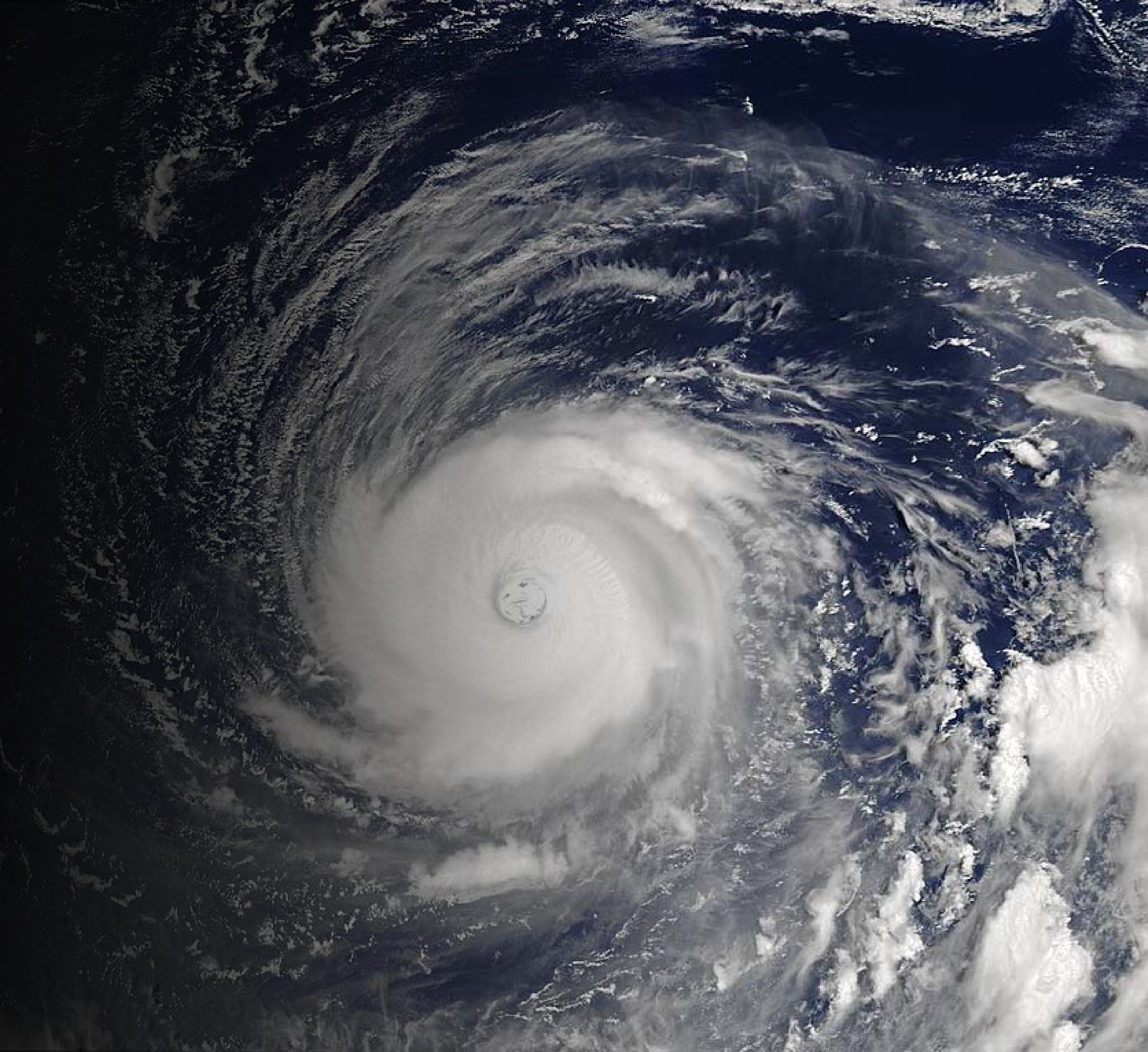
Typhoon: A Slice-Scrambled In-Place LSD Sort

Zelun Liu, Arif Arman, Dmitri Loguinov

Texas A&M University



Agenda

I. **Motivation**

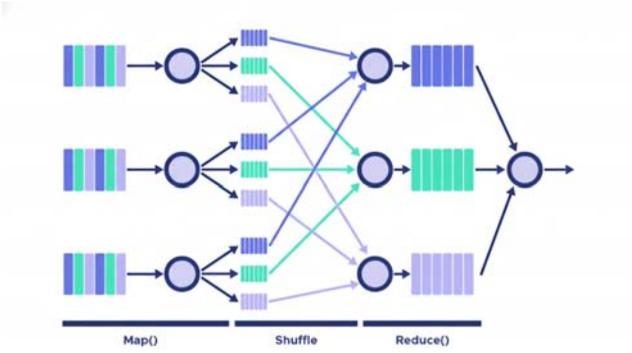
II. Static Typhoon (S-Typhoon)

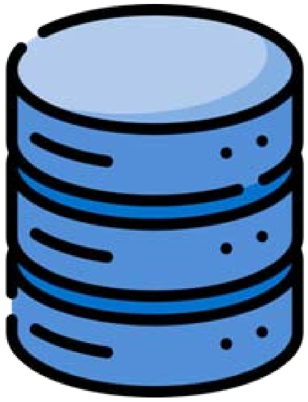
III. Typhoon

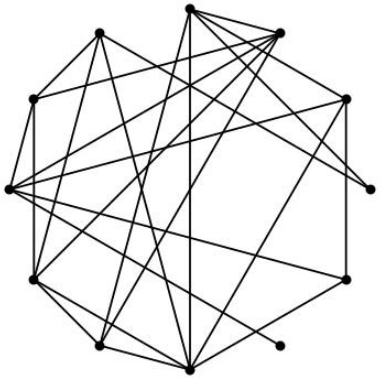
IV. Experiments

Motivation

- Sorting has become a ubiquitous building block behind many big-data computational frameworks and distributed systems







MapReduce
Sorts key-value pairs

Databases
ORDER BY, GROUP BY,
sort-merge join

Graph Mining
PageRank,
graph inversion

Motivation

- Sort performance can be formalized into 5 main parameters
  - Single-threaded speed
  - Robustness against non-uniform keys
  - Memory usage
  - Stability
  - Multi-core scaling behavior
- Existing methods exhibit tradeoffs between these objectives
  - Some are fast, but unstable or single-threaded only
  - Others are out-of-place or slow
  - Yet others can be fast on uniform keys & in-place, but slow on skewed distributions

Motivation

- LSD radix sort is stable and insensitive to key distribution
- However, for n input items
  - 2n RAM usage (out-of-place)
  - Histogram pass on each level
  - Chokes on bursts of keys going into the same bucket
- 4n memory traffic per level
  - 1n histogram
  - 1n read input
  - 1n read for ownership on destination buckets
  - 1n write to output
- Can we do better?

Algorithm 1: Textbook LSD

```
Func LSD(Item *input, int n)
    allocate aux array of size n
    for (L = 0; L < ⌈w/b⌉; L++) do
        if (L & 1) == 0 then
            Split(input, n, aux, L);    ▷ even level
        else
            Split(aux, n, input, L);    ▷ odd level

Func Split(Item *in, int n, Item *out, int L)
    buck = Histogram(in, out, L);    ▷ set up pointers in out array
    for (i = 0; i < n; i++) do
        idx = ExtractIdx(in[i], L);    ▷ bucket index
        *buck[idx]++ = in[i];    ▷ write item, increment pointer
```

Agenda

I. Motivation

II. Static Typhoon (S-Typhoon)

III. Typhoon

IV. Experiments

S-Typhoon: Overview

- Omit histogram pass
  - Static buckets allocated by oracle to correct size
- Avoid read-for-ownership using Write-Combine (WC)
  - First write to in-cache tmp memory, then stream data using *non-temporal* stores to RAM
  - 2n memory traffic per level
- Examine fastest prior solution from Vortex (ASPLOS 2020)
  - Call this WCv1
  - Significant speed reduction on runs of duplicate keys

Algorithm 2: WCv1

```
Func Split(Item *in, int n, Item *out, int L)
    buck = Histogram(in, out, L);
    for (i=0; i < n; i++) do
        prefetch (in + i + D);
        idx = ExtractIdx(in[i], L);
        p = tmpBuckets + idx*B;
        p[tmpSize[idx]] = in[i];
        if ++tmpSize[idx] == B then
            OffloadAVX(buck[idx], p);
            buck[idx] += B;
            tmpSize[idx] = 0;

Func OffloadAVX(__m256i *dest, __m256i *src)
    for (i=0; i < R / sizeof(__m256i); i++) do
        x = _mm256_load_si256(src + i);
        _mm256_stream_si256(dest + i, x);
```

TABLE I
WCv1A Speed

run len	M/sec	c/key
1	1,121	4.2
4	938	5.0
16	826	5.7
512	883	5.3

S-Typhoon: Read-After-Write Dependencies

- We uncover that load-to-store forwarding stalls are responsible for loss of performance
- New solution WCv2
  - Simultaneously reads multiple keys and loads their buckets pointers
  - Uses conditional moves (cmov) to resolve conflicts (i.e., adjacent keys going to the same bucket)
  - Avoids read-after-write dependencies using a branchless solution
- No reduction in performance compared to uniform keys

Algorithm 5: WCv2

```
Func Split(Item *in, int n, Item **buck, Item **t, int L)
    for (x = in; x < in + n; x += 2) do
        prefetch (x + D);
        MOVE(x);

Macro MOVE(x)
    idx0 = *((uint8*)x+L);
    idx1 = *((uint8*)(x+1)+L);
    p0 = t[idx0]; p1 = t[idx1];
    WRITE(x[0], p0, idx0);
    p1 = (idx0 == idx1) ? p0 : p1;
    WRITE(x[1], p1, idx1);

Macro WRITE(key, p, idx)
    *p++ = key;          ▷ store item
    if (p & (R-1) == 0) then      ▷ overflow?
        p −= B;      ▷ roll back to start of bucket
        OffloadAVX(buck[idx], p);
        buck[idx] += B;
    t[idx] = p;
```

TABLE IV
WCv2 Speed

run len	M/sec	c/key
1	1,128	4.2
4	1,128	4.2
16	1,118	4.2
512	1,302	3.6

S-Typhoon: Histogram

- The same performance problem arises for basic histograms (Hv1)
  - 60% loss of speed on bursty input
- This can be improved using parallel updates to k histograms (Hv3)
  - Better performance, but not ideal
  - Exhibits 4K aliasing and L1 cache-set conflicts
- By offsetting the start of each histogram
  - Speed remains constant for all run lengths
  - Even 30% faster on uniform compared to Hv1

Algorithm 3: Histogram Hv1

```
Func Hist(Item *in, int n)
    for (i=0; i < n; i ++) do
        prefetch (in + i + D);
        idx = *(uint8*)(in + i);
        hist[idx]++;
```

TABLE II
Hv1 Speed

run len	M/sec	c/key
1	2,250	2.1
4	1,817	2.6
16	1,454	3.2
512	927	5.1

Algorithm 6: Histogram Hv3

```
Func Hist(Item *in, int n)
    for (x = in; x < in+n; x += 4) do
        prefetch (x + D);
        idx0 = *(uint8*)x;
        idx1 = *(uint8*)(x+1);
        idx2 = *(uint8*)(x+2);
        idx3 = *(uint8*)(x+3);
        hist0[idx0]++;
        hist1[idx1]++;
        hist2[idx2]++;
        hist3[idx3]++;
```

TABLE V
Hv3 Speed

run len	M/sec	c/key
offset = 0		
1	2,912	1.6
4	2,688	1.7
16	2,215	2.1
512	1,904	2.5
offset = 8		
1	2,941	1.6
4	2,941	1.6
16	2,941	1.6
512	2,941	1.6

Agenda

I. Motivation

II. Static Typhoon (S-Typhoon)

III. Typhoon

IV. Experiments

Typhoon: Memory Management

- We now deal with dynamic resizing output buckets
  - Typhoon treats the available memory as a sequence of *slices*, which are contiguous regions of RAM consisting of multiple physical pages
- After finishing an input slice, its pointer is released into the stack
- When an output bucket runs out of space
  - Slices are popped from the free stack to extend the bucket
  - A special *slice database* keeps track of slices allocated to each bucket
- WCv3 is a slice-aware WCv2
  - Surprisingly, it runs 23% slower
  - Incorrect software/hardware prefetch
- Novel non-linear prefetch in WCv4

WCv2 (static)	WCv3		WCv4	
	4 KB	8 KB	4 KB	8 KB
1,128	872	939	1,117	1,139

Typhoon: Histogram

- The histogram is almost 3× faster than the splitter
  - Impact of incorrect prefetch becomes even worse – 47% drop in speed
  - Non-linear prefetch improves the result to 90% of static speed, but this is still not ideal
- Instead of jumping over slices in the order keys were stored in each bucket
  - We identify all *contiguous* runs of data within the original buffer and call Hv3 on each of them
  - This reaches 100% of the static speed

Typhoon: Multi-Threading

- Threads mostly run independently of each other
  - Each of them maintains its own local stack of free slices, bucket pointers, and slice database
- However, after each level of LSD, slice imbalance occurs
  - Some threads have more slices than average, others less
  - This leads to starvation in later levels
- To address this problem
  - Typhoon runs a global stack of free slices, which is used after each level to rebalance the individual stacks
- Additional caveats (see the paper)
  - Special effort is needed during the last level to properly allocate border slices shared across adjacent threads

Typhoon: Slice Reshuffle

- After the last level of LSD, the sorted data is stored in slices randomly scattered in RAM
- To put them in correct order
  - Typhoon internally keeps track of the PFNs (physical frame numbers) of allocated pages and slices they belongs to
  - All slices are first unmapped using OS virtual-memory primitives
  - And then remapped back to the same space using a permuted array of PFNs
- Remapping operations are performed by all threads concurrently

Agenda

I. Motivation
II. Static Typhoon (S-Typhoon)
III. Typhoon
IV. Experiments

Experiments: Typhoon vs. S-Typhoon

Intel i7-7820X, 8-core Skylake-X CPU, 4.7 GHz, quad-channel DDR4-3200 RAM

Level	Single core								All cores							
	32-bit keys				64-bit key-value pairs				32-bit keys				64-bit key-value pairs			
	Static	4KB	8KB	16KB	Static	4KB	8KB	16KB	Static	4KB	8KB	16KB	Static	4KB	8KB	16KB
0	1,128	1,162	1,182	1,183	815	831	854	865	8,308	8,902	8,944	8,846	4,381	4,504	4,505	4,513
1	1,110	1,126	1,158	1,161	812	790	820	846	8,289	8,355	8,554	8,575	4,386	4,309	4,413	4,456
2	1,131	1,134	1,167	1,174	828	794	833	853	8,298	8,379	8,554	8,541	4,391	4,327	4,411	4,460
3	2,941	2,955	2,933	2,934	2,174	2,037	2,035	2,036	20,831	17,197	17,927	18,286	10,354	9,291	9,896	10,217
4	1,124	1,121	1,148	1,161	814	788	821	846	8,132	8,129	8,345	8,407	4,352	4,219	4,305	4,344
0-4	**256**	259	265	266	**187**	182	189	193	1,878	1,878	1,919	1,922	990	971	991	1,002
5		9,323	11,441	14,451		4,682	5,798	7,370		59,005	68,237	75,655		28,718	33,789	38,347
0-5		252	**259**	261		175	183	**188**		1,820	1,866	1,874		939	963	976

- Typhoon shows no performance loss compared to S-Typhoon using slices as small as 8-16KB

Experiments: Typhoon Scaling

Level	1 core	2 cores		3 cores		4 cores		5 cores		6 cores		7 cores		8 cores	
0	1,183	2,340	2.0×	3,533	3.0×	4,680	4.0×	5,824	4.9×	6,978	5.9×	8,041	6.8×	8,846	7.5×
1	1,161	2,312	2.0×	3,462	3.0×	4,605	4.0×	5,714	4.9×	6,813	5.9×	7,788	6.7×	8,575	7.4×
2	1,174	2,340	2.0×	3,490	3.0×	4,628	3.9×	5,752	4.9×	6,835	5.8×	7,822	6.7×	8,541	7.3×
3	2,931	5,256	1.8×	7,747	2.6×	10,163	3.5×	12,479	4.3×	14,651	5.0×	16,643	5.7×	18,286	6.2×
4	1,161	2,327	2.0×	3,456	3.0×	4,609	4.0×	5,733	4.9×	6,827	5.9×	7,750	6.7×	8,407	7.2×
0-4	266	524	2.0×	783	2.9×	1,039	3.9×	1,290	4.9×	1,536	5.8×	1,755	6.6×	1,922	7.2×
5	14,451	28,731	2.0×	39,097	2.7×	49,160	3.4×	58,511	4.0×	65,985	4.6×	67,140	4.6×	75,655	5.2×
0-5	261	515	2.0×	768	2.9×	1,018	3.9×	1,262	4.8×	1,501	5.7×	1,711	6.6×	1,874	7.2×

- 1GB of uniform 32-bit keys, 16KB slices
  - Splitter scales perfectly until it starts saturating RAM bandwidth
  - OS fails to linearly scale its remapping speed on the last level
- Next, we examine full sorts using six datasets
  - D1 = uniform, D2 = almost sorted, D3 = Zipf frequency, D4 = Gaussian, D5 = uniform floats, G = IRLbot domain graph in-degree computation and inversion

Experiments: 32-bit Keys

single-threaded

Sort	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{G}$
Gorset [15]	37	52	39	38	42	44
Polychroniou [26]	34	34	34	32	30	–
Ska [30]	40	96	73	51	84	84
Regions [23]	77	58	–	79	96	85
Voracious [25]	79	80	84	81	84	86
Vortex [16]	150	122	130	135	147	127
IPS^{2}Ra [5]	46	107	107	58	101	127
Dovetail [11]	103	99	99	103	102	99
Reinald [27]	96	100	101	101	100	111
Fast-Radix [32]	69	68	71	70	70	72
DFR [31]	76	69	131	67	79	129
pdqsort [24]	34	55	58	34	53	56
Blacher-256 [7]	133	109	136	133	133	131
IPS^{4}o [5]	36	50	58	36	50	55
Highway-512 [14]	115	128	132	115	115	140
Intel-512 [18]	149	158	80	154	153	78
Origami-512 [3]	131	131	131	131	131	131
Typhoon-16KB	257	259	261	260	259	261
	1.7×	1.6×	1.9×	1.7×	1.7×	1.9×

- Typhoon wins in all six columns, runs in-place, and posts a 60-90% improvement over the best prior methods
- It operates using mostly scalar instructions and still doubles the speed of prior AVX-512 efforts

multi-threaded

Sort	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{G}$
Regions [23]	689	667	731	700	675	761
Voracious [25]	581	906	586	597	587	688
IPS^{2}Ra [5]	526	967	991	650	777	816
Dovetail [11]	312	350	270	339	326	267
IPS^{4}o [5]	327	432	480	327	417	458
Origami-512 [3]	919	927	930	939	946	931
Typhoon-16KB	1,879	1,879	1,920	1,891	1,915	1,912
	2.0×	1.9×	1.9×	2.0×	2.0×	2.0×

Experiments: 64-bit Key-Value Pairs

- Typhoon improvement reaches 2.8x compared to best prior work
- Multi-threaded, it runs into RAM bottlenecks, but still posts a 1.3-2x speedup

single-threaded

Sort	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{G}$
Gorset [14]	21	46	21	24	21	20
Polychroniou [25]	27	20	11	25	14	–
Ska [29]	36	83	68	38	67	32
Raduls2 [19]	82	65	56	92	58	53
Regions [22]	49	45	70	56	72	29
Voracious [24]	57	54	59	53	53	56
Vortex [15]	120	102	68	117	63	57
IPS^2Ra [4]	45	96	104	45	88	38
Dovetail [10]	67	67	67	67	68	62
Reinald [26]	39	39	39	38	40	37
Fast-Radix [31]	40	40	43	40	43	38
DFR [30]	49	47	–	48	33	–
pdqsort [23]	31	48	32	31	31	30
IPS^4o [4]	31	38	41	30	39	27
Highway-512 [13]	57	57	58	57	57	54
Intel-512 [17]	76	73	75	76	76	69
Origami-512 [3]	55	55	55	55	55	53
Typhoon-16KB	184	188	193	186	202	192
	1.5×	1.8×	1.9×	1.6×	2.3×	2.8×

multi-threaded

Sort	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{G}$
Raduls2 [19]	656	478	394	737	433	491
Regions [22]	365	382	359	351	296	291
Voracious [24]	402	510	397	405	340	298
IPS^2Ra [4]	409	737	623	418	466	318
Dovetail [10]	198	206	177	197	197	130
IPS^4o [4]	286	366	351	291	341	286
Origami-512 [3]	380	389	395	394	395	392
Typhoon-16KB	986	989	998	986	1,001	997
	1.5×	1.3×	1.6×	1.3×	2.1×	2.0×

Experiments: In-Place & Cross-Platform

Model	Year	Family	RAM	GB
Intel Xeon E5-2690	2012	Sandy Bridge (SB)	DDR3-1333	256
Intel Xeon E5-2680v2	2013	Ivy Bridge (IB)	DDR3-1866	192
Intel Xeon E5-2680v4	2016	Broadwell (BW)	DDR4-2400	128
Intel i7-8700K	2017	Coffee Lake (CL)	DDR4-3200	64
Intel i5-12600K	2021	Alder Lake (AL)	DDR5-6400	32
AMD 7950X	2022	Raphael (Zen4)	DDR5-6400	32
AMD 9600X	2024	Granite Ridge (Zen5)	DDR5-6400	32

32-bit keys

Sort	SB	IB	BW	CL	AL	Zen4	Zen5
Gorset [14]	25	26	24	48	46	71	73
Polychroniou [25]	20	21	23	35	44	49	53
Ska [29]	40	43	41	84	99	116	120
Regions [22]	53	57	52	89	124	121	142
Vortex [15]	54	56	53	162	178	203	265
IPS^{2}Ra [4]	–	–	47	90	109	110	121
pdqsort [23]	23	24	24	33	40	45	47
IPS^{4}o [4]	–	–	22	33	34	46	50
Highway [13]	21	22	42	77	106	149	185
Intel [17]	–	–	63	118	167	177	240
std::sort	7	7	7	9	10	13	13
Typhoon-16KB	120	129	129	265	328	388	491
	2.2×	2.3×	2.0×	1.6×	1.8×	1.9×	1.8×

64-bit key-value pairs

Sort	SB	IB	BW	CL	AL	Zen4	Zen5
Gorset [14]	16	16	15	32	28	47	48
Polychroniou [25]	14	15	15	27	30	33	34
Ska [29]	31	32	33	66	71	80	79
Regions [22]	36	39	47	74	84	96	99
Vortex [15]	29	31	31	133	175	187	239
IPS^{2}Ra [4]	–	–	36	77	80	85	88
pdqsort [23]	17	19	20	29	41	47	47
IPS^{4}o [4]	–	–	19	31	34	44	47
Highway [13]	10	11	18	35	46	92	123
Intel [17]	–	–	22	43	60	89	135
std::sort	7	7	7	9	10	13	13
Typhoon-16KB	76	79	83	197	233	321	404
	2.1×	2.0×	1.8×	1.5×	1.3×	1.7×	1.7×

Conclusion

- Across a range of desktop/server generations, Intel/AMD CPU offerings, and SSE/AVX2/AVX-512 instruction sets, Typhoon delivers the best performance
  - 38x faster than std::sort on AMD Zen5
  - Its speed is insensitive to key distribution
  - The only method that is both stable and in-place