F5: A Robust SIMD-Accelerated MSD Radix Sort

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

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

***Abstract*—Sorting is a building block of many data-intensive applications, databases, MapReduce pipelines, and large-scale distributed systems. In this paper, we focus on MSD (most-significant digit first) radix sort, in which we identify three main bottlenecks – slow small-bucket sorts at the end of recursion, partitioning algorithms susceptible to read-after-write stalls, and unnecessarily deep recursion chains on non-uniform distributions. We overcome these issues in a framework we call F5, which consists of a) vectorized small sorts that perform well on a wide range of input sizes; b) a partitioning engine that delivers bursts of keys into buckets at significantly increased speed; c) a prefix extractor and sortedness checker that allow jumping over common bits and terminating recursion early; and d) an adaptive radix selector that optimally decides the number of bits for each level of partitioning. Results show that F5 often doubles the speed of prior methods, including recent AVX-512 endeavors from Google [13] and Intel [19], on both uniform/skewed distributions, while also remaining *in-place*.**

I. Introduction

Many sorting algorithms have been developed over the last century. Among them, distribution-based divide-and-conquer methods are responsible for some of the fastest approaches today, including MSD (most-significant digit first) radix sort [16], [23], [28] and SIMD (single-instruction multiple-data) quick sort [8], [13], [19], [37]. Their high-level operation is shown in Algorithm 1, which works by splitting incoming items (Line 2) into k buckets $B_0, \ldots, B_{k-1}$, such that keys in B_i are no larger than those in B_{i+1}, and sorting them separately by either diverting to *small sorts* (Line 6) or recursing into the same function (Line 8), depending on the value of threshold $\mathcal{T}$. Because MSD methods generally work faster than the alternatives, especially on 64-bit and longer keys, they will be our focus here.

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

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

One prominent bottleneck in Algorithm 1 is the small sort in Line 6, which in the fastest methods [16] accounts for 45% of the total cost. To address this problem, our **first contribution** is a family of novel SIMD *micro-sorts* that handle diversion for buckets that can be held entirely in CPU vector registers. This is the first streamlined solution to sorting of non-square SIMD matrices, including cases with an odd number of rows, that achieves top performance across a full range of m. On 64-bit keys and Intel Skylake-X, results show that our micro-sorters are 5× faster than [13], [16], [23] and more than triple the speed of the corresponding components in [3], [19].

For out-of-register scenarios, our **second contribution** is a suite of SIMD *mini-sorts* that use a novel vectorized binary merge starting with a micro-sort of size m' and producing sequences of length $2m', 4m', \ldots, m$. The main caveat of this approach lies in rearranging the vectorized merge network to reduce register spill (i.e., evictions to the cache). On Skylake-X and 64-bit keys, our mini-sorts are faster than prior MSD approaches [23] by an order of magnitude and existing SIMD methods [3], [8], [13], [19] by 2-8×. By raising threshold $\mathcal{T}$, we also eliminate the massive performance drop common to MSD methods [16], [23] under adversarial workloads.

The next performance issue in Algorithm 1 comes from the MSD partitioning engine in Line 2, which works well on uniform keys, but chokes when batches of adjacent keys are sent into the same bucket [16], [23]. To address this problem, our **third contribution** proposes a new method for updating bucket pointers that avoids store-to-load forwarding stalls and delivers long bursts of items into the same bucket at a substantially improved speed. Results show that the new engine achieves a 10% speed-up on uniform keys and up to a 9× boost on skewed distributions.

The last limitation of Algorithm 1 is the lack of a rigorous foundation for selecting the number of buckets k in Line 8, as well as inability to avoid maximum recursion depth in certain cases of interest. To this end, our **final contribution** is a set of low-overhead techniques that a) use dynamic programming to adaptively calculate k at each level of recursion to ensure that small sorts run with an optimal size m; b) detect already sorted buckets and terminate recursion early; and c) compute the number of common leading bits among the keys in each bucket and skip over them during the next level. This leads to further speed increase (i.e., up to 2.5×) under skewed distributions, but without compromising performance on uniform keys.

We put these ideas into an in-place MSD radix sort we call F5. It outperforms the best prior approaches by 70-180% on uniform keys and 60-230% on non-uniform, including multiple hardware platforms and SIMD instruction sets.

II. Diversion

A. Background

Traditional MSD methods divert using a variety of general-purpose algorithms – insertion sort [2], [14], [22], [24], [26],

2375-026X/26/$31.00 ©2026 IEEE
DOI 10.1109/ICDE65706.2026.00195

[27], [30], quicksort [34], shell sort [24], [27], merge sort [12], block quicksort [23], pdqsort [29], and American Flag [5], [28], [34]. In more optimized approaches, such as Raduls2 [23] and Vortex [16], each sort size $m \geq 2$ is handled by a dedicated scalar sorting network [6]. Because m varies unpredictably between buckets, these methods have to maintain $\mathcal{T} - 1$ function pointers in some array f and invoke each small sort through a call to $f[m]$. By unrolling the entire branchless sequence of comparisons within each function, this solution allows the CPU pipeline to maximally reorder instructions and execute all independent swaps in parallel.

While sorting networks are significantly faster than the classical alternatives, their performance in MSD frameworks still leaves much to be desired. For example, on a Skylake-X CPU, Vortex's $f[8]$ needs 3c/key (cycles/key), which is 8.5× faster than insertion sort; however, calls to $f[m]$, where m is a random variable with an expected value of 8, runs three times slower (i.e., 9c/key) due to the high penalty for mispredicted indirect branches. Overcoming this problem requires increasing m to such values that make the relative cost of branch misprediction negligible compared to the amount of work done by $f[m]$. Since doing so with scalar approaches is prohibitively expensive, we next investigate how to leverage SIMD (vectorized) instructions to build faster networks.

B. Vectorizing Sorting Networks

For a given array $(a_0, \ldots, a_{m-1})$, define a $swap(u, v)$ to be an elementary operation that exchanges the values of a_u and a_v if $a_u > a_v$ and leaves them untouched otherwise. A common implementation of a swap consists of two branchless instructions $t = \min(a_u, a_v); a_v = \max(a_u, a_v); a_u = t$. Then, a *scalar sorting network* Φ_m is a sequence of swap index pairs (u, v), where $u < v$ and $0 \leq u, v \leq m - 1$, such that all input arrays of size m are sorted after application of the corresponding swaps. Note that adjacent swaps that can execute in parallel (i.e., do not share a common index) comprise a *layer*. For example, the *odd-even* network $\{(02)(13) \mid (01)(23) \mid (12)\}$ sorts $m = 4$ elements in 5 swaps and the *bitonic* network $\{(01)(23) \mid (03)(12) \mid (01)(23)\}$ does the same in 6, both using 3 layers separated by $|$.

Grouping independent comparators in Φ_m and performing multiple swaps in one SIMD instruction gives rise to *vectorized* sorting networks, which are currently the most promising approach to speeding up small sorters. Assume an SIMD architecture with vector registers capable of holding w items each. After loading the input array into $r = \lceil m/w \rceil$ CPU registers and padding the last $m \pmod{w}$ items with infinity, we obtain an $r \times w$ matrix, where each row corresponds to a register. Most SIMD operations run concurrently along the columns of the matrix (e.g., blend, min/max, compare), while a few specialized instructions can also move data *between* columns (e.g., shuffle, unpack, permute).

While there are several reasonably efficient options for scalar networks Φ_{rw} that can produce sorts of arbitrary size (e.g., odd-even & bitonic [6]), converting them to SIMD, while optimizing the vectorized swap count and the shuffle/blend overhead, faces many challenges – predicting how SIMD transformations affect future states of the matrix, working around the various constraints baked into each instruction, deciding among the large number of possible matrix operations at each step, and dealing with the non-greedy nature of the problem. Given the large depth of the decision tree (e.g., 191 swaps in the odd-even sort32 [6]), it is currently unknown how to optimally solve this task, which leads to a variety of heuristics in the field, often with vastly reduced performance.

In particular, traditional SIMD approaches [11], [18], [32], [40] focus on merge sort and do not attempt to handle the entire matrix in-register. Instead, they organize keys into a square $w \times w$ grid, sort the columns, transpose the result, and write the sorted rows to RAM to be later merged using out-of-register techniques. Another approach [8] is to iteratively decide the swaps as the program progresses through the sort, which stifles the CPU's ability to unroll the pipeline, predict branches, and achieve high levels of instruction-level parallelism. More recent developments operate by padding the matrix with dummy data to make it into some predetermined shape, e.g., [3], [19] round r to the nearest power of 2, which essentially cuts speed in half for certain sizes m, and [13], [37] additionally pad the matrix to 16 rows, dramatically reducing performance for $m \ll 16w$. Furthermore, even for the values of m that require no padding, these approaches execute a large number of unnecessary operations. Finally, there are also proposals [9], [39] that take advantage of specialized AVX-512 instructions to decompose the sort into a sequence of row-major sorts, but this inherently leads to low efficiency since SIMD is not well-suited for horizontal operations.

III. Micro-sorts (In-Register)

Proofs omitted from the paper and further discussion can be found in the technical report [4].

A. Matrix Binary Merge

Since Batcher's sorting networks [6], which we use as a building block for our SIMD approach, are constructed using repeated binary merges, our first task is to perform an efficient merge between two sorted sequences (A, B), each given as a matrix with $k \geq 1$ columns and $r \geq 1$ rows. We arrange input elements into an order we call *k-zigzag* that constructs pairs of rows by alternating between items at even and odd indexes within each sequence. For vector A, this is illustrated in Figure 1(a) using $k = 4$ and $r = 5$, where the first row contains $A[0], A[2], A[4], A[6]$. When r is odd, this pattern breaks in the last row, which we initialize to contain all remaining values in increasing order. For two sorted sequences (A, B) of length rk each, we build their *joint matrix* by concatenating the rows of individual zigzag matrices, as illustrated in part (b) of the figure, where indexes $(0, \ldots, rk - 1)$ correspond to items in A and $(rk, \ldots, 2rk - 1)$ to those in B. Merging the two sequences is now equivalent to sorting the joint matrix.

Next, we define two elementary operations on joint matrices. The first transformation begins by reversing each odd row as shown in Figure 1(c). When r is odd, the last row $r - 1$ needs

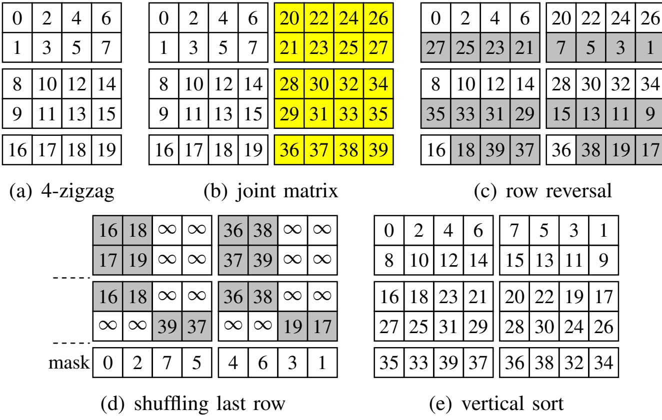


Fig. 1. Usage of zigzag matrices ($r = 5, k = 4$).

a custom permutation that is determined using the steps in Figure 1(d). On top of the figure, we create two virtual rows that contain the k largest indexes of each sequence in zigzag order, padded with infinity. After reversing the bottom virtual row, we have the correct column order for each of the $2k$ finite items (i.e., 16, 18, 39, 37, 36, 38, 19, 17). This yields the shuffle mask at the bottom of Figure 1(d) that needs to be applied to the bottom row of subfigure (b) to produce the result in (c). For SIMD to support row reversal, it is sufficient to assume existence of a macro SHUF(i, mask) that rearranges the items in row i in the order given by the shuffle mask.

The second operation is a *vertical sort*, which orders the items in each column in ascending order, leading to the result in Figure 1(e). This is accomplished using an SIMD macro SWAP(i, j) that performs a column-wise comparison between rows i and j, moving the smaller values to the former and the larger ones to the latter.

Theorem 1. *Application of row reversal and vertical sort to an $r \times k$ joint matrix produces the following: a) the largest item in each row i is no bigger than the smallest in row $i+1$; and b) items in each row, rearranged in round-robin order $(0, k-1, 1, k-2, \ldots)$, form a bitonic sequence.*

This result shows that a merge of (A, B) can be completed using a three-step process: i) odd-row reversal; ii) vertical sort along each column; and iii) horizontal sort of each register. The cost of performing i) is simply $\lceil r/2 \rceil$ calls to SHUF, but the other two steps need more analysis. A full vertical sort in ii) is expensive; instead, we note that each column interleaves items from two sorted sequences. As an example, column 0 of Figure 1(c) contains (0, 8, 16) from A and (27, 35) from B. Since each of A, B is already sorted, a 3×2 merge would be sufficient. This requires only 5 swaps [6], while a full sort on 5 elements would need 9.

For even r, the merge pattern is the same across all columns, which makes it ideally suited for SIMD; however, odd r leads to complications since some columns require an $\lceil r/2 \rceil \times \lfloor r/2 \rfloor$ merge, while others need the opposite, i.e., $\lfloor r/2 \rfloor \times \lceil r/2 \rceil$. An example of the latter can be seen in column 2 of Figure 1(c),

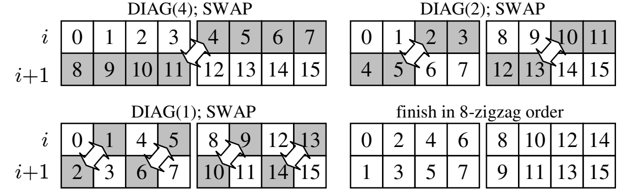


Fig. 2. Horizontal bitonic sort ($k = 4$).

where the two sorted sequences are (4, 12) and (23, 31, 39). We next show how to deal with this situation.

Theorem 2. *Suppose a column of a row-reversed joint matrix is partitioned into $r_1 = \lceil r/2 \rceil$ items from the even rows and $r_2 = \lfloor r/2 \rfloor$ items from the odd rows. Then, applying Batcher's odd-even $r_1 \times r_2$ merge to these two sequences sorts the entire column for any r.*

After a vertical column merge, we now deal with step iii) that applies a third operation – *horizontal bitonic sort* – to each row, after which the matrix will contain a sorted sequence of length $2rk$ that represents a merge of the two original vectors. Recall that a sequence is said to be *bitonic* [6] if it is first monotonically non-decreasing and then monotonically non-increasing, or can be rotated to become such. For a vector $c = (c_0, \ldots, c_{2k-1})$, a *half-cleaner* on c is a series of scalar swaps $(i, i+k)$ for $i = 0, 1, \ldots, k-1$ [6]. After running a half-cleaner on a bitonic sequence c: 1) each of its halves $(c_0, \ldots, c_{k-1})$ and $(c_k, \ldots, c_{2k-1})$ is bitonic; and 2) items in the first half are no larger than those in the second, i.e., $\max(c_0, \ldots, c_{k-1}) \leq \min(c_k, \ldots, c_{2k-1})$ [6]. Since $2k$ represents the width of the joint matrix, which is the result of multiple prior binary merges that start with 1-zigzag order, $2k$ has to be a power of 2, i.e., $2k = 2^p$. Therefore, recursive application of half-cleaners to each smaller subsequence leads to a sorted result after p layers of comparators. Note that the j-th layer, where $j \in [1, p]$, runs its k comparators (u_i, v_i) at a fixed distance $|v_i - u_i| = 2k/2^j$. For $2k = 8$ in Figure 1, this yields 12 scalar swaps in 3 layers – {(04)(15)(26)(37) | (02)(13)(46)(57) | (01)(23)(45)(67)}.

When each row of length $2k$ is bitonic, we can execute horizontal sorts on pairs of adjacent rows using a simple technique. In particular, define DIAG(i, d), where $2k/d$ is even, to be an SIMD macro that performs a *diagonal exchange* between the t-th contiguous block of size d in row i and the $(t-1)$-st block in row $i+1$, for $t = 1, 3, ..., 2k/d - 1$. This is illustrated in Figure 2, where a series of calls to {DIAG(i, d_j); SWAP(i, i+1)]} for each layer $j \in [1, p]$ and distance $d_j = 2k/2^j$, not only horizontally sorts both rows i and $i+1$, but also puts them in $(2k)$-zigzag order.

IV. Mini-sorts (Out-of-Register)

A. Slide Merge

We now deal with sort sizes m big enough to not fit into SIMD registers, but still below threshold $\mathcal{T}$ in Algorithm 1. We build these *mini-sorts* using out-of-register binary merges

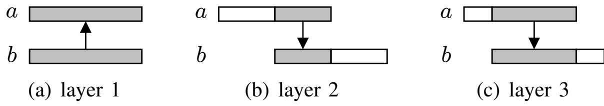


Fig. 3. Odd-even slide merge.

from SIMD networks of the previous section. Because odd-even networks are shorter than bitonic [6], we focus on them here. Given two sorted scalar sequences $(a_0, \ldots, a_{t-1})$ and $(b_1, \ldots, b_{q-1})$, the classical (recursive) definition of an odd-even merge separates them into odd-index and even-index elements to create four subsequences, merging each pair separately [6]. However, if the data is manipulated using SIMD registers, this fragmentation and later reassembly are expensive propositions. Instead, it would be beneficial to keep the order of adjacent elements in each of a and b unchanged to allow SIMD operations on contiguous chunks of each vector.

To accomplish this, we next describe a generator that produces SIMD-friendly scalar $t \times q$ merge networks and later extend it to leverage vectorized instructions. The generator works on *support* sequences – a of length t and b of size q – both initially filled with zeros. Because the odd-even merge algorithm [6] requires both sequence lengths to be a power of 2, define $k = 2^p$, where $p = \lceil \log_2(\max(t, q)) \rceil$, to be the smallest such value. After padding (a, b) with ∞ to length k, we call a swap *real* if both elements being compared are finite and *dummy* otherwise. Real swaps are included in the final merge kernel, while dummy ones are needed only to ensure correct alignment for later swaps.

With this setup, the first layer of the merge runs all pairs of swaps (a_u, b_u) for $u = 1, \ldots, k$. Out of these, only $\min(t, q)$ are real and the rest are dummy. From this point, the *direction* of the swaps (i.e., the side receiving the minimum) is reversed and layer $j \geq 2$ executes comparators (b_u, a_{u+d_j}) for $u = 0, \ldots, k-1-d_j$, where $d_j = k/2^{j-1}$. This finishes after d_j reaches 1, in total performing $1 + \log_2 k$ layers of swaps. To visualize this process, it is convenient to position a on top of b, as shown in Figure 3(a), and use an arrow to indicate swap direction. For the second layer in Figure 3(b), the bottom array slides to the right by half the sequence length (i.e., $k/2$), the direction is reversed, and only swaps that line up vertically in the shaded area of each sequence are performed. In the next layer, vector b moves left by $k/4$ and the process executes $k/2 + k/4$ vertical swaps, as illustrated in Figure 3(c). Subsequent layers continue sliding b left in exponentially decreasing steps until reaching 1. Equivalency of this method to traditional odd-even merging follows from [17], [25].

To vectorize this approach, which we call *F5-mini*, let A be a pointer of SIMD datatype initialized to read support sequence a. Pointer B is defined similarly for array b. For now, we design an SIMD merge network using an unlimited number of virtual registers, represented by A_i and B_j, and convert this program to use real registers later in the section. When SIMD width w is a power of 2, which it usually is, the number of registers in each of (A, B), i.e., $r = k/w$, is also a power of 2. This allows the first $1 + \log_2 r$ out of $1 + \log_2 k$ layers of the scalar network sketched in Figure 3 to proceed without shuffles. We call this phase of the merge *clean* and show its operation in Lines 2-7 of Algorithm 2. After setting up the (A, B) pointers and converting item count k to the number of SIMD registers (Line 2), the method performs all upward swaps in one pass over both vectors (Lines 2-4), where macro SWAP(i, j, UP) detects if the requested swap between A_i and B_j is real (i.e., there exists a column in which both registers contain a finite value) and, if so, records it into the SIMD merge network. The generator also executes a vectorized swap between support registers A_i and B_j to ensure infinity is properly tracked. This is followed by $\log_2 r$ downward layers of swaps in Lines 5-7.

Algorithm 2: F5-mini generator for merging SIMD sequences

```
1  Function Merge(a, b, k)
2    A = a; B = b; r = k/w;    ▷ r = number of SIMD registers
3    for (i = 0; i < r; i++) do    ▷ first layer
4      SWAP(i, i, UP)
5    for (d = r / 2; d ≥ 1; d /= 2) do    ▷ iterate over slide distance
6      for (i = 0; i < r – d; i++) do    ▷ run all swaps in this layer
7        SWAP(i + d, i, DOWN)
8    B[r] = B[r – 1]    ▷ create an aux register
9    for (i = r – 1; i ≥ 0; i––) do    ▷ first dirty layer, go backwards
10     B[i] = SLIDE(max(i – 1, 0), i, w / 2);
11     SWAP(i, i, DOWN);
12   B[r] = SLIDE(r, r, w / 2);
13   for (s = w / 4; s ≥ 1; s /= 2) do    ▷ all remaining dirty layers
14     for (i = 0; i < r; i++) do    ▷ go forward
15       B[i] = SLIDE(i, i + 1, s);
16       SWAP(i, i, DOWN);
17     B[r] = SLIDE(r, r, s);
18   for (i = 0; i < r; i––) do    ▷ final alignment and transpose
19     B[i] = SLIDE(i, i + 1, 1);
20     Transpose2(A[i], B[i]);    ▷ unpack to row-major order
```

In the second phase, which we call *dirty*, we now have to slide the bottom scalar sequence in Figure 3 by progressively smaller fractions of SIMD width, starting from $w/2$ and exponentially decreasing to 1. For this operation, define macro SLIDE(i, j, s) to combine the lower $w - s$ items of B_i with the upper s items of B_j. Because B_0 will be partially occupied throughout the dirty phase, Line 8 creates an extra register B_r to hold a copy of B_{r-1} and perform left rotations on it. After this, the loop in Lines 9-11 executes the first layer of dirty swaps, going backwards since each register i is assembled from B_{i-1} and B_i.

The last register, originally saved in B_r, needs to be shifted left during each layer to be consistent with the rest of the sequence. This is done in Lines 12 and 17. For the remaining dirty layers, where B_i is assembled from itself and B_{i+1}, it is more convenient to run through the vector in the forward direction, which we do in Line 14. After this is done, Lines 18-20 shift each element in B left by one item and transpose pairs (A_i, B_i) to row-major order.

B. Spill Optimization

Now we face the task to optimally allocate physical CPU registers to support the constructed SIMD merge network.

Recalling that the SIMD merge uses $2r$ virtual registers across sequences (A, B), suppose the CPU contains only R physical registers, where $R < 2r$. Then, given an SIMD program that operates on $2r$ virtual registers, the goal is to 1) reorder SIMD instructions without violating their dependencies; and 2) insert loads/stores that bring registers from/to memory such that the number of *evictions* (i.e., temporary storage into the cache) is minimized. In compiler literature [10], this is known as *spill minimization*. Since the optimal solution is NP-complete, even in much simpler cases than ours [1], various heuristics are employed by compilers and offline optimizers. They have shown promise when tackling small regions of code or using a custom solution designed for a specific type of computation; however, given the large dependency graph in our problem (e.g., a 512×512 merge contains 1153 SSE swaps), ability of existing methods to reorder long dependency chains of merge networks remains limited.

Instead, we design a new low-overhead approximation that leverages the structure of the slide merge. Define *chain* C_i for a pair (A_i, B_i) to contain all instructions on which these two registers depend. Further suppose set S_i includes the virtual registers used by the instructions in C_i. Algorithm 2 guarantees that once chain C_i finishes, registers (A_i, B_i) are no longer needed in other chains and can be permanently offloaded to output. Assuming ρ is a permutation of $(0, \ldots, r-1)$ that specifies the order in which the chains are executed by the CPU, the i-th chain $\rho(i)$ produces peak register demand $R_i = \left|\bigcup_{j=0}^{i} S_{\rho(j)}\right| - 2j$, where $2j$ accounts for finalized pairs sent to output. If chains are short enough and/or permuted in a way that makes adjacent sets $S_{\rho(i)}$ and $S_{\rho(i+1)}$ overlap in register usage, it may be possible that the entire merge can complete without any evictions, i.e., $\max_i R_i \leq R$.

While it is inefficient to test all $r!$ permutations ρ, we use the following approach inspired by Algorithm 2. In the dirty phase, SLIDE(i, i+1) in Lines 15 & 19 constructs B_i from $\log_2 w$ rotated versions of B_{i+1}. If these rotations were performed immediately prior to doing chain i, they can be kept in physical registers and reused with zero memory traffic. This suggests that going through the chains in *reverse order* is a sound strategy. In addition, because adjacent chains C_i, C_{i+1} generally share a large number of common clean swaps, the reverse permutation naturally helps maximize register recycling between them.

The final component of the algorithm handles register spill by reading sequentially the output of the reordering module described above and mapping virtual registers to physical. When the latter are exhausted, it decides to evict the virtual register that will not be used the longest in the future, which is trivially optimal [7].

V. Partitioning

A. Load-Store Dependencies

We now deal with the partition function in Algorithm 1, which splits input items into $k = 2^b$ buckets using the next b bits of each key. When input is less than half the cache size, state-of-the-art splitters [16], [23] output directly to the destination pointers; otherwise, they write into k temporary buckets, each allocated to hold up to B bytes, where kB fits in the cache, and then offload overflowing buckets into destination arrays using non-temporal (streaming) stores. Pumping data through an intermediate buffer that groups items destined to the same bucket is known as *software write-combine* (WC) [30], [32], [33]. Its dual purpose is to a) avoid a storm of TLB misses when 2^b exceeds the capacity of the TLB; and b) bypass read-for-ownership on the destination cache lines, which are overwritten in their entirety by the partitioning engine. For convenience, we assume B is a power of 2 and each tmp bucket is aligned to B bytes.

Algorithm 3: Fastest existing partitioning engine [16]

```
1  bp = WC ? tmpBuckets : dest;
2  Function Partition-v1(input, n)
3      for (i = 0; i < n; i++) do
4          prefetch(input + i + D);
5          key = input[i]; idx = (key ≫ shift) & (k – 1);
6          p = bp[idx];
7          MOVE(key, p, idx);
8  ________________________________
9  Macro MOVE(key, p, idx)
10     *p++ = key;    ▷ store key & increment pointer
11     if (p & (B – 1) == 0) then    ▷ bucket overflow?
12         p –= B / keySize;    ▷ roll back to start of tmp bucket
13         OFFLOAD(p, dest[idx]);    ▷ copy tmp bucket to RAM
14     bp[idx] = p;    ▷ update bucket pointer
```

Algorithm 3 shows a high-level operation of the partitioning engine from Vortex [16], covering both in-cache and WC cases. In Line 1, it sets up an array of bucket pointers bp to refer to tmp buckets or final destinations depending on whether WC is used. The main loop starts in Line 4 with a hint to the CPU to prefetch the input at some distance D, followed by loading of the next key and extraction of its bucket index. Line 6 obtains the target address from bp[idx], which is given to the MOVE macro that saves the key into p and increments the pointer in Line 10. The next step is to examine if p is now past the end of the bucket. When not using WC, we set $B = \infty$ to never trigger the branch in Line 11; otherwise, an overflow causes a roll back of p to the start of the tmp bucket and an offload to a pointer dest[idx] using SIMD streaming instructions. Regardless of whether the branch is taken, the new value of p is saved back into bp in Line 14.

When adjacent items move to different buckets, the CPU reorder buffer allows the splitter to work on different keys in parallel. In particular, a fetch of bp[idx] in Line 6 can proceed before the previous store in Line 14 is finished. This increases instruction-level parallelism and minimizes pipeline stalls waiting for memory disambiguation. On the other hand, bursts of back-to-back keys that flow into the same bucket encounter *read-after-write dependencies*, where loads of bp[idx] must wait for the preceding updates to the pointer to finish. Fetching the required item from the CPU store buffer, which is known as *store-to-load forwarding*, incurs high cost (e.g., 4-5 cycles [38]) and creates a latency-bound bottleneck.

We overcome this issue by replacing some of the store-forwarded loads using conditional moves. Algorithm 4 grabs

Algorithm 4: Avoiding store-to-load forwarding stalls

```
1 Function Partition-v2(input, n)
2     for (i = 0; i < n; i += 2) do
3         prefetch(input + i + D);
4         key0 = input[i]; idx0 = (key ≫ shift) & (k – 1);
5         key1 = input[i+1]; idx1 = (key ≫ shift) & (k – 1);
6         p0 = bp[idx0]; p1 = bp[idx1];    ▷ read both pointers
7         MOVE(key0, p0, idx0);
8         p1 = (idx0 == idx1) ? p0 : p1;
9         MOVE(key1, p1, idx1);
```

two keys per iteration of the loop, decides their indexes, and simultaneously obtains both pointers in Line 6. After dispatching the first key with the MOVE macro, it decides the pointer for the second key using a branchless cmov instruction in Line 8, which moves items between two registers if the condition is true and does nothing otherwise. If the compiler refuses to issue a cmov for the ternary operator ?, other options are available (e.g., assembly). If the two buckets are the same, p1 is overwritten by p0; otherwise, it stays unchanged from the earlier load. Because there is no dependency between the store of p0 and the load of p1, this method avoids dependencies in all adjacent pairs of keys.

B. *Longer Bursts*

Algorithm 4 prevents a sharp drop in speed when adjacent keys hit the same bucket, which can occur at any level of recursion. This is a good start; however, there is an opportunity to do better for longer bursts. In particular, if we can monitor keys passing through the splitter and detect long runs destined into the same bucket, it will be possible to switch to SIMD instructions that load multiple items, perform vectorized shifts/masking, and store entire registers into one pointer. This avoids unnecessary reloading of p and all associated loop-carried dependencies. The main constraint of this solution is that it must not cause a noticeable overhead for uniform keys or be susceptible to thrashing (i.e., oscillations between scalar and SIMD regimes of operation).

This gives rise to Algorithm 5. It works by processing input in *chunks*, which are units of data small enough to allow quick detection of bursts, but also large enough to prevent frequent loop interruptions. Line 3 examines the first key of the chunk and records its current bucket pointer p. After running Partition-v2 on the chunk, Line 5 checks if keys went into the same bucket. If so, the algorithm switches to a streaming engine that facilitates quicker delivery of keys into this bucket. Using prefix v_ for SIMD variables, it begins by reading w items from input (Line 10) and computing the corresponding buckets using a vectorized shift-right followed by a bitwise *and* (Line 11). To verify that all keys are going to the same destination, Line 12 broadcasts the index of the first key to all columns of an SIMD register, which is then used by Line 13 to compare against the w original indexes and break out of the loop if at least one bucket is different (Lines 14-15).

At this point, all w items in v_keys are going to the same address; however, this shared bucket may be different from the one used in the previous iteration of the loop. Thus, Line 16

Algorithm 5: Handling bursty input

```
1 Function Partition-v3(input, n)
2     for (i = 0; i < n; ) do
3         idx = (input[i] ≫ shift) & (k – 1); p = bp[idx];
4         Partition-v2(input + i, chunk); i += chunk;
5         if (bp[idx] – p == chunk) then    ▷ all keys in one bucket?
6             i = StreamEngine(input, i, n, p, idx);
7
8 Function StreamEngine(input, i, n, p, idx)
9     while i < n do
10        v_keys = LOAD(input + i);
11        v_idx = AND(SHR(v_keys, shift), k – 1);    ▷ bucket IDs
12        v_first = BROADCAST(v_idx);    ▷ replicate first element
13        v_diff = SUB(v_first, v_idx);    ▷ subtract two vectors
14        if (TEST(v_diff, v_diff)) then    ▷ test register for zero
15            break;    ▷ one of the buckets is different
16        next = EXTRACT(v_idx, 0);    ▷ extract first element
17        if (next ≠ idx) then
18            bp[idx] = p; idx = next; p = bp[idx];
19        STORE(p, v_keys); p += w; i += w;
20    p[idx] = p;
21    return i;
```

extracts one of the bucket IDs to a scalar register and switches destination pointers in Line 18 if that index no longer equals the previous value. At this stage, Line 19 can finally write the items and update the corresponding pointers. When using WC, there are additional caveats. Regular stores in Line 19 should be replaced by streaming instructions that bypass the cache and go directly to RAM, for which the destination pointer p should be initialized to dest[idx] and aligned to SIMD-width w. Alignment can be achieved before the main loop begins by moving a sufficient number of keys into this bucket using a scalar splitter. We omit these details for brevity.

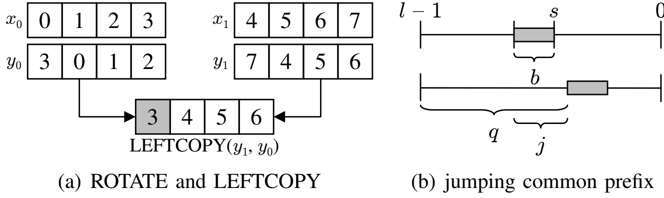
Normally, MSD/LSD radix sorts run out-of-place and require a histogram pass over the keys. Both are significant drawbacks, which we avoid by employing the infinite-bucket abstraction from Vortex [16]. It reserves enough memory for each stream, catches page faults as items are written into buckets, and transparently remaps freed pages from input to output. Because Vortex streams do not support concurrent producers/consumers on each bucket, which is a non-trivial problem to overcome, we currently run F5 single-threaded.

VI. RECURSION DEPTH

A. *Sortedness Check*

We now aim to reduce recursion depth in Algorithm 1 on non-uniform inputs. Our main constraint on these methods is that they should not deteriorate speed when dealing with uniform keys. Thus, for example, computing a histogram over each bucket, or similar methods that carry non-trivial overhead regardless of key distribution, would be out of the question.

Our first observation in Algorithm 1 is that any bucket B_i that is already sorted can be directly appended to output, bypassing small sorters and additional recursion. To detect such cases, assume an input sequence $a = (a_0, \ldots, a_{n-1})$ is scanned using SIMD registers of width w, where the i-th register is given by $x_i = (a_{iw}, \ldots, a_{iw+w-1})$. Define macro ROTATE in application to SIMD vectors to perform a circular shift to the right by one element, i.e., x_i becomes



(a) ROTATE and LEFTCOPY (b) jumping common prefix

Fig. 4. SIMD detection of sortedness and common prefix.

Algorithm 6: Sortedness checker

```
1  Function SortedCheck(input, n)
2    v_prev = LOAD(input);
3    for (i = 0; i < n; i += w) do
4      v_keys = LOAD(input + i);   ▷ (4567)
5      v_tmp = ROTATE(v_keys);   ▷ (7456)
6      v_rotated = LEFTCOPY(v_tmp, v_prev);   ▷ (3456)
7      v_mask = CMP(v_rotated, v_keys);   ▷ (34)(45)(56)(67)
8      if (TEST(v_mask, v_mask)) then   ▷ order violation?
9        return false;
10     v_prev = v_tmp;   ▷ (7456)
11   return true;
```

$y_i = (a_{iw+w-1}, a_{iw}, \ldots, a_{iw+w-2})$, which is accomplished using an in-register shuffle. This is illustrated on top of Figure 4(a). Next, let a LEFTCOPY between two vectors (y_i, y_{i-1}) be an operation that replicates (i.e., blends) the left-most element from the latter into the former, as shown at the bottom of Figure 4(a). Then, blending y_{i-1} into y_i and comparing the result against x_i detects order violations between (x_j, x_{j+1}) for $j \in [iw-1, iw+w-2]$, i.e., across the boundary between x_{i-1} and x_i, as well as between all $w-1$ adjacent pairs in x_i. Algorithm 6 implements this idea, where variable v_prev keeps track of y_{i-1}, v_tmp corresponds to y_i, and CMP in Line 7 performs a column-wise comparison for greater-than, setting individual columns to 0xFF when there is an order violation and 0 otherwise.

B. Prefix Computation

Assume s is the shift variable in Algorithm 5, $k = 2^b$ is the number of destination buckets at this level of recursion, and l is the key length in bits. Normally, the splitter decides the bucket index by extracting from each key a group of b adjacent bits in the range $(s+b, s]$, where bits are numbered from the least-significant (bit 0) to the most-significant (bit $l-1$), as illustrated on top of Figure 4(b). If there exists a $j \geq 1$ such that all input keys have identical bit values in the range $(s+b, s+b-j]$, our second observation is that the splitter can jump over these bits and avoid distributing keys into fewer than 2^b buckets and thus prolonging recursion. We therefore augment Algorithm 6 with additional SIMD instructions that compute a bitwise XOR between x_i and x_0 to clear out the common bits, followed by a bitwise OR of the result with an accumulator register. After the scan is over, the w items remaining in the accumulator are OR'ed with each other and a count of leading zero bits q is produced using lzcnt, as shown at the bottom of Figure 4(b). Then, $j = q - (l - (s+b))$ is the number of bits that can be skipped, which is done by subtracting $\max(j, 0)$ from s.

While the baseline algorithm for bit-jumping is a good start, it does not have a mechanism to stop the scan as soon as it becomes clear that j is not sufficiently large to justify further checking, which is an important feature that prevents this optimization from impacting sort speed on inputs that do not allow jumps, such as uniform keys. Therefore, we examine the result of each OR operation to verify that it is consistent with j being at least as large as some threshold. This entails two instructions (i.e., a bitwise AND, followed by a TEST) per SIMD register. Putting the two scan algorithms together, we break out of the loop as soon as the input is detected as not sorted *and* jump j is found to be below the threshold.

C. Adaptive Radix

To further guard our framework against non-uniformity, the last observation is that split factor $k = 2^{b_i}$, also known as *radix*, can be heterogeneous between recursion levels, and even between buckets, in order to land the partitioning engine on the value of m that gives the best overall performance. There are two aspects to this optimization. The first one is avoidance of sorts with exceptionally poor performance, such as sort32 in Vortex [16], which is 17× slower than sort8. The second one is deciding a fundamental tradeoff in MSD sorting – either pursuing deeper recursion to hit faster micro-sorts or aiming for fewer levels to finish with slower mini-sorts.

Assume $(b_0, \ldots, b_{i-1})$ is a sequence of bit groups in each radix used during partitioning of a given bucket up to the current recursion depth i. Then, the objective is to dynamically select the next radix length b_i to minimize the expected remaining cost of sorting this bucket. The majority of the field [5], [12], [14], [23], [28], [29] uses a fixed $b_i = 8$ for all levels and all buckets, while Vortex [16] statically decides the sequence such that $\log_2 n - \sum_i b_i = 3$, i.e., hits $m = 8$ on average at the end of recursion, which is then applied on a per-level basis (i.e., all buckets on level i have the same b_i).

To address this problem more rigorously, define S_b to be the number of cycles/key needed by the splitter when writing into 2^b buckets and $s(m)$ to be the same taken by a small sorter of size m. Note that we set $s(m) = \infty$ for values of m larger than recursion threshold $\mathcal{T}$. Now, supposing the current bucket has size $m = 2^p$, the two options are either to perform a split on b bits, which produces 2^b buckets of average size 2^{p-b}, or run a small sort immediately. This leads to the following dynamic-programming formulation

$$\alpha[p] = \min\Big[\min_b\Big(S_b + \alpha[p-b]\Big), s(2^p)\Big], \tag{1}$$

where $p = 1, 2, ..., \log_2 n$, the range of viable choices for b is determined by the CPU (e.g., 1-10 bits), and $\alpha[p] = \infty$ for $p \leq 0$. Optimal choices of b for each p are recorded into a companion table $\beta[p]$, whose positive values specify the number of bits to use when splitting a bucket of size 2^p and zeros indicate running a small sort. Even though the β table is small (e.g., 27 entries for 1-GB inputs of 64-bit keys), it has to be precomputed ahead of time to avoid wasting CPU resources during the sort. A lookup to determine the next-best

TABLE I
PERFORMANCE OF MICRO-SORTS FOR $w = 4$

	SSE 32-bit											AVX2 64-bit											
	swaps				non-swaps			cycles/sort				swaps				non-swaps				cycles/sort			
m	[16]	[3]	[13]	F5	[3]	[13]	F5	[16]	[3]	[13]	F5	[3]	[13]	[19]	F5	[3]	[13]	[19]	F5	[3]	[13]	[19]	F5
4	5	5	60	3	0	0	5	8	12	98	9	3	60	3	3	5	0	6	5	24	247	18	8
8	19	19		6	0		11	24	30		10	14		9	6	34		14	10	67		53	17
12	41	20		13	54		18	59	78		19	20		25	13	40		32	18	127		123	36
16	60			17			22	87			22				17				20				41
20	97	64	108	28	80	64	29	145	127	157	37	64	108	67	28	80	64	72	28	230	449	240	67
24	127			33			33	199			42				33				30				78
28	161			44			40	256			55				44				38				90
32	191			49			44	414			61				49				40				93
36	268	158	172	64	176	160	51	532	278	237	79	158	172	172	64	192	160	160	48	533	713	602	114
40	305			70			55	562			86				70				50				120
44	347			85			62	671			104				85				58				133
48	384			91			66	746			113				91				60				142
52	423			106			73	907			129				106				68				156
56	464			114			77	996			137				114				70				161
60	506			128			84	1,062			154				128				78				178
64	543			134			88	1,184			162				134				80				188
av	246	99	128	62	115	96	47	491	183	182	76	99	128	107	62	123	96	103	44	346	530	381	101

b_i requires converting m to the nearest power of 2 using lzcnt, followed by a load from L1 cache, resulting in ∼1c per bucket.

VII. EXPERIMENTS

A. Setup and Datasets

We use an Intel i7-7820X (Skylake-X), clocked at a fixed 4.7 GHz for all instruction sets, with 32 GB of DDR4-3200 RAM. Our method F5 is compiled in Visual Studio (VS) 2022, while previous work is reported using the fastest outcome among Clang 19, Intel OneAPI C++ 2025 (ICX), and VS. Benchmarks run on Windows Server 2016 and Ubuntu 24.04, depending on the requirements of each method, using a single thread.

We use seven synthetic datasets – $\mathcal{D}_1$ contains uniform Mersenne Twister integers; $\mathcal{D}_2$ is obtained by sorting $\mathcal{D}_1$ and then overwriting every 7-th key with INT_MAX; $\mathcal{D}_3$ consists of uniform numbers with all bits cleared to zero except the leading and trailing 7 bits; $\mathcal{D}_4$ is constructed by taking uniform numbers and replicating each of them k times, where k is a random variable drawn from the Zipf distribution $1 - (1 + x/\beta)^{-\alpha}$, where $\alpha = 1$ and $\beta = 7$, truncated at 10K, and then shuffling the result; $\mathcal{D}_5$ uses integer keys drawn from a normal distribution with mean INT_MAX/2 and standard deviation one-third of that; $\mathcal{D}_6$ packs floats uniformly distributed between 0 and FLT_MAX; and $\mathcal{D}_7$ is an adversarial sequence for Vortex [16] that contains $n/(\mathcal{R} + d)$ groups of almost-identical keys, where d is the maximum depth of recursion, crafted in a way that forces MSD methods to take small buckets of size $\mathcal{R} + d$ to full depth d (see Algorithm 7).

We also run three real-world graph-mining benchmarks – $\mathcal{G}_1$ computes the in-degree of each 32-bit node in the IRLbot domain out-graph [21], which requires a 7-GB sort on node IDs; $\mathcal{G}_2$ performs inversion of the same graph, which entails sorting 14 GB of 64-bit (src, dest) pairs; and $\mathcal{G}_3$ computes one iteration of PageRank over the 64-bit version of the graph, which can be solved by sorting 28 GB of key-value pairs consisting of 64-bit integer IDs and 64-bit doubles.

Algorithm 7: Dataset D_7 (adversarial for MSD)

```
1 Function Generate(input, n, R)
2     d = sizeof(KeyType);   ▷ max depth of recursion
3     for (i = 0; i < n; i += R + d) do
4         x = random(0, KEY_MAX) ;   ▷ uniform key
5         for (k = 0; k < R; k++) do
6             input[i + k] = x ˆ k;   ▷ batch of R similar keys
7         for (k = 0; k < d; k++) do
8             mask = 0xFF ≪ (8*k);   ▷ flips the k-th byte
9             input[i + R + k] = x ˆ mask;   ▷ rogue key for each level
```

B. Micro-sorts

We start by analyzing performance of our micro-sorts. Recall that their size m is limited to Rw, where R is the total number of SIMD registers, each containing w items. Both 128-bit SSE and 256-bit AVX use $R = 16$, while AVX-512 expands this number to 32. Table I compares F5 to prior approaches in two scenarios with $w = 4$, showing the number of swaps performed by each method, non-swap SIMD matrix transformations (i.e., shuffles, permutes, blends), and the CPU cycles taken by each sort. When a method posts the same result in multiple rows due to rounding to the next power of 2, we combine the corresponding cells and show a single number that reflects its performance in this entire range.

In the first column is Vortex [16], which is currently the fastest representative of the MSD family. Its scalar sorting networks do pretty well for $m \leq 8$; however, this range of bucket sizes is unusable in practice due to the ∼40c (cycle) penalty for each mispredicted indirect branch (i.e., function-pointer call). By the time Vortex gets to $m = 32$, it needs 191 swaps and 414c per sort. In contrast, F5 can perform the same work with 4× fewer swaps, which is optimal given $w = 4$, and 6.8× less runtime (i.e., in 61c). This gap widens to 7.3× by the time we reach $m = 64$.

TABLE II
AVERAGE IMPROVEMENT FACTOR OF F5'S MICRO-SORTS COMPARED TO PRIOR METHODS

	32-bit									64-bit								(64+64)-bit					
	SSE		AVX2				AVX-512			SSE		AVX2			AVX-512			SSE		AVX2		AVX-512	
	[3]	[13]	[8]	[3]	[13]	[19]	[3]	[13]	[19]	[3]	[13]	[3]	[13]	[19]	[3]	[13]	[19]	[3]	[13]	[3]	[13]	[3]	[13]
swaps	1.6	2.1	1.6	1.9	2.2	1.8	2.3	2.3	1.9	1.6	2.0	1.6	2.1	1.7	1.9	2.2	1.8	1.4	2.3	1.6	2.0	1.8	2.1
non-swaps	2.4	2.0	2.6	2.7	2.7	2.4	2.6	2.0	2.4	2.1	1.5	2.8	2.2	2.3	2.4	1.8	2.5	–	–	1.8	1.3	2.5	1.5
cycles	2.4	2.4	3.1	2.7	2.5	2.1	2.4	2.3	1.9	3.9	4.3	3.4	5.2	3.8	2.8	3.0	2.6	2.2	2.9	2.0	3.6	2.5	2.6

Origami [3] in the second column does better than Vortex in terms of swaps; however, many of them are still unnecessary. For example, it issues 64/49 = 1.3× more swaps than F5 at $m = 32$ and 158/134 = 1.2× more at $m = 64$. But a bigger problem is rounding of m to the next power of 2, which results in even worse performance in the remaining rows. Depending on the random value of m at which Algorithm 1 calls small sorters, Origami may end up using 158/64 = 2.5× more swaps and 278/79 = 3.4× more cycles than F5 at $m = 36$. To provide a more-balanced assessment and condense the vast amount of numbers into a single metric, the last row of Table I shows the expected cost of each method averaged over all values of m in the table. With this metric, Origami needs 1.6× more swaps and 2.4× more non-swaps & cycles than F5.

Highway [13] in the next column follows a similar trend, performing better at powers of 2 and suffering reduced performance for intermediate values of m. Because it pushes the number of rows r to 16 regardless of m and packs items into the matrix in column-major order, it needs 60 swaps from the Green's network [15] to perform a vertical column sort for all cases of $m \leq 16$. This leads to a ∼10× lower speed than F5 at $m \leq 8$. In the rest of the table, Highway does better and eventually finishes in the last row with double the number of swaps/non-swaps and 2.4× more cycles than F5.

The right half of Table I covers another case of $w = 4$ using 64-bit AVX2. While the general structure of these results is similar to SSE, the gains of F5 are more pronounced, e.g., 4.7× over Origami [3] at $m = 36$, 6.3× over Highway [13], and 5.3× over Intel-sort [19]. The longer version of the paper [4] goes into more detail on port usage and 64-bit optimizations in F5, which account for some portion of this speedup. The rest of the advantage comes from loading the CPU with less work, i.e., 1.6-2.1× fewer swaps and 2.2-2.8× fewer shuffles/blends, as shown in the last row.

In total, we examined nine combinations between SSE/AVX2/AVX-512 and item size 32-bit/64-bit/(64+64) key-value pairs. Two of these are already discussed in Table I and four additional cases are shown in [4]. The rest are summarized in Table II using the *average improvement factor*, which is the ratio between swaps, non-swaps, and cycles in the last row of each table (i.e., relative reduction in the expected cost between F5 and each method for a random m between w and Rw). In each category of Table II, we highlight the top competitor with a shaded background. These results show that F5 delivers 1.4-1.9× fewer swaps, 1.3-2.4× non-swaps, and 1.9-3.9× cycles compared to the best related work, across a broad range of instruction sets and item size.

TABLE III
OUT-OF-REGISTER $m \times m$ MERGE (SSE, 32-BIT KEYS)

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

TABLE IV
AVERAGE SPEED (M/SEC) FOR 64-BIT MICRO/MINI-SORTS

	SSE			AVX2				AVX-512			
m	[3]	[13]	F5	[3]	[13]	[19]	F5	[3]	[13]	[19]	F5
16	453	355	1521	529	267	535	1718	592	492	1133	2212
32	323	362	1318	532	272	509	1487	572	454	984	1647
64	281	111	871	388	330	390	1573	734	515	804	1607
128	196	82	643	319	168	220	1083	639	653	649	1808
256	159	68	484	214	133	189	832	515	332	533	1679
512	133	61	376	173	115	163	653	414	271	417	1055
1024	134	55	290	143	103	146	523	268	237	369	731

C. Mini-sorts

We now examine performance of mini-sorts using threshold $\mathcal{T} = 1024$ in Algorithm 1. Our first test is to understand the effectiveness of chain reorder and spill optimization in F5. To this end, we construct an SIMD merge network using Algorithm 2 and let three mainstream compilers reorder independent instructions in order to reduce register spill (which we earlier called *evictions*) and maximize temporal locality of register usage. For this benchmark, we use an $m \times m$ merge such that $2m > Rw$, i.e., the CPU cannot hold the result of the merge fully in registers, and assume the merge begins with each sorted sequence residing in the L1 cache.

Table III compares F5 to Visual Studio (VS) 2022, Clang 19, and ICX 2025. In the first row is a 64×64 merge, which contains $2R = 32$ registers. F5's chain-reordering algorithm and spill management run this case with 6 evictions, finishing 97 swaps and 82 shuffles in 106c. Since this CPU can do at most 1 swap/cycle, F5 is within 9% of the best possible outcome. On the other hand, the three compilers require 20× more evictions and ∼2.5× higher runtime. Moving on to larger merges, F5 continues to exhibit significantly better performance, both in terms of register spill and cycles/merge. In the last row, it almost triples the compiler speed and drops the eviction count by 7×.

Armed with F5's out-of-register sorts, we next compare them against in-register counterparts, examine the impact of larger SIMD width w on speed, and analyze performance of

TABLE V
SORT SPEED (M/SEC) ON ADVERSARIAL INPUT
IN MSD METHODS (AVX-512, 1 GB OF 64-BIT KEYS)

		Run-length $\mathcal{R}$ in $\mathcal{D}_7$							
Sort	$\mathcal{D}_1$	32	64	128	256	384	512	1K	2K
Ska [34]	47	47	47	28	32	33	33	35	36
Raduls2 [23]	114	68	49	45	43	22	30	33	35
Regions [28]	49	9	12	15	20	22	24	31	33
Vortex [16]	144	13	22	36	48	54	58	65	70
IPS^{2}Ra [5]	52	52	51	28	32	35	37	40	40
F5	226	220	216	211	219	231	222	128	145

TABLE VI
PARTITIONING SPEED OF F5 (M/SEC)

		32-bit			64-bit			(64+64)-bit		
		$\mathcal{D}_1$	$\mathcal{A}$	diff	$\mathcal{D}_1$	$\mathcal{A}$	diff	$\mathcal{D}_1$	$\mathcal{A}$	diff
cache	v1	1252	832	−34%	1087	800	−26%	847	665	−21%
	v2	1340	1526	14%	1129	1405	24%	914	1225	34%
	v3	1363	7433	445%	1152	3716	223%	914	2743	200%
WC	v1	1096	893	−19%	751	619	−18%	471	497	6%
	v2	1120	1284	15%	774	1165	51%	486	750	54%
	v3	1121	3071	174%	781	1555	99%	488	756	55%

existing methods. For convenience of discussion, we convert cycles into speed, average the result across all sort sizes in $(m/2, m]$, where m is a power of 2, and show the outcome in Table IV using 64-bit keys. The shaded cells in the table correspond to micro-sorts, i.e., $m \leq Rw$, and the rest are mini-sorts. Compared to prior work, F5-mini is 2.2-7.8× faster on SSE, 3.4-6.4× on AVX2, and 2-3.9× on AVX-512.

In MSD methods, mini-sorts allow earlier termination of recursion, but they also protect against drastic speed reduction on adversarial inputs (i.e., by raising threshold $\mathcal{T}$). To demonstrate this effect, Table V sweeps through a range of values $\mathcal{R}$ in dataset $\mathcal{D}_7$ of Algorithm 7, marking the lowest speed in each row with a gray background (i.e., when $\mathcal{R} = \mathcal{T}$). Among prior approaches, Vortex suffers the highest reduction in speed (i.e., 11× at $\mathcal{R} = 32$), but other MSD methods also exhibit non-negligible deterioration (e.g., Raduls2 and Regions drop by 5× at $\mathcal{R} = 384$ and $\mathcal{R} = 32$, respectively). On the other hand, F5's larger $\mathcal{T}$ and faster out-of-register sorting networks allow it to remain robust for all $\mathcal{R} < \mathcal{T}$, where its performance stays fairly constant until $\mathcal{R} = 512$. On the worst-case of $\mathcal{R} = 1024$, F5 drops 43% compared to uniform keys $\mathcal{D}_1$, but then begins a gradual recovery starting at $\mathcal{R} = 2048$.

It should be noted that MSD radix sort is not the only technique susceptible to adversarial inputs; the technical report [4] shows that SIMD quicksorts [8], [19] can also be slowed down by 2.7-4.3× using carefully crafted sequences.

D. Partitioning

We next contrast the splitter speed on two types of inputs – uniform $\mathcal{D}_1$, which allows the CPU to fetch bucket pointers bp[idx] without waiting for all preceding updates, and all-same $\mathcal{A}$, which has the opposite effect. The top half of Table VI shows the L2-cache performance of Algorithm 3 (v1), Algorithm 4 (v2), and Algorithm 5 (v3), as well as their difference in speed between the two datasets. The baseline v1 splitter from Vortex [16] runs pretty well on $\mathcal{D}_1$, but exhibits a 21-34% loss on $\mathcal{A}$. Even worse, this behavior can be triggered by input that contains relatively short runs (i.e., 5-10 keys) going into each bucket. Next, the table shows that v2 not only overcomes the latency issues of v1, but also delivers a 40-90M/sec speedup on $\mathcal{D}_1$ and 600-700M/sec on $\mathcal{A}$. This is surpassed by v3, which improves v2 in every column, achieving a massive 4.9× boost on the 32-bit $\mathcal{A}$. An important observation is that v3 does not sacrifice speed on uniform keys to realize these gains.

The bottom half of the table runs the three methods over a 1-GB input using the write-combine (WC) optimization. For all item sizes, v2 is again vastly superior to v1, which in turn is overshadowed by v3. Also note that the bottom row saturates the single-threaded memcpy speed on $\mathcal{A}$, which prevents v3 from pushing the speed any further.

TABLE VII
SPEED OF AVX-512 PREFIX/SORTEDNESS CHECK IN F5 (M/SEC)

	32-bit		64-bit		(64+64)-bit	
	L1	RAM	L1	RAM	L1	RAM
sort check	32073	4077	16339	2042	12774	1034
prefix extract	53256	4188	26479	2092	20468	1041
both methods	22196	3690	11131	1841	8947	982

E. Impact of Improvements

We analyze the sortedness checker (Algorithm 6) in the first row of Table VII. When running in L1 cache, which models buckets near the end of recursion, the method hits 32B keys/sec on 32-bit keys, with a 2× reduction for 64-bit items and 2.5× for 64+64. When operating in RAM, performance is mostly limited by the ability of memory to fetch data, which is capped at ∼16 GB/s. In the second row, the prefix extractor goes ∼65% faster in cache and 3% faster in RAM compared to the first row. When both algorithms are enabled, their in-cache rate exceeds 22B/sec on 32-bit keys and 11B/sec on 64-bit, while the out-of-cache speed stays close to RAM bandwidth.

To evaluate the various improvements in F5, Table VIII progressively adds features to the baseline Vortex framework [16] and observes the impact on speed using datasets $\mathcal{D}_1 - \mathcal{D}_7$. On uniform keys, Vortex delivers a solid 144M/sec in the first row; however, it degrades to 83M/sec on mid-zeros ($\mathcal{D}_3$), ∼60M/sec on Pareto-frequency ($\mathcal{D}_4$) and floats ($\mathcal{D}_6$), and 13M/sec on adversarial ($\mathcal{D}_7$). The last case is especially damaging since it forces Algorithm 1 to check 255 empty buckets just to find roughly 40 keys in one of the non-empty buckets, which happens at every level of recursion and for every 40 keys. As seen earlier, the main solution to these types of attacks is to increase $\mathcal{T}$ to larger values.

Towards this end, the next row adds our micro-sorts, but keeps the old radix calculation that aims to finish recursion at $m = 8$. This bumps the uniform case to 168M/sec, increases the normal distribution ($\mathcal{D}_5$) and floats ($\mathcal{D}_6$) by 30M/sec, jumps a few others by ∼10M/sec, and recovers the adversarial case in the last column to almost full speed (121M/sec). With proper calculation of radix bits at each level, the next row

TABLE VIII
SORT SPEED (M/SEC) IN F5 (AVX2, 1 GB OF 64-BIT KEYS)

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

TABLE IX
32-BIT SORT SPEED (M/SEC)

Sort	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{D}_6$	$\mathcal{G}_1$
Polychroniou [30]	34	34	31	35	32	30	–
Ska [34]	40	96	100	76	51	84	81
Regions [28]	77	58	154	88	79	96	87
Voracious [29]	79	80	123	84	81	84	93
Vortex [16]	150	122	148	130	135	147	124
IPS^{2}Ra [5]	46	107	153	107	58	101	128
Dovetail [12]	103	99	91	99	103	102	99
Reinald [31]	96	100	101	101	101	100	106
Fast-Radix [36]	69	68	84	71	70	70	86
DFR [35]	76	69	–	131	67	79	123
Blacher[8]	133	108	233	136	133	133	138
IPS^{4}o [5]	36	50	101	58	36	50	55
Highway [13]	115	128	192	132	115	115	127
Intel-sort [19]	149	158	220	80	154	153	79
Origami [3]	131	131	130	131	131	131	121
F5	306 2.0×	527 3.3×	374 1.6×	279 2.1×	299 1.9×	248 1.6×	233 1.7×

delivers a spike for uniform keys by another 25M/sec, while adding 7-13M/sec to almost-sorted ($\mathcal{D}_2$) and normal ($\mathcal{D}_5$). After enabling mini-sorts, $\mathcal{D}_4$ and $\mathcal{D}_6 - \mathcal{D}_7$ experience a 30-40M/sec jump, which is caused by terminating recursion for values of m above $Rw = 64$, but below $\mathcal{T} = 1024$. Once partition-v2 kicks in, the uniform case obtains another 13M/sec speedup, as predictably do the rest of the datasets.

With partition-v3 in the next row, the main beneficiaries are datasets with runs of keys going into the same bucket, i.e., $\mathcal{D}_2 - \mathcal{D}_4$, where the boost is another 10-20M/sec. Next, the prefix checker helps skip the 50 zero bits in the middle of each key in $\mathcal{D}_3$, doubling performance. Since all copies of each key in $\mathcal{D}_4$ eventually cluster in the same bucket, which normally ends up taking recursion to maximum depth, the prefix check helps stop those cases early, leading to a 40M/sec hike in this column. The table ends with a sortedness checker, which substantially increases performance on almost-sorted ($\mathcal{D}_2$), adding 75M/sec to that column.

F. Full Sorts

We are now ready to examine the speed of complete sorts across 8-GB datasets $\mathcal{D}_1 - \mathcal{D}_6$, as well as graph-analytics jobs $\mathcal{G}_1 - \mathcal{G}_3$. Table IX begins with 32-bit keys, where dashes represent inability to finish the sort (e.g., crashing, failing sortedness checks, unsupported input size). Prior work is organized chronologically in the order of publication and grouped

TABLE X
64-BIT SORT SPEED (M/SEC)

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

TABLE XI
(64+64)-BIT SORT SPEED (M/SEC)

Sort	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{D}_6$	$\mathcal{G}_3$
Ska [34]	31	74	39	38	28	26	35
Raduls2 [23]	45	58	37	40	59	47	–
Regions [28]	22	38	45	21	26	26	20
Voracious [29]	30	31	35	30	30	30	–
IPS^{2}Ra [5]	40	72	44	47	36	31	42
Dovetail [12]	19	20	24	23	22	20	–
IPS^{4}o [5]	33	33	51	35	33	33	35
Highway [13]	27	31	45	32	27	17	29
Origami [3]	25	24	24	25	25	26	–
F5	126 2.8×	155 2.1×	139 2.7×	104 2.2×	124 2.1×	85 1.8×	83 2.0×

into four categories – MSD, LSD, quicksort, and mergesort. On uniform keys ($\mathcal{D}_1$), F5 delivers 306M/sec single-threaded, doubling performance of the second place (i.e., Vortex with 150M/sec). AVX-512 methods Google Highway [13], Intel-sort [19], and Origami [3], as well as AVX2 Blacher [8], are not far behind, reaching 115-149M/sec. Their lack of more sophisticated detection of special cases becomes noticeable on the almost-sorted case $\mathcal{D}_2$, where F5 hits 527M/sec by stopping recursion in already-sorted buckets, while the other SIMD methods are 3-5× slower. On the mid-zero $\mathcal{D}_3$, quicksorts speed up at least 50% due to the small number of unique keys (i.e., only 2^{14} out of 2^{31}); however, prefix jumping allows F5 to speed up even more, reaching 374M/sec. The rest of the table demonstrates similar trends, with F5 posing a 1.6-2.1× improvement over the nearest competitor.

Table X shows results with 64-bit keys. While Vortex is still the fastest prior method on uniform data, it now suffers a massive performance loss on $\mathcal{D}_3 - \mathcal{D}_4$. A similar effect, which is caused by 8 levels of recursion and long runs of duplicate keys going into the same bucket, occurs on graph inversion $\mathcal{G}_2$. In contrast, F5 remains largely immune to these effects, at least doubling the Vortex speed on each of the datasets and delivering an overall improvement between 1.7× and 3.1× compared to the fastest prior work. In addition,

TABLE XII
MACHINE SPECIFICATIONS

Model	Year	Family	RAM	SIMD
Intel E5-2690	2012	Sandy Bridge (SB)	DDR3-1333	SSE
Intel E5-2680v2	2013	Ivy Bridge (IB)	DDR3-1866	SSE
Intel E5-2680v4	2016	Broadwell (BW)	DDR4-2400	AVX2
Intel i7-8700K	2017	Coffee Lake (CL)	DDR4-3200	AVX2
Intel i5-12600K	2021	Alder Lake (AL)	DDR5-6400	AVX2
AMD 7950X	2022	Raphael (Zen4)	DDR5-6400	AVX-512
AMD 9600X	2024	Granite Ridge (Zen5)	DDR5-6400	AVX-512

TABLE XIII
UNIFORM 32-BIT SORT SPEED (M/SEC)

Sort	SB	IB	BW	CL	AL	Zen4	Zen5
Polychroniou [30]	23	25	24	36	45	50	54
Ska [34]	29	29	32	48	54	71	67
Regions [28]	53	54	55	86	100	115	133
Voracious [29]	24	25	23	107	136	155	110
Vortex [16]	61	65	57	155	200	203	256
IPS^{2}Ra [5]	–	–	27	47	56	76	73
Dovetail [12]	24	25	23	126	145	193	126
Reinald [31]	25	26	24	133	157	210	140
Fast-Radix [36]	24	24	22	103	131	154	98
DFR [35]	17	18	21	74	133	114	134
Intel IPP [20]	16	16	18	89	105	133	168
Blacher [8]	–	–	85	153	211	255	299
IPS^{4}o [5]	–	–	24	37	45	48	52
Highway [13]	23	25	49	84	112	168	204
Intel-sort [19]	–	–	76	130	183	202	276
std::sort	8	8	8	10	10	13	13
Origami [3]	30	31	61	112	124	107	157
F5	114	129	151	288	390	470	597
	1.9×	2.0×	1.8×	1.9×	1.8×	1.8×	2.0×

TABLE XIV
UNIFORM 64-BIT SORT SPEED (M/SEC)

Sort	SB	IB	BW	CL	AL	Zen4	Zen5
Polychroniou [30]	12	14	11	18	23	27	32
Ska [34]	26	27	27	42	46	65	60
Raduls2 [23]	43	51	55	90	118	133	152
Regions [28]	23	23	23	35	43	48	53
Voracious [29]	15	15	13	77	93	100	79
Vortex [16]	42	45	37	139	177	190	221
IPS^{2}Ra [5]	–	–	24	42	51	72	67
Dovetail [12]	8	7	7	50	60	84	64
Reinald [31]	8	8	7	54	65	89	65
Fast-Radix [36]	8	8	7	55	66	88	62
DFR [35]	12	13	12	68	100	85	95
Intel IPP [20]	8	8	9	39	46	61	75
IPS^{4}o [5]	–	–	22	30	33	47	49
Highway [13]	11	12	20	36	47	105	133
Intel-sort [19]	–	–	29	55	64	97	139
std::sort	8	8	8	10	10	13	13
Origami [3]	17	17	22	48	41	63	71
F5	79	81	98	192	275	352	450
	1.8×	1.6×	1.8×	1.4×	1.6×	1.9×	2.0×

TABLE XV
UNIFORM (64+64)-BIT SORT SPEED (M/SEC)

Sort	SB	IB	BW	CL	AL	Zen4	Zen5
Ska [34]	23	25	25	39	49	65	64
Raduls2 [23]	25	26	33	59	71	67	93
Regions [28]	16	17	17	26	34	36	39
Voracious [29]	9	9	9	49	48	51	54
IPS^{2}Ra [5]	–	–	22	43	53	65	70
Dovetail [12]	4	4	8	30	36	50	48
IPS^{4}o [5]	–	–	25	32	36	52	57
Highway [13]	6	7	11	21	24	44	52
std::sort	8	8	8	10	10	13	13
Origami [3]	10	11	16	27	34	20	21
F5	40	41	48	116	166	223	266
	1.6×	1.6×	1.5×	2.0×	2.3×	3.3×	2.9×

SIMD methods (Highway, Intel-sort, Origami) are no longer competitive since their sorts almost entirely rely on vectorized instructions, where speed depends inverse proportionally on key size. This is not the case for F5, which runs only small sorts in SIMD and whose speed reduction between Tables IX to X is a moderate 33%.

Going to 64+64 keys-value pairs, Table XI shows another dominant performance by F5, with a 2.8× higher speed on uniform keys and 1.8-2.7× on $\mathcal{D}_2 - \mathcal{D}_6$. For the PageRank computation on $\mathcal{G}_3$, where the dataset takes up almost the entire RAM (i.e., 28 out of 32 GB), only the in-place methods are able to complete the job. This limits Table XI to five frameworks besides F5, where our method posts a 97% speedup over the second-fastest approach and 186% over the only SIMD representative remaining (i.e., Highway).

We now investigate sort performance on seven additional hardware platforms shown in Table XII. The top three rows are Xeon server CPUs running memory in a quad-channel configuration, while the rest are dual-channel desktop models. The table covers a wide range of memory speed (1333-6400 MHz), RAM type (DDR3-DDR5), and SIMD generations (from SSE to AVX-512), spanning 12 years of CPU design. We use an 8-GB uniform dataset $\mathcal{D}_1$ and show the result in Tables XIII-XV, which for completeness also include C++ std::sort and a low-level, heavily optimized library from Intel *Integrated Performance Primitives* (IPP) [20] that contains an LSD sort.

Compared to the best prior algorithms, F5 begins with a 50-100% speedup on older platforms and gradually improves towards 100-230% as the CPUs become faster and more sophisticated, eventually beating std::sort by 46× on 32-bit keys and by 35× on 64-bit using AMD Zen5. In the end, F5 takes the top spot in every comparison, all examined input distributions and SIMD instruction sets, and multiple server/desktop architectures, delivering a substantial improvement over the best existing methods.

VIII. CONCLUSION

We presented a suite of novel algorithms for sorting SIMD matrices, extended them to out-of-register operation, improved the MSD partitioning engine, created low-overhead mechanisms for stopping recursion early, and solved the problem of optimal selection of radix at each level/bucket. This resulted in significant speed gains over prior work on both uniform and skewed workloads, across multiple platforms and SIMD instruction sets. Future work involves redesigning Vortex streams to support concurrent producers/consumers on each bucket, which will open avenues towards multi-threading F5.

AI-Generated Content Acknowledgement

No AI was used.

References

[1] A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code generation for expressions with common subexpressions," *J. ACM*, vol. 24, no. 1, pp. 146–160, Jan. 1977.

[2] A. Al-Badarneh and F. El-Aker, "Efficient Adaptive In-Place Radix Sorting," *Informatica*, vol. 15, no. 3, pp. 295–302, Aug. 2004.

[3] A. Arman and D. Loguinov, "Origami: A High-Performance Mergesort Framework," in *Proc. VLDB*, Sep. 2022, pp. 259–271.

[4] A. Arman and D. Loguinov, "F5: A Robust SIMD-Accelerated MSD Radix Sort," Tech. Rep., Mar. 2026. [Online]. Available: http://irl.cs.tamu.edu/publications.

[5] 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, Mar. 2022.

[6] K. E. Batcher, "Sorting Networks and their Applications," in *Proc. Spring Joint Computer Conference*, Apr. 1968, pp. 307–314.

[7] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, Jun. 1966.

[8] 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.

[9] B. Bramas, "A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake," *arXiv preprint arXiv:1704.08579*, 2017.

[10] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *Proc. ACM SIGPLAN*, Jun. 1982, pp. 98–105.

[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] 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.

[13] Google, "Highway Sort," 2025. [Online]. Available: https://github.com/google/highway/tree/master/hwy/contrib/sort.

[14] E. Gorset, "In-place Radix Sort," Apr. 2011. [Online]. Available: https://github.com/gorset/radix.

[15] M. W. Green, "Some Improvements in Nonadaptive Sorting Algorithms," in *Proc. Princeton Conference on Information Sciences and System*, Mar. 1972, pp. 387–391.

[16] 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.

[17] Z. Hong and R. Sedgewick, "Notes on Merging Networks (Prelimiary Version)," in *Proc. ACM STOC*, May 1982, pp. 296–302.

[18] K. Hou, H. Wang, and W.-C. Feng, "ASPaS: A Framework for Automatic Simdization of Parallel Sorting on x86-based Many-core Processors," in *Proc. ACM International Conference on Supercomputing*, Nov. 2015, pp. 383–392.

[19] Intel Corporation, "x86-simd-sort," 2025. [Online]. Available: https://github.com/intel/x86-simd-sort.

[20] Intel Corporation, "Intel Integrated Performance Primitives," Jan. 2026. [Online]. Available: https://software.intel.com/en-us/intel-ipp.

[21] IRL Web Crawling Datasets. [Online]. Available: http://irl.cs.tamu.edu/projects/web/.

[22] D. Jiménez-González, J. J. Navarro, and J.-L. Larriba-Pey, "CC-radix: A Cache Conscious Sorting Based on Radix Sort," in *Proc. Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2003, pp. 101–108.

[23] 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.

[24] 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.

[25] M. Kumar and D. Hirschberg, "An Efficient Implementation of Batcher's Odd-even Merge Algorithm and its Application in Parallel Sorting Schemes," *IEEE Transactions on Computers*, vol. 100, no. 3, pp. 254–264, Mar. 1983.

[26] A. Maus, "ARL, A Faster In-place, Cache Friendly Sorting Algorithm," *Norsk Informatik Konferranse NIK*, vol. 2002, no. 85-95, p. 7, 2002.

[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. Obeya, E. Kahssay, E. Fan, and J. Shun, "Theoretically-efficient and Practical Parallel In-place Radix Sorting," in *Proc. ACM Symposium on Parallelism in Algorithms and Architectures*, Jun. 2019, pp. 213–224.

[29] A. Piot, "Voracious Sort," 2020. [Online]. Available: https://github.com/lakwet/voracious_sort.

[30] 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.

[31] A. Reinald, P. Harris, R. Rohrer, and J. Dirk. Radix sort. [Online]. Available: http://www.cubic.org/docs/download/radix_ar_2011.cpp.

[32] 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.

[33] 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.

[34] M. Skarupke, "Ska Sort," 2017. [Online]. Available: https://github.com/skarupke/ska_sort.

[35] S. Thiel, "The Diverting Fast Radix Algorithm," Ph.D. dissertation, Concordia University, 2021. [Online]. Available: https://spectrum.library.concordia.ca/id/eprint/988207/1/Thiel_PhD_S2021.pdf.

[36] 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.

[37] J. Wassenberg, M. Blacher, J. Giesen, and P. Sanders, "Vectorized and Performance-portable Quicksort," *Software: Practice and Experience*, vol. 52, no. 12, pp. 2684–2699, Dec. 2022.

[38] H. Wong, "Store-to-Load Forwarding and Memory Disambiguation in x86 Processors," 2014. [Online]. Available: https://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/.

[39] T. Xiaochen, K. Rocki, and R. Suda, "Register Level Sort Algorithm on Multi-Core SIMD Processors," in *Proc. Workshop on Irregular Applications: Architectures and Algorithms*, Nov. 2013, pp. 1–8.

[40] Z. Yin, T. Zhang, A. Müller, H. Liu, Y. Wei, B. Schmidt, and W. Liu, "Efficient Parallel Sort on AVX-512-Based Multi-Core and Many-Core Architectures," in *Proc. IEEE HPCC/SmartCity/DSS*, Aug 2019, pp. 168–176.