Origami: A High-Performance Mergesort Framework

Arif Arman
Texas A&M University
College Station, TX
arman@tamu.edu

Dmitri Loguinov
Texas A&M University
College Station, TX
dmitri@cs.tamu.edu

ABSTRACT

Mergesort is a popular algorithm for sorting real-world workloads as it is immune to data skewness, suitable for parallelization using vectorized intrinsics, and relatively simple to multi-thread. In this paper, we introduce Origami, an in-memory mergesort framework that is optimized for scalar, as well as all current SIMD (single-instruction multiple-data) CPU architectures. For each vector-extension set (e.g., SSE, AVX2, AVX-512), we present an in-register sorter for small sequences that is up to 8x faster than prior methods and a branchless streaming merger that achieves up to a 1.5x speed-up over the naive merge. In addition, we introduce a cache-residing quad-merge tree to avoid bottlenecking on memory bandwidth and a parallel partitioning scheme to maximize thread-level concurrency. We develop an end-to-end sort with these components and produce a highly utilized mergesort pipeline by reducing the synchronization overhead between threads. Single-threaded Origami performs up to 2x faster than the closest competitor and achieves a nearly perfect speed-up in multi-core environments.

PVLDB Reference Format:
doi:10.14778/3489496.3489507

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at http://irl.cs.tamu.edu/projects/streams.

1 INTRODUCTION

Over the years, mergesort has emerged as a highly appealing platform for tackling real-world sorting tasks, with many benefits and features. Its first important characteristic is distribution insensitivity, i.e., constant speed on all inputs. Among the alternatives, some of the methods (e.g., most-significant byte first (MSB) radix sort [25]) perform quite poorly unless the keys are uniform. Others (e.g., quicksort, sample sort, comb sort) have certain worst-case inputs that degrade the sort by either worsening its asymptotic complexity or inflating the constants in the $O(n \log n)$ upper bound [3, 4, 5, 22, 29, 31], neither of which is desirable. The second benefit of mergesort is the support for streaming operation, i.e., sequential processing of input/output data, which is a highly useful feature for certain large-scale applications that involve external-memory (i.e., disk) and/or distributed (i.e., network) computation. With PCIe 5.0 I/O rates reaching 64 GB/s, faster in-memory sorters will soon be in demand. Third, mergesort is well-suited for multi-core parallelization because it admits low-overhead load-balancing of tasks across threads, even under non-uniform keys, and trading of memory traffic for cache hits as the amount of parallelism increases. The alternatives are usually harder to scale without taking a performance hit. A prime example is least-significant byte first (LSB) radix sort [25], where rewriting the entire dataset in RAM during each pass causes it to saturate memory bandwidth with just 2-4 threads. Finally, research into mergesort usually yields new optimized kernels for small inputs, which helps speed up applications that deal with short chunks of data (e.g., neighbor lists in graph algorithms).

Many mergesort variants have been proposed in the last two decades with the goal to maximize data/thread-level parallelism [6, 14, 15, 17, 18, 26, 27, 30, 32, 33]; however, they leave room for improvements in terms of speed and usability. First, none of the papers examine how to optimize each individual phase of the sort pipeline. With many partial benchmarks and disjoint techniques, it is unclear which of them can be improved, by how much, and where the bottlenecks are. Additionally, the majority of available code is either single-threaded or, if parallel, bottleneck on memory bandwidth, which sheds little light on the best performance of mergesort in multi-core environments. Second, the existing frameworks do not offer a unifying mergesort solution that is simultaneously optimized for scalar, SSE, AVX2, and AVX-512 architectures. In fact, some of them [30], [32], [33] inherently work only in the extended instruction set of AVX-512, with back-porting either impossible or requiring an expensive set of substitute instructions. Depending on CPU availability and user preferences (e.g., lower power consumption), it may be desirable to have access to the fastest sort in each category rather than the fastest overall.

To address these issues, we introduce a highly optimized, distribution-insensitive, parallel mergesort framework that we call Origami. We first formalize operation of mergesort using a four-phase computational model and examine how to achieve maximum speed during each step of the sort. This leads to a number of novel algorithms, improvements, and corresponding benchmarks. We then develop our end-to-end sort by efficiently connecting these optimized components together and generalizing the underlying algorithms to work for scalar, SSE, AVX2, and AVX-512 CPU architectures. Results show that the Origami framework is by far the fastest mergesort on both small and large input sequences, reaching a 1.5-2x speed-up over the best existing methods. After parallelization, it gains a nearly perfect scaling in multi-core settings.

2 PIPELINE OVERVIEW

Suppose a sort algorithm operates on fixed-size items, which are either keys or key-value pairs, depending on the application. The following notation will be useful for the rest of the discussion:
**N**: Number of items to sort  
**C**: Number of items fitting into L2 cache (typically $2^{16} - 2^{18}$)  
**T**: Number of threads (typically double the core count)  
**W**: Number of items per SIMD register (typically 4-16)  
**R**: Number of SIMD registers per core (typically 16 or 32)  
**B**: Size of each item in bits (typically 32, 64, or 128)

Mergesort can be broken down into four phases, which we call $P_1$, $P_2$, $P_3$, and $P_4$. For better cache utilization, which in turn increases performance, merge-based sorts [6], [14], [27] usually divide the input into blocks of size $C$ and sort them individually in the cache to produce $N/C$ sorted lists. A series of $k$-way merge operations on these blocks, where $k > 2$, then yields the final sorted list. In the rest of this section, we briefly overview this pipeline, examine the operation of each phase, highlight the limitations of existing methods, and explain our contributions.

**(P₁)** Tiny sorters. Define phase $P_1$ to be a process that converts each input block into a sequence of sorted runs of length $m \geq 2$, i.e., items in each range $(i m, (i + 1) m)$ are organized in ascending order, where $i = 0, 1, \ldots, C/m - 1$. While mergesort is quite efficient for small values of $m$ (e.g., below 128), much faster alternatives, which we call tiny sorters, exist (e.g., insertion sort [24], sorting networks [2], [23], and various SIMD generalizations [6], [14]).

Letting $S_1(m)$ be the speed at which $P_1$ executes, we show that the runtime of the full sort is determined by the cost function $f(m) = S_{merge}/S_1(m) - log_2 m$, where $S_{merge}$ is the speed of the binary merge. Prior work typically operates at a fixed point $m = W$ and does not explore avenues for minimizing $f(m)$. In contrast, Origami offers novel algorithms that expand $m$ to the entire range $[W, R \times W]$ and significantly increase $S_1(m)$ compared to existing approaches, both of which leads to lower $f(m)$ and much faster overall runtime. This is achieved by saturating all $R$ available registers with data, representing them as an $R \times W$ matrix, and developing new SIMD-friendly methods for sorting such rectangular structures entirely within the CPU.

**(P₂)** In-cache merge. At the end of $P_1$, every run of $m$ keys in a block is sorted. A sequence of $log_2(C/m)$ binary merges, which comprises phase $P_2$, then sorts all $C$ items in the block. While many efforts exist for moving pointers along two sorted arrays during merging (e.g., branching [6], [14], [15], [17], [18], [26], [27]), branchless [30], partially branchless [16], SIMD-aided [33]), the issue of how to further increase performance of this step has remained open for many years. To this end, we develop a new merge technique that is not only faster than all prior solutions, but also applicable to both scalar and vectorized architectures. It relies on a novel design for advancing stream pointers using conditional move instructions and interleaved comparisons from multiple merge pairs to lower the latency caused by instruction/data dependency.

**(P₃)** Out-of-cache independent merge. Each thread begins the next phase, which we call $P_3$, with owning several sorted sequences of $C$ items each. It then independently merges them until reaching some threshold after which coordination with other threads becomes necessary. Because the data no longer fits in the cache, binary merges are not suitable for this step as they operate at speed that can easily exceed RAM bandwidth, especially across $T$ threads. Much of the prior work [14], [26], [30], [32], [33] ignores this issue and produces poor performance in this phase. The remaining efforts [15], [17] perform a $k$-way merge in $P_3$, which reduces memory traffic by a factor of $\log_2 k$, but their performance is often suboptimal. Specifically, [15] always uses $k = 4$ without regard to memory bandwidth and [17] requires an insertion sort that may result in $N^2$ complexity for non-inform keys.

In Origami, we develop a new $k$-way merge tree that relies on our algorithms in $P_2$. Unlike previous literature [6], [17], where each node performs a binary merge, our approach executes optimized $4$-way merges at each step to achieve better throughput. In contrast to expensive circular queues in [6], [18], [27], we use simpler data structures that exhibit lower management cost. And most importantly, Origami computes the optimal value of $k$ based on the memory bandwidth achievable across $T$ threads instead of hardcoding an ad-hoc constant (e.g., $k = 4$ in [15], $k = 32$ in [17]).

**(P₄)** Out-of-cache cooperative merge. In this phase, which we call $P_4$, multiple threads work together to merge the final $k$ lists. Note that some papers, e.g., [6], [18], [27], omit phase $P_3$ and directly execute $P_4$ on $k = N/C$ sorted buffers, which suffers from hefty synchronization cost. Additionally, some of the techniques [14], progressively shrink the number of working threads as the computation goes forward and others [30], [32], [33] continue running binary merges, which results in suboptimal multi-core utilization. The remaining methods [15], [17] split the merge size across threads evenly; however, they do not parallelize the partitioning step, fail to perfectly load-balance stragglers, and run into performance issues when the final number of sorted streams (i.e., $T$) is insufficient to prevent memory bottlenecks.

Origami overcomes these problems by multi-threading the array split, load-balancing across threads even when equal-size jobs consume different amounts of time (e.g., due to OS scheduling delays, difference in key distribution), and using our $k$-way merge tree from $P_3$, which keeps memory traffic just below RAM bandwidth.

### 3 TINY SORTERS ($P_1$)

#### 3.1 Principles

Traditional mergesort [8][p. 13] performs $\log_2 N$ binary merges starting from sorted runs of size 1. In practice, however, it is better to first presort the items in small groups using a different algorithm and then execute binary merges on these chunks. Over the years, sorting networks [2] have proven to be the fastest option for such tiny sorts. Recall that a sorting network is a sequence of min-max operations, each of which we call a swap.

**Definition 1 (swap).** Given two (possibly vector) registers $x$ and $y$, the swap($x, y$) macro performs the following operations

\[
\text{tmp} = \min(x, y); \quad y = \max(x, y); \quad x = \text{tmp};
\]

Normally, a sorting network would run over scalar variables (e.g., integers, doubles), but modern computers can do better. With their ubiquitous support of SIMD (single-instruction multiple-data) operations, commonly known as streaming or advanced vector extensions (i.e., SSE, AVX, AVX2, AVX-512), a single register can hold multiple scalar values. For example, AVX2 has 256-bit registers that can fit $W = 8$ integers or $W = 4$ doubles. A single vector instruction, which we indicate by prefix _mm_, can then apply a
Figure 1: Approaches to sorting short lists ($W = 4$).

particular operation (e.g., min/max) to all $W$ values at once. As a result, these architectures can perform $W$ scalar swaps with only a pair of _mm_min,_mm_max intrinsics. For a more in-depth overview, see [6], [17], [27].

For maximum speed (i.e., to avoid of branch mispredictions) both min/max functions in swap must be branchless. In scalar code, this is achieved using the cmov (conditional move) Assembly instruction or the ternary operator $? \times \leq y \times ?$ which implements the min. For SIMD, all vectorized min/max intrinsics are automatically branchless. An added benefit of sorting networks and branchless code is their insensitivity to key distribution, i.e., similar speed on all inputs, which is a desirable characteristic for real-life workloads that are frequently skewed/non-uniform.

With this in mind, we can stack multiple SIMD registers and vertically sort $W$ columns in parallel, which is a common technique we call csort.

Definition 2 (csort). Given an $r \times c$ matrix $M$, csort orders each column of $M$ using a vectorized sorting network of size $r$.

Define $m$ to be the length of sorted runs generated by $P_1$. As shown in Figure 1(a), where the shade of each cell indicates its relative numeric value (i.e., darker is larger), the best existing methods [6], [27], [14], [33] load $W^2$ items in $W$ SIMD registers and view them as a $W \times W$ matrix. Next, they sort the items independently within each column with csort, transpose the matrix, and obtain $W$ sorted lists of $W$ items each. Other approaches exist, but they are less efficient. In particular, [30] uses only four registers, [32] only two, and [15], [17] only one.

There are several limitations to the method in Figure 1(a). First, it uses just $W$ out of $R$ available registers. With $R$ either 16 (i.e., SSE) or 32 (i.e., AVX2, AVX-512), this leaves 50–94% of the registers unused depending on item size $B$. Second, this technique loads $W^2$ keys per iteration, but outputs sorted runs of dismal size $m = W$. To sort all $W^2$ keys, another $\log_2 W$ merge passes are needed in the cache. In contrast, Origami $P_1$ stuffs data into all $R$ registers, sorts the whole rectangular $R \times W$ matrix using a variety of new algorithms, including a faster transpose, and outputs runs of $m = RW$ items. This is illustrated in Figure 1(b) and detailed next.

3.2 Matrix-Column Merge

Origami begins $P_1$ by loading $RW$ input items into $R$ registers and sorting columns of the corresponding matrix using csort. The complexity of this step is given by the length of the underlying sorting network (e.g., 19 invocations of swap for $R = 8$). The next objective is to continue sorting this matrix in column-major order, by which we mean the following.

Definition 3. An $r \times c$ matrix $M$ is considered sorted in column-major order if values within each column are non-decreasing and no larger than those in the next column, i.e., $M(i, j) \leq M(i + 1, j)$ and $M(r, j) \leq M(1, j + 1)$ for all valid indexes $i, j$.

An example is shown in Figure 2(a), where both $4 \times 2$ matrices $M_0, M_1$ satisfy this definition. We can view the result of the initial csort as producing $W$ matrices sorted in column-major order, each $R \times 1$ in size. From this point forward, a sequence of specialized operations, which we call matrix-column merge (mcmerge), lead to progressively larger matrices sorted in column-major order. This design considering two key characteristics of SIMD: (i) multiple columns can be manipulated with one instruction; and (ii) re-arranging keys across columns and registers is expensive.

Therefore, it is advantageous to keep the items in column-major order and delay the transpose as long as possible.

Definition 4 (mcmerge). Given two $r \times c$ matrices $M_0, M_1$ sorted in column-major order, mcmerge reorders the keys such that max($M_0$) $\leq$ min($M_1$) and both matrices remain sorted in column-major order.

This algorithm is rather complex; we thus relegate its details and mapping to the various instruction sets to the code [1]. Its essence boils down to the following. Suppose we split $M_0$ into $r/2 \times 1$ partial columns $v_1, \ldots, v_2c$ such that max($v_i$) $\leq$ min($v_{i+1}$). Note that each $v_i$ is internally sorted. Applying the same operation to $M_1$, we get additional columns $v_{2c+1}, \ldots, v_{4c}$. In Figure 2(a), $b_0 = (6, 14), v_1 = (19, 28), \ldots, v_7 = (50, 53)$. We now use an odd-even merge network over these 4c elements, coupled with vectorized swap operations, to order the entire sequence of columns such that max($v_{2j}$) $\leq$ min($v_{2j+1}$) holds for all $j \in [1, 4c]$. This is illustrated in Figure 2, which runs mcmerge over two $4 \times 2$ matrices to produce a $4 \times 4$ result sorted in column-major order. Part (b) of the figure performs the initial shuffle to exchange $v_5$ with $v_3$ and $v_4$ with $v_7$, which facilitates a vertical merge given next.
Definition 5 (cswap). Given a $r \times c$ matrix $M$, where $r$ is even and its half-columns are sorted, i.e., $M(i, j) \leq M(i + 1, j)$ for $i \in [1, r/2 - 1] \cup [r/2 + 1, r - 1], j \in [1, c]$, cswap applies a vectorized merge network of size $r$ to make the columns of $M$ fully sorted.

Figure 2(c) shows the effect of running a single cswap over the result in (b). After the next shuffle, i.e., $v_2 \leftrightarrow v_4, v_0 \leftrightarrow v_5$, and another cswap, we obtain the result in (d). The next shuffle is the most complicated, i.e., $v_2 \to v_8, v_3 \to v_4, v_4 \to v_6, v_5 \to v_3, v_6 \to v_2, v_7 \to v_5, v_8 \to v_7$, but, after another cswap, it produces the correct partial columns in (e), but in the wrong locations. After another round of shuffling, we obtain the final result in (f).

It should be noted that the number of steps required for mcmerge is the depth (i.e., number of parallel layers) of the underlying merge network of size $4c$, where each step contains one cswap and one or more shuffles. For $c = 2$ in Figure 2, we have a size-8 merge network with 3 layers. Furthermore, the number of regular swaps contained in each cswap is the length of its size-$r$ merge network (i.e., 3 for $r = 4$ in the figure). Thus, larger $r$ or $c$ make the process slower. However, when $W > 2c$, vectorization allows mcmerge to apply simultaneous operations on all $W/(2c)$ pairs of matrices. For example, $W = 16$ performs the transformations in Figure 2 on four pairs of $4 \times 2$ matrices at no extra cost.

For back-to-back mcmerges, the last reordering (e)$\to$(f) can be omitted; instead, the sorted half-columns can be used in their current locations if the merge network of the following mcmerge is adjusted accordingly. It may also appear that (e)$\to$(f) is unavoidable at the final mcmerge; however, for $R \geq W$, it can also be omitted by simply renumbering the registers after the transpose.

Origami uses $r = R$, applying mcmerge repeatedly with $c = 1, 2, 4, \ldots$ At some point before reaching $W/2$, $c$ becomes large enough that an alternative mechanism can continue merging faster than mcmerge. This is related to the growing depth of size-$4c$ merge networks and the corresponding shuffle cost. At that point, it is better to perform a transpose to convert the matrix into row-major order and switch to another algorithm, which we explain next.

### 3.3 Matrix-Row Merge

Our second method, which we call matrix-row merge (mrmerge), is similar in spirit to mcmerge, except it maintains sorted keys in row-major order rather than column-major.

Definition 6 (mrmerge). Given two $r/2 \times c$ matrices $M_0, M_1$ sorted in row-major order, mrmerge reorders the keys such that $\max(M_i) \leq \min(M_j)$ and both matrices remain sorted in row-major order.

This algorithm is derived from a modular merge sorting networks [21] and its generalization for keys stored in a grid [7]. At a high level, mrmerge performs the following steps:

1. Reverse each row of the second matrix $M_1$;
2. Stack the matrices on top of each other and run cswap;
3. Sort each row of the resulting $r \times c$ matrix $M$.

To understand this better, consider Figure 3 which uses transposed versions of $M_0, M_1$ from Figure 2(a), i.e., input consists of $2\times4$ matrices. This would be equivalent to Origami stopping mcmerge after $c = 1$. In Figure 3(a), we reverse the rows of $M_1$ and run cswap on the stacked matrix to obtain the result in (b). After this, it is guaranteed that the largest key in row $j$ is no bigger than the smallest key in row $j+1$. What is left at this point is to simply sort the rows. For SSE and AVX2, this is accomplished by transposing the full matrix $M$, running csort, and transposing it back. For AVX-512, it is faster to use its new masking instructions, which are not supported on earlier platforms, and perform the sort in place leveraging the observation that each row is a bitonic sequence. At the end of this step, all $R$ registers are stored to memory and the process repeats with the next $R/2$ items.

It should be noted that mrmerge is not significantly affected by the increasing complexity of the merge networks as $r$ or $c$ increase, which results in excellent scalability to large matrix sizes. However, it has a non-negligible minimum cost (e.g., two transposes) that makes it inefficient for short sequences. This is in contrast to mcmerge, where the overhead is low to begin with but increases rapidly as the network becomes more complex. It is therefore beneficial to switch between them at some critical threshold, which we determine experimentally later in the paper.

### 3.4 Matrix Transpose

Column-wise SIMD operations leave sorted lists scattered in different registers in column-major order. This organization must usually be fixed with a matrix transpose before the data can be written back to memory. An SIMD transpose is performed with $\log_2 W$ levels of diagonal exchanges, where at level $j = 0, 1, \ldots, W-1$ exchange $2^j$ bits. Present work [6], [14], [26], [33] achieves this through a pair of shuffle or permute intrinsics for each diagonal exchange. However, this puts pressure on port 5 in Intel CPUs and becomes a performance bottleneck [9]. To avoid this, we replace some of the shuffles with blend instructions, which are executed in ports 0, 1, and 5. This yields better IPC (instructions per cycle) performance, which is especially useful during multiple independent exchanges where the CPU’s out-of-order execution engine can issue instructions to different ports. The following code segment shows how we can achieve this for diagonally exchanging 64 bits.

We refer to these as transpose_v0 and v1 respectively.

```c
// transpose_v0: two shuffles
_v0 = __mm256_shuffle_ps(v0, v1, 0x33);
_v1 = __mm256_shuffle_ps(v0, v1, 0xCC);

// transpose_v1: one shuffle and two blends
v = __mm256_blend_ps(v0, v1, 0xCC);
_v0 = __mm256_blend_ps(v0, v1, 0x33);
```

![Figure 3: Operation of mrmerge on 2 \times 4 matrices.](image-url)
3.5 Optimal Run Length

Note that Origami is flexible enough to allow a variety of run lengths \( m \in [W, W^2] \). At what point \( m \) should the algorithm be operating? As given in the next result, this depends on the speed \( S_1(m) \) at which it can produce sorted runs during \( P_1 \) and the in-cache merge speed \( S_{merge} \) of phase \( P_2 \).

**Theorem 3.1.** The optimal value of \( m \) for \( P_1 \) minimizes

\[
f(m) = S_{merge}/S_1(m) - \log_2 m.
\]

There is tradeoff in the cost function \( f(m) \) — larger \( m \) increases the log term being subtracted, but also reduces the speed \( S_1(m) \). And thus the sweet spot usually lies somewhere in the middle.

4 IN-CACHE MERGE (\( P_2 \))

4.1 Merge Kernel

The main building block of merge-based sorts is the binary merge, which we call \( \text{bmerge} \). Its purpose is to combine two large inputs (i.e., significantly longer than \( W \)) sorted sequences into one. With non-trivial input sizes (e.g., over 1 GB), there can be 20-30 passes of \( \text{bmerge} \) over the data. Therefore, its speed plays an important role in the overall performance of the sort. The main difference between \( \text{bmerge} \) in phase \( P_2 \) (in-cache) and \( P_3 \) (out-of-cache) is whether the algorithm needs to keep memory traffic below some threshold.

One component of \( \text{bmerge} \) is its kernel, i.e., an algorithm that merges two sorted registers.

**Definition 7 (rswap).** Given two sorted SIMD registers \( x \) and \( y \), \( rswap(x,y) \) rearranges the items such that both registers are still sorted and \( \max(x) \leq \min(y) \).

Note that if \( x \) is loaded from an input stream \( A \) and \( y \) from another stream \( B \), \( rswap \) shuffles the data such that the smallest \( W \) items out of \( 2W \) end up in \( x \), which is then written to the output. In more general cases, illustrated in Algorithm 1, we can load \( k \geq 1 \) registers from each of \( A \) and \( B \) and run a sequence of \( rswap \)s from any merge network (e.g., odd-even, bitonic) of size \( 2k \). This produces a sorted sequence of \( 2kW \) items, whose lower half \( r_0, \ldots, r_{k-1} \) is stored to the output \( C \) and the upper half \( r_k, \ldots, r_{2k-1} \) is retained for the next iteration. We then reload the emptied registers from the stream with smaller values at the front, advance its pointer, and repeat until one of the two streams is exhausted.

For scalar keys, \( rswap \) is identical to the regular swap. For wider registers, existing work uses SIMD merge kernels based on either the bitonic [6], [14], [18], [27], [33] or odd-even network [15]; however, their speed leaves room for improvement. The fastest prior kernel, which comes from [17], uses a series of \( \text{rotate} \) and \( \text{swap} \) operations; however, it works only for SSE. We extend this method to AVX2/AVX-512 and compare it against \( \text{rrmerge} \). The former requires \( W \) stages compared to \( \log_2 W + 1 \) in the latter. But \( \text{rotate} \)s are cheaper than transpose, which usually makes this method faster. For AVX-512, however, larger register width produces a significant difference between \( \text{wshuf} \) and \( \log_2 W + 1 \). In addition, introduction of \text{mask}_{min} \text{ and mask}_{max} \text{ intrinsics in AVX-512 gives more flexibility in data movement across registers and enables faster sort compared to older extension sets [32]. In the benchmark section, we evaluate which method is faster and deploy the winner in Origami.

**Algorithm 1:** Outline of generalized \( \text{bmerge} \)

```
Function bmerge (Item *A, "endA", *B, "endB", *C)
load registers \( r_0, \ldots, r_{k-1} \) from \( A \); \( A += kW; \)
load registers \( r_k, \ldots, r_{2k-1} \) from \( B \); \( B += kW; \)
while \( A \neq \text{endA} \) and \( B \neq \text{endB} \) do
    rswaps from a merge network of size \( 2k \);
    store \( r_0, \ldots, r_{k-1} \) to \( C \); \( C += kW; \)
    reload \( r_0, \ldots, r_{k-1} \) from either \( A \) or \( B \);
    move \( A \) or \( B \) forward by \( kW; \)
end
merge keys left in registers and the unfinished list;
```

4.2 Advancing Pointers

Performance of Algorithm 1 depends on \( rswap \) and the last two lines of the loop (i.e., deciding which stream to load from and moving the pointers). When \( kW \) is small, pointer management plays a pivotal role in determining the speed. The majority of work in the SIMD literature [6], [14], [16], [27], [32] relies on a branching comparison to decide which of the two pointers to advance:

\[
\text{if } (\ 'A' < 'B' \text{) do reload } r_0, \ldots, r_{k-1} \text{ from } A; A += kW; \}
\text{else } \{ \text{reload } r_0, \ldots, r_{k-1} \text{ from } B; B += kW; \}
\]

This version, which we call \( \text{bmerge}_v0 \), sometimes leads to a significant misprediction penalty and bottlenecks the sort. Other alternatives [15], [17] dismiss mergesort for in-cache operation in favor of comb sort (i.e., an extension of bubble sort), which runs in quadratic time for certain inputs [4]. Finally, the remaining papers [30], [33] use expensive intrinsics (e.g., \text{scatter}, \text{gather}, \text{mask}_{cmp}) that are not only slower than our approach below, but also inapplicable to certain instruction sets (e.g., scalar, SSE).

For non-SIMD sorts, there were several attempts at developing a branchless \( \text{bmerge} \). For example, [12] argues that the compiler will generate \( \text{cmov} \) (conditional move) instructions for short if statements, but this is usually not the case in practice. Other techniques include [13], which replaces the branch with a binary flag and multiplication, and [16], which runs a hybrid set-intersection algorithm that removes difficult-to-predict branches in favor of those that are easy to predict. These methods are usually 40-50% faster than the branching version; however, Origami develops an even faster, purely branchless \( \text{bmerge} \) as we explain next.

The first improvement to \( \text{bmerge}_v0 \) is to attempt replacing the if block with a sequence of ternary operators to force the compiler to generate \( \text{cmov} \) instructions during load. We call this version \( \text{bmerge}_v1 \):

\[
\text{flag} = 'A' < 'B';
\text{if } \text{flag} \{ \text{load}(A+iW) ; \text{load}(B+iW); i \in [0, k-1] \}
A += \text{flag} ? kW : 0; B += \text{flag} ? 0 : kW;
\]

While this works fine for scalar with \( k = 1 \), the compiler gets confused for \( k > 1 \) and opts for a branch instead of multiple \( \text{cmovs} \). It also computes the flag three times, which may be related to its inability to hold both incremented pointers in registers. On top of that, SIMD \_mm_load intrinsics do not support conditional moves, which leads to branches as well. The first and third issues can be mitigated by using \( \text{cmovs} \) to control the pointer to where the data is.
Algorithm 2: Origami bmerge

```
Function bmerge_v3(item A, B, endA, endB)
  load r0, ..., rk−1 as in Algorithm 1;
  loadFrom = A + k'W; opposite = B + k'W;
  while loadFrom ≠ endA and loadFrom ≠ endB do
    rswaps from a merge network of size 2k;
    store r0, ..., rk−1, to C; C == k'W;
    flag = loadFrom < opposite;
    tmp = flag ? loadFrom : opposite;
    opposite = flag ? opposite : loadFrom;
    loadFrom = tmp;
    load r0, ..., rk−1 from loadFrom; loadFrom += k'W;
  end

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

coming from. This version, which we call bmerge_v2, is completely branchless:

```
src = flag ? A : B;
ri = load(src + i'W); i ∈ [0, k − 1]
A += flag ? k'W : 0; B += flag ? 0 : k'W;
```

While this algorithm is 50% faster than v0, the compiler still has a redundant computation of the flag. To overcome this issue, we introduce our final method in Algorithm 2, which we call bmerge_v3. It runs two pointers loadFrom and opposite, where the former always points to the array from which the next load will take place and the latter points to the current position in the other array. The pointers are swapped based on the flag using one mov instruction and two cmovs, which increases efficiency and yields a 25% faster merge than v2 and 86% faster than v0. In addition, Algorithm 2 is distribution-insensitive since the branchless merge removes speculation from the control flow and runs at a nearly constant speed for all inputs.

Even though the vectorized merge network in Algorithm 2 allows multiple rswaps to proceed in parallel, it still periodically runs into pipeline stalls when there is dependency between adjacent operations in the merge network. To push performance even further, it is beneficial to run multiple independent merge networks to take advantage of the CPU’s instruction-level parallelism. To this end, Origami unrolls bmerge to simultaneously read several pairs of input lists in a single thread. This interleaves the instructions of the rswaps and reduces the duration of the stall cycles in the pipeline, which for AVX2 increases performance by 77–94%.

### 4.3 Scalar Merge Optimizations

We can speed up the scalar bmerge further by reducing the number of swaps needed by the merge network. Assume that registers r0, ..., rk−1 are loaded from A and rk, ..., rk+1 from B. A 2k-size merge network will aim to reorder the keys such that register r0 ≤ ... ≤ rk−1, i.e., it sorts the entire collection of 2k items. Since each iteration of the loop stores the smallest k keys to the output, the lower half of this sequence must be sorted; however, the upper half can remain in some partially sorted state that allows the next iteration to properly extract the smallest k items. For k that is a power of 2, it turns out that we can skip any swaps that involve the second half, i.e., registers rk, ..., rk+1, which can be proven using the zero-one principle [19]. This novel result allows Origami to reduce the number of swaps from 9 to 8 for k = 4 and from 25 to 20 for k = 8. Note that this optimization does not work for vector registers since multiple keys reside in each r.

### 5 OUT-OF-CACHE MERGE

Merging lists that do not fit in the cache requires consideration of memory-bandwidth limitations. In addition, proper load balancing is needed to maximize thread-level parallelism. In this section, we discuss the design decisions in Origami for out-of-cache merge using single and multiple threads.

#### 5.1 Independent Merge (P3)

Phase P3 finishes when each thread obtains a number of sorted lists of L2-cache-size C. Assuming the number of these streams is still significant, we can continue merging them separately in each thread, which constitutes phase P3. An out-of-cache merge involves loading items from main memory, running rswap over them, and storing the result back to RAM. The maximum achievable speed for this step is that of memcpy, which we define as the rate at which T threads can concurrently copy large chunks of memory (e.g., 100 MB) without any synchronization. For example, Skylake-X i7 CPUs with DDR4-3200 quad-channel memory max out at 37 GB/sec. Our vectorized bmerge_v3 can exhaust this bandwidth with just 3 threads, even though this CPU family comes with many more cores (i.e., between 6 and 18). For older computers with lower RAM clocks and those with dual-channel memory, the saturation point may be approached even with a single Origami thread, which severely limits the overall performance on these systems.

The majority of prior SIMD mergesorts [14], [26], [30], [32], [33] ignore this issue and continue with binary merges in P3. The main other alternative is to run a multi-way merge to get around this bottleneck. Observe that a k-way merge reduces the number of out-of-cache passes over the data from \( \log_k (N/C) \) to \( \log_k (N/C) \). This reduces memory-bandwidth demand by a factor of \( \log_k k \). One technique [6] is to use a merge tree that resides in the L3 cache and implements an \( N/C \)-way merge through a series of binary merges at each node. Each internal node stores partial merge results in a circular-buffer queue. A node is marked ready if both of its children’s queues contain a threshold number of keys. Threads draw elements from a global pool of ready nodes and process their merges in parallel. Besides requiring inter-thread synchronization and inter-core data traffic, which would we like to avoid in P3, this method fails to utilize dedicated core caches (i.e., L2) and uses relatively slow queues at each node. Another approach [17] uses a 32-way merge tree that fits into the L2 cache and fixed 4-KB intermediate buffers instead of queues. It encodes the stream from which the key originated in the upper 5 bits of each item and runs insertion sort to break ties, which leads to not only extensive overhead in coding/decoding the index bits, but also quadratic complexity on certain inputs.

In Origami, we develop an L2-cache-residing k-way merge tree, which we call mtree. Unlike prior work, where k is fixed, our approach uses it as a tuning parameter that can be adapted to the characteristics of the architecture on which the sort is running. To facilitate faster operation, each node in mtree performs a 4-way
merge instead of binary. This is done with tiny intermediate buffers inside each 4-way node (i.e., 64 – 128 bytes), while buffers at the root and leaves remain large. We use our branchless bmerge_v3 from Section 4.2 within the tree. The optimal choice of \( k \), which can be determined experimentally at runtime or in advance, depends on \( T \), memory bandwidth, and L2 cache size. If the optimizer returns \( k = 2 \), we run binary merges using bmerge_v3; otherwise, we invoke mtree_v2, with hyper-threading enabled to better utilize the CPU pipeline.

For comparison purposes, we refer to the baseline binary merge tree that internally uses bmerge_v0 at each node as mtree_v0, binary tree with bmerge_v3 as mtree_v1, and our final quad-way tree with bmerge_v3 as mtree_v2.

### 5.2 Cooperative Merge (\( P_3 \))

Mergesort inherently allows easy utilization of thread-level parallelism since sorted sequences can be merged independently. At a certain point, however, the number of remaining lists becomes insufficient to continue independent merging (i.e., fewer than one per thread). This calls for the final phase \( P_3 \), where \( T \) threads cooperatively process the remaining items. Most prior work [6], [15], [17], [30], [32], [33] begins this phase with \( k = T \) streams and uses a binary-search partitioning method that splits each list into \( T \) segments \( [aij, b_{ij}] \), where \( i = 1, \ldots, k \) and \( j = 1, \ldots, T \), such that \( \sum_{i=1}^{k} (bij - aij) = N/T \). Other approaches either avoid the issue of load-balancing by shrinking \( T \) as the merge nears the end [14] or do not multi-thread the code at all [26].

To avoid bottlenecks on memory bandwidth, our observation is that the merge must utilize at least \( k \) sequences, where \( k \) is selected optimally by mtree in \( P_3 \). For example, if \( k = 64 \) and \( T = 8 \), prior work would lose half the achievable performance in \( P_3 \). Other issues in previous approaches include single-threaded partitioning and poor management of stragglers, i.e., threads that take longer to finish despite having the same amount of work. Instead, Origami terminates \( P_3 \) at the stage when at least \( k \) streams remain, performs parallel partitioning of the lists, and creates a large number of smaller jobs (e.g., \( 16T \)) to reduce the time gap between the fastest and slowest threads. The jobs are added into a shared queue, from which all threads draw their workload in parallel.

### 6 EXPERIMENTS

#### 6.1 Setup

Benchmarks run all code compiled with Visual Studio 2019 in Windows Server 2016 on an 8-core Intel Skylake-X (i7-7820X) CPU with a fixed 4.7 GHz clock on all cores, 1 MB L2 cache, and 32 GB of DDR4-3200 quad-channel RAM. When AVX-512 was used, BIOS default to a 400-MHz lower clock (i.e., 4.3 GHz), which is known as the AVX offset implemented by many motherboard manufacturers to keep core temperature under control.

We enable the maximum level of optimization and use appropriate flags (e.g., \(/\text{arch}:\text{AVX512} \) to ensure the compiler uses all available SIMD registers. Performance profiling is done with the help of EMON, which is a low-level command-line tool that is part of the Intel VTune profiler [10]. EMON leverages the counters from CPU performance monitoring units to collect event information. Unless mentioned otherwise, all keys are uniformly random.

### 6.2 Tiny Sorters

#### Matrix Transpose

We start by testing the speed of matrix transpose as it is a key component of our in-register sort. The test loads a \( W \times X \) matrix with random keys and performs a billion transposes. Table 1 shows the result across all SIMD extension sets, where \( B = 32 \) uses 4-byte keys, \( B = 64 \) runs 8-byte keys, and \( B = 64 + 64 \) uses 16-byte key-value pairs. Note that gray shading in the table shows the fastest result for each architecture and value of \( B \).

As discussed in Section 3.4, transpose_v1 achieves better pipeline utilization by distributing the workload among ports 0, 1, and 5. We see a reduction in CPU cycles per transpose by up to 23% in SSE, 27% in AVX2, and 11% in AVX-512. The result for key-value pairs, however, demands further explanation. For SSE, 128-bit pairs consume entire registers, where transposing a 1 x 1 matrix is not needed. In AVX-2, the transpose is a single 128-bit diagonal exchange. The two permutes of v0 take 4 cycles to achieve this since each has latency 3 and throughput 1. On the other hand, v1 also requires 4 cycles because the two bEnds (latency 1, throughput 0.5) following the permute can be performed in 1 cycle when issued to separate ports. In this case, reducing port 5 pressure therefore does not improve performance.

From the table, transposing key-value pairs in AVX-512 with v1 is 19% slower than with v0. This transpose pipeline consists of a pair of diagonal exchanges – one shuffles 128 bits and the other 256 bits. Results show that this step takes 4.1 cycles in v0 and 4.5 cycles in v1. This is possibly due to AVX-512 using mask intrinsic to perform the mask_blend intrinsic. For each pair of bEnds, a mask register is loaded from a general-purpose register with kmov, which can be the number of pairs of sorted sequences in the matrix. Then, a \( k \)-merge operation uses SIMD to produce a binary merge over all \( X \) pairs, replacing them with size-\( 2k \) sorted sequences. For example, a merge that begins with \( X = 16 \) sorted sequences of length \( K = 16 \) produces \( X/2 = 8 \) sorted sequences of size \( 2K = 32 \).

### Table 1: Matrix transpose cost (CPU cycles/transpose)

<table>
<thead>
<tr>
<th></th>
<th>SSE</th>
<th>AVX2</th>
<th>AVX-512</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>2.25</td>
<td>2.25</td>
</tr>
<tr>
<td>64+64</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Terminates when at least \( k \) streams remain, performs parallel partitioning of the lists, and creates a large number of smaller jobs (e.g., \( 16T \)) to reduce the time gap between the fastest and slowest threads. The jobs are added into a shared queue, from which all threads draw their workload in parallel.
Each test performs a billion $K$-merges within the matrix. Table 2 shows the result, where cells with a dash denote an invalid merge condition. The outcome is consistent with our earlier discussion — mrmerge is up to 4.65× faster than mrmerge for small $K$ as it utilizes the data-level parallelism of SIMD registers better (i.e., by performing column-major merges). However, it suffers an almost exponential decay in speed as $K$ gets larger. The reason is that its merge networks get longer and the overhead of expensive cross-column permutes becomes higher. While mrmerge begins at a lower rate, the length of the sequence being merged does not significantly impact its performance. While $K$ affects the number of merge-network swaps, the remaining elements of the algorithm have constant cost, i.e., exactly $R/2$ reverses and $2R/W$ transposes, irrespective of $K$.

### 6.3 In-cache Merge

$rswap$. We next discuss the performance of $rswap$. We benchmark this by repeatedly performing a merge of two sorted sequences of size $W$ within the matrix. Table 3 compares the different implementations of $rswap$ for various $B$. It shows that for both SSE and AVX2, running $W$ levels of rotate/swap instructions outperforms both mrmerge and the bitonic network. AVX-512, on the other hand, runs much faster with mrmerge, where specialized mask instructions allow rapid sorting within the rows. Finally, 128-bit key-value pairs occupy an entire SSE register, in which case $rswap$ is equivalent to a regular $swap$, and all three methods produce the same result. The data in Table 3 is in line with our discussion in Section 4.1. As a side note, scalar $rswap$, which is absent from the table, increases speed by 5%–15% using our optimized size-8 merge network from Section 4.3. This is the benefit of saving one out of every nine swaps.

Table 4 shows the effect of running multiple independent $rswaps$ on separate input streams. For all extension sets, there is a significant performance improvement from unrolling (up to 2.86× in SIMD, 1.38× in scalar). For SIMD, this is because each swap is dependent on the previous rotate or permutation. Without unrolling, the CPU fails to take advantage of its out-of-order execution engine. While running multiple $rswaps$, the pipeline can reorder the instructions to use the available execution units and utilize the cycles otherwise wasted in pipeline stalls. The performance gain for scalar in row $L$ is smaller as the number of stalls is already minimized by using a size-8 merge network. After a certain threshold, further unrolling leads to performance degradation due to the lack of registers.

### $bmerge$

Our next test benchmarks the in-cache $bmerge$. The input arrays are filled with $C/4$ random keys, where $C$ is the L2 cache size, and sorted separately before running the merge. To prevent cache misses due to core hoppings, we bind the merging thread to a fixed core. The left half of Table 5 compares performance of Origami’s $bmerge_v3$ (Algorithm 2) with that of the naive branched version $v0$. It also shows the effect of loading $k$ registers per input stream and running a size-$2k$ $rswap$ merge network. The $v3$ setup gains up to 92% in scalar and up to 72% in vectorized $bmerge$ over the corresponding $v0$. The margin of improvement, however, gets narrower for larger networks and wider registers. In these cases, executing the $rswaps$ constitutes the majority of instructions in $bmerge$ and the relative penalty of branch misprediction diminishes.

For scalar merges in Table 5 under $B = 32$ and $B = 64$, Origami reaches peak speed at $k = 4$, but then performance takes a dive. There are two reasons for this. First, some of the 16 available registers are used to store pointers and loop variables of $bmerge_v3$. Using a larger $k$ would result in some of these contents being spilled to memory (i.e., the stack). Second, larger merge networks have more complexity per key they output. For example, a classical size-8 merge network [2] runs 9 swaps to output 4 keys, while a size-10 network executes 15 swaps to spit out 5 keys. This increases the cost per output key from 2.25 swaps to 3. Not surprisingly, key-value pairs (i.e., $B = 64 + 64$) exhibit $k = 2$ as optimal for scalar since each swap now uses double the amount of registers.

As discussed in Section 4.2, we run multiple independent $bmerge$ operations in a single thread to achieve even higher throughput. The right side of Table 5 shows the effect of unrolling $bmerge_v3$ to multiple pairs of streams, all of which still fit into the L2 cache. Except for scalar, where we either run out of registers or already fully pack the pipeline, unrolling significantly improves performance. To be specific, we gain up to 47%, 96%, and 59% in SSE, AVX2, and AVX-512, respectively, over the corresponding merges without any unroll. At a certain point, additional unroll begins to hurt performance, which is similar to earlier observations in Table 4. Origami selects the best option for $bmerge_v3$ based on Table 5.
Table 5: In-cache bmerge speed (M/s); the left half of the table compares Origami optimized branchless merge (v3) with naive branched merge (v0); the right half shows further improvement from unrolling v3 to merge multiple sequences

<table>
<thead>
<tr>
<th>k</th>
<th>v8</th>
<th>Scalar</th>
<th>v3</th>
<th>SSE</th>
<th>AVX2</th>
<th>AVX-512</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>858</td>
<td>575</td>
<td>1402</td>
<td>1585</td>
<td>1966</td>
<td>2006</td>
</tr>
<tr>
<td>2</td>
<td>511</td>
<td>1136</td>
<td>1893</td>
<td>2284</td>
<td>1821</td>
<td>1888</td>
</tr>
<tr>
<td>3</td>
<td>791</td>
<td>1213</td>
<td>1701</td>
<td>1904</td>
<td>1526</td>
<td>1537</td>
</tr>
<tr>
<td>4</td>
<td>922</td>
<td>1527</td>
<td>1869</td>
<td>2016</td>
<td>1651</td>
<td>1662</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>k</th>
<th>v8</th>
<th>Scalar</th>
<th>v3</th>
<th>SSE</th>
<th>AVX2</th>
<th>AVX-512</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>858</td>
<td>575</td>
<td>1402</td>
<td>1585</td>
<td>1966</td>
<td>2006</td>
</tr>
<tr>
<td>128</td>
<td>511</td>
<td>1136</td>
<td>1893</td>
<td>2284</td>
<td>1821</td>
<td>1888</td>
</tr>
<tr>
<td>64</td>
<td>791</td>
<td>1213</td>
<td>1701</td>
<td>1904</td>
<td>1526</td>
<td>1537</td>
</tr>
<tr>
<td>32</td>
<td>922</td>
<td>1527</td>
<td>1869</td>
<td>2016</td>
<td>1651</td>
<td>1662</td>
</tr>
</tbody>
</table>

Table 6 compares our optimal bmerge against the existing work. We directly use the source code published by the authors of [14], [26], [30], [32], porting everything to Windows and building with maximum compiler optimizations. Some methods [13], [16], [33] do not have a reference implementation; however, the papers provide enough code snippets and details for us to make one ourselves. SSE is left out of this table since older papers either do not use phase P2 for in-core sorting [6], [18], [27] or rely on distribution-sensitive combsort [15]. Results in the table show that Origami achieves a substantial improvement over prior methods, exceeding their merge speed by 1.27 – 2.85x.

6.4 Out-of-cache Merge

bmerge. During P2, each thread has many pairs of sorted lists to merge, which allows usage of unrolled bmerge_v3 throughout; however, the situation changes once input arrays become larger than L2 and threads put pressure on the memory controller. We first discuss the bandwidth usage by bmerge_v3 running in a single thread over sequences that cannot be kept in the cache. The benchmark setup is similar to earlier in-cache bmerge tests except the total output size is now 1 GB. Table 7 displays the throughput of the optimized and parameter-tuned bmerge_v3, as well as that of C++ std::memcpy. We drill into these numbers in more detail next.

On a single core, the RAM bandwidth is limited by the number of line-fill buffers (LFBs) and RAM latency. Skylake-X has 12 LFBs [28] and its latency on our test machine is reported as 61 ns by the Intel MLC [11]. This gives us an ideal one-directional throughput (i.e., reads or writes) as 12 x 128/61 = 25.2 GB/s. Dividing this in half produces an estimated upper bound on memcpy rate – 12.6 GB/s. As the table shows, Origami comes within 60% of this number using scalar merging and 92-95% using SIMD. It also beats memcpy since bmerge_v3 switches to streaming (i.e., non-temporal) SIMD store instructions to bypass the cache on large inputs. By comparing Table 7 with the top speed in Table 5, where unrolled Origami hits 16 GB/s on SSE and 17.8 GB/s on AVX2, it is easy to see how a single merge thread can exhaust the available bandwidth of its core.

With multiple threads, the situation gets worse. The memcpy bandwidth across all cores on this machine is limited to 37 GB/s. With 8 threads, Origami could exceed this rate by as much as 3.8x. It is therefore beneficial to explore alternative ways to merge out-of-cache data that avoid these bottlenecks, which we do with our k-way merge tree.

mtree. Next we discuss the performance of mtree. The test is prepared by generating a 1 GB buffer, dividing it into k chunks, and then sorting them individually before feeding the result into the leaves of the tree. While the tree stays in L2 cache, the data is directly served from and written to RAM. Table 8 compares the results of different mtree versions for 32-bit keys over various k. To recap, both v0 and v1 trees use binary nodes, where the former runs the branching bmerge_v0 and the latter uses our fastest branchless bmerge_v3. The third variant is mtree_v2 that internally operates 4-way merge nodes, except for the root node, which may be 2-way depending on the desired value of k. As seen in the table, mtree_v2 is the best option for all SIMD extension sets. It beats v0 by bigger margins than previously in Table 5 (e.g., 19% in the first row of AVX-512) and comes out ahead of mtree_v1 by 5-12% due to better clustering of data in small buffers.
within each 4-way node. It should be noted that \texttt{bmerge\_v3} inside the tree cannot be unrolled due to complex dependencies that control buffer refill at each node and its children. As a result, the extra cost of running mtree\_v2 compared to repeated binary merges can be assessed by diving the speed of non-unrolled \texttt{bmerge\_v3} on the left half of Table 5 by \log_2 k. Generally, the tree is slower than the corresponding estimates from binary merges, except for AVX-512 in the first three rows of Table 8. For example, its 4-way merge speed 1482M/s manages to beat the predicted 2770/2 = 1385M/s. Reduction in performance is negligible in the first few rows of Table 8, but then becomes more noticeable as \( k \) grows. It is therefore beneficial to use the smallest \( k \) allowed by the memory bandwidth for a given number of threads \( T \).

Scalar results are absent from the table as the speed remains similar for both mtree\_v1 and mtree\_v2. With general-purpose registers used to maintain tree variables and perform the merge, the advantage of 4-way merging is lost due to register scarcity.

### 6.5 End-to-End Sort

For convenience of presentation, the section breaks the full Origami sort, which consists of the fastest components outlined above, into so-called checkpoints \( C_i \), each of which represents execution of phases \( P_1 \) through \( P_t \). The source code is available from [1].

\textit{Origami \( C_1 \) (Tiny Sorters).} We now compare the \( C_1 \) performance of Origami against that of prior work. We first define a \textit{chunk-c sort} to be an algorithm that sorts every \( c \) continuous items in an input of size \( N \), where \( c \leq N \). Note that this section uses \( c \) up to \( m = R \cdot W \), i.e., it sorts the entire chunk in registers (at least from the compiler’s perspective, see below), but later sections expand \( c \) to be much larger. Our next test generates a block of random items that fit in the L2 cache (i.e., \( N = C \)) and measures the chunked sort speed for different \( c \). Note that Origami loads \( R \cdot W \) keys into its matrix and sorts each group of \( \min(R \cdot W, c) \) items in registers. It begins the sort with ncmerge and then, if necessary, moves to nmerge at the optimal switch point from Table 2. Since none of the prior work sorts more than \( W \) keys in register, achieving \( c > W \) requires them to run nmerge from phase \( P_2 \).

Table 9 shows the result on 32-bit keys (most prior implementations do not support larger \( B \)). Note that dashes represent cases that require either too many registers (SSE and AVX2) or fewer than one (AVX-512). In all cases, Origami demonstrates a significant improvement over prior work, posting a 5x faster speed in third row for SSE, 4.3x for AVX2, and 1.5x for AVX-512. As chunk size increases, our SSE advantage shrinks to 16% at \( c = 256 \), where the

vectorized combsort from [15] is quite efficient; however, its quadratic complexity on certain inputs makes it potentially unsuitable for sorting real-world (i.e., skewed) data. For AVX2 and AVX-512, Origami finishes the table with a 2.3 – 2.4x advantage over the nearest competitor. The extra cache traffic and their merge being expensive for small sequences render previous methods inefficient in this benchmark.

For scalar, checkpoint \( C_1 \) simply runs a sequence of swaps in a size-\( c \) sorting network over each chunk. Since prior SIMD work does not consider scalar sorting as a viable option, we only discuss Origami results. For \( c = 8 \), the speed of its \( C_1 \) is 1804M/s for \( B = 32 \) and 1590M/s for \( B = 64 \). Note that this performance more than doubles that of [15] in the first row of Table 9. Dealing with \( B = 64 + 64 \) key-value pairs, where each item is packed in a struct, is the next question. Trivially, we could overload the \(< \) operator in C++ and use the scalar swap from Section 3. However, compilers generate a significant amount of load/store instructions with operator overload, even when sorting only \( c = 8 \) pairs. Instead, we modify the swap to move the keys and values as separate entities, forcing the compiler to generate a single cmp instruction and four cmove, which is optimal. This results in a 20% speed-up over operator overload.

\textit{Origami \( C_2 \) (In-Cache).} We now set up a benchmark to find the optimal switch point from \( P_1 \) to \( P_2 \) in Origami. To achieve a chunk-\( c \) sort, there are two competing options. First, we could run \( P_1 \) all the way to \( c \), assuming the compiler allows usage of this many registers. The speed for this step comes from Table 9. Alternatively, we could run \( P_1 \) to \( c/2 \) and then execute a binary merge with \texttt{bmerge\_v3}. For each \( c = 8, 16, \ldots \), we test both versions and select the winner, which represents the Origami algorithm for checkpoint \( C_2 \). Note that once the dilemma is resolved in favor of \texttt{bmerge\_v3}, all remaining chunks are processed via binary merge as well. This is because each additional phase in Origami \( P_1 \) gets slower as \( c \) increases, while \texttt{bmerge\_v3} runs at a constant rate.

Table 10 shows results for \( C_2 \), where the cells marked with asterisks denote the first time \texttt{bmerge\_v3} wins in each column. For example, Origami SSE runs \( P_1 \) up to \( c = 32 \) and then switches to binary merge. This yields a 33% improvement compared to the results in Table 9 by the time we get to \( c = 256 \). Table 10 also demonstrates that in some of the cases, it is beneficial to run \( P_1 \) with more
than RW items, even though the compiler cannot keep all of them in registers. This happens because the latency to offload some of them onto the stack (i.e., L1 cache) is effectively hidden by the CPU instruction-reorder buffer. Thus, it is sometimes possible to push out $m > RW$ sorted items in each iteration of $P_i$ and achieve an overall speed-up. This is shown for AVX-512 in Table 10, where Origami uses 128 registers (i.e., $c = 2K$), way more than available in the system (i.e., 32), just before switching to $P_2$. This allows it to achieve run lengths of $m = 2K$ by using tiny sorters.

Compared to related work, Origami continues posting significant improvement margins, finishing the table with 22% better performance under SSE, 71% under AVX2, and 65% under AVX-512.

**Origami C3 (Independent Out-Of-Cache).** We now inspect Origami performance in checkpoint C3 using a single thread. The benchmark still runs a chunked sort, except the total data size is 1 GB, i.e., exceeds cache capacity. As shown in Table 11, we achieve a $1.5-2.1 \times$ speed-up over the fastest competitor. A few interesting observations can be made about these results.

First, the discrepancy for $c = 128K$ between Tables 10 and 11 is because the former sorts entirely in cache while the latter slides a $c$-size window across a large buffer, which incurs a lot of main-memory traffic. Second, a notable slowdown is visible for [15] compared to Table 10. This is because the combsort in [15] includes a bubble-sort phase at the end to put the data in order. As it turns out, the number of bubble-sort passes may vary widely across chunks, even with **uniformly distributed keys**. Figure 4(a) shows that some of the 128K-size chunks trigger a huge number of passes, therefore significantly affecting the overall sort speed. To test sensitivity of combsort to input workloads, we run additional tests with the following distributions:

- $(D_1)$ Uniformly random, generated by Mersenne Twister
- $(D_2)$-4 All the same, sorted, and reverse-sorted
- $(D_3)$ Almost sorted, where every 7th key is set to $KEY\_MAX$
- $(D_4)$ Pareto, generated as $\min(cell(\beta/(1 - u)), 1)$, where $\beta = 7$ and $u \sim \text{uniform}[0, 1]$
- $(D_5)$ Bursts of same keys, where the length of each subsequence is drawn from $D_k$ and the key from $D_k$
- $(D_6)$ Random shuffle, generated by randomly permuting $D_7$
- $(D_7)$ Fibonacci, wrapped around when it overflows $N$

Figure 4(b) indicates that vectorized combsort performs well when all keys are the same, but then deteriorates into extremely slow regimes with other types of input. In contrast, Table 12 confirms an earlier prediction that Origami is largely distribution-insensitive. The numbers broadly follow the results from Table 5 and provide several points for discussion. First, for all vector extension sets and key type, we achieve a near constant speed for all $D_1 - D_6$. In scalar, Origami receives a performance boost for skewed distributions, particularly where the merged keys have long back-to-back runs from one of the input streams (e.g., when all keys are the same). These bursts of streaming memory reads allows the relatively-slow scalar bmerge to become faster. Since SIMD bmerge is already compute-bound, we do not get any advantage from possibly faster memory access.

Second, within the same extension set, we may expect a constant 2X speed drop when going from $B = 32$ to $B = 64$. However, this is not the case. For scalar, the two sorts remain within 10% of each other since the number of CPU instructions (e.g., comparisons, memory loads/stores) stays the same. Even though the memory traffic of bmerge doubles, it is nowhere near the memory bandwidth. Thus, the drop in speed is only minor. For vector registers, the speed reduction is $1.8 - 2.6 \times$ since not only is $W$ reduced by half and the memory traffic is higher, but certain instructions become more expensive on some of the architectures (e.g., SSE and AVX2 do not have a 64-bit min/max). The penalty is more noticeable for AVX2/AVX-512 because they generally execute faster and get a significant performance boost from the unrolled bmerge under 32-bit keys, as shown in Table 5, but not as much under 64-bit.

Third, the speed-up factor between the platforms is quite a bit lower than the ratio of their $W$ (e.g., AVX2 is only $1.33 \times$ faster than

---

**Table 11: Chunked speed in C3 (M/s); $N = 256M$, $B = 32$**

<table>
<thead>
<tr>
<th>chunk size</th>
<th>SSE</th>
<th>AVX2</th>
<th>AVX-512</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 K</td>
<td>63</td>
<td>176</td>
<td>26</td>
</tr>
<tr>
<td>256 K</td>
<td>64</td>
<td>177</td>
<td>26</td>
</tr>
<tr>
<td>512 K</td>
<td>59</td>
<td>138</td>
<td>26</td>
</tr>
<tr>
<td>1 M</td>
<td>57</td>
<td>131</td>
<td>26</td>
</tr>
<tr>
<td>2 M</td>
<td>55</td>
<td>124</td>
<td>26</td>
</tr>
<tr>
<td>4 M</td>
<td>54</td>
<td>118</td>
<td>26</td>
</tr>
<tr>
<td>8 M</td>
<td>52</td>
<td>112</td>
<td>26</td>
</tr>
<tr>
<td>16 M</td>
<td>50</td>
<td>107</td>
<td>26</td>
</tr>
<tr>
<td>32 M</td>
<td>48</td>
<td>102</td>
<td>26</td>
</tr>
<tr>
<td>64 M</td>
<td>47</td>
<td>98</td>
<td>26</td>
</tr>
<tr>
<td>128 M</td>
<td>45</td>
<td>95</td>
<td>26</td>
</tr>
<tr>
<td>256 M</td>
<td>44</td>
<td>91</td>
<td>26</td>
</tr>
</tbody>
</table>

**Table 12: Origami single-threaded speed (M/s) for 1 GB**

<table>
<thead>
<tr>
<th>$B$</th>
<th>$D_1$</th>
<th>$D_2$</th>
<th>$D_3$</th>
<th>$D_4$</th>
<th>$D_5$</th>
<th>$D_6$</th>
<th>$D_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>43</td>
<td>47</td>
<td>47</td>
<td>44</td>
<td>44</td>
<td>43</td>
<td>44</td>
</tr>
<tr>
<td>64</td>
<td>43</td>
<td>47</td>
<td>47</td>
<td>44</td>
<td>44</td>
<td>43</td>
<td>44</td>
</tr>
<tr>
<td>128</td>
<td>43</td>
<td>47</td>
<td>47</td>
<td>44</td>
<td>44</td>
<td>43</td>
<td>44</td>
</tr>
<tr>
<td>256</td>
<td>43</td>
<td>47</td>
<td>47</td>
<td>44</td>
<td>44</td>
<td>43</td>
<td>44</td>
</tr>
</tbody>
</table>

---

**Figure 4:** (a) Number of bubble-sort passes for SSE combsort ($N = 256M$, chunk = 128K, $B = 32$); (b) the same for different distributions.
Table 13: Origami parallel speed (M/s) on Skylake-X, 1 GB

<table>
<thead>
<tr>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1C</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>64×64</td>
</tr>
</tbody>
</table>

Table 14: Origami parallel speed (M/s) on dual Xeons, 64 GB

<table>
<thead>
<tr>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1C</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>64×64</td>
</tr>
</tbody>
</table>

SSE for B = 32 despite doubling register size. For small item sizes, using the latest extension set yields better performance; however, this does not hold for larger items. For example, SSE wins on 64+64 key-value pairs, whereas AVX-512 is almost as slow as scalar. This is due to expensive cross 128-bit lane data movements and AVX-512 mask instructions.

**Origami C4 (Cooperative Out-of-Cache).** To inspect our parallel-sort performance, we use a 1-GB input of uniformly random keys and set the number of threads to twice the number of physical cores. This utilizes hyper-threading to maximally load up the CPU pipeline and reduce stall durations. We also set each thread affinity masks to a specific core to ensure better cache utilization.

Table 13 shows the full sort rate for multiple cores and the corresponding speed-up factors over the serial (i.e., single-threaded) sort. For scalar, the improvement is relatively poor because of the slow mtree, as discussed in Section 6.4. Note that the single-threaded sort here uses the fast unrolled bmerge_v3, which cannot be utilized in mtree. Therefore, in this scenario, it is impossible to achieve perfect scaling. For SIMD sorting, the situation is much better. With hyper-threading and usage of load-balancing queues, Origami avoids stragglers and produces excellent multi-core speedup, which reaches 7.7× in one case, but generally stays in the low-to-mid 7s. Both AVX2 and AVX-512 max out at roughly 1B/s with 32-bit keys. As a baseline, std::sort posts around 11M/s regardless of item size, which is 13.5× slower than single-threaded Origami on 32-bit keys, 5.9× on 64-bit, and 3.2× on 64+64.

Not many publicly available SIMD implementations support multi-threaded operation. The only exceptions are [14], which reaches 179M/sec on all 8 cores of our machine using AVX2, and [32], which peaks at 423M/sec using AVX-512, both limited to B = 32. Additional results, shown in Table 14, focus on larger (i.e., 64+64) sorts on dual-socket server CPUs (Intel Xeon E5-2690) with more cores (i.e., 16 total). Because this computer does not support AVX2 or AVX-512, we limit benchmarks to SSE. The table demonstrates an almost linear speed-up, reaching 15.6× on 64-bit keys. Compared to Skylake-X used in prior experiments, each core of this Xeon is 1.8 – 3.5× slower (depending on key size), which explains the generally low numbers in the table.

### 6.6 Database Queries

Our final test involves running SQL queries over existing DBMSs and comparing their performance to that of Origami on the same task. The first dataset comes from IRLbot crawls [20], from which we take the out-degree domain graph G and perform anti-spam ranking on it. The goal is to organize the nodes in descending order of in-degree, but first eliminate all edges from (likely malicious) sources whose out-degree exceeds 1M. This is done by creating two files – a table A containing (src, out-degree) pairs and a table B corresponding to a specific core to ensure better cache utilization.

The SQL query for this task is given by:

```sql
SELECT dst, COUNT(*) as cnt
FROM A INNER JOIN B ON A.src = B.src
WHERE A.outdeg < 1000000
GROUP BY dst
ORDER BY cnt DESC;
```

Our second dataset comes from the database benchmark TPC-H, for which we run standard queries Q1 and Q4. We modify their filters to preserve almost all of the records and test performance on 10, 30, and 100 GB datasets. We use the Xeon server specified above, which has sufficient RAM (i.e., 256 GB of DDR3-1333) to keep all of the data in memory. We also configure the DBMS to avoid spilling the tables to disk. The result is shown in Table 15, including MariaDB (which can run only a single thread per query), MonetDB, and PostgreSQL. As seen in the table, Origami SSE is 37 – 60× faster in single-core scenarios and 30 – 113× faster than the closest competitor in multi-core.

### 7 CONCLUSION

Origami offers a highly optimized mergesort framework that runs at constant speed for different data distributions and gains a nearly linear speed-up in multi-core environments. The proposed components are flexible enough to accommodate future SIMD extension sets as the programmer is only required to write a few small functions with architecture-specific intrinsics, while the remaining algorithms remains unchanged. Future work will examine external-memory sorting, longer key/value pairs, and incorporation of Origami into existing DBMSs.