

Typhoon: A Slice-Scrambled In-Place LSD Sort

Zelun Liu
Texas A&M University
zelunliu@tamu.edu

Arif Arman
Texas A&M University
arman@tamu.edu

Dmitri Loguinov
Texas A&M University
dmitri@cse.tamu.edu

Abstract—This paper designs a novel in-place LSD (least significant digit first) radix sort for data-intensive applications. Our framework, which we call *Typhoon*, drops the histogram pass on each level of the sort except the last one, incorporates a high-performance architecture for dynamically expanding output buckets using low-overhead memory blocks we call *slices*, and includes a number of optimizations that reduce pipeline stalls due to cache conflicts and read-after-write bottlenecks. Because Typhoon scatters slices in each bucket randomly across RAM, it has to employ novel mechanisms for non-linear prefetch that combat tendencies of the CPU to pollute the cache on each jump. At the end of the sort, Typhoon uses OS virtual-memory primitives to unscramble the slices and put them in correct order within the input buffer. Results show that Typhoon achieves a significant single/multi-core improvement over the existing methods, including recent AVX-512 efforts from Google [18] and Intel [23], often doubling or tripling their performance.

I. INTRODUCTION

Sorting has become a ubiquitous building block behind many big-data computational frameworks and distributed systems, including various MapReduce platforms [3], [4], [14], large-scale databases [8], and external-memory graph analytics [13]. After decades of research, improving sort performance has become a difficult target, which we formalize in *five main parameters* – single-threaded speed, robustness against adversarial/non-uniform inputs, RAM usage during the sort (i.e., in-place vs out-of-place), stability (i.e., preservation of original order between duplicate keys), and performance in multi-core environments.

In particular, single-threaded speed measures an algorithm’s useful work per CPU cycle, which is a top priority in scenarios that are not bottlenecked by the total RAM bandwidth of the system. This may include HBM (High Bandwidth Memory) server architectures (e.g., 900 GB/s per socket [17]), optimizing for power consumption, and/or sorting on fewer than all available cores. Resilience against non-uniformity guarantees predictably high performance on real-world datasets, which are often skewed, while in-place operation either saves on hardware cost (i.e., requires half the RAM) or allows fewer passes in external memory compared to out-of-place methods. Stability is an important property in key-value sorts (e.g., in databases and MapReduce), where it is crucial to ensure that an existing order of values is not disturbed by subsequent sorts of the data. Finally, scaling behavior in multi-core settings reflects the algorithm’s synchronization overhead and combined memory traffic across multiple threads, which in some cases can become a separate choke point.

Unfortunately, prior work exhibits a tradeoff between these objectives, which includes sensitivity to key distribution [10],

[20], [25], out-of-place and/or unstable operation [5], [10], [18], [20], [25], [36], [37], low speed [7], [12], [31], and non-trivial complexity in achieving efficient multi-threading [10], [18], [20]. Our goal in this paper is to develop a sorting framework that not only rivals the existing methods in terms of robustness, stability, and RAM usage, but also surpasses them in single/multi-core performance. Since the LSD (least-significant digit first) radix sort is stable and insensitive to input distribution, it is an excellent starting point for our investigation.

To establish an upper bound on performance of an LSD sort, we first assume that buckets are pre-allocated by an oracle to never overflow in a method we call *Static Typhoon* (S-Typhoon). This allows the algorithm to skip the histogram pass at each level, except the last one where the keys must be returned to the input buffer. Under these conditions, we focus on optimizing the LSD partitioning loop and the histogram to reduce CPU pipeline stalls due to read-after-write dependencies and cache-associativity conflicts, create a non-temporal streaming engine that allows destination buckets to be unaligned to SIMD vector width during software write-combine offloads, and parallelize the resulting architecture.

To make S-Typhoon practical in terms of memory usage, we next augment this approach with a novel dynamically-resizing and in-place bucket architecture, calling the combined framework *Typhoon*. This method treats the available memory as a sequence of *slices*, which are contiguous regions of RAM consisting of multiple physical pages each. After finishing an input slice, its pointer is released into a free stack, which is then used to extend output buckets as they become full. Our main priority here is to keep the partitioning loop at a complexity similar to that of S-Typhoon, which we accomplish by careful optimization of the various operations on the free stack and the slice database.

While this is a good start, additional challenges are created by the non-contiguous nature of slices. Because keys within each bucket are kept at disjoint locations in virtual memory, reading through them causes the CPU hardware prefetchers to pollute the cache with irrelevant data. To overcome this setback, we design a novel non-linear prefetch logic for Typhoon that anticipates leaps through memory and loads the future data into the cache ahead of each jump. At the end of the sort, keys lie in randomly shuffled slices. To unscramble this mishmash, Typhoon unmaps all slices via OS primitives and remaps them back to the input array in correct order, avoiding the need for `memcpy` or additional space.

Result shows that Typhoon with 16-KB slices yields perfor-

mance within 2% of S-Typhoon, despite having to deal with data randomly scattered across RAM. For an 8-GB input of 32-bit uniform keys on an Intel Skylake-X i7-7820X, its single thread sorts at 256M/sec, exceeding the speed of the fastest existing LSD sort [32] by $2.7\times$, the top AVX-512 quick sort [23] by $1.7\times$, the fastest AVX-512 merge sort [5] by $2\times$, and the closest MSD radix sort [20] by $1.7\times$. Using 8 threads, Typhoon hits 1,869M/sec, which is $2\times$ faster than the nearest multi-core implementation [5] in prior work. For 32-bit keys coupled with 32-bit values, Typhoon beats the top single-threaded [20] and multi-threaded [25] competitors by $1.5\times$. Furthermore, in a real-world benchmark that requires stable key-value sorting (i.e., graph inversion), Typhoon delivers $2\text{--}2.8\times$ better performance.

II. BACKGROUND

Historically, sorting algorithms have focused mostly on speed, sacrificing certain characteristics that may be important in practice – *robustness*, *in-place operation*, and *stability*. It is thus not surprising that prior work usually achieves only a subset of these objectives. This complicates comparison due to emergence of multiple tradeoffs that do not have a clear winner unless application requirements and data properties are known ahead of time. Instead, our aim is to design a framework that simultaneously optimizes all five objectives. We next highlight the various considerations that go into making such an approach feasible.

Speed. Assume an input array of n items, each consisting of a w -bit key and an optional value. The fastest methods in related work can be split into three broad avenues – scalar radix sort [7], [12], [20], [25], [26], [27], [28], [30], [31], [32], [33], [35], [36], [37], SIMD merge sort [5], [11], [21], and SIMD quick sort [10], [18], [39]. Radix sort is generally appealing because its high fan-out factor (i.e., 2^b -way partitioning using groups of b bits) is coupled with low computational cost per key (i.e., bit shifts and masking). Whether it examines bits starting from the MSD (most-significant digit) or LSD (least-significant), radix sort requires at most $\lceil w/b \rceil$ partitioning passes over the data. In contrast, merge/quick sorts execute a lot more scans (i.e., $\log_2 n$); however, recent results [5], [18] show that vectorization, i.e., comparison of multiple keys per SIMD instruction, can boost each binary pass to $3\text{--}4\times$ the rate of those in radix sort, which helps reduce the negative impact of the low fan-out, making merge/quick sort a viable alternative.

Robustness. Distribution-insensitivity (i.e., consistent performance regardless of data skew) trivially holds for merge sort and LSD. Even though quick sort has quadratic complexity in the worst case, SIMD methods [10], [18], [39] typically implement defenses against adversarial inputs (e.g., median-of-medians sampling, detection of duplicates, fallback to $n \log n$ sorts at large recursion depth), which makes them fairly robust. In contrast, MSD is highly vulnerable to key non-uniformity and bucket imbalance, which can result in orders-of-magnitude performance degradation.

Memory usage. For the discussion below, a sort is called *in-place* if it uses $n + O(1)$ space as $n \rightarrow \infty$ and *out-of-place* otherwise [20]. Even though in-place sorts are highly desirable for large workloads, only two traditional approaches achieve this property by design – quick sort [10], [18], [39] and MSD variants [19], [28], [31], [35] derived from the classical American Flag Sort (AFS) [27], although AFS-based methods are usually not competitive in terms of speed.

There are several focused efforts to convert out-of-place methods (e.g., streaming MSD [7], [12] and sample sort [6], [7]) to in-place, but they require a hefty performance cost to manage overflow buckets, reshuffle memory blocks, and memcpy the keys to establish proper ordering. A more promising approach is Vortex [20], which is an MSD sort that catches page faults in user space to transparently extend output buckets, detect which blocks can be freed, and map them to new locations in virtual memory. One of the drawbacks to this framework is that it requires relatively large blocks (i.e., 1 MB) in order for the OS memory manager to not bog down the sort, resulting in ~ 135 MB of extra memory under uniform keys and up to 1 GB under non-uniform. Additionally, manipulating the page table in the kernel, which includes expensive locks and TLB flushes, leads to a 20% speed penalty in the MSD partitioning engine, while multi-threading of Vortex streams is currently an open problem.

For LSD, prior attempts [2] to achieve in-place operation have resulted in dismal performance, while the fastest implementations [31], [32], [33], [36], [37] all work out-of-place.

Stability. A sort is called *stable* if it preserves the original order between duplicate keys, which is a major benefit during multi-pass/multi-key sorting. One example that we use later in the paper is graph inversion, where an out-graph consisting of adjacency lists (s_i, L_i) needs to be converted into an in-graph. In this notation, s_i is a 32-bit ID of the i -th source node in non-decreasing order (i.e., $s_i \geq s_{i-1}$) and L_i is a sorted list of its out-neighbors. A stable sort can accomplish this task using 64-bit (dest, src) key-value tuples, while an unstable sort must treat each pair as a monolithic 64-bit key, which is often more expensive.

Unfortunately, stability is rarely guaranteed by existing methods. For example, a scalar merge sort satisfies this requirement; however, faster SIMD versions [5], [11], [21] rely on sorting networks [9], which are inherently unstable. Similarly, a streaming MSD radix sort can be stable in theory, but fast implementations [20], [25] invoke sorting networks once bucket size drops below a threshold. Finally, SIMD quick sorts [10], [18], [23] are unstable because pivot-based partitioning fails to preserve the original order of tuples and unstable sorting networks are necessary at the end of recursion for speed. One exception is [15], which combines MSD partitioning with a comparison sort at the end of recursion instead of using faster, but inherently unstable, sorting networks [20], [25]. As we show below, this approach comes at a significant performance loss. The other alternative is LSD [32], [33], [37], which is stable by design. Since development of both *efficient* and *stable* sorting networks is an open problem, LSD

Algorithm 1: Textbook LSD

```

1 Func LSD(Item *in, int n)
2   allocate aux array of size n
3   for (L = 0; L <  $\lceil w/b \rceil$ ; L++) do
4     if (L & 1) == 0 then
5       Split(input, n, aux, L);  ▷ even level
6     else
7       Split(aux, n, input, L);  ▷ odd level
8
9 Func Split(Item *in, int n, Item *out, int L)
10  buck = Histogram(in, out, L);  ▷ set up pointers in out array
11  for (i = 0; i < n; i++) do
12    idx = ExtractIdx(in[i], L);  ▷ bucket index
13    *buck[idx] += in[i];  ▷ write item, increment pointer

```

is currently the main option for achieving both stability and maximum performance.

Summary. Considering the various options, LSD provides the most promising platform towards our goals. In what follows, we examine how to achieve additional improvements in speed, which means omitting the counting pass, reducing RAM traffic, helping the CPU with memory disambiguation, and minimizing TLB misses, how to redesign the method to maintain an $n + O(1)$ memory footprint, and how to reduce the dependency on the OS and enable efficient execution with small block sizes.

III. STATIC TYPHOON (S-TYPHOON)

A. Baseline LSD

Assume n input keys, each consisting of w bits. Algorithm 1 shows a textbook out-of-place version of LSD (least significant digit first) radix sort. After creating an auxiliary array of size n (Line 2), the method runs $\lceil w/b \rceil$ passes (levels) that alternate between partitioning the input into the aux buffer and vice versa (Lines 3-7), where $b \geq 1$ is the number of bits examined at each level. The Split function begins with a histogram (Line 10) that sets up 2^b destination bucket pointers buck, where $\text{buck}[0] = \text{out}$ and $\text{buck}[i+1] - \text{buck}[i]$ is the number of keys that will be written into the i -th bucket. Following this, Line 12 isolates the b bits that represent the bucket index of each key $\text{in}[i]$ and Line 13 writes the item into the corresponding memory location, updating the destination pointer $\text{buck}[\text{idx}]$ in the process.

On each level, Algorithm 1 reads n items from RAM during the histogram pass and another n during splitting. On top of that, it writes n items to the output buckets, which causes the CPU to additionally read for ownership all destination cache lines in the output buffer. Thus, Algorithm 1 ends up with a total of $4n$ keys of memory traffic per level. One can do significantly better by utilizing software *write-combine* (WC) [8], [20], [25], [31], [33], [34], [40], which initially stores data into small tmp buckets contained in the L1/L2 cache and then offloads them to RAM using non-temporal (streaming) stores that bypass the cache. For efficiency reasons, tmp bucket size B is usually assumed to be a multiple of cache-line size. Because streaming avoids read-for-ownership, this reduces RAM traffic to $3n$ per level, i.e., 25% lower than the naive

Algorithm 2: WCv1

```

1 Func Split(Item *in, int n, Item *out, int L)
2   buck = Histogram(in, out, L);
3   for (i=0; i < n; i++) do
4     prefetch (in + i + D);
5     idx = ExtractIdx(in[i], L);
6     p = tmpBuckets + idx*B;
7     p[tmpSize[idx]] = in[i];
8     if ++tmpSize[idx] == B then
9       OffloadAVX(buck[idx], p);
10      buck[idx] += B;
11      tmpSize[idx] = 0;
12
13 Func OffloadAVX(__m256i *dest, __m256i *src)
14   for (i=0; i < R / sizeof(__m256i); i++) do
15     x = _mm256_load_si256(src + i);
16     _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

approach. Assuming 2^b is larger than the TLB size, which is commonly the case, this also decreases the number of TLB misses from one per key in Algorithm 1 to one per B keys.

Drawing inspiration from [20], which is currently the fastest implementation of WC, assume tmpBuckets is an array of $(2^b \cdot B)$ items and $\text{tmpSize}[i]$ stores the current number of keys in bucket i . Algorithm 2, which we call WCv1, shows the baseline partitioning function of an optimized LSD. Its Line 4 runs a prefetch at some distance D in the input buffer, Line 8 detects tmp bucket overflow, and Line 9 uses a non-temporal memcopy from the start of the tmp bucket (i.e., pointer p) to the corresponding location in RAM (i.e., $\text{buck}[\text{idx}]$). Note that $R = B \cdot \text{ItemSize}$ in Line 14 is the length of each bucket in bytes, which is assumed to be a multiple of 256-bit AVX register data type _m256i .

To make discussion more focused, it should be noted that $b = 8$ is currently the optimal value for LSD, in both Algorithm 1 and 2. Larger values of b (such as 10 or 11) may be used to reduce the number of passes; however, each pass becomes significantly slower due to the increased number of TLB misses and page-table walks, making performance of the whole sort noticeably worse. Therefore, most of the examples below assume $b = 8$ and 256-way partitioning. In this setup, sorting 32-bit keys (i.e., four levels) requires $16n$ memory traffic in Algorithm 1 and $12n$ in Algorithm 2.

B. Overview of S-Typhoon

While the basic partitioning engine in Algorithm 2 is a good starting point, we are interested in the question of achieving the absolute maximum performance, both in terms of CPU cycles per key and RAM traffic, in order to establish a definitive upper bound on LSD speed. At a minimum, each pass of the sort has to read through the input keys, decide on their buckets, and send them to output, i.e., the work done in Lines 3-11 must remain. However, the histogram in Line 2 can be omitted unless keys in adjacent output buckets must appear contiguously in RAM, which is a requirement only for the final level. Avoiding the histogram not only reduces the amount of CPU cycles by $\sim 30\%$, but also leads to lower RAM pressure in multi-threaded scenarios. If this idea can be implemented

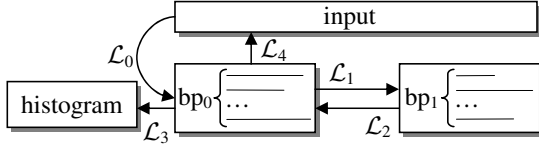


Fig. 1. S-Typhoon workflow overview (32-bit keys).

efficiently in practice, 32-bit keys would require three levels of $2n$ memory traffic (i.e., non-counting) and one level of $3n$ (i.e., counting), for a total of $9n$ per full sort, dropping the memory load of Algorithm 2 by another 25%.

Eliminating the histogram requires expanding buckets dynamically without knowing their final size. We delay this issue until the next section, but in the meantime assume that an oracle pre-allocates two sets of static buckets, $bp[0]$ and $bp[1]$, that never overflow. In this notation, $bp[L+1][i]$ points to bucket i during level L . Then, Static Typhoon (S-Typhoon) proceeds in five steps shown in Fig. 1, where partitioning levels $\mathcal{L}_0 - \mathcal{L}_2$ are non-counting, level $\mathcal{L}_3$ is the histogram on the most-significant digit, and the final distribution pass $\mathcal{L}_4$ returns all items back into the input buffer. Note that buckets during $\mathcal{L}_0 - \mathcal{L}_2$ are kept in disjoint memory locations, while those in $\mathcal{L}_4$ are tightly packed (i.e., appear with no gaps).

C. Read-After-Write Bottlenecks

Our first topic is to analyze WCv1 under skewed key distributions. While the method works well for uniform keys, our results show that it inexplicably reduces speed when multiple adjacent keys are sent into the same bucket, which is a common occurrence for non-uniform inputs. Define WCv1a to be Algorithm 2 without the histogram pass in Line 2. Then, Table I demonstrates this issue with 1 GB of 32-bit keys and an Intel Skylake-X i7-7820X clocked at a fixed 4.7 GHz, where the run length in the first column specifies how many back-to-back copies of each random key are generated. The speed begins at 1,121M keys/sec (4.2 cycles/key) in the first row (all unique), drops to 826M/sec in third row (5.7c/key), and finally settles on 883M/sec (5.3c/key) for sufficiently long runs. The worst case is $1.4\times$ slower than the best, which is not a negligible drop. We next examine this problem in more detail and develop a solution that allows S-Typhoon to remain robust against such cases.

A key component of data partitioning, whether it uses write-combine or not, is to perform updates to shared counters as items arrive from input. These can be bucket pointers in Algorithm 1 (Line 13) or tmp bucket sizes in Algorithm 2 (Line 8). To better understand the challenges the CPU faces in these cases, first consider a simplified problem, also of relevance to S-Typhoon, whose purpose is to compute a histogram using the first byte of each item in an array. A baseline solution [25], [31], which we call Hv1, is illustrated by Algorithm 3. Its performance using the same setup as before is shown in Table II. In this case, the reduction in speed is even sharper, i.e., $2.4\times$ between all-unique and all-duplicate.

Algorithm 3: Histogram Hv1

```

1 Func Hist(Item *in, int n)
2   for (i=0; i < n; i++) do
3     prefetch (in + i + D);
4     idx = *(uint8*)(in + i);
5     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

To delve deeper, consider the CPU pipeline for Hv1, which repeats a pattern of two loads, an increment, and a store:

```

idx0 = *(uint8*)in; c0 = hist[idx0]; inc c0; hist[idx0] = c0;
idx1 = *(uint8*)(in+1); c1 = hist[idx1]; inc c1; hist[idx1] = c1;

```

where $idx0$ - $idx1$, $c0$ - $c1$ are registers. For performance reasons, the CPU's out-of-order execution engine attempts to hoist loads ahead of preceding stores unless its memory-disambiguation module detects a conflict. As the pipeline decodes the uops, it sees a store to $hist[idx0]$, followed by a load from $hist[idx1]$, both from yet-unknown buckets $idx0$, $idx1$. Without additional hints, the CPU cannot decide whether this presents a conflict and optimistically assumes that these uops are independent, which causes it to reorder the load from $hist[idx1]$ to precede the store into $hist[idx0]$.

When adjacent keys refer to different buckets, load hoisting allows higher levels of instruction-level parallelism as counters from multiple locations can be fetched and incremented concurrently. However, doing the same for pairs of keys with $idx0 = idx1$ leads to consistency violations, which are detected by the CPU just before these instruction retire, causing expensive pipeline flushes. It is speculated [16] that Intel maintains a history of mispredictions for each load and temporarily disables hoisting after a threshold of violations is reached.

While hoisting is inoperable and duplicate keys are still arriving, the histogram runs into dependency chains between each load from the L1 cache and the preceding store. Going into a latency-bound regime (i.e., 4c/load for Skylake-X) is already a major bottleneck, but an additional problem arises from having to search through the store buffer and forward loads out of it [22], which sometimes has an even higher latency [41]. Breaking loop-carried dependencies and reducing frequency of pipeline stalls is our next topic.

D. Reducing Store-Forwarding Costs

This analysis gives rise to the following idea. Dependency between k adjacent store-load pairs can be resolved by reading k histogram counters upfront, performing comparison across all $k(k-1)/2$ pairs of bucket indexes, and incrementing the relevant counters using the result of the comparison. For $k = 2$, this is demonstrated by Algorithm 4, which simultaneously reads two indexes from input and obtains their counters at the start of each iteration (Lines 4-6), keeping them in registers. It then updates counter $c0$ (Line 7) and decides the value of $c1$ based on whether the two buckets are the same (Line 8). To keep the algorithm branchless, ternary operator $?$ is implemented using a *conditional move* CPU instruction `cmov`, which needs only 0.5c/iteration (i.e., 0.25c/key). If the

Algorithm 4: Histogram Hv2

1 Func Hist(Item *in, int n)		TABLE III		
2 for (x = in; x < in + n; x += 2) do		Hv2 SPEED		
3	prefetch (x + D);	run len	M/sec	c/key
4	idx0 = *(uint8*)x;	1	2,496	1.9
5	idx1 = *(uint8*)(x + 1);	4	2,478	1.9
6	c0 = hist[idx0]; c1 = hist[idx1];	16	2,275	2.1
7	c0++; hist[idx0] = c0;	512	1,402	3.4
8	c1 = (idx0 == idx1) ? c0 : c1;			
9	c1++; hist[idx1] = c1;			

compiler does not issue `cmov`, assembly can be used instead.

Table III displays the performance of Hv2. For unique keys in the first row, the speed goes up 11% compared to Hv1. As the run length increases, performance gains become more substantial, where Hv2 finishes with a $1.5\times$ advantage in the last row compared to Table II. The gradual reduction in speed as the burst length increases in Table III can be explained by Hv2's success at eliminating store-load dependencies *within* each pair of items, but not *between* pairs. Unfortunately, for $k \geq 3$, the quadratic cost of doing all-to-all scalar comparisons becomes prohibitively high, i.e., no further improvement is currently possible under the same umbrella.

It should be noted, however, that with vector intrinsics of AVX-512, the majority of this work can be reduced to four instructions, i.e., `vpgather`, `vpconflict`, `vpopcnt`, and `vpscatter`, that respectively collect counters from k disjoint locations, perform all $k(k-1)/2$ comparisons to create a mask, count the number of set bits in the mask, and distribute the result back to memory. With a few extra steps, the algorithm can offload $k = 16$ keys per iteration of the loop. The main problem is that conflict/scatter instructions are still expensive, amounting to roughly 2.7c per key [1], which is not competitive against the majority of rows in Table III. Additionally, the AVX-512 histogram fails to deliver useful lessons towards improving WCv1, linearly drops speed as keys become larger, and has difficulties when item and counter sizes mismatch.

We therefore leverage conflict detection in Algorithm 4 by applying the same principles to key distribution. This is shown in Algorithm 5 under the name of WCv2. Unlike the previous version WCv1a, bucket addresses in RAM are passed into the function in the third argument (i.e., array of pointers `buck`), which refers to either `bp[0]` or `bp[1]` depending on the level. Additionally, the fourth argument specifies an array of pointers `t` to the starting position in each tmp bucket. The algorithm further assumes that the number of bytes in each tmp bucket $R = B \cdot \text{ItemSize}$ is a power of two and the start of `tmpBuckets` is aligned to R bytes.

For each pair of items, macro `MOVE` reads their indexes from the corresponding byte of the key (Lines 7-8) and obtains both pointers `p0`, `p1` (Line 9) without yet knowing if there exists a conflict. It then calls a macro `WRITE`, which stores the first key into `p0` and checks for the end of the tmp bucket (Line 16). If so, it jumps `p0` to the start of that bucket and offloads its B items to RAM. At the end, it updates `t[idx0]` in Line 20 to reflect the new position. At this point, `p1` can

Algorithm 5: WCv2

1 Func Split(Item *in, int n, Item **buck, Item **t, int L)		TABLE IV		
2 for (x = in; x < in + n; x += 2) do		WCv2 SPEED		
3	prefetch (x + D);	run len	M/sec	c/key
4	MOVE(x);	1	1,128	4.2
5 Macro MOVE(x)		4	1,128	4.2
6	idx0 = *((uint8*)x+L);	16	1,118	4.2
7	idx1 = *((uint8*)(x+1)+L);	512	1,302	3.6
8	p0 = t[idx0]; p1 = t[idx1];			
9	WRITE(x[0], p0, idx0);			
10	p1 = (idx0 == idx1) ? p0 : p1;			
11	WRITE(x[1], p1, idx1);			
12 Macro WRITE(key, p, idx)				
13	*p++ = key; ▷ store item			
14	if (p & (R-1) == 0) then ▷ overflow?			
15	p -= B; ▷ roll back to start of bucket			
16	OffloadAVX(buck[idx], p);			
17	buck[idx] += B;			
18	t[idx] = p;			

be computed using a conditional move in Line 11 and the process repeats for the second key. Note that WCv2 avoids store forwarding within each pair of keys because `t[idx1]` is read (Line 9) before `t[idx0]` is written (Line 20).

Table IV shows the resulting speed. Compared to WCv1a in Table I, this version gains a few M/sec in the first row, improves by 20% in the second, 35% in the third, and 47% in the last one. While Table IV shows no major slowdown when dealing with duplicates, the same cannot be said about the Hv2 histogram in Table III. The reason for this discrepancy is that WCv2 even at its best speed (i.e., 3.6c/key) it is still slower than Hv2 at its worst (i.e., 3.4c/key). This suggests that the CPU executes movement of keys largely in parallel with intra-pair conflict resolution and that the former is the bottleneck, which makes WCv2 mostly insensitive to presence of duplicates.

E. Cache Conflicts

Another non-uniform key pattern that breaks performance of WCv1 relates to repetitive (round-robin) writing into tmp buckets. On Intel CPUs, the optimal size of each tmp bucket is $R = 256$ bytes, which serves as a middle ground between offloads interrupting the CPU pipeline with mispredicted branches too frequently and the tmp buckets becoming too big to fit in the cache. Across all $2^b = 256$ buckets, the tmp buffer occupies 64 KB, i.e., double the L1 cache size, and thus requires evicting keys into L2. Under uniform loads, this presents no problem as evicted cache lines carry enough useful data to keep the L1→L2 pipeline faster than the splitter. However, non-uniform keys can break this pattern and result in significant performance loss. We look into this issue next.

Recall that L1 in modern CPUs is a k -way set-associative cache, where the lowest c bits of a virtual address specify the offset within the cache line and the next r bits are used as an index that determines the corresponding cache set. When data items collide on a cache set, the CPU can accommodate up to k cache lines without creating a conflict, after which it must

TABLE V
ROUND-ROBIN SPLITTER SPEED (M/SEC)

skip s	C_S	C_L	WCv1a	WCv2	sWCv1a	sWCv2
1	16	16	536	594	1,104	1,237
2	8	16	470	534	1,128	1,241
4	4	16	268	361	1,119	1,256
8	2	16	206	218	1,099	1,245
16	1	16	160	199	1,122	1,242
32	1	8	1,137	1,234	1,230	1,328

evict one of the cache lines from the set. Thus, there are 2^r cache sets, each containing k cache lines of size 2^c bytes, i.e., $k2^{r+c}$ is the total size of L1. While the exact operation of CPU caches is proprietary, it is not difficult to construct examples that trigger the worst-case, i.e., collision of all buckets on just one cache set. In this situation, the application evicts a full cache line for every item written into tmp buckets.

To examine how this works in practice, we construct a 32-bit key sequence that causes level 0 of LSD to round-robin through every s -th bucket, i.e., the i -th key is $x_i = s \cdot i \pmod{2^b}$, where $2^b/s$ is an integer. Note that $s = 1$ has the same effect on the cache as a fast random-number generator called the LCG (linear congruential generator), which makes this case of special interest. Performance is broadly determined by two factors – the number of unique cache sets C_S being hit and the number of cache lines C_L the splitter attempts to simultaneously put into each set. As long as $C_L > k$, there will be a possibility for conflict eviction. Since sR is the number of bytes between adjacent stores into the tmp buckets, we get that $C_S = \max(2^{r+c}/(sR), 1)$, from where the cache-set occupancy is the number of non-empty buckets divided across the available cache sets, i.e., $C_L = 2^b/(sC_S)$.

For Intel Skylake-X with its 32-KB L1 cache, $k = 8$, and $c = 6$, we get $r = 6$. Table V examines performance of WCv1a/WCv2 for different values of s . In the first row, the splitter uses $C_S = 16$ cache sets out of the 64 available and attempts to stuff 16 cache lines into each, resulting in a drastic performance reduction compared to the uniform case in Tables I and IV. While WCv2 is still faster than WCv1a, it is definitely not immune to L1-L2 cache thrashing. As s increases and C_S reduces, the speed undergoes additional deterioration, eventually bottoming out below 200M/sec with $s = 16$. As predicted, this row elicits the worst behavior from the cache since $C_S = 1$ and $C_L > k$ are simultaneously true. Even though the next row also has $C_S = 1$, it places only 8 cache lines into each set, leading to no unnecessary evictions (due to $C_L \leq k$) and therefore much faster speed.

Since robustness is one of the goals for S-Typhoon, behavior of WCv1a/WCv2 in Table V is unacceptable. In order to remain distribution-insensitive, S-Typhoon applies a randomized stagger to the address of the first item in each tmp bucket (i.e., `tmpSize[i]` in WCv1a and `t[i]` in WCv2) to ensure that the initial bucket pointers sweeps all 2^r cache sets. Assuming each tmp bucket is a multiple of cache line, i.e., $R/2^c$ is an integer, Algorithm 6 computes the byte-increment `stag[i]` that needs to be added to the start of tmp bucket i before each

Algorithm 6: Optimal cache stagger

```

1 Func ComputeStagger(int *stag)
2   groupSize = max( $2^{r+c} / R$ , 1);  $\triangleright$  buckets with equal
   stagger
3   for (i=0; i < nBuckets; i++) do
4     | stag[i] = ((i / groupSize) * cacheLineBytes) % R;
5   RandomShuffle(stag, nBuckets)

```

Algorithm 7: WCv2L4, dealing with level $\mathcal{L}_4$

```

1 Func AdjustBucket(int idx)
2   x = buck[idx] % cacheLineBytes;
3   stag[idx] += x; stag[idx] /= ItemSize;
4   t[idx] = tmpBuckets + idx*B + stag[idx];
5
6 Func OffloadL4(Item *dest, Item *src, int idx)
7   if (stag[idx] > 0) then  $\triangleright$  first offload of this bucket?
8     | OffloadScalar (dest, src + stag[idx]);
9     | stag[idx] = 0;  $\triangleright$  remove stagger from later offloads
10  else
11    | OffloadAVX(dest, src);  $\triangleright$  switch to regular AVX
    | offload

```

level of LSD begins. It first decides in Line 2 on the number of adjacent buckets that are guaranteed to hit different cache sets. It then applies in Line 4 the same stagger within each group, which ensures that none of them collide on the same cache set, and increases the stagger by 1 cache line between the groups. Finally, Line 5 makes the stagger unpredictable to prevent adversarial inputs that undo the effects of Algorithm 6.

We prefix splitters running over optimally staggered buckets with the letter s , i.e., sWCv1a and sWCv2. Their performance is shown in the last two columns of Table V. Results demonstrate that sWCv1a recovers back to the uniform speed of WCv1a, while sWCv2 runs ~ 100 M/sec faster on round-robin inputs than previously in the first row of Table IV. It also beats sWCv1a in Table V by roughly the same amount.

F. Offload Alignment

We now deal with architectural nuances of streaming stores in OffloadAVX, which require alignment of the destination pointer to SIMD vector length. For convenience, we align the input buffer and all buckets to cache-line size (i.e., 64 bytes). Thus, for levels $\mathcal{L}_0 - \mathcal{L}_2$ of the S-Typhoon workflow in Fig. 1, dealing with stagger is simple – function OffloadAVX streams full tmp buckets, including the `stag[i]` dummy bytes in the front. The leading garbage is ignored on the next level by pushing the starting pointer of the bucket forward by `stag[i]`. On the other hand, the situation gets more complicated during the last level $\mathcal{L}_4$, where a) destination buckets are no longer aligned to SIMD width; and b) the dummy stagger bytes may destroy the contents of the preceding bucket.

While prior work [25] can tackle a) by rolling back the destination pointer to the nearest multiple of SIMD width (e.g., 32 bytes for AVX), this solution does not work with constraint b). Instead, Algorithm 7 summarizes the changes in sWCv2 needed to address both problems. Function AdjustBucket, called once for each bucket before $\mathcal{L}_4$ begins, computes the

offset x within the cache line of the destination pointer and uses it to adjust stagger of the corresponding tmp bucket. This ensures that proper SIMD alignment will be reached after the first offload on this bucket. Next, Line 18 of Algorithm 5 gets replaced with a call to `OffloadL4`, shown at the bottom of Algorithm 7, which either offloads using a scalar loop when `stag[idx]` is non-zero or falls back to AVX otherwise. Note that the scalar branch in Line 8 is taken at most once per bucket, i.e., mispredicted ≤ 256 times out of millions of offloads.

G. Histograms Revisited

We now deal with the issue of designing the histogram for level $\mathcal{L}_3$, keeping in mind that we no longer need this solution to be applicable to a key splitter. Our next approach, illustrated as Hv3 in Algorithm 8, unrolls the loop to grab multiple keys at once and writes updated counters into *separate* histograms. While this example shows unrolling to $r = 4$ keys and $h = 4$ histograms, other combinations are possible as well. Furthermore, when r exhausts general-purpose registers, our implementation of Hv3 uses SIMD (SSE/AVX) to hold the keys. A crucial element of this technique is the use of an *offset*, which specifies the distance (in bytes) between the start of each histogram and the end of the previous one. Since each histogram is exactly $256 \times 8 = 2$ KB, offsets are needed to avoid 4K aliasing and conflicts in set-associative caches [22].

Using $r = 16$ keys and $h = 8$ histograms, the upper half of Table VI shows that a zero offset can produce a 16-35% improvement over Hv2, but the resulting method still chokes on duplicate keys, losing over a billion keys/sec between the first and last rows. On the other hand, offsetting the histograms by 8 bytes yields a drastically different result, i.e., a constant 1.6c/key, as also shown in the table.

H. Multi-Threading

Assume a joint sort across T threads, each holding its own dual set of RAM buckets `bp[0]`, `bp[1]` and local tmp buckets in the corresponding L1/L2 cache, possibly with a different stagger. Suppose matrix M consists of all sub-buckets written by the threads after a particular level of splitting, i.e., M_{ij} represents the contents of bucket i created by thread j . To identify a sub-bucket, it is sufficient to specify its 2D index (ij) . Furthermore, let triple (ijr) refer to the r -th key in bucket M_{ij} . Then, a *row-major order on keys* is defined as $(xyr) \prec (uvt)$ iff $(x < u) \vee (x = u, y < v) \vee (x = u, y = v, r < t)$. Note that for level $\mathcal{L}_0$, we assume $M = (M_{00})$ is a 1×1 matrix consisting of the input array.

Multi-threading requires assigning each thread $p = 1, 2, \dots, T$ a set of keys Δ_p consisting of n/T triples (ijr) such that $\{\Delta_1, \dots, \Delta_T\}$ forms a partition on M . Two rules must be satisfied in order to ensure correctness: a) each thread processes keys assigned to it in row-major order; and b) if $(xyr) \in \Delta_p$ and $(uvt) \in \Delta_q$, where $p < q$, then $(xyr) \prec (uvt)$ must hold. This guarantees stability, i.e., that the next level of LSD does not break the relative order established within each bucket on the previous level. To achieve b), our load-balancing algorithm views all n keys

Algorithm 8: Histogram Hv3

```

1 Func Hist(Item *in, int n)
2   for ( $x = \text{in}; x < \text{in} + n; x += 4$ ) do
3     prefetch ( $x + D$ );
4     idx0 = *(uint8*)x;
5     idx1 = *(uint8*)(x+1);
6     idx2 = *(uint8*)(x+2);
7     idx3 = *(uint8*)(x+3);
8     hist0[idx0]++;
9     hist1[idx1]++;
10    hist2[idx2]++;
11    hist3[idx3]++;

```

TABLE VI
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

as a one-dimensional array in the row-major order of M and assigns its p -th consecutive batch of size n/T to thread p .

The cost of computing these boundaries depends on the number of buckets 2^b and thread count T , but this is usually negligible (i.e., under 10 μ s) compared to the sort time. Since threads require data in sub-buckets M_{ij} created by other threads, each level $\mathcal{L}_0 - \mathcal{L}_4$ ends with a barrier that synchronizes the threads. In summary, S-Typhoon runs three passes $\mathcal{L}_0 - \mathcal{L}_2$ using `SWCv2`, which combines Algorithm 5 with stagger in Algorithm 6, a single counting pass at $\mathcal{L}_3$ using Hv3 in Algorithm 8, and another pass at $\mathcal{L}_4$ using a modified `SWCv2` from Algorithm 7, for a total of $9n$ memory traffic.

IV. TYPHOON

To make the framework developed in the previous section practical, the first challenge is to design a low-overhead technique for dynamically expanding output buckets as they are written to. While this problem has been touched upon in prior work [6],[7], [20], [31], [36], performance of these solutions leaves much to be desired, as we discussed in Section II. Our objective here is to create a new bucket-management infrastructure that runs $\mathcal{L}_0 - \mathcal{L}_4$ at almost the same speed as S-Typhoon, but without using an oracle to statically pre-allocate the buckets. After $\mathcal{L}_4$, the data will end up in a number of disjoint locations in RAM, where the second challenge is to restore proper order between the keys using a novel unscrambling level we call $\mathcal{L}_5$.

Define a *slice* to be contiguous region of S bytes in virtual memory starting from an address that is aligned to S . Slices come from two places – the input buffer and some auxiliary space that is needed to provide support to partially filled slices during the split. To speed up detection of end-of-slice, we assume S is a power of 2, and to prevent offloads from crossing slice boundaries, let S be a multiple of tmp bucket size R . For reshuffling at $\mathcal{L}_5$, S must also be a multiple of page size.

A. Data Structures

We start by considering single-threaded execution. Suppose the sort maintains a stack of free slices, which is an array of 64-bit pointers to the start of each slice. Compared to other data structures, stacks have an advantage in their low push/pop cost, i.e., one `stackTail` pointer, a load/store instruction, and a register increment/decrement. Additionally, stacks achieve high temporal locality because of immediate reuse of slices

Algorithm 9: WCv4 (simplified)

```

1 Func Split(Item **, int ns, Item **bp, Item **, int L)
2   for (j=0; j < ns; j++) do  ▷ iterate over all slices
3     cur = s[j]; next = s[j+1]; dist = next - cur;
4     do
5       prefetch (cur + dist);  ▷ prefetch next slice
6       MOVE(cur);
7       cur += 2;
8     while cur & (sliceSize - 1);

```

between input and output. As we see below, this allows Typhoon to run certain levels of the sort faster than S-Typhoon. At the start of the sort, the free stack is initialized to A auxiliary slices, where A determines the $O(1)$ constant in $n + O(1)$ memory usage. Each level of the sort requires at least 2^b slices in the free stack, plus the space wasted due to stagger, i.e., $R/2$ bytes per tmp bucket. Thus, $A \geq 2^b(1 + R/(2S))$ must hold. We refine this bound later in the section.

To keep track of the slices assigned to each bucket, suppose *slice database* $\mathbf{sd}$ contains in $\mathbf{sd.p}[i][j]$ the address of the j -th slice in bucket i . For each new slice popped from the free stack, the splitter records tuple (bucket idx, slice pointer) into a separate pre-allocated buffer. After the level is over, this array is processed to count the number of slices that went into each bucket i , which allows easy construction of $\mathbf{sd}$. Since Typhoon alternates between two sets of buckets, each set requires a separate slice database, which we call $\mathbf{sd}[0]$ and $\mathbf{sd}[1]$. If the sort reads slices from $\mathbf{sd}[k]$, where k is either 0 or 1, it keeps track of the new ones in $\mathbf{sd}[1-k]$.

B. Aligned Splitter ($\mathcal{L}_0 - \mathcal{L}_2$)

We extend the S-Typhoon WCv2 splitter, which we now call WCv3, to accept an array of slice pointers $\mathbf{s}[]$ rather than one large buffer. These pointers typically come from a single bucket, where the first slice may be offset by stagger, all intermediate slices are full, and the last one may be partially filled. The main loop in Algorithm 5 remains essentially the same, except it gets interrupted every S bytes to load the next input slice pointer. This involves four CPU instructions per slice and a mispredicted branch.

On output, the OffloadAVX function has to check if the destination pointer $\mathbf{buck}[\mathbf{idx}]$ is aligned to slice boundary using bitwise masking. If so, a free slice is popped from the stack and added to the array $\mathbf{sd}[k].\mathbf{p}[\mathbf{idx}]$. To ensure proper operation at the start, all buckets begin such that $\mathbf{buck}[\mathbf{idx}] = \text{NULL}$, which causes a trip to the stack on first access to each destination pointer. In total, output slice management requires one mask instruction and a well-predicted branch *per offload*, which adds at most $1/4$ cycle per R bytes, as well as 6 additional mov/add/sub instructions *per slice*, which add ~ 1.5 cycles per S bytes. Because of the simplicity of its data structures, WCv3 can fit all pointers and variables into 13 general-purpose registers, leaving three unused. This ensures no register pressure, spills to the stack, or reloads.

Considering the low cost of managing the slice database and the stack, it is perhaps unexpected that WCv3 runs a

TABLE VII
LEVEL $\mathcal{L}_1$ SPLITTER SPEED (M/SEC)

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

lot slower than WCv2 during $\mathcal{L}_1$. With 4-KB slices and 32-bit keys, Intel i7-7820X shows a reduction in speed from 1,128M/sec to 872M/sec, a loss of 23%! Further investigation reveals that this issue is caused by two compounding effects – software prefetch at distance D in Line 3 of Algorithm 5, which pollutes the cache with irrelevant data ahead of each jump, and CPU hardware prefetchers that detect scans and also load some amount of garbage following each slice.

There is not much we can do about the latter issue, but the former can be alleviated by introducing a non-linear prefetch into WCv3, which keeps both the current slice pointer $\mathbf{cur} = \mathbf{s}[j]$ and the next one $\mathbf{s}[j+1]$ in registers, prefetching at address $(x + D)$ when $x + D < \mathbf{cur} + S$ and $(\mathbf{next} + x - \mathbf{cur} + D - S)$ otherwise. In cases when the CPU allows a range of prefetch options sufficiently-far in the future to work at optimal speed, which is the case for Intel and AMD, the loop can be further simplified to always prefetch from the next slice. For these situations, Algorithm 9 shows a high-level operation of the new approach WCv4. Note that separate logic is needed to handle stagger on the first slice and the partially filled slice at the end of $\mathbf{s}$, which we omit for brevity.

Generally, it is expected that larger slices are faster because the CPU prefetches less garbage compared to the amount of useful data in the slice. Additionally, the cost of managing the free stack and database $\mathbf{sd}$ becomes smaller as well. Results in Table VII confirm this observation using 4 and 8 KB slices. Even with 8-KB slices, WCv3 struggles to match the static speed. On the other hand, WCv4 essentially ties S-Typhoon with 4KB slices and exceeds its performance using 8KB slices. As mentioned earlier, stack-based reuse of slices sometimes gives Typhoon an advantage over S-Typhoon. Additional cases will be illustrated later in the paper.

C. Histogram ($\mathcal{L}_3$)

As level $\mathcal{L}_3$ runs almost $3\times$ faster than the splitter, the effect of incorrect prefetch becomes even worse. In particular, the speed of Hv3 drops by 47%, i.e., from 2,941M/sec in Table VI to 1,560M/sec (4-KB slices). Applying the trick from Algorithm 9 improves the result to 2,744M/sec, but this is still 200M/sec slower than static.

In contrast to the splitter, which must process the items in row-major order within matrix M , the histogram is not constrained in how it visits the keys. Hence, it is sufficient to identify all *contiguous* runs of keys in virtual memory and call Hv3 on each of them. To determine these regions, one option is to sort all slice pointers in the $\mathbf{sd}$ database and merge the ones next to each other. However, sorting 256K slices per GB of data is too expensive, which makes the result slower than random jumping using the prefetch of WCv4.

On the other hand, the same outcome can be obtained by extracting the chunks of *unused* space from the free stack and *partial* slices in *sd*, sorting them, and computing the contiguous runs of valid keys by complementing the empty space. Note that a bucket can have up to two partial slices (i.e., the first one due to stagger and the last one), which combined with the empty stack amounts to at most A separate regions of free memory. Thus, the sort involves a few hundred integers, regardless of n . We call this method *Hv4* and note that it easily hits the speed of S-Typhoon even with 4-KB slices.

D. Unaligned Splitter ($\mathcal{L}_4$)

For $\mathcal{L}_4$, a new challenge arises due to stagger and the possibility that function *OffloadL4* may cross slice boundaries. The general structure for this level follows *WCv4* with modifications in Algorithm 7, except both *OffloadAVX* and *OffloadScalar* need to detect when the destination pointer moves to the next slice. Because the latter runs only once per bucket, this check can be done for every key; however, the former needs further optimizations.

In particular, *OffloadAVX* examines if the current slice can accommodate another R bytes; if so, it runs the standard (uninterrupted) offload loop. Otherwise, it moves enough keys to finish the current slice, obtains a new one from the stack, and completes the offload there, in both cases using AVX. The frequency of taking the slice-crossing branch is determined by the ratio of tmp bucket size to slice length, i.e., R/S . With $S = 4$ KB, the probability to take the slower branch on Intel CPUs is 6.25% (i.e., $R = 256$). Combined with the extra cost of branch misprediction, this explains why Typhoon's $\mathcal{L}_4$ runs slightly slower than its $\mathcal{L}_1 - \mathcal{L}_2$.

E. Multi-Threading

We now deal with slice management across T threads. Assume each of them maintains a local state consisting of a free stack, a tmp bucket buffer, two sets of bucket pointers *bp*[0], *bp*[1], and two slice databases *sd*[0], *sd*[1]. The goal of this setup is to make threads run with as little interaction with each other as possible. To handle partitioning of matrix M , we extend the row-major order introduced earlier to organize keys by (bucket, thread, slice). For speed reasons, threads never share slices from each sub-bucket, but the rest of the load-balancing algorithm (Section III-H) remains the same.

For $\mathcal{L}_0 - \mathcal{L}_2$, if a thread begins a level with $A_0 = 2^b(1 + R/(2S))$ free slices in its local stack, it has sufficient extra memory to cover the wasted bytes due to stagger and leave one almost-empty slice at the end of each bucket. Thus, *WCv4* can run independently and without modification within each thread using its local stack. However, because a thread reads other thread's sub-buckets, its stack size at the end of a level can be anywhere from zero to the total number of partial slices in the sub-buckets it visited. Thus, each level introduces a size-imbalance into thread stacks, which, if left uncorrected, eventually leads to a crash.

To address this problem, Typhoon introduces a global stack that contains the remaining slices not currently assigned to

any thread. After finishing a level, but before the barrier, each thread returns excess slices (i.e., those above A_0) to the global stack. Similarly, when a level begins (i.e., after the barrier), each thread acquires the missing slices to bring its local stack size back to A_0 . Under these rules, a thread can either return or acquire slices at a particular level, but not both.

This works well for $\mathcal{L}_0 - \mathcal{L}_2$, but additional difficulties arise during $\mathcal{L}_4$. Because keys must be contiguous in space at the end of the sort, there can be up to $T \cdot 2^b$ slices that are shared across bucket boundaries, including between different threads. The main challenge here is to prevent allocation of redundant slices when *WCv4* moves into the last (partial) slice of each bucket. To address this, the last thread of Typhoon that reaches the barrier at the end of $\mathcal{L}_3$ pre-allocates all shared slices and then kicks off $\mathcal{L}_4$. This entails examining $T \cdot 2^b$ boundaries between sub-buckets, assigning each a slice from the global free stack, and notifying the threads that these slices have been pre-allocated.

The notification is done through the *sd* database – any time *WCv4* aims to obtain a new slice, it loads the pointer x for the current slice j of bucket i for its thread p ; if this value is NULL, it gets a new slice from its local stack; otherwise, it uses x as a pointer to the pre-allocated slice. Not only that, but x specifies the exact location within the slice where sub-bucket (ip) begins. Combining this logic with generalized offloads that can move across slices (Section IV-D) leads to our final *WCv4L4* splitter in Typhoon.

The worst-case memory usage happens at $\mathcal{L}_4$. Observe that threads may hold $T \cdot A_0$ slices in matrix M before the level begins, they are given $T \cdot 2^b$ additional shared slices in pre-allocated memory, and they request $T \cdot 2^b$ new slices immediately after starting the level. As a result, the smallest number of auxiliary slices A that the sort needs is $T \cdot 2^b(3 + R/(2S))$. With 4-KB slices and $R = 256$ bytes per tmp bucket, this leads to 3 MB per thread. Compared to 1 GB/thread worst-case in Vortex [20], this is a major improvement.

It should also be noted that the modified histogram *Hv4* applies only to the single-threaded case since complementing the empty space does not reveal which thread is responsible for which slice. Additionally, $T \geq 2$ leads to tight interleaving of slices, where the length of contiguous regions assigned to each thread, even if they could be determined efficiently, is often no more than 2 slices, which negates the sought-after benefits. Therefore, we run histogram *Hv4* only for $T = 1$ thread and *Hv3* otherwise.

F. Reshuffle ($\mathcal{L}_5$)

After $\mathcal{L}_4$, the data is located in $\lceil n \cdot \text{ItemSize}/S \rceil$ slices, which are randomly scattered in RAM. Assume $\{v_1, v_2, \dots\}$ is a list of pointers in row-major order within matrix M recorded in the slice database during $\mathcal{L}_4$, ignoring duplicate slices shared across adjacent buckets. Now the remaining task is to make each slice v_i appear at offset iS in the input buffer. To accomplish this, the Typhoon constructor obtains a chunk of memory *buf*, big enough to hold both n input items and

TABLE VIII
TYPHOON SPEED (M/SEC) ON 1 GB INPUT (UNIFORM KEYS)

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

TABLE IX
STRONG SCALING OF TYPHOON SPEED (M/SEC) ON 1 GB OF 32-BIT KEYS (16-KB SLICES, UNIFORM KEYS)

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×

TABLE X
STRONG SCALING OF TYPHOON SPEED (M/SEC) ON 1 GB OF 64-BIT KEY-VALUE PAIRS (16-KB SLICES, UNIFORM KEYS)

Level	1 core	2 cores	3 cores	4 cores	5 cores	6 cores	7 cores	8 cores
0	865	1,718 2.0×	2,557 3.0×	3,380 3.9×	4,144 4.8×	4,538 5.2×	4,564 5.3×	4,513 5.2×
1	846	1,683 2.0×	2,495 2.9×	3,279 3.9×	4,010 4.7×	4,421 5.2×	4,468 5.3×	4,456 5.3×
2	853	1,692 2.0×	2,504 2.9×	3,306 3.9×	4,034 4.7×	4,433 5.2×	4,484 5.3×	4,460 5.2×
3	2,034	3,427 1.7×	5,084 2.5×	6,700 3.3×	8,154 4.0×	9,290 4.6×	9,940 4.9×	10,217 5.0×
4	846	1,677 2.0×	2,493 2.9×	3,295 3.9×	3,945 4.7×	4,267 5.0×	4,333 5.1×	4,344 5.1×
0-4	193	377 2.0×	559 2.9×	737 3.8×	897 4.7×	986 5.1×	1,003 5.2×	1,002 5.2×
5	7,370	13,849 1.9×	19,683 2.7×	24,427 3.3×	29,339 4.0×	32,811 4.5×	36,358 4.9×	38,347 5.2×
0-5	188	367 2.0×	544 2.9×	716 3.8×	870 4.6×	957 5.1×	976 5.2×	976 5.2×

A auxiliary slices, using OS primitives that allow physical pages to be mapped/unmapped within this virtual space. For Windows, this translates into a call to `VirtualAlloc` with the `MEM_PHYSICAL` flag. Typhoon then grabs enough physical pages using `AllocateUserPhysicalPages` and maps them to this buffer via `MapUserPhysicalPages`. The `buf` array is then given to the user to fill in the keys.

Note that the OS provides the PFN (physical frame number) of each allocated page, which Typhoon stores internally in the `pfn` array for later use during remapping. Level $\mathcal{L}_5$ begins with unmapping all slices in `buf` using T parallel threads. The information within these pages does not get destroyed, but becomes temporarily inaccessible. After finishing the unmap, threads jointly construct the `nextPfn` array that specifies the page frames that need to appear in each position in `buf`.

In more detail, assuming P is page size in bytes, observe that $(v[i] - \text{buf})/P$ is the offset in the `pfn` array that contains the S/P page frames from slice v_i . Similarly, the offset in the `nextPfn` buffer where v_i should be mapped to is given by $i*S/P$. Therefore, construction of `nextPfn` is a sequence of operations `memcpy(nextPfn + i*S/P, pfn + (v[i] - buf)/P, S/P*sizeof(void*))` for all i . After the threads are done

with unmapping and `memcpy`, they synchronize on a barrier and call `MapUserPhysicalPages` with their assigned portion of `nextPfn`. Another barrier follows, after which Typhoon finishes the sort by swapping `pfn` and `nextPfn` pointers in preparation for the next iteration (if needed). The sort can be called repeatedly any number of times, reusing `buf` and other data structures, without causing allocating of new memory.

V. EVALUATION

Our primary benchmark platform is an Intel i7-7820X, which is an 8-core Skylake-X CPU with a 32-KB L1, 256-KB L2, and 16-MB L3, clocked for these experiments at a fixed 4.7 GHz on each core. We run 32 GB of DDR4-3200 RAM in a quad-channel memory configuration, which yields a peak non-temporal AVX `memcpy` bandwidth of 37 GB/s and a maximum AVX read speed of 86 GB/s across 8 cores. Single-threaded Typhoon is bottlenecked by the splitter's 4.2c/key in Table IV, while the performance of the multi-core version is upper bounded by four `memcpy` passes and one read pass, i.e., 2088M keys/sec for 32-bit items and half of that for 64-bit.

Typhoon, whose source code is available from [38], is compiled in Visual Studio 2019, while prior methods are

reported using the best achievable speed among Clang 19, Intel oneAPI C++ 2025 (ICX), and VS 2019. Benchmarks run on Windows Server 2016 and Ubuntu 24.04.

A. Static vs Sliced Typhoon

Our first topic is to examine Typhoon in comparison to its static version and assess performance loss due to slicing and remapping. Table VIII shows this result using 4-16 KB slices and breaks down the speed for individual levels $\mathcal{L}_0, \dots, \mathcal{L}_5$. For 32-bit keys in the first four columns, the single-threaded Typhoon shows a 1-5% advantage over the static version due to slice reuse and ties S-Typhoon at $\mathcal{L}_3$. Even 4-KB slices allow Typhoon to finish the sort on levels 0-4 faster than S-Typhoon (i.e., 259 vs 256M/sec). Adding remapping at $\mathcal{L}_5$, which is 8-13 $\times$ faster than the splitter, yields a final Typhoon speed between 252 and 261M/sec, depending on slice size. This is quite competitive against S-Typhoon; in fact, slices 8 KB or larger lead to no loss of performance.

For 64-bit items (i.e., 32-bit keys with 32-bit values), the number of records per slice is reduced by half, which means that all slice-related activities occur twice as frequently on a per-key basis. Thus, it is not surprising that in these cases Typhoon needs double the slice size to achieve the same relative performance (e.g., 16 KB to match S-Typhoon on 64-bit items vs 8 KB on 32-bit, shown in bold in Table VIII). For multi-threaded cases on the right side of the table, Typhoon again generally runs faster than S-Typhoon when splitting on $\mathcal{L}_0 - \mathcal{L}_2$; however, it now loses up to 18% on the histogram pass (i.e., level $\mathcal{L}_3$). This is because the optimized version Hv4 does not work with $T \geq 2$ threads and the sort has to use the slower Hv3. Adding the cost of $\mathcal{L}_5$, where the OS struggles to maintain linear scaling of remapping speed, results in 1-5% loss on the full sort. Nevertheless, Typhoon-16KB hits $1874/2088 = 89\%$ of memcpy bandwidth using 32-bit items and $976/1044 = 93\%$ using 64-bit.

Table IX examines strong scaling of 32-bit speed as the number of threads increases, including a multiplicative factor improvement compared to the single-threaded version. We fix the input size at 1 GB and set the affinity mask to one thread per core, which yields the best result. From the table, observe that splitter speed (levels $\mathcal{L}_0 - \mathcal{L}_2, \mathcal{L}_4$) scales almost perfectly until 6 cores, but then slows down to 6.7 $\times$ at 7 cores and 7.3 $\times$ at 8 cores as it starts approaching RAM bandwidth. At the peak, the splitter reaches 8.8B keys/sec, or 35.3 GB/s. On the other hand, the histogram at $\mathcal{L}_3$ shows a noticeably worse scaling behavior, which arises from the fact that the single-threaded version Hv4 has an 11% advantage over the multi-threaded Hv3. In particular, the former runs at 2,931M/sec (first column), while the latter delivers at most $5,256/2 = 2,628$ M/sec per core (second column), eventually saturating at 18.3B/sec, or 73 GB/s, in the last column. It is also interesting to observe that the OS fails to linearly scale its remapping speed on $\mathcal{L}_5$, finishing with a 5.2 $\times$ speed-up on 8 cores. Considering all these factors, Typhoon’s final speed in the bottom row is quite reasonable.

Table X examines the same benchmark on 64-bit key-value pairs. Given the larger item size, RAM-bandwidth saturation occurs earlier, where 6 threads are essentially enough. As shown in the first row, the splitter’s best rate (i.e., 4.5B/sec, or 36.5 GB/s) is only barely faster than the 32-bit result in Table IX, but the 64-bit histogram manages to hit 82 GB/s (i.e., 10.2B/sec) in the last column, exceeding the 32-bit rate by a much bigger margin (i.e., 9 GB/s). This was expected since the amount of CPU cycles per key is about the same in both cases, but larger items generate more RAM traffic. In fact, this can be gleaned by comparing the single-threaded speed of $\mathcal{L}_3$ across both tables – the 32-bit histogram reads at 12 GB/s, while the 64-bit version goes 16 GB/s. This faster RAM scanning also explains the increased penalty from incorrect prefetch in Hv3, which in Table X runs 16% slower than Hv4. The parallel component of the sort from Amdahl’s law can be estimated as $p = 0.99$ in Table IX and 0.98 in Table X.

To analyze weak scaling, we vary the number of cores $c = 1, 2, \dots, 8$ and set input size to nc , where $n = 2$ GB, which produces sorts that range from 2 to 16 GB. Compared to Table IX, the biggest difference in the new setup is a 2% improvement in $\mathcal{L}_0$ speed at 8 cores, which reaches 9,035M/sec (i.e. 36.1 GB/s). The overall sort speed increases slightly, but remains within 1% of the previous results. Compared to Table X, the 64-bit outcome shows even less variation, with all numbers staying within 0.7%. To avoid clutter, we omit these tables. Curve-fitting Gustafson’s law to the data, the parallel component of the sort comes out to $p = 0.93$.

B. Baseline Sorts

For the next group of tests, we use five synthetic 8-GB datasets: uniformly random integers ($\mathcal{D}_1$); a sorted sequence of uniform numbers, where every 7-th key is set to `UINT_MAX` ($\mathcal{D}_2$); uniformly random keys, each repeated U times, where U is drawn from the Zipf distribution with $\alpha = 1, \beta = 7$, then shuffled randomly ($\mathcal{D}_3$); integer keys drawn from a normal distribution with mean `UINT32_MAX/2` and standard deviation equal to 1/3 of the mean ($\mathcal{D}_4$); and uniformly random floats between 0 and `FLT_MAX` ($\mathcal{D}_5$). We also use one real-world dataset, which is an inter-domain out-graph $\mathcal{G}$ from the IRLbot web crawl [24], consisting of 89M nodes and 1.8B edges. We leverage $\mathcal{G}$ for two standard applications – *computation of in-degree*, which entails sorting 7.2 GB of out-neighbor adjacency lists, and *graph inversion*, which requires either a stable key-value sort on 14.4 GB of (dest, src) pairs, where each node ID is 32-bit, or an unstable 64-bit key sort.

Table XI shows the speed on 32-bit keys, where we partition prior work into four groups (top to bottom) – MSD (most significant digit) radix sort, LSD (least significant digit) radix sort, quick/sample sort, and merge sort – each in chronological order of publication. When a method relies on SIMD, we specify after its name the vector width (i.e., 128, 256, or 512) used in the benchmark. We highlight the fastest prior approach in each column with a gray background, run Typhoon with 16-KB slices (i.e., 12 MB of aux memory per thread), and show

TABLE XI
SINGLE-CORE SPEED (M/SEC) ON 32-BIT KEYS

Sort	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{G}$
Gorset [19]	37	52	51	38	42	44
Polychroniou [31]	34	34	32	32	30	–
Ska [35]	40	96	81	51	84	84
Regions [28]	77	58	93	79	96	85
Voracious [30]	79	80	86	81	84	86
Vortex [20]	150	122	128	135	147	127
IPS ² Ra [7]	46	107	121	58	101	127
Dovetail [15]	103	99	94	103	102	99
Reinald [32]	96	100	103	101	100	111
Fast-Radix [37]	69	68	71	70	70	72
DFR [36]	76	69	97	67	79	129
pdqsort [29]	34	55	80	34	53	56
Blacher-256 [10]	133	109	117	133	133	131
IPS ⁴ o [7]	36	50	51	36	50	55
Highway-512 [18]	115	128	176	115	115	140
Intel-512 [23]	149	158	48	154	153	78
Origami-512 [5]	131	131	131	131	131	131
Typhoon-16KB	257 1.7×	259 1.6×	261 1.5×	260 1.7×	259 1.7×	261 1.9×

the speed-up factor against the best alternative in the bottom row. Dashes indicate inability to finish the sort (e.g., crashing, failing sortedness checks, unsupported input size).

There are three types of methods that stand out – the Vortex MSD [20], an SIMD quick sort from Blacher [10], with more general implementations at Google [18] and Intel [23], and the Origami SIMD merge sort [5]. All three perform quite well, delivering over 130M/sec on at least one dataset, but there is no clear winner between them. Origami posts remarkably stable speed in all columns, but it is neither the fastest nor in-place. Vortex wins on uniform data, but drops 19% between $\mathcal{D}_1$ and $\mathcal{D}_2$. Blacher/Highway have similar levels of fluctuation, while Intel takes a 68% dive on $\mathcal{D}_3$ and 48% on $\mathcal{G}$. In contrast, Typhoon wins in all six columns, runs in-place, and posts a 50-90% improvement over the best prior methods. Furthermore, its speed on non-uniform data is never slower than on $\mathcal{D}_1$, while the deviation between the max and the min is only 1.9%. This was expected from its robustness against key non-uniformity.

Among the 17 prior methods in Table XI, only six support multi-threading. Their all-core speed is displayed in Table XII. Across related work, Origami wins in the uniform case by a large margin and delivers the best result in three additional columns; however, it needs double the RAM of Typhoon and uses power-hungry AVX-512 intrinsics to achieve this level of performance. With just SSE, its speed drops by 1.6×, while the other AVX-512 methods in Table XI sustain even more damage (e.g., Highway is 2.8× slower with SSE than with AVX-512). In contrast, Typhoon in Table XII operates mostly using scalar instructions and still almost doubles the speed of the best prior work in all six cases. This outcome is even faster than we saw previously in Table VIII because of larger input (i.e., 8 GB instead of 1 GB), where longer runs of full slices in each bucket lead to higher efficiency.

For the next experiment, we use 64-bit items composed of 32-bit key-value pairs. Some of the prior work does not have

TABLE XII
ALL-CORE SPEED (M/SEC) ON 32-BIT KEYS

Sort	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{G}$
Regions [28]	689	667	689	700	675	761
Voracious [30]	581	906	566	597	587	688
IPS ² Ra [7]	526	967	1049	650	777	816
Dovetail [15]	312	350	257	339	326	267
IPS ⁴ o [7]	327	432	450	327	417	458
Origami-512 [5]	919	927	930	939	946	931
Typhoon-16KB	1,879 2.0×	1,879 1.9×	1,920 1.8×	1,891 2.0×	1,915 2.0×	1,912 2.1×

TABLE XIII
SINGLE-CORE SPEED (M/SEC) ON 64-BIT KEY-VALUE PAIRS

Sort	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{G}$
Gorset [19]	21	46	21	24	21	20
Polychroniou [31]	27	20	11	25	14	–
Ska [35]	36	83	68	38	67	32
Raduls2 [25]	82	65	56	92	58	53
Regions [28]	49	45	70	56	72	29
Voracious [30]	57	54	59	53	53	56
Vortex [20]	120	102	68	117	63	57
IPS ² Ra [7]	45	96	104	45	88	38
Dovetail [15]	67	67	67	67	68	62
Reinald [32]	39	39	39	38	40	37
Fast-Radix [37]	40	40	43	40	43	38
DFR [36]	49	47	–	48	33	–
pdqsort [29]	31	48	32	31	31	30
IPS ⁴ o [7]	31	38	41	30	39	27
Highway-512 [18]	57	57	58	57	57	54
Intel-512 [23]	76	73	75	76	76	69
Origami-512 [5]	55	55	55	55	55	53
Typhoon-16KB	184 1.5×	188 1.8×	193 1.9×	186 1.6×	202 2.3×	192 2.8×

a separate provision for this case, requiring that such items be treated as monolithic 64-bit keys. Reasons include faster performance (e.g., in comparison-based SIMD methods) and ability to use unstable sorts to achieve common database tasks that would otherwise need stability (e.g., graph inversion). In this comparison, we omit Blacher [10] since it only works with 32-bit keys and add another MSD method Raduls2 [25], which was absent previously as it requires key length to be a multiple of 8 bytes.

Table XIII shows the single-threaded outcome. First notice that, compared to 32-bit cases in Table XI, the three AVX-512 methods take a huge performance hit, sinking from 115-149M/sec to 55-76M/sec. Second, even though Vortex delivers excellent results for the uniform case $\mathcal{D}_1$ (i.e., 120M/sec), it degrades to ~65M/sec on $\mathcal{D}_3$ and $\mathcal{D}_5$. Graph inversion on $\mathcal{G}$ also gets derailed, achieving only 57M/sec, i.e., a 2.1× reduction compared to the uniform case. This highlights our earlier point that real-world datasets are often skewed in a way that can heavily destabilize performance of MSD methods. In another similar case, Raduls2 suffers a 1.5× speed drop on $\mathcal{G}$. Things get worse on adversarial inputs, where Vortex slows down to a measly 13M/sec. In contrast, Typhoon in Table XIII shows consistent performance across the columns, delivering a speed-up that ranges from 1.5× on $\mathcal{D}_1$ to 2.8× on $\mathcal{G}$.

Scalability to multiple threads is shown in Table XIV.

TABLE XIV
ALL-CORE SPEED (M/SEC) ON 64-BIT KEY-VALUE PAIRS

Sort	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{G}$
Raduls2 [25]	656	478	394	737	433	491
Regions [28]	365	382	359	351	296	291
Voracious [30]	402	510	397	405	340	298
IPS ² Ra [7]	409	737	623	418	466	318
Dovetail [15]	198	206	177	197	197	130
IPS ⁴ o [7]	286	366	351	291	341	286
Origami-512 [5]	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×

TABLE XV
SINGLE-CORE $\mathcal{L}_0$ SPEED (M/SEC)

Sort	32-bit			64-bit		
	$\mathcal{D}_1$	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{D}_1$	$\mathcal{N}_1$	$\mathcal{N}_2$
Gorset [19]	113	128	178	75	90	157
Ska [35]	253	300	224	187	270	206
Raduls2 [25]	—	—	—	462	358	140
Vortex [20]	649	555	160	652	501	155
IPS ² Ra [7]	547	504	163	370	341	163
Dovetail [15]	266	224	242	175	166	178
Reinald [32]	383	291	142	227	229	122
Fast-Radix [37]	215	182	93	128	121	74
DFR [36]	227	215	66	160	158	56
Typhoon-16KB	1,183	1,189	1,211	865	1,050	862

Among prior work, Raduls2 wins in three columns, while IPS²Ra owns the top spot for the remaining cases. As encountered before, decision between previous methods is difficult, especially considering that IPS²Ra is in-place, while Raduls2 is not. However, with the introduction of Typhoon, this choice becomes simpler – our method reliably, and by a wide margin, yields the best speed. Its current performance is stifled by insufficient RAM bandwidth, where linear scaling of the numbers in Table XIII suggest a peak rate of $184 \times 8 = 1472\text{M/sec}$, i.e., $1.5 \times$ more than shown in Table XIV. Its auxiliary memory overhead is just 1.2% of the input, which can dropped to 0.3% using 4-KB slices at the cost of 3-4% speed reduction.

C. Raw Splitter Speed

To better understand Typhoon’s splitter performance in the context of prior methods, Table XV benchmarks the available radix-partitioning implementations using uniform numbers from dataset $\mathcal{D}_1$, uniform numbers repeated back-to-back with run length 64, which we call dataset $\mathcal{N}_1$, and the round-robin input with skip factor $s = 16$, which we call $\mathcal{N}_2$. The last two tests are similar to those we performed for Tables IV-V using S-Typhoon.

In the $\mathcal{D}_1$ column, observe that Typhoon’s main algorithm (i.e., WCV4) defeats all prior methods by a large margin. Reasons for the slow performance in existing work include the need for a histogram pass [15], [19], [25], [35], [32], expensive dynamic bucket expansion [7], [20], [37], [36], and inefficient interaction with RAM (e.g., lacking write-combine [7], [15], [19], [32], [35], [37], [36] and/or latency-related bottlenecks during key swapping [19], [35]).

Algorithm 10: Dataset $\mathcal{N}_2$ (LSD order, 32-bit keys)

```

1 Function Generate(Item *in, int n, int s)
2   d1 = 256/s; d2 = d1 * d1; d3 = d2 * d;
3   for (i = 0; i < n; i++) do
4     i1 = (i / d3) % d1; i2 = (i / d2) % d1;
5     i3 = (i / d1) % d1, i4 = i % d1;
6     in[i] = ((i1*s) << 24) + ((i2*s) << 16) + ((i3*s) << 8) + i4*s;

```

TABLE XVI
IMPACT OF SKEWED DISTRIBUTIONS (1 GB)

Family	Sort	$\mathcal{D}_1$	Dataset	Speed	Change
LSD	Reinald [32]	113	$\mathcal{N}_2$ (32-bit)	36	-68%
	Fast-Radix [37]	85		31	-63%
	DFR [36]	123		29	-76%
	Typhoon-16KB	260		260	0%
MSD	Ska [35]	47	$\mathcal{N}_3$ (64-bit)	28	-40%
	Raduls2 [25]	114		22	-81%
	Vortex [20]	147		13	-91%
	IPS2Ra [7]	52		28	-46%
	Typhoon-16KB	187		198	6%
Quicksort	Blacher-256 [10]	153	$\mathcal{N}_4$ (32-bit)	56	-76%
	Typhoon-16KB	260		289	11%

For $\mathcal{N}_1$, some of the prior methods avoid store-to-load forwarding stalls by using a much slower splitter, while the faster approaches [20], [25], [32] take a significant (i.e., up to 24%) performance hit. Things get worse in $\mathcal{N}_2$, where the penalty of cache/TLB conflict evictions affects 6 of the 9 methods, even if they do not use write-combine. For example, Vortex suffers a $4.2 \times$ drop, while Raduls2, DFR, and IPS²Ra slash speed by $3.3 \times$. In contrast, Typhoon remains robust in both situations, delivering over 1180M/sec on 32-bit items and 860M/sec on 64-bit.

D. Robustness

We next examine the impact of skewed distributions on sort speed and resilience of the methods against non-uniformity. We first extend the dataset $\mathcal{N}_2$ to implement round-robin in every byte of the key. For LSD methods, this is shown in Algorithm 10, which increments the lowest byte by s until that byte overflows, adds s to the next byte, and restarts the process from the lowest byte. Applying this input to the three LSD methods under consideration yields the result in the first section of Table XVI. Note that this table uses smaller (1-GB) inputs than before, which yields faster runtimes than in Table XI. Compared to the uniform case $\mathcal{D}_1$, all three methods take a significant hit, dropping performance 60-75%. In contrast, Typhoon remains at 260M/sec on both datasets due to the introduced stagger of tmp buckets in Section III-E.

For MSD methods, we construct a 64-bit sequence $\mathcal{N}_3$, which takes a uniform key x and uses it to produce a batch of m keys $y_1, \dots, y_m$. Batches have an important property that a) at every level of recursion, they split off one “rogue” key into some bucket and send the remaining items into another; and b) the batch size at each level exceeds threshold $\mathcal{T}$ for stopping recursion and switching to another sort. This guarantees maximum depth of recursion for $m - d$ keys, where $d = \lceil w/b \rceil$ is the number of levels needed for a w -bit key

Algorithm 11: Dataset $\mathcal{N}_3$

```

1 Function Generate(Item *in, int n, int  $\mathcal{T}$ )
2   d = sizeof(KeyType);  $\triangleright$  depth of recursion
3   for (i = 0; i < n; ) do
4     x = random(0, KEY_MAX);
5     for (k = 0; k <  $\mathcal{T}$ ; k++) do
6       in[i + k] = x ^ k;  $\triangleright$  batch stays together
7     for (k = 0; k < d; k++) do
8       mask = 0xFF << (8*k);  $\triangleright$  flips the k-th byte
9       in[i +  $\mathcal{T}$  + k] = x ^ mask;  $\triangleright$  rogue key
10    i +=  $\mathcal{T}$  + d;

```

Algorithm 12: Dataset $\mathcal{N}_4$

```

1 Function Generate(Item *in, int n)
2   for (i=0; i < n; i++) do
3     p = random(0, 1);
4     if p < 0.92 then
5       in[i] = n;
6     else if p < 0.94 then
7       in[i] = i * i;
8     else
9       in[i] = n - i;

```

under b -bit splitting. Furthermore, this construction defeats *sortedness checks* that can stop recursion early and *prefix jumping*, i.e., detection of a common prefix on all keys in a bucket and skipping over it. Finally, by setting m to its minimum value $\mathcal{T} + d$, we can also ensure maximum relative overhead in scanning 2^b buckets to determine which ones are non-empty compared to the work being done during key partitioning of size- m buckets.

Algorithm 11 shows how to generate the baseline $\mathcal{N}_3$, which is then randomly shuffled to separate the keys in each batch from each other. Performance reduction of existing MSD methods on this dataset is shown in the middle of Table XVI. While all of them are affected to some degree, none drop speed more than Vortex, which suffers an $11\times$ slowdown. This can be explained by a $2.7\times$ increase in recursion depth (i.e., from 3 to 8) and the tiny threshold $\mathcal{T} = 32$ used in the method, where scanning 256 buckets at each level to find a batch of $\mathcal{T} + d = 40$ keys makes the splitter significantly more expensive. In contrast, Typhoon gains a 6% speed-up on this dataset since LSD is not sensitive to bucket sparsity.

We round up this discussion with Blacher’s SIMD quicksort [10], for which dataset $\mathcal{N}_4$ in Algorithm 12 produces much deeper recursion chains than the expected $\log_2 n$. This is done by ensuring that no pivot can be selected in each bucket to produce load-balanced partitions, which is achieved by mixing a large number of duplicates with two sequences of unique keys that grow in opposite directions. Even with protection against quadratic worst-case scenarios, [10] exhibits a 76% reduction in speed due to imbalance in partition size, as shown at the bottom of Table XVI. On the other hand, Typhoon is not effected by these types of input, which was expected, and actually delivers an 11% speed-up.

TABLE XVII
MACHINE SPECIFICATIONS FOR IN-PLACE TESTS

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

TABLE XVIII
SINGLE-CORE IN-PLACE SPEED (M/SEC) ON 32-BIT KEYS

Sort	SB	IB	BW	CL	AL	Zen4	Zen5
Gorset [19]	25	26	24	48	46	71	73
Polychroniou [31]	20	21	23	35	44	49	53
Ska [35]	40	43	41	84	99	116	120
Regions [28]	53	57	52	89	124	121	142
Vortex [20]	54	56	53	162	178	203	265
IPS ² Ra [7]	—	—	47	90	109	110	121
pdqsort [29]	23	24	24	33	40	45	47
IPS ⁴ o [7]	—	—	22	33	34	46	50
Highway [18]	21	22	42	77	106	149	185
Intel [23]	—	—	63	118	167	177	240
std::sort	7	7	7	9	10	13	13
Typhoon-16KB	120 2.2×	129 2.3×	129 2.0×	265 1.6×	328 1.8×	388 1.9×	491 1.8×

E. Other Platforms and In-Place Experiments

We next examine performance across seven additional CPU architectures whose characteristics are shown in Table XVII. The first three entries are server CPUs that use quad-channel memory, while the other four are dual-channel desktop chips. Sandy/Ivy Bridge implement SIMD instruction sets up to AVX, Broadwell/Coffee/Alder Lake support up to AVX2, and Zen4/Zen5 allow AVX-512. For each of the configurations, we leave 5 GB for background processes and fill the rest with uniform keys. This leads to sort sizes 27-251 GB and requires methods that can operate in-place.

Only 10 prior implementations satisfy this criterion. Their performance on 32-bit keys is shown in Table XVIII, where `std::sort` is added for reference. Note that we remove Blacher [10] since its usage of AVX2 `gather` instructions restricts array indexes to 4 bytes, which limits the sort to 16 GB. In the first two columns of Table XVIII, Highway [18] is forced to use SSE, resulting in worse speed (i.e., 21M/sec) than a basic implementation [19] of the American Flag Sort [27] in the first row. This is in contrast to earlier Skylake-X results (Table XI), where [18] was $3.1\times$ faster. The Intel version [23] of the same algorithm does not support SSE, while both IPSxx methods crash on inputs above ~ 128 GB. In the end, Vortex [20] squeaks out a win on Sandy Bridge and Regions [28] on Ivy Bridge, but Typhoon in the last row manages to more than double their performance.

Once AVX2 kicks in on Broadwell, SIMD methods become more competitive in the third column, with Intel climbing to the top. Vortex mounts a comeback on Coffee/Alder Lake in the next two columns, but is still 1.6 - $1.8\times$ slower than

TABLE XIX
SINGLE-CORE IN-PLACE SPEED (M/SEC) ON 64-BIT KEY-VALUE PAIRS

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

Typhoon. With a jolt from AVX-512 in the last two columns, Intel almost catches up to Vortex; however, both methods are still roughly half of Typhoon’s 388M/sec on Zen4 and 491M/sec on Zen5. Similar observations hold for 64-bit key-value pairs in Table XIX. Outside of Vortex, which uses up to 1 GB of extra memory and exhibits high volatility on non-uniform keys, the other prior methods are at least $2.6\times$ slower on Coffee Lake, $3.3\times$ on Zen4, and $3\times$ on Zen5.

Overall, results show that across a range of desktop/server generations, Intel/AMD CPU offerings, and SSE/AVX2/AVX-512 instruction sets, Typhoon delivers the best performance, consistently taking the top spot in every comparison and finishing 32-bit sorts $38\times$ faster than `std::sort` on AMD Zen5. Furthermore, it is highly skew-resilient, as well as the only method in this comparison that is both stable and in-place.

VI. ACKNOWLEDGMENTS

The authors are grateful to Benjamin Ramon and Carson Hanel for initial prototyping and insightful discussion.

VII. CONCLUSION

We developed a novel LSD sort called Typhoon and demonstrated that it worked remarkably fast across a variety of CPU architectures, memory configurations, single/multi-core scenarios, input skew, and array sizes. Not only that, but Typhoon is also stable, in-place, and distribution-insensitive. Future work involves handling longer keys and testing Typhoon in big-data frameworks/databases.

REFERENCES

- [1] A. Abel and J. Reineke, “uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures,” in *Proc. ACM ASPLOS*, Apr. 2019, pp. 673–686. [Online]. Available: <https://uops.info/table.html>.
- [2] A. Al-Badarneh and F. El-Aker, “Efficient Adaptive In-Place Radix Sorting,” *Informatica*, vol. 15, no. 3, pp. 295–302, Jan. 2004.
- [3] Apache Hadoop. [Online]. Available: <http://hadoop.apache.org/>.
- [4] Apache Spark. [Online]. Available: <https://spark.apache.org/>.
- [5] A. Arman and D. Loguinov, “Origami: A High-Performance Mergesort Framework,” in *Proc. VLDB*, Sep. 2022, pp. 259–271.
- [6] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, “In-place Parallel Super Scalar Samplesort (IPS4o),” in *Proc. European Symposium on Algorithms (ESA)*, Sep. 2017, pp. 9:1–9:14.
- [7] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, “Engineering In-place (Shared-memory) Sorting Algorithms,” *ACM Transactions on Parallel Computing*, vol. 9, no. 1, pp. 1–62, Jan. 2022.
- [8] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, “Multi-core, Main-memory Joins: Sort vs. Hash Revisited,” *VLDB Endow.*, vol. 7, no. 1, pp. 85–96, Sep. 2013.
- [9] K. E. Batchner, “Sorting Networks and their Applications,” in *Proc. Spring Joint Computer Conference*, 1968, pp. 307–314.
- [10] M. Blacher, J. Giesen, and L. Kühne, “Fast and Robust Vectorized In-Place Sorting of Primitive Types,” in *Proc. International Symposium on Experimental Algorithms (SEA)*, Jun. 2021, pp. 3:1–3:16.
- [11] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, “Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture,” *VLDB Endow.*, vol. 1, no. 2, pp. 1313–1324, Aug. 2008.
- [12] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, and R. Puri, “PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort,” *VLDB Endow.*, vol. 8, no. 12, pp. 1518–1529, Aug. 2015.
- [13] Y. Cui, D. Xiao, D. B. Cline, and D. Loguinov, “Improving I/O Complexity of Triangle Enumeration,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 4, pp. 1815–1828, Apr. 2022.
- [14] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” in *Proc. USENIX OSDI*, Dec. 2004, pp. 137–150.
- [15] X. Dong, L. Dhulipala, Y. Gu, and Y. Sun, “Parallel Integer Sort: Theory and Practice,” in *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Mar. 2024, pp. 301–315.
- [16] T. Downs, “Memory Disambiguation on Skylake,” 2021. [Online]. Available: <https://github.com/travisdowns/uarch-bench/wiki/Memory-Disambiguation-on-Skylake>.
- [17] K. Fukazawa and R. Takahashi, “Performance Evaluation of the Fourth-Generation Xeon with Different Memory Characteristics,” in *Proc. HPC Asia Workshops*, Jan 2024, pp. 55–62.
- [18] Google, “Highway Sort,” 2024. [Online]. Available: <https://github.com/google/highway/tree/master/hwy/contrib/sort>.
- [19] E. Gorset, “In-place Radix Sort,” Apr. 2011. [Online]. Available: <https://github.com/gorset/radix>.
- [20] C. Hanel, A. Arman, D. Xiao, J. Keech, and D. Loguinov, “Vortex: Extreme-Performance Memory Abstractions for Data-Intensive Streaming Applications,” in *Proc. ACM ASPLOS*, Mar. 2020, pp. 623–638.
- [21] H. Inoue and K. Taura, “SIMD-and cache-friendly algorithm for sorting an array of structures,” *PVLDB*, vol. 8, no. 11, pp. 1274–1285, Jul. 2015.
- [22] Intel Corporation, “Intel 64 and IA-32 Architectures Optimization Reference Manual Volume 1,” [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- [23] Intel Corporation, “x86-simd-sort,” 2024. [Online]. Available: <https://github.com/intel/x86-simd-sort>.
- [24] IRL Web Crawling Datasets. [Online]. Available: <http://irl.cs.tamu.edu/projects/web/>.
- [25] M. Kokot, S. Deorowicz, and A. Debudaj-Grabysz, “Even Faster Sorting of (Not Only) Integers,” in *Proc. International Conference on Man-Machine Interactions*, Oct. 2017, pp. 481–491.
- [26] M. Kokot, S. Deorowicz, and A. Debudaj-Grabysz, “Sorting data on ultra-large scale with RADULS,” in *Proc. Beyond Databases, Architectures and Structures*, Sep. 2017, pp. 235–245.
- [27] P. M. McIlroy, K. Bostic, and M. D. McIlroy, “Engineering Radix Sort,” *Computing Systems*, vol. 6, no. 1, pp. 5–27, 1993.
- [28] O. Obeyea, E. Kahssay, E. Fan, and J. Shun, “Theoretically-efficient and Practical Parallel In-place Radix Sorting,” in *Proc. ACM SPAA*, Jun. 2019, pp. 213–224.
- [29] O. Peters, “Pattern-defeating Quicksort (pdqsort),” 2021. [Online]. Available: <https://github.com/orlp/pdqsort>.
- [30] A. Piot, “Voracious Sort,” 2020. [Online]. Available: https://github.com/lakwet/voracious_sort.
- [31] O. Polychroniou and K. A. Ross, “A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort,” in *Proc. ACM SIGMOD*, Jun. 2014, pp. 755–766.
- [32] A. Reinald, P. Harris, R. Rohrer, and J. Dirk, Radix sort. [Online]. Available: http://www.cubic.org/docs/download/radix_ar_2011.cpp.
- [33] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, and P. Dubey, “Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort,” in *Proc. ACM SIGMOD*, Jun. 2010, pp. 351–362.
- [34] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich, “On the Surprising Difficulty of Simple Things: The Case of Radix Partitioning,” *VLDB Endow.*, vol. 8, no. 9, pp. 934–937, May 2015.

- [35] M. Skarupke, “I Wrote a Faster Sorting Algorithm,” 2016. [Online]. Available: <https://probablydance.com/2016/12/27/i-wrote-a-faster-sorting-algorithm/>.
- [36] S. Thiel, “Implementing the Diverting Fast Radix Algorithm,” Aug. 2022. [Online]. Available: <https://arxiv.org/abs/2207.14334>.
- [37] S. Thiel, G. Butler, and L. Thiel, “Improving GraphChi for Large Graph Processing: Fast Radix Sort in Pre-Processing,” in *Proc. ACM IDEAS*, Jul. 2016, pp. 135–141.
- [38] Typhoon. [Online]. Available: <http://irl.cs.tamu.edu/projects/streams>.
- [39] J. Wassenberg, M. Blacher, J. Giesen, and P. Sanders, “Fast and Robust Vectorized In-Place Sorting of Primitive Types,” *Software: Practice and Experience*, vol. 52, no. 12, pp. 2684–2699, 2022.
- [40] J. Wassenberg and P. Sanders, “Engineering a Multi-core Radix Sort,” in *Proc. Euro-Par*, Aug. 2011, pp. 160–169.
- [41] H. Wong, “Store-to-Load Forwarding and Memory Disambiguation in x86 Processors,” 2014. [Online]. Available: <https://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/>.