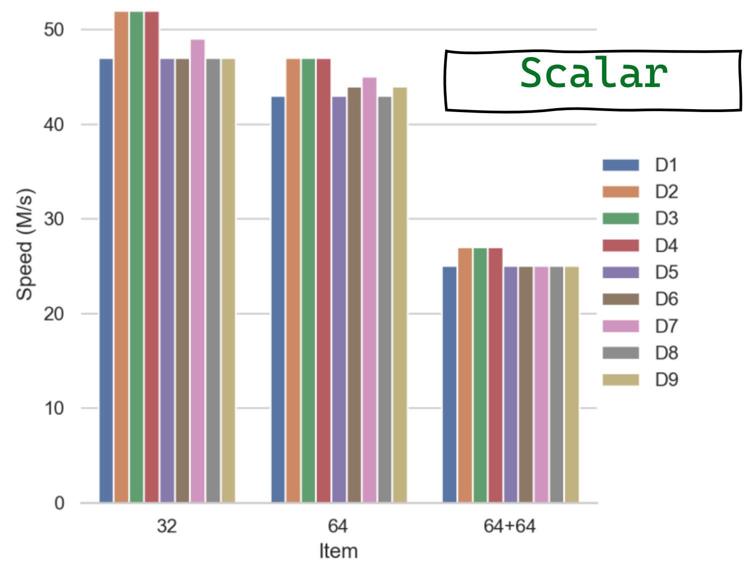
chunk size c	SSE		AVX2			AVX-512			
	[15]	C_3	[14]	[26]	C_3	[30]	[32]	[33]	C_3
128 K	63	176	53	139	228	40	198	140	295
256 K	61	147	47	128	210	33	184	130	269
512 K	59	138	44	120	195	30	172	113	249
1 M	57	131	41	109	183	28	160	102	232
2 M	55	124	39	92	174	25	150	95	216
4 M	54	118	37	81	168	23	140	88	203
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

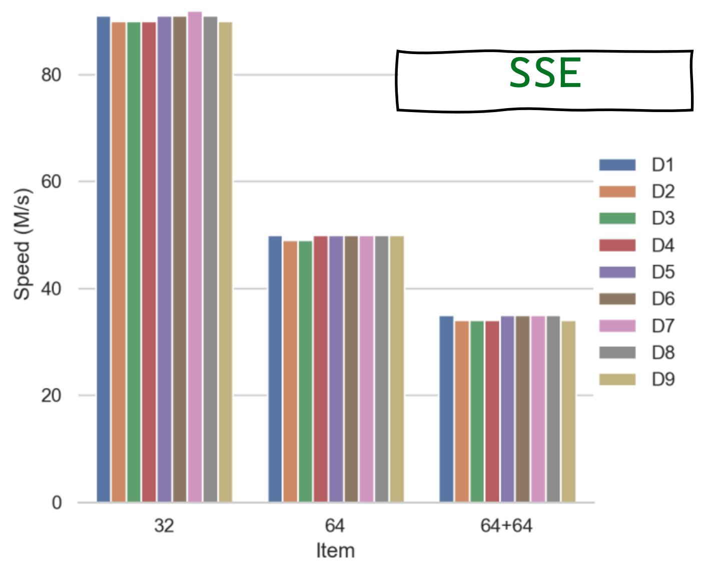
SSE: 110%

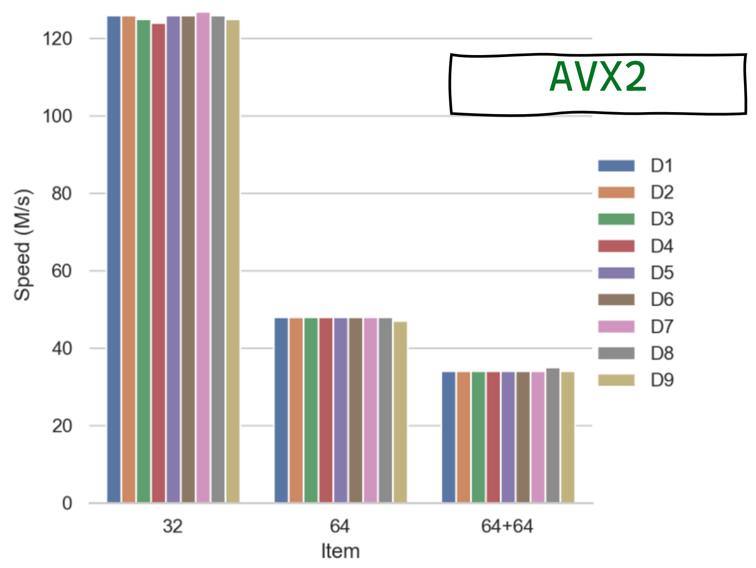
AVX2: 100%

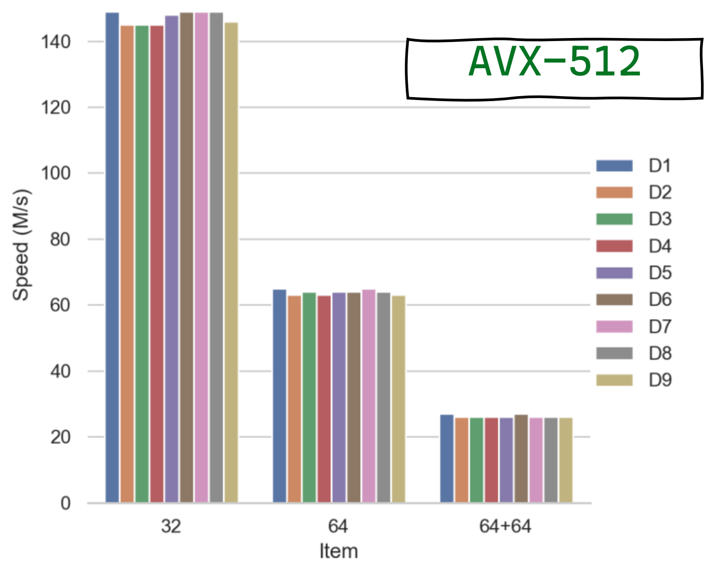
AVX-512: 53%

Distribution Insensitivity





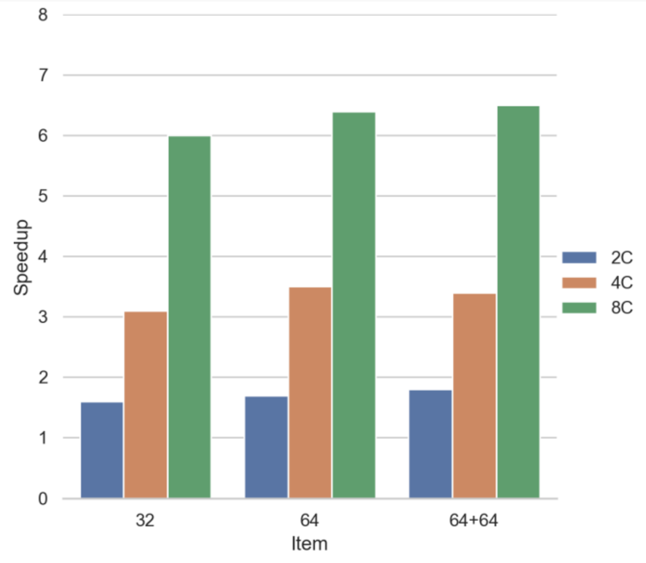




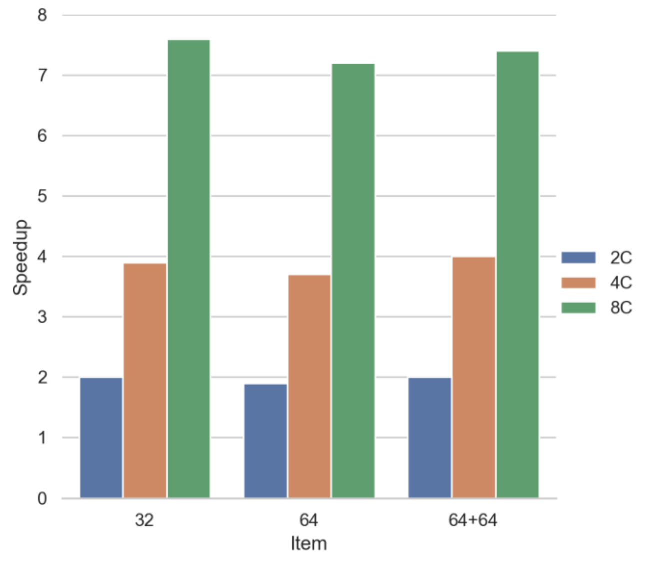
D1: Uniform
D2: All same
D3: Sorted
D4: Reverse sorted
D5: Almost sorted (7th = MAX)
D6: Pareto
D7: Bursts of same keys (length from D6, key from D1)
D8: Random shuffle of D7
D9: Fibonacci

Multi-core Speedup

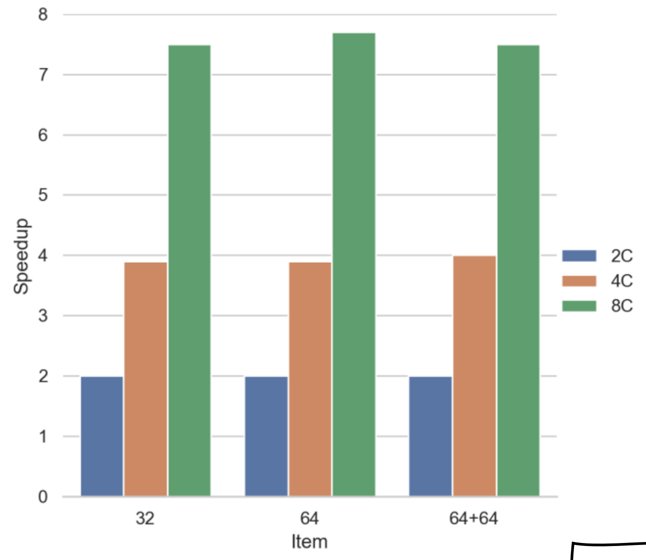
Scalar



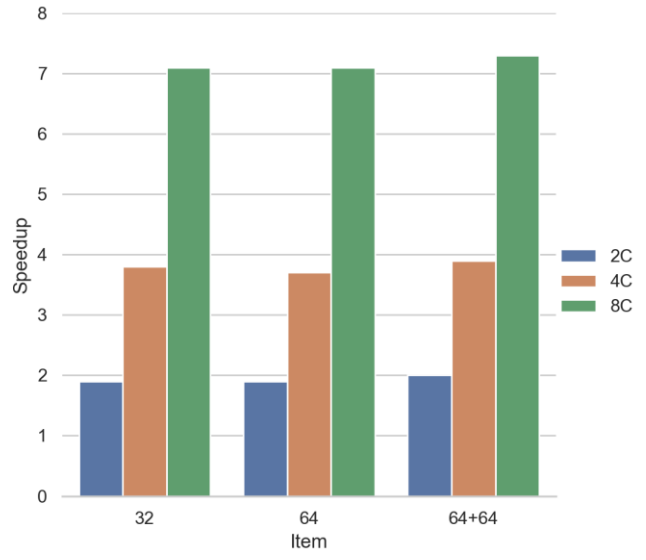
SSE



AVX2



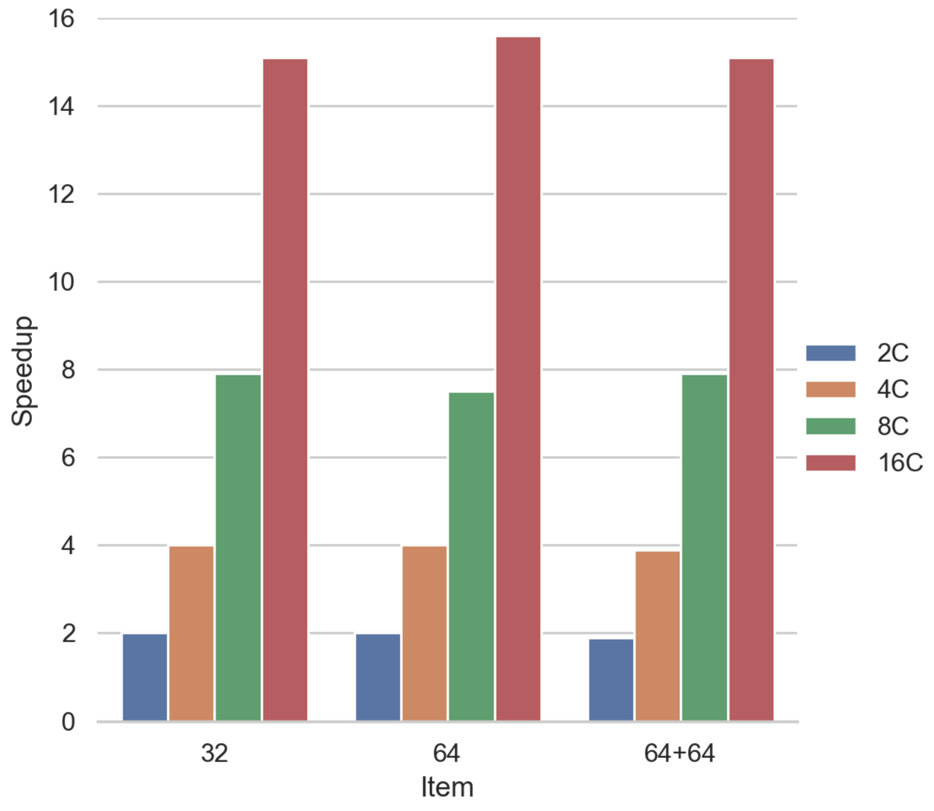
AVX-512



1 GB

Multi-core Speedup (Xeons)

SSE

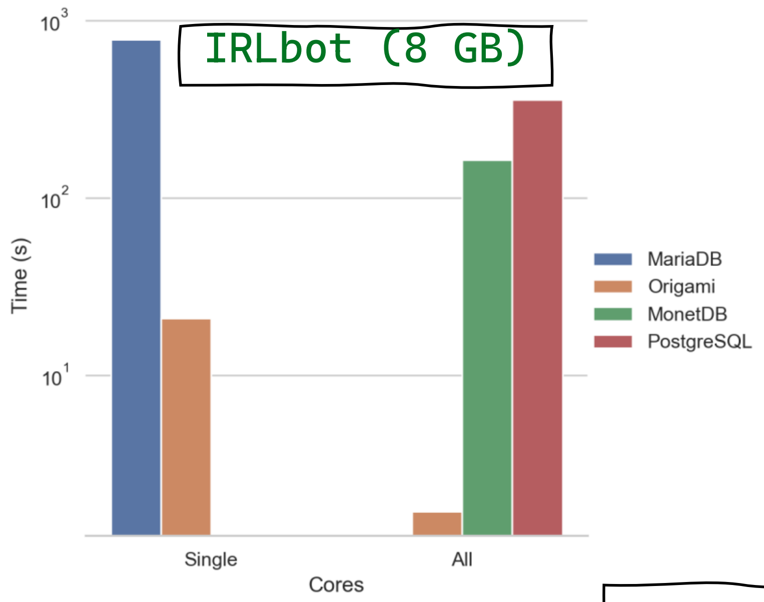


64 GB

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
```

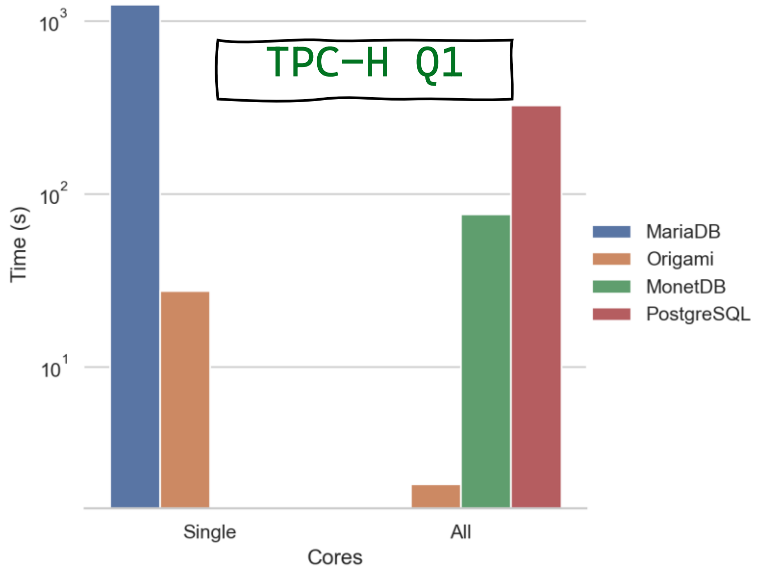


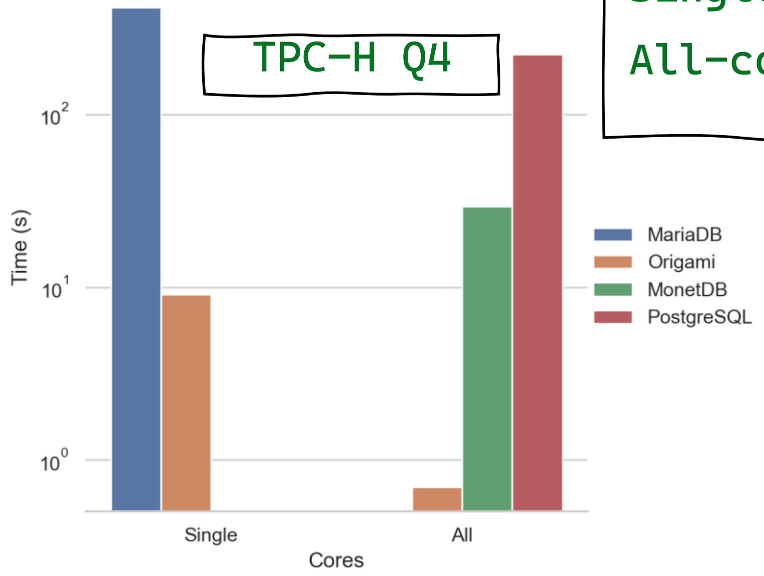
Single-core: 37-60x

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