ICDE 2026
MONTRÉAL, CANADA

F5: A Robust SIMD-Accelerated MSD Radix Sort

Arif Arman and Dmitri Loguinov

TEXAS A&M
UNIVERSITY

Agenda

- **Introduction and Motivation**
- Micro-Sorts (In-Register)
- Experiments
- Conclusion

Introduction

- Sorting is a ubiquitous building block of many data engineering applications
- MSD (most-significant digit first) radix sort methods are the fastest for longer keys (e.g., 64-bit)

Algorithm 1: Distribution-based divide-and-conquer sort

```
1 Function Sort(input, n)
2     [B_0, ..., B_{k-1}] = Partition(input, n);    ▷ create k buckets
3     for (i = 0; i < k; i++) do
4         m = size of bucket B_i;
5         if m ≤ T then    ▷ below threshold?
6             SmallSort(B_i, m);    ▷ sort & append to output
7         else
8             Sort(B_i, m);    ▷ recurse on the bucket
```

Motivation

- Existing SmallSorts are inefficient and lack generality
- Accounts for 45% of total cost in SOTA
- First contribution: A family of novel SIMD (Single Instruction, Multiple Data) micro-sorts for buckets fitting entirely in CPU vector registers
  - Streamlined to sort non-square SIMD matrices, including cases with odd number of rows
- Second contribution: Out-of-register SIMD mini-sorts to extend SmallSort threshold
  - Starting with a micro-sort of size m', produce sequences of $2m', 4m', \ldots$ using a novel vectorized merge with reduced register spill

Motivation

- Performance issues in existing MSD partitioning
- Works well on uniform keys, but chokes when batches of adjacent keys go into the same bucket
- Third contribution: A novel partition engine that includes
  - A pointer update method to avoid store-to-load forwarding stalls
  - A SIMD stream engine for rapid processing of long burst of keys into identical target bucket

Motivation

- Lack of rigorous foundation for selecting bucket count k
- In certain cases, cannot avoid maximum recursion depth
- Final contribution: A set of low-overhead techniques
  - Adaptive selection of k at each level of recursion
  - Detect common prefix to avoid redundant partitioning
  - Detect already sorted buckets to terminate early

- Due to time limitations, only the first contribution is presented

Agenda

- Introduction and Motivation
- **Micro-Sorts (In-Register)**
- Experiments
- Conclusion

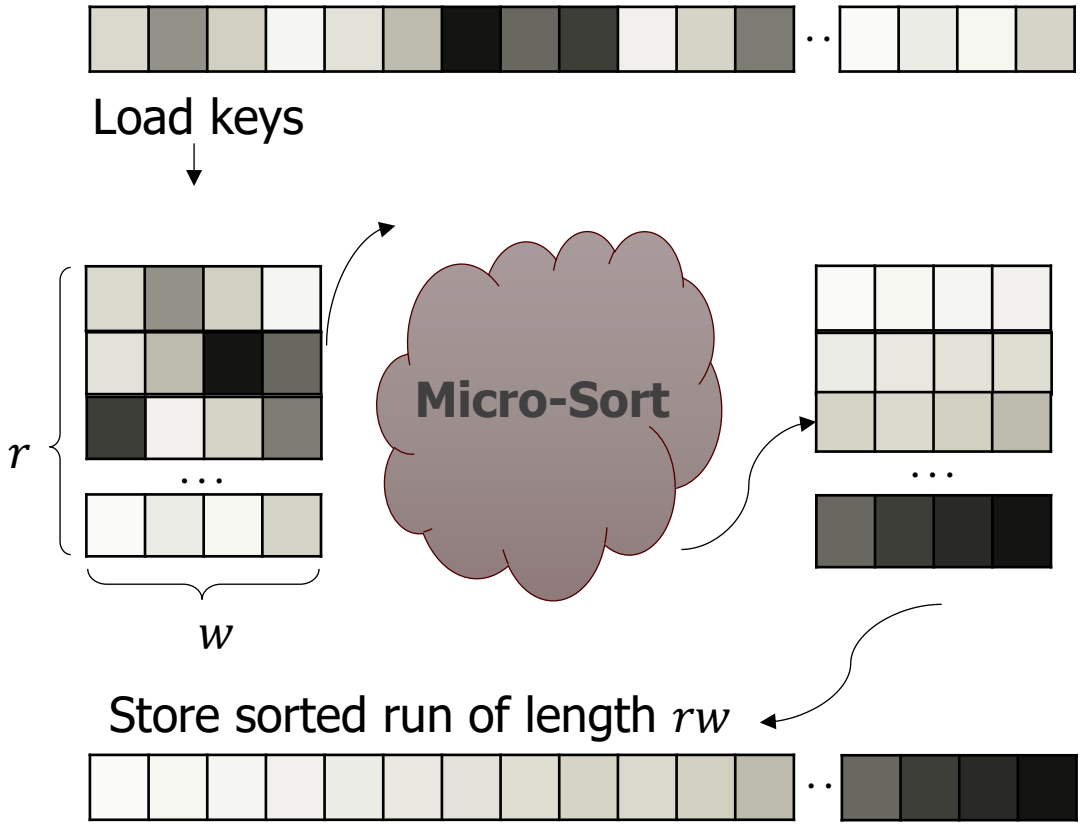
Sorting Networks

- For a given array $a_0, \ldots, a_{m-1}$, define $swap(u, v)$:
  - If $a_u > a_v$, exchange a_u and a_v
  - Else, leave them untouched
- A scalar sorting network ϕ_m is a fixed sequence of swap index pairs $(u, v): u < v$ and $0 \leq u, v \leq m - 1$
- Swaps can be implemented with branchless min-max
- Unrolling the branchless sequences of a network allows
  - CPU pipeline to maximally reorder instructions
  - Execute independent swaps in parallel
- As a result, for small m, sorting networks are significantly faster than classical alternatives

SIMD Sorting Networks

- Benefits of sorting networks in existing MSD frameworks are greatly reduced due to high misprediction penalty
  - Invoke a network through function pointer $f[m]$
  - m varies unpredictably in $[2, \mathcal{T}]$
  - Example: scalar $f[8]$ goes from 3 cycles/key to 9 cycles/key
- Solution: Increase m to reduce the relative cost of branch misprediction compared to useful work done by $f[m]$
  - Prohibitively expensive with scalar approaches
  - We leverage SIMD to build faster networks for sufficiently large m
- Vector registers in modern CPUs enable
  - Performing multiple swaps in ϕ_m with one SIMD instruction
  - Holding KBs of data in-register (e.g., 2 KBs with AVX-512)

Micro-Sorts (In-Register)



r_0 | 14 | 6 | 8 | 11
r_1 | 4 | 12 | 2 | 15

$swap(r_0, r_1)$

r_0 | 4 | 6 | 2 | 11
r_1 | 14 | 12 | 8 | 15

$r_0 = blend(r_0, r_1, b_{mask})$

r_0 | 4 | 6 | 8 | 15

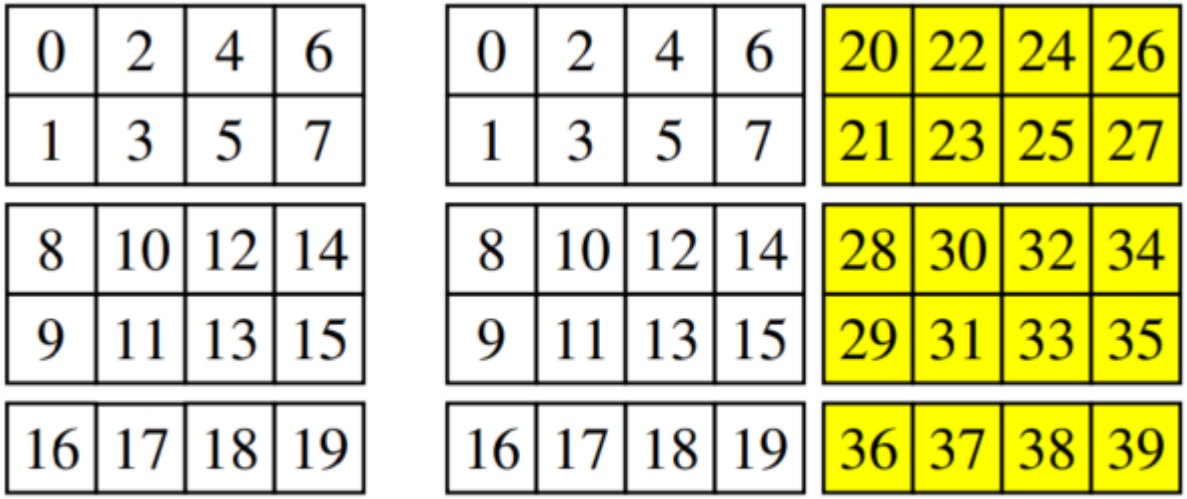
$r_0 = shuffle(r_0, s_{mask})$

r_0 | 6 | 4 | 15 | 8

» Goal is efficient SIMD conversion of ϕ_{rw} while optimizing vectorized swap count and the shuffle/blend overhead

Matrix Binary Merge

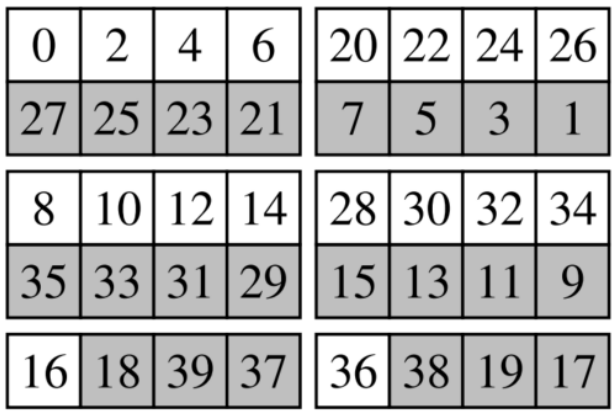
- We use Batcher's sorting network as our building block
  - Constructed using repeated binary merges
  - Each sorted sequence is arranged as a $r \times k$ matrix in a special k-zigzag order, with a special odd-row pattern
  - We build a joint matrix by combining the k-zigzag matrices
  - Sorting the joint matrix ≡ merging the two sequences



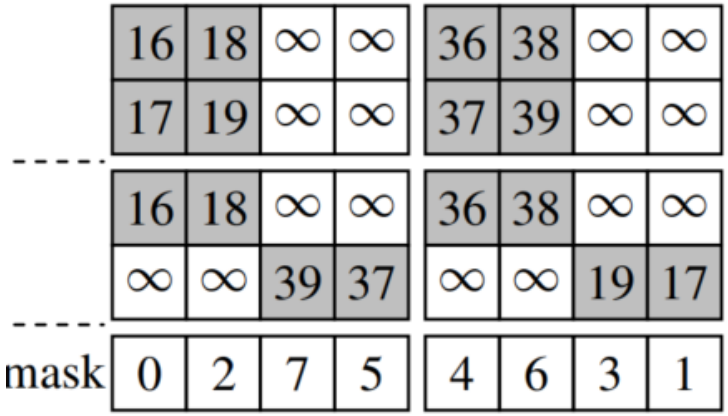
(a) 4-zigzag (b) joint matrix

Matrix Binary Merge

- Merge is completed using a three-step process
  - Reverse every odd row
    - For odd r, the last row $r-1$ needs a custom permutation



(c) row reversal



(d) shuffling last row

Matrix Binary Merge

- Merge is completed using a three-step process
  - Reverse every odd row
    - For odd r, the last row $r-1$ needs a custom permutation
  - Vertical sort along each column
    - Each row becomes a bitonic sequence
    - Largest item in $r_i \leq$ smallest item in r_{i+1}

0	2	4	6	20	22	24	26
27	25	23	21	7	5	3	1
8	10	12	14	28	30	32	34
35	33	31	29	15	13	11	9
16	18	39	37	36	38	19	17

(c) row reversal

0	2	4	6	7	5	3	1
8	10	12	14	15	13	11	9
16	18	23	21	20	22	19	17
27	25	31	29	28	30	24	26
35	33	39	37	36	38	32	34

(e) vertical sort

Matrix Binary Merge

- Merge is completed using a three-step process
  - Reverse every odd row
    - For odd r, the last row $r-1$ needs a custom permutation
  - Vertical sort along each column
    - Each row becomes a bitonic sequence
    - Largest item in $r_i \leq$ smallest item in r_{i+1}
  - Horizontal bitonic sort on each pair of rows

0	2	4	6	7	5	3	1
8	10	12	14	15	13	11	9
16	18	23	21	20	22	19	17
27	25	31	29	28	30	24	26
35	33	39	37	36	38	32	34

(e) vertical sort

0	2	4	6	8	10	12	14
1	3	5	7	9	11	13	15
16	18	20	22	24	26	28	30
17	19	21	23	25	27	29	31
32	33	34	35	36	37	38	39

(f) finish in 8-zigzag order

Agenda

- Introduction and Motivation
- Micro-Sorts (In-Register)
- **Experiments**
- Conclusion

Micro-Sorts Performance

» Intel i7-7820X, 8-cores each @ 4.7 GHz, 32 GB DDR4-3200 Quad Channel RAM

» Performance of Micro-Sorts for 32-bit keys on SSE ($w = 4$)

Vortex (Scalar) Origami Highway

	SSE 32-bit										
	swaps				non-swaps			cycles/sort			
m	[16]	[3]	[13]	F5	[3]	[13]	F5	[16]	[3]	[13]	F5
4	5	5	60	3	0	0	5	8	12	98	9
8	19	19		6	0		11	24	30		10
12	41	20		13	54		18	59	78		19
16	60			17			22	87			22
20	97	64	108	28	80	64	29	145	127	157	37
24	127			33			33	199			42
28	161			44			40	256			55
32	191			49			44	414			61
36	268	158	172	64	176	160	51	532	278	237	79
40	305			70			55	562			86
44	347			85			62	671			104
48	384			91			66	746			113
52	423			106			73	907			129
56	464			114			77	996			137
60	506			128			84	1,062			154
64	543			134			88	1,184			162
av	246	99	128	62	115	96	47	491	183	182	76

1.6X 2X 2.4X

Out-of-Register Merge Performance

- Benchmark $m \times m$ merge such that $2m > Rw$, where R is the physical register count
- Compare F5's chain-reordering and spill management with suboptimal compiler generated mergers
- Out-of-Register $m \times m$ merge (SSE, 32-bit keys, $w = 4$)

20X

2.5X

			evictions				cycles/merge			
m	swap	shuf	VS	Clang	ICX	F5	VS	Clang	ICX	F5
64	97	82	117	122	120	6	248	266	252	106
128	225	162	411	407	408	35	779	773	759	240
256	513	322	1142	1071	1031	120	2029	1961	1915	614
512	1153	642	2793	2542	2339	354	4785	4426	4275	1537

7X

2.8X

Impact of Contributions

- Using Vortex as baseline, assess the impact of each proposed component of F5 on different datasets
- Sort speed (M keys/sec) for 1 GB of 64-bit keys

Improvements	$\mathcal{D}_1$ Uniform	$\mathcal{D}_2$ Almost Sorted	$\mathcal{D}_3$ Mid-zeros	$\mathcal{D}_4$ Pareto Freq.	$\mathcal{D}_5$ Normal	$\mathcal{D}_6$ Uniform Double	$\mathcal{D}_7$ MSD Adversarial
Vortex [16]	144	127	83	58	139	68	13
+ micro-sorts	168	141		69	169	97	121
+ adaptive radix	193	148			182		
+ mini-sorts				101	185	143	160
+ partition-v2	206	190	98	126	200	155	199
+ partition-v3		207	110	144		161	
+ prefix check		221	242	186			
+ sort check		296					
final speedup	1.4×	2.3×	2.9×	3.2×	1.4×	2.4×	15×

Full Sort Performance (64-bit Keys)

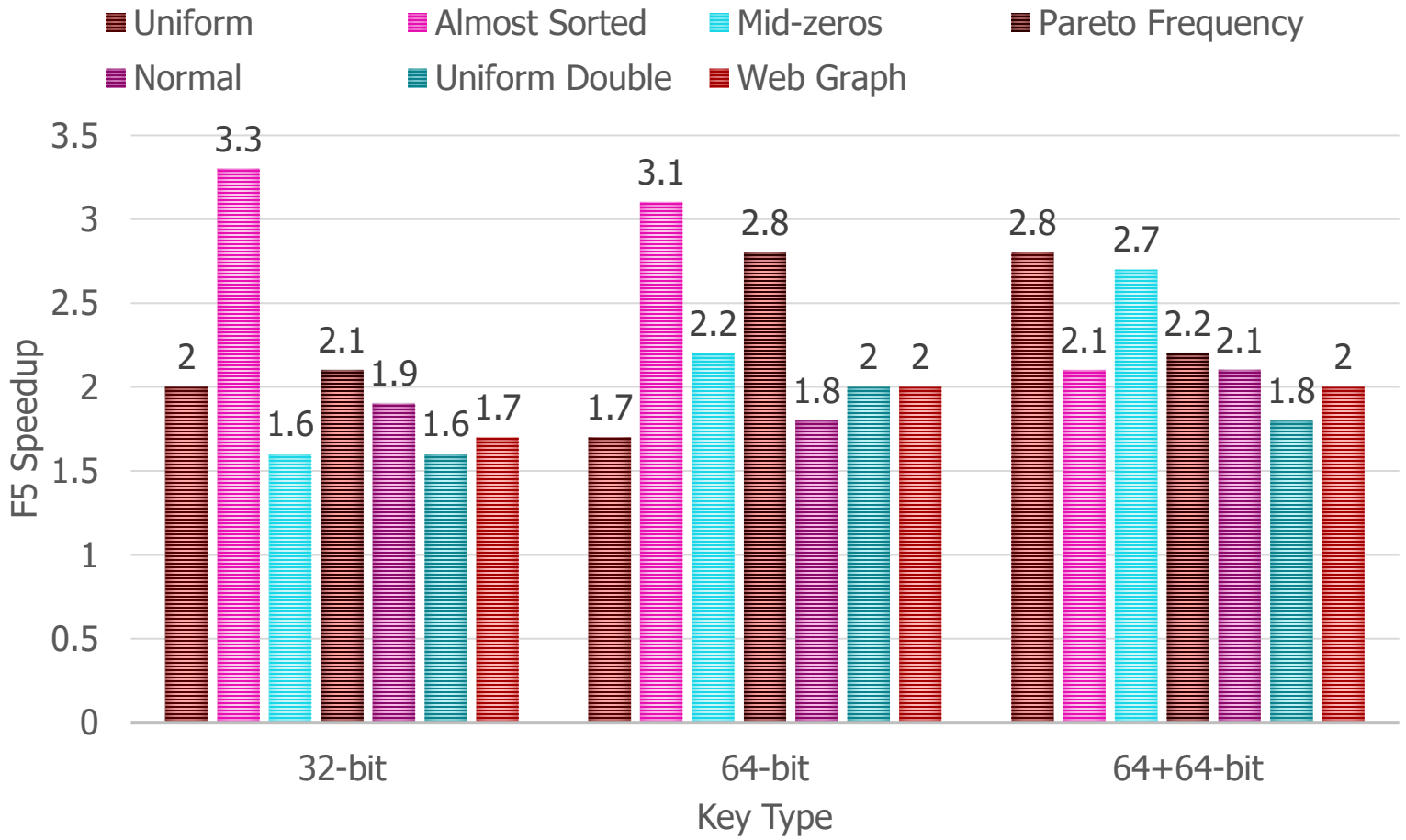
» Sort speed (M keys/sec) for 8 GB of 64-bit keys

Graph inversion on Web Graph

Sort	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{D}_6$	$\mathcal{G}_2$
Polychroniou [30]	27	24	11	23	26	12	–
Ska [34]	36	79	56	51	36	34	32
Raduls2 [23]	82	65	49	48	96	67	53
Regions [28]	49	31	70	27	42	38	29
Voracious [29]	57	60	55	65	58	45	56
Vortex [16]	120	98	56	52	114	78	57
IPS^{2}Ra [5]	45	87	58	60	40	40	38
Dovetail [12]	37	37	37	36	38	37	62
Reinald [31]	39	38	41	39	40	38	37
Fast-Radix [36]	40	40	49	46	38	39	38
DFR [35]	49	52	102	–	49	36	–
IPS^{4}o [5]	31	41	86	52	29	29	27
Highway [13]	57	63	94	65	55	55	54
Intel-sort [19]	76	79	109	46	73	74	69
Origami [3]	56	55	57	56	56	56	53
F5	206	308	235	182	206	156	136
	1.7×	3.1×	2.2×	2.8×	1.8×	2.0×	2.0×

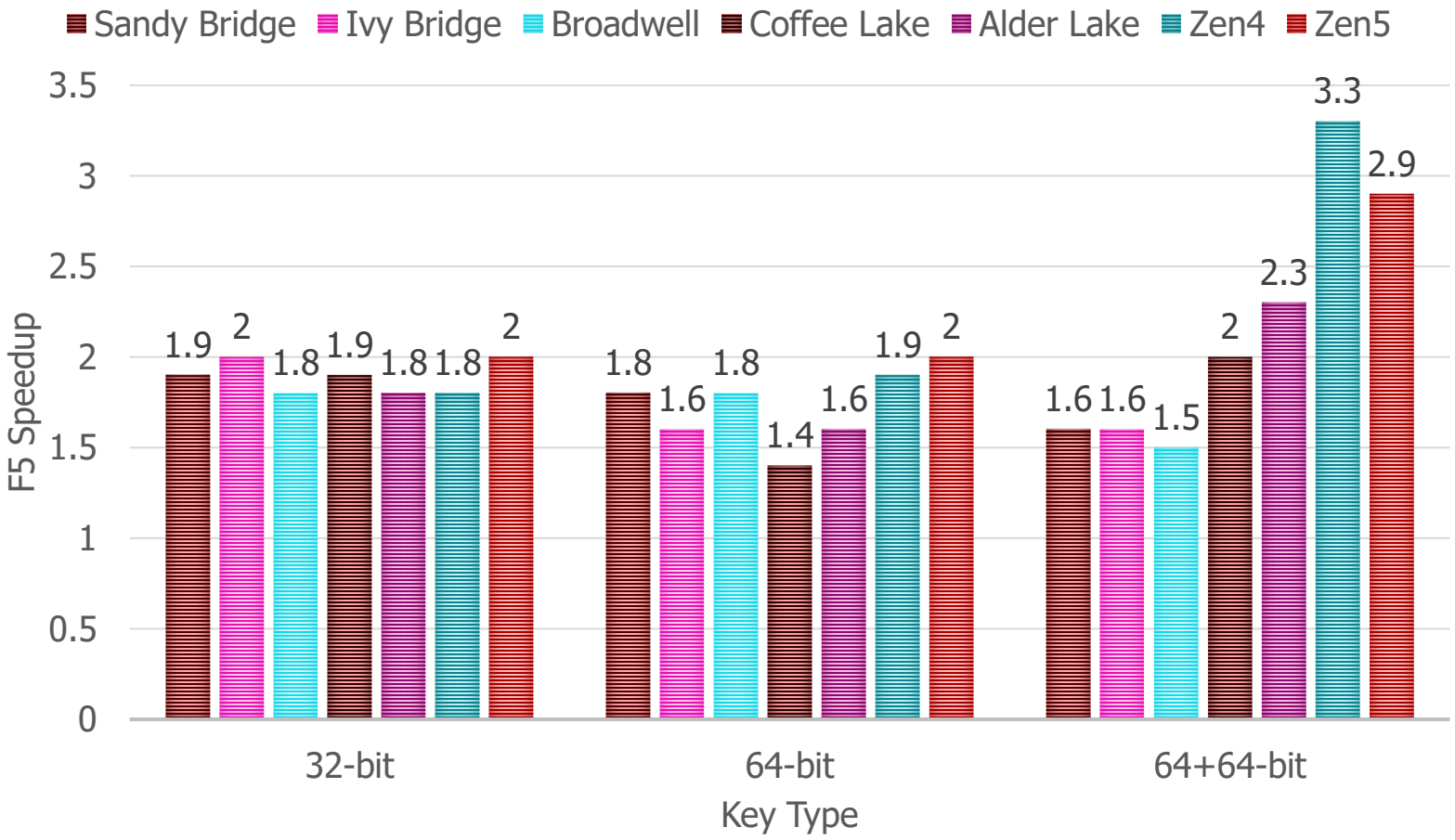
Full Sort Performance (Summary)

» F5 speedup over the corresponding closest competitor for each dataset on 8 GB sorts, with varying key types



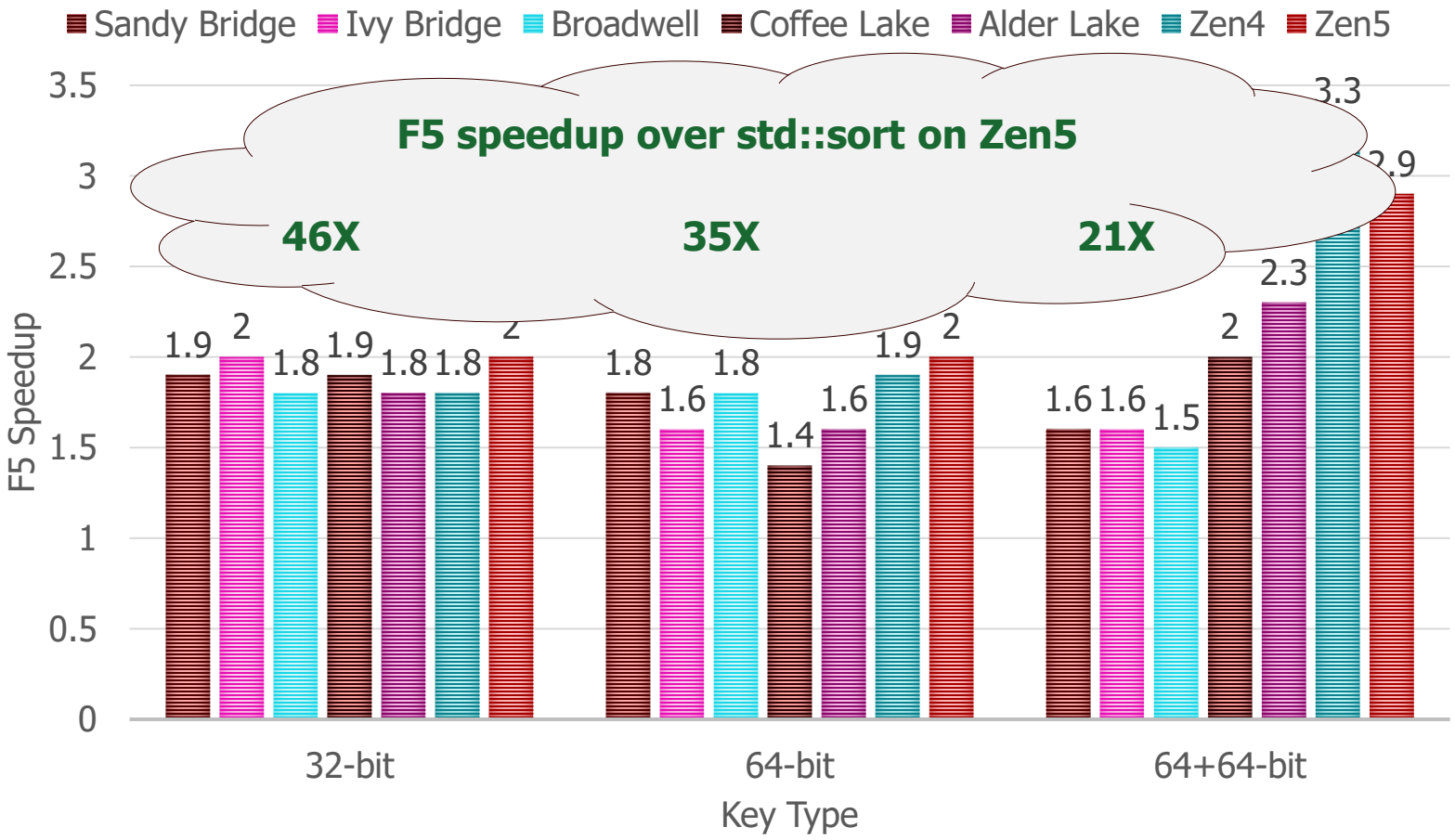
Full Sort Performance (Cross Platform Summary)

» F5 speedup over the corresponding closest competitor on each platform for 8 GB uniform data, with varying key types



Full Sort Performance (Cross Platform Summary)

» F5 speedup over the corresponding closest competitor on each platform for 8 GB uniform data, with varying key types



Agenda

- Introduction and Motivation
- Micro-Sorts (In-Register)
- Experiments
- **Conclusion**

Conclusion

- F5 presents a suite of novel algorithms for
  - Sorting SIMD matrices in-register
  - Extending them to out-of-register operation
  - Improving MSD partitioning engine
  - Solving optimal selection of radix bits
  - Stopping recursion early with low-overhead detections
- Results show significant speed gain over prior work
  - On uniform, skewed, and real-life workloads
  - On different key/key-value types
  - On multiple platforms
- Future work involves multi-threading F5, dealing with even longer records etc.

Conclusion

Questions?