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 8× faster than prior methods and a branchless streaming merger that achieves up to a 1.5× 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 2× faster than the closest competitor and achieves a nearly perfect speed-up in multi-core environments.

PVLDB Reference Format:
Arif Arman and Dmitri Loguinov. Origami: A High-Performance Mergesort Framework. PVLDB, 15(2): 259-271, 2022.
doi:10.14778/3489496.3489507

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/arif-arman/origami-sort.

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, samplesort, combsort) 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).

Proceedings of the VLDB Endowment, Vol. 15, No. 2 ISSN 2150-8097.
doi:10.14778/3489496.3489507

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, bottlenecks 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 a 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 1.5-2× 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:

$\mathcal{N}$:	Number of items to sort
C:	Number of items fitting into L2 cache (typically $2^{16} - 2^{18}$)
$\mathcal{T}$:	Number of threads (typically double the core count)
$\mathcal{W}$:	Number of items per SIMD register (typically 4-16)
$\mathcal{R}$:	Number of SIMD registers per core (typically 16 or 32)
$\mathcal{B}$:	Size of each item in bits (typically 32, 64, or 128)

Mergesort can be broken down into four phases, which we call $P_1 - P_4$. 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 $\mathcal{N}/C$ sorted lists. A series of k-way merge operations on these blocks, where $k \geq 2$, 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.

(P_1) 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 $[im, (i+1)m)$ are organized in ascending order, where $i = 0, 1, \ldots, C/m - 1$. While mergesort is quite inefficient 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 $m = \mathcal{W}$ and does not explore avenues for minimizing $f(m)$. In contrast, Origami offers novel algorithms that expand m to the entire range $[\mathcal{W}, \mathcal{R}\mathcal{W}]$ 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 $\mathcal{R}$ available registers with data, representing them as an $\mathcal{R} \times \mathcal{W}$ matrix, and developing new SIMD-friendly methods for sorting such rectangular structures entirely within the CPU.

(P_2) 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.

(P_3) 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 $\mathcal{T}$ 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 $\mathcal{N}^2$ 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 $\mathcal{T}$ threads instead of hardcoding an ad-hoc constant (e.g., $k = 4$ in [15], $k = 32$ in [17]).

(P_4) 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 $k = \mathcal{N}/C$ 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., $\mathcal{T}$) 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)

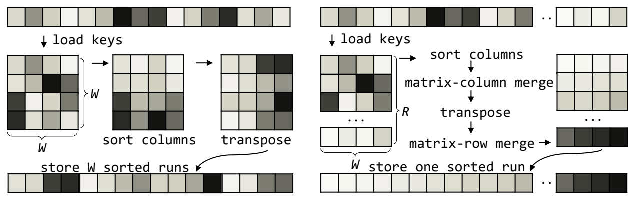
3.1 Principles

Traditional mergesort [8][p. 13] performs $\log_2 \mathcal{N}$ 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*

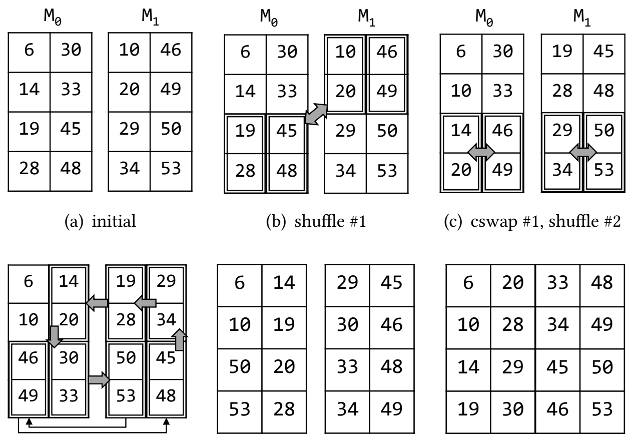
$$tmp = min(x, y);\ y = max(x, y);\ x = tmp;$$

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 $\mathcal{W} = 8$ integers or $\mathcal{W} = 4$ doubles. A single vector instruction, which we indicate by prefix `_mm_`, can then apply a



(a) prior work (b) Origami

Figure 1: Approaches to sorting short lists ($\mathcal{W} = 4$).



(a) initial (b) shuffle #1 (c) cswap #1, shuffle #2

(d) cswap #2, shuffle #3 (e) cswap #3 (f) final

Figure 2: Operation of mcmerge on 4×2 matrices.

particular operation (e.g., min/max) to all $\mathcal{W}$ values at once. As a result, these architectures can perform $\mathcal{W}$ 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 ? in C/C++. For example, `tmp = x < y ? x : y` 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 $\mathcal{W}$ columns in parallel, which is a common technique we call `csort`.

Definition 2 (csort). *Given an $r \times c$ matrix $\mathcal{M}$,* `csort` *orders each column of $\mathcal{M}$ 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 $\mathcal{W}^2$ items in $\mathcal{W}$ SIMD registers and view them as a $\mathcal{W} \times \mathcal{W}$ matrix. Next, they sort the items independently within each column with `csort`, transpose the matrix, and obtain $\mathcal{W}$ sorted lists of $\mathcal{W}$ 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 $\mathcal{W}$ out of $\mathcal{R}$ available registers. With $\mathcal{R}$ either 16 (i.e., SSE) or 32 (i.e., AVX2, AVX-512), this leaves $50 - 94\%$ of the registers unused depending on item size $\mathcal{B}$. Second, this technique loads $\mathcal{W}^2$ keys per iteration, but outputs sorted runs of dismal size $m = \mathcal{W}$. To sort all $\mathcal{W}^2$ keys, another $\log_2 \mathcal{W}$ merge passes are needed in the cache. In contrast, Origami P_1 stuffs data into all $\mathcal{R}$ registers, sorts the whole rectangular $\mathcal{R} \times \mathcal{W}$ matrix using a variety of new algorithms, including a faster transpose, and outputs runs of $m = \mathcal{R}\mathcal{W}$ items. This is illustrated in Figure 1(b) and detailed next.

3.2 Matrix-Column Merge

Origami begins P_1 by loading $\mathcal{R}\mathcal{W}$ input items into $\mathcal{R}$ 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 $\mathcal{R} = 8$). 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 4×2 matrices M_0, M_1 satisfy this definition. We can view the result of the initial `csort` as producing $\mathcal{W}$ matrices sorted in column-major order, each $\mathcal{R} \times 1$ in size. From this point forward, a sequence of specialized operations, which we call *matrix-column merge* (`mcmerge`), lead to progressively larger matrixes 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 $\max(v_j) \leq \min(v_{j+1})$. Note that each v_j is internally sorted. Applying the same operation to M_1, we get additional columns $v_{2c+1}, \ldots, v_{4c}$. In Figure 2(a), $v_0 = (6, 14), v_1 = (19, 28), \ldots, v_7 = (50, 53)$. 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 $\max(v_j) \leq \min(v_{j+1})$ holds for all $j \in [1, 4c]$.

This is illustrated in Figure 2, which runs `mcmerge` over two 4×2 matrices to produce a 4×4 result sorted in column-major order. Part (b) of the figure performs the initial shuffle to exchange v_1 with v_5 and v_4 with v_7, which facilities 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., $M(i,j) \le M(i+1,j)$ 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., $v_2 \leftrightarrow v_4, v_6 \leftrightarrow v_8$, and another `cswap`, we obtain the result in (d). The next shuffle is the most complicated, i.e., $v_2 \to v_8, v_3 \to v_4, v_4 \to v6, v_5 \to v_3, v_6 \to v_2, v_7 \to v_5, v_8 \to v_7$, 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 $\mathcal{W} > 2c$, vectorization allows `mcmerge` to apply simultaneous operations on all $\mathcal{W}/(2c)$ pairs of matrices. For example, $\mathcal{W} = 16$ performs the transformations in Figure 2 on *four pairs of* 4×2 *matrices at no extra cost.*

For back-to-back `mcmerges`, the last reordering (e)→(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)→(f) is unavoidable at the final `mcmerge`; however, for $\mathcal{R} \ge \mathcal{W}$, it can also be omitted by simply renumbering the registers after the transpose.

Origami uses $r = \mathcal{R}$, applying `mcmerge` repeatedly with $c = 1, 2, 4, \ldots$ At some point before reaching $\mathcal{W}/2$, 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,* `mcmerge` *reorders the keys such that* $\max(M_0) \le \min(M_1)$ *and both matrices remain sorted in row-major order.*

This algorithm is derived from modulo 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 2×4 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 $\mathcal{R}$ registers are stored to memory and the process repeats with the next $\mathcal{RW}$ items.

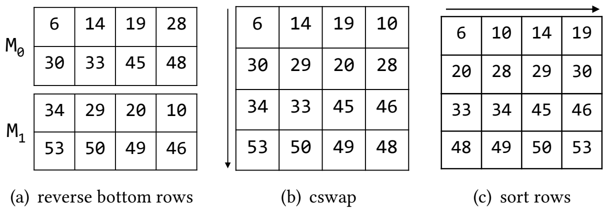


Figure 3: Operation of mrmerge on 2×4 matrices.

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 $\log_2 \mathcal{W}$ levels of diagonal exchanges, where at level $j = 0, 1, \ldots$ rows $(i, i + 2^j)$ exchange $2^j \mathcal{B}$ bits. Present work [6], [14], [26], [33] achieves this through a pair of `shuffle` or `permute` instrinsics 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.

```
// transpose_v0: two shuffles
_v0 = _mm256_shuffle_ps(v0, v1, 0x44);
_v1 = _mm256_shuffle_ps(v0, v1, 0xEE);

// transpose_v1: one shuffle and two blends
v = _mm256_shuffle_ps(v0, v1, 0x4E);
_v0 = _mm256_blend_ps(v0, v, 0xCC);
_v1 = _mm256_blend_ps(v1, v, 0x33);
```

3.5 Optimal Run Length

Note that Origami is flexible enough to allow a variety of run lengths $m \in [\mathcal{W}, \mathcal{RW}]$. 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*

$$f(m) = S_{merge}/S_1(m) - \log_2 m. \tag{1}$$

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 bmerge. Its purpose is to combine two large (i.e., significantly longer than $\mathcal{W}$) sorted sequences into one. With non-trivial input sizes (e.g., over 1 GB), there can be 20-30 passes of bmerge over the data. Therefore, its speed plays an important role in the overall performance of the sort. The main difference between 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 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* $\max(x) \le \min(y)$.

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 $\mathcal{W}$ items out of $2\mathcal{W}$ end up in x, which is then written to the output. In more general cases, illustrated in Algorithm 1, we can load $k \ge 1$ registers from each of A, B and run a sequence of rswaps from any merge network (e.g., odd-even, bitonic) of size $2k$. This produces a sorted sequence of $2k\mathcal{W}$ items, whose lower half $r_0, \dots, r_{k-1}$ is stored to the output C and the upper half $r_k, \dots, r_{2k-1}$ 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 rotate and swap operations; however, it works only for SSE. We extend this method to AVX2/AVX-512 and compare it against mrmerge. The former requires $\mathcal{W}$ stages compared to $\log_2 \mathcal{W} + 1$ in the latter. But rotates are cheaper than transpose, which usually makes this method faster. For AVX-512, however, larger register width produces a significant difference between $\mathcal{W}$ and $\log_2 \mathcal{W} + 1$. In addition, introduction of mask_min and mask_max 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 bmerge

```
Function bmerge (Item *A, *endA, *B, *endB, *C)
    load registers $r_0, \dots, r_{k-1}$ from $A$;    $A$ += $k\mathcal{W}$;
    load registers $r_k, \dots, r_{2k-1}$ from $B$;    $B$ += $k\mathcal{W}$;
    while $A \ne endA$ and $B \ne endB$ do
        rswaps from a merge network of size $2k$;
        store $r_0, \dots, r_{k-1}$ to $C$;    $C$ += $k\mathcal{W}$;
        reload $r_0, \dots, r_{k-1}$ from either $A$ or $B$;
        move $A$ or $B$ forward by $k\mathcal{W}$;
    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 $k\mathcal{W}$ 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:

```
if (*A < *B) { reload $r_0, \dots, r_{k-1}$ from $A$; $A$ += $k\mathcal{W}$; }
else { reload $r_0, \dots, r_{k-1}$ from $B$; $B$ += $k\mathcal{W}$; }
```

This version, which we call 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 combsort (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., scatter, gather, 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 bmerge. For example, [12] argues that the compiler will generate 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 bmerge as we explain next.

The first improvement to bmerge_v0 is to attempt replacing the if block with a sequence of ternary operators to force the compiler to generate cmov instructions during load. We call this version bmerge_v1:

```
flag = *A < *B;
$r_i$ = flag ? load($A + i\mathcal{W}$) : load($B + i\mathcal{W}$); $i \in [0, k-1]$
$A$ += flag ? $k\mathcal{W}$ : 0; $B$ += flag ? 0 : $k\mathcal{W}$;
```

While this works fine for scalar with $k = 1$, the compiler gets confused for $k > 1$ and opts for a branch instead of multiple 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 cmovs to control the pointer to where the data is

Algorithm 2: Origami bmerge

```
Function bmerge_v3 (Item *A, *endA, *B, *endB, *C)
    load $r_0, \ldots, r_{2k-1}$ as in Algorithm 1;
    loadFrom = $A + k\mathcal{W}$; opposite = $B + k\mathcal{W}$;
    while loadFrom ≠ endA and loadFrom ≠ endB do
        rswaps from a merge network of size $2k$;
        store $r_0, \ldots, r_{k-1}$ to $C$;    $C$ += $k\mathcal{W}$;
        flag = *loadFrom < *opposite;
        tmp = flag ? loadFrom : opposite;
        opposite = flag ? opposite : loadFrom;
        loadFrom = tmp;
        load $r_0, \ldots, r_{k-1}$ from loadFrom; loadFrom += $k\mathcal{W}$;
    end
    merge keys left in registers and the unfinished list
```

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

```
src = flag ? $A$ : $B$;
$r_i$ = load(src + $i\mathcal{W}$); $i \in [0, k-1]$
$A$ += flag ? $k\mathcal{W}$ : 0; $B$ += flag ? 0 : $k\mathcal{W}$;
```

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 `cmov`s, 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 `rswap`s 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 `rswap`s 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 `swap`s needed by the merge network. Assume that registers $r_0, \ldots, r_{k-1}$ are loaded from A and $r_k, \ldots, r_{2k-1}$ from B. A $2k$-size merge network will aim to reorder the keys such that $r_0 \leq \ldots \leq r_{2k-1}$, 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 `swap`s that involve the second half, i.e., registers $r_k, \ldots, r_{2k-1}$, which can be proven using the zero-one principle [19]. This novel result allows Origami to reduce the number of `swap`s 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_i.

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 (P_3)

Phase P_2 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 P_3. 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 $\mathcal{T}$ 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 P_3. 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 $\log_2(\mathcal{N}/C)$ to $\log_k(\mathcal{N}/C)$. This reduces memory-bandwidth demand by a factor of $\log_2 k$. One technique [6] is to use a merge tree that resides in the L3 cache and implements an $\mathcal{N}/C$-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 we would like to avoid in P_3, 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 $\mathcal{T}$, 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_4)

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_4, where $\mathcal{T}$ threads cooperatively process the remaining items. Most prior work [6], [15], [17], [30], [32], [33] begins this phase with $k = \mathcal{T}$ streams and uses a binary-search partitioning method that splits each list i into $\mathcal{T}$ segments $[a_{ij}, b_{ij})$, where $i = 1, \ldots, k$ and $j = 1, \ldots, \mathcal{T}$, such that $\sum_{i=1}^{k}(b_{ij} - a_{ij}) = \mathcal{N}/\mathcal{T}$. Other approaches either avoid the issue of load-balancing by shrinking $\mathcal{T}$ as the merge nears the end [14] or do not multi-thread the code at all [26].

To avoid bottlenecking 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 $\mathcal{T} = 8$, prior work would lose half the achievable performance in P_4. 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., $16\mathcal{T}$) 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 defaulted 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., `/arch: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.

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

	SSE			AVX2			AVX-512		
$\mathcal{B}$	$\mathcal{W}$	`v0`	`v1`	$\mathcal{W}$	`v0`	`v1`	$\mathcal{W}$	`v0`	`v1`
32	4	8	6.2	8	24.1	17.6	16	63.9	58.9
64	2	2.25	2	4	9.1	8.27	8	24.5	21.4
64+64	1	–	–	2	4	4	4	9.3	11.1

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 $\mathcal{W}\times\mathcal{W}$ matrix with random keys and performs a billion transposes. Table 1 shows the result across all SIMD extension sets, where $\mathcal{B} = 32$ uses 4-byte keys, $\mathcal{B} = 64$ runs 8-byte keys, and $\mathcal{B} = 64 + 64$ uses 16-byte key-value pairs. Note that gray shading in the table shows the fastest result for each architecture and value of $\mathcal{B}$.

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 1×1 matrix is not needed. In AVX2, the transpose is a single 128-bit diagonal exchange. The two `permute`s 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 `blend`s (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* registers to perform the `mask_blend` intrinsic. For each pair of `blend`s, a mask register is loaded from a general-purpose register with `kmov` (latency 1, throughput 1) that also uses port 5. The good news for `v1` is that the overhead of `kmov` is mostly hidden when executing multiple independent exchanges (i.e., unrolling the transpose loop to operate on multiple matrices simultaneously). For example, with 4× unrolling, `v0` takes 2 cycles per exchange, while `v1` executes in 1.75. This translates to `v1` becoming 11% faster, i.e., 7.1 cycles per transpose vs 8 cycles for `v0`. Because Origami almost always performs $\mathcal{R}/\mathcal{W}$ independent transposes simultaneously (i.e., 8 for $\mathcal{B} = 64 + 64$), `v1` is still the clear choice, even for AVX-512.

Matrix Merges. We next analyze the speed of `mcmerge` and `mrmerge`. The benchmark loads an $\mathcal{R} \times \mathcal{W}$ matrix in $\mathcal{R} = 32$ registers. These $m = \mathcal{RW}$ numbers are set up to consist of $m/\mathcal{K}$ sorted sequences of $\mathcal{K}$ items each, where $\mathcal{K} = \{8, 16, 32, 64\}$. For example, for $\mathcal{R} = 32$ and $\mathcal{W} = 8$, we can represent the matrix as 32 sorted sequences of 8 items or four sequences of 64 items. Define $\mathcal{X} = m/(2\mathcal{K})$ to be the number of pairs of sorted sequences in the matrix. Then, a $\mathcal{K}$-merge operation uses SIMD to produce a binary merge over all $\mathcal{X}$ pairs, replacing them with size-$2\mathcal{K}$ sorted sequences. For example, a merge that begins with $\mathcal{X} = 16$ sorted sequences of length $\mathcal{K} = 16$ produces $\mathcal{X}/2 = 8$ sorted sequences of size $2\mathcal{K} = 32$.

Table 2: Merge speed (B keys/s) in a $32 \times \mathcal{W}$ matrix

$\mathcal{B}$	$\mathcal{K}$	SSE			AVX2			AVX-512		
		$\mathcal{X}$	mc	mr	$\mathcal{X}$	mc	mr	$\mathcal{X}$	mc	mr
32	8	8	10.39	3.75	16	24.11	5.19	32	21.98	-
	16	4	6.26	3.52	8	13.82	5.21	16	16.92	7.63
	32	2	2.81	3.24	4	6.24	5.02	8	7.53	7.23
	64	1	1.58	2.83	2	3.98	4.74	4	5.04	6.71
64	8	4	3.51	1.96	8	4.66	2.36	16	10.98	3.22
	16	2	2.45	1.71	4	3.21	1.99	8	8.46	3.07
	32	1	1.06	1.41	2	1.41	1.83	4	3.53	2.88
	64	–	–	–	1	0.93	1.49	2	2.33	2.68
64+64	8	2	1.44		4	2.08	1.23	8	3.61	1.26
	16	1	1.06		2	1.43	1.08	4	3.06	1.13
	32	–	–	–	1	0.66	0.92	2	1.32	1.03
	64	–	–	–	–	–	–	1	0.89	1.01

Each test performs a billion $\mathcal{K}$-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 – `mcmerge` is up to 4.65× faster than `mrmerge` for small $\mathcal{K}$ 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 $\mathcal{K}$ 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 $\mathcal{K}$ affects the number of merge-network `swaps`, the remaining elements of the algorithm have constant cost, i.e., exactly $\mathcal{R}/2$ reverses and $2\mathcal{R}/\mathcal{W}$ transposes, irrespective of $\mathcal{K}$.

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 $\mathcal{W}$ within two registers. Table 3 compares the different implementations of `rswap` for various $\mathcal{B}$. It shows that for both SSE and AVX2, running $\mathcal{W}$ 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 `permute`. 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 last row is smaller since 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.

Table 3: rswap speed (B keys/s) for a size-$2\mathcal{W}$ merge

	SSE			AVX2			AVX-512		
$\mathcal{B} \rightarrow$	32	64	64+64	32	64	64+64	32	64	64+64
bitonic	2.34	1.38	1.81	2.93	1.14	0.76	3.31	1.52	0.51
rotate	4.26	1.81	1.81	3.61	1.31	1.01	3.38	1.56	0.69
mr	2.21	1.19	1.81	2.29	1.21	0.78	5.61	2.27	0.74

Table 4: Speed-up factors over the best speed from Table 3 for running simultaneous independent rswaps (*unrolling*)

$\mathcal{B} \rightarrow$	32			64			64 + 64		
Unroll →	2×	3×	4×	2×	3×	4×	2×	3×	4×
SSE	1.57	1.90	2.07	1.57	2.09	2.45	1.63	1.94	2.15
AVX2	1.59	2.07	2.27	1.74	2.33	2.72	1.93	2.43	2.86
AVX-512	1.49	1.49	1.49	1.38	1.44	1.47	1.54	1.68	1.68
Scalar	1.05	1.11	1.09	1.11	1.12	1.06	1.38	1.36	1.35

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 hopping, 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 $\mathcal{B} = 32$ and $\mathcal{B} = 64$, 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 this 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., $\mathcal{B} = 64 + 64$) 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

$\mathcal{B}$	k	Scalar		SSE		AVX2		AVX-512		Scalar	SSE			AVX2			AVX-512		
		`v0`	`v3`	`v0`	`v3`	`v0`	`v3`	`v0`	`v3`	2×	2×	3×	4×	2×	3×	4×	2×	3×	4×
32	1	308	575	1402	1788	1966	2006	2639	2770	830	2441	2430	2338	3068	3552	3292	3422	3796	3776
	2	591	1136	1893	2284	1821	1888	2424	2503	1074	2324	2282	2224	2305	2203	2120	2526	2542	2514
	3	791	1213	1701	1904	1526	1537	1877	1877	986	1926	1893	1796	1623	1581	1592	1905	1901	1868
	4	922	1327	1869	2016	1651	1662	1892	1904	1002	1935	1908	1795	1668	1627	1596	1903	1850	1845
64	1	309	569	616	931	686	698	1042	1065	805	1396	1396	1316	1176	1355	1278	1430	1495	1488
	2	573	1016	864	912	671	674	988	988	1058	1422	1397	1260	864	836	804	1041	1059	1066
	3	814	1133	822	828	611	613	779	781	916	1101	1068	980	606	598	590	799	811	808
	4	961	1270	914	966	609	611	789	790	1004	1048	1013	954	622	620	608	804	811	809
64+64	1	263	481	290	503	489	557	342	353	542	844	872	697	961	1113	962	547	561	546
	2	448	674	526	761	520	520	319	327	531	1030	1017	928	834	831	703	370	370	359
	3	494	544	671	780	498	499	265	269	448	967	922	784	559	585	545	288	279	275
	4	463	528	764	926	557	567	287	290	371	955	907	747	579	539	515	297	289	289

Table 6: In-cache bmerge speed (M/s); $\mathcal{B} = 32$

Scalar			AVX2			AVX-512			
[13]	[16]	`v3`	[14]	[26]	`v3`	[30]	[32]	[33]	`v3`
465	481	1327	720	1995	3552	395	2997	1849	3796

Table 7: Single-threaded memory throughput (GB/s)

bmerge_v3					
$\mathcal{B}$	Scalar	SSE	AVX2	AVX-512	`memcpy`
32	4.83	8.82	10.26	11.99	
64	7.39	9.75	8.69	10.84	10.81
64+64	7.53	11.62	11.58	8.22	

Table 8: Single-threaded mtree speed (M/s); $\mathcal{B} = 32$

	SSE			AVX2			AVX-512		
k	`v0`	`v1`	`v2`	`v0`	`v1`	`v2`	`v0`	`v1`	`v2`
4	843	1048	1101	986	987	1093	1244	1303	1482
8	521	627	694	617	644	718	843	858	955
16	379	456	501	465	477	528	628	638	689
32	292	346	396	364	366	413	484	488	549
64	240	284	303	303	302	331	398	398	433
128	202	237	251	251	253	278	331	336	361
256	174	199	214	211	212	235	269	267	301
512	151	171	190	192	190	203	230	235	259
1024	133	147	165	166	168	178	196	203	223

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 P_2 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 – 2.85×.

6.4 Out-of-cache Merge

bmerge. During P_2, 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 $12 \times 128/61 = 25.2$ 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 3.8×. 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

Table 9: Chunked speed in C_1 (M/s); $\mathcal{N}$ = 128K, $\mathcal{B}$ = 32

chunk	SSE		AVX2			AVX-512			
size c	[15]	C_1	[14]	[26]	C_1	[30]	[32]	[33]	C_1
8	809	4416	7043	2987	7462	913	-	-	-
16	562	3137	978	1622	5599	534	1740	4615	5682
32	499	2082	493	964	4198	372	1545	2921	4412
64	454	1036	302	659	2377	288	923	1519	2642
128	435	653	212	494	1431	233	697	889	1976
256	362	421	167	391	1053	197	558	633	1457
512	–	–	137	333	757	167	463	467	1184
1K	–	–	–	–	–	117	399	380	951

Table 10: Chunked speed in C_2 (M/s); $\mathcal{N}$ = 128K, $\mathcal{B}$ = 32

chunk	SSE		AVX2			AVX-512			
size c	[15]	C_2	[14]	[26]	C_2	[30]	[32]	[33]	C_2
8	809	4416	7043	2987	7462	913	-	-	-
16	562	3137	978	1622	5599	534	1740	4615	5682
32	499	2082	493	964	4198	372	1545	2921	4412
64	454	*1089	302	659	2377	288	923	1519	2642
128	435	729	212	494	1431	233	697	889	1976
256	362	563	167	391	1053	197	558	633	1457
512	319	458	137	333	*767	167	463	467	1184
1K	308	386	114	292	650	117	399	380	951
2K	286	333	98	259	545	90	341	303	750
4K	266	294	87	232	474	74	306	260	*646
8K	222	263	78	208	414	62	279	235	557
16K	204	237	70	190	370	58	257	211	480
32K	189	217	63	175	326	50	236	189	425
64K	161	198	58	161	297	44	217	172	378
128K	150	183	54	150	257	40	203	157	335

within each 4-way node. It should be noted that `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 `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 `memcpy` bandwidth for a given number of threads $\mathcal{T}$.

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_i. The source code is available from [1].

Origami C_1 (Tiny Sorters). We now compare the C_1 performance of Origami against that of prior work. We first define a *chunk-c sort* to be an algorithm that sorts every c continuous items in an input of size $\mathcal{N}$, where $c \le \mathcal{N}$. Note that this section uses c up to $m = \mathcal{RW}$, 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., $\mathcal{N} = \mathcal{C}$) and measures the chunked sort speed for different c. Note that Origami loads $\mathcal{RW}$ keys into its matrix and sorts each group of $\min(\mathcal{RW}, c)$ items in registers. It begins the sort with `mcmerge` and then, if necessary, moves to `mrmerge` at the optimal switch point from Table 2. Since none of the prior work sorts more than $\mathcal{W}$ keys in register, achieving $c > \mathcal{W}$ requires them to run `bmerge` from phase P_2.

Table 9 shows the result on 32-bit keys (most prior implementations do not support larger $\mathcal{B}$). 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 5× faster speed in third row for SSE, 4.3× for AVX2, and 1.5× 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 − 2.4× advantage over the nearest competitor. The extra cache traffic and their `bmerge` 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 $\mathcal{B} = 32$ and 1590M/s for $\mathcal{B} = 64$. Note that this performance more than doubles that of [15] in the first row of Table 9. Dealing with $\mathcal{B} = 64+64$ 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 `cmovs`, which is optimal. This results in a 20% speed-up over operator overload.

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 `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 `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 `bmerge_v3` runs at a constant rate.

Table 10 shows results for C_2, where the cells marked with asterisks denote the first time `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

Table 11: Chunked speed in C_3 (M/s); $\mathcal{N}$ = 256M, $\mathcal{B}$ = 32

chunk	SSE		AVX2			AVX-512			
size c	[15]	C_3	[14]	[26]	C_3	[30]	[32]	[33]	C_3
128 K	63	176	53	139	228	40	198	140	295
256 K	61	147	47	128	210	33	184	130	269
512 K	59	138	44	120	195	30	172	113	249
1 M	57	131	41	109	183	28	160	102	232
2 M	55	124	39	92	174	25	150	95	216
4 M	54	118	37	81	168	23	140	88	203
8 M	52	112	35	77	162	21	131	83	191
16 M	50	107	33	73	153	20	122	78	181
32 M	48	102	32	70	145	19	115	72	172
64 M	47	98	30	67	138	18	109	69	163
128 M	45	95	29	65	132	17	103	66	156
256 M	44	91	28	63	126	17	97	64	149

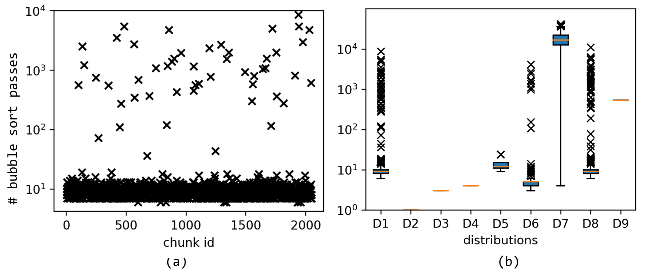


Figure 4: (a) Number of bubble-sort passes for SSE combsort ($\mathcal{N}$ = 256M, *chunk* = 128K, $\mathcal{B}$ = 32); (b) the same for different distributions.

than $\mathcal{RW}$ 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 $m > \mathcal{RW}$ sorted items in each iteration of P_1 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 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 C_3 (Independent Out-of-Cache). We now inspect Origami performance in checkpoint C_3 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× 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

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

	$\mathcal{B}$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{D}_6$	$\mathcal{D}_7$	$\mathcal{D}_8$	$\mathcal{D}_9$
Scalar	32	47	52	52	52	47	47	49	47	47
	64	43	47	47	47	43	44	45	43	44
	64+64	25	27	27	27	25	25	25	25	25
SSE	32	91	90	90	90	91	91	92	91	90
	64	50	49	49	50	50	50	50	50	49
	64+64	35	34	34	34	35	35	35	35	34
AVX2	32	126	126	125	124	126	126	127	126	125
	64	48	48	48	48	48	48	48	48	47
	64+64	34	34	34	34	34	34	35	34	34
AVX-512	32	149	145	145	145	148	149	149	149	146
	64	65	63	64	63	64	64	65	64	63
	64+64	27	26	26	26	26	27	26	26	26

significantly affecting the overall sort speed. To test sensitivity of combsort to input workloads, we run additional tests with the following distributions:

($\mathcal{D}_1$) *Uniformly random*, generated by Mersenne Twister
($\mathcal{D}_{2-4}$) *All the same, sorted, and reverse-sorted*
($\mathcal{D}_5$) *Almost sorted*, where every 7th key is set to `KEY_MAX`
($\mathcal{D}_6$) *Pareto*, generated as $\min(\text{ceil}(\beta(1/(1-u)-1)), 10000)$, where $\beta = 7$ and $u \sim \text{uniform}[0,1]$
($\mathcal{D}_7$) *Bursts of same keys*, where the length of each subsequence is drawn from $\mathcal{D}_6$ and the key from $\mathcal{D}_1$
($\mathcal{D}_8$) *Random shuffle*, generated by randomly permuting $\mathcal{D}_7$
($\mathcal{D}_9$) *Fibonacci*, wrapped around when it overflows $\mathcal{N}$

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 $\mathcal{D}_1 - \mathcal{D}_9$. 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 2× speed drop when going from $\mathcal{B}$ = 32 to $\mathcal{B}$ = 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 `memcpy` bandwidth. Thus, the drop in speed is only minor. For vector registers, the speed reduction is 1.8 – 2.6× since not only is $\mathcal{W}$ 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 $\mathcal{W}$ (e.g., AVX2 is only 1.33× faster than

Table 13: Origami parallel speed (M/s) on Skylake-X, 1 GB

	$\mathcal{B}$	Speed (M/s)				Speed-up		
		1C	2C	4C	8C	2C	4C	8C
Scalar	32	47	74	147	282	1.6	3.1	6.0
	64	43	75	149	273	1.7	3.5	6.4
	64+64	25	44	84	162	1.8	3.4	6.5
SSE	32	91	179	352	687	2.0	3.9	7.6
	64	50	94	185	361	1.9	3.7	7.2
	64+64	35	70	139	260	2.0	4.0	7.4
AVX2	32	126	248	495	950	2.0	3.9	7.5
	64	48	95	189	369	2.0	3.9	7.7
	64+64	34	70	137	254	2.1	4.0	7.5
AVX-512	32	149	286	565	1062	1.9	3.8	7.1
	64	65	122	242	462	1.9	3.7	7.1
	64+64	27	53	105	197	2.0	3.9	7.3

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

	$\mathcal{B}$	Speed (M/s)					Speed-up			
		1C	2C	4C	8C	16C	2C	4C	8C	16C
SSE	32	38	76	153	301	581	2.0	4.0	7.9	15.1
	64	16	32	63	118	249	2.0	4.0	7.5	15.6
	64+64	10	19	38	78	150	1.9	3.9	7.9	15.1

SSE for $\mathcal{B} = 32$ 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_blend` intrinsics.

Origami C_4 (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 $\mathcal{B} = 32$. Additional results, shown in Table 14, focus on larger (i.e., 64-GB) 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 keys size), which explains the generally low numbers in the table.

Table 15: Query runtime (sec) on dual Xeons

	Size	Single core		All cores		
	(GB)	MariaDB	Origami	MonetDB	PostgreSQL	Origami
IRLbot	2	201.6	4.8	45.2	108.3	0.4
	8	777.8	20.9	162.9	357.1	1.7
TPC-H Q1	10	129.1	2.1	6.8	32.1	0.2
	30	374.5	7.4	20.1	96.5	0.6
	100	1248.4	27.5	76.4	326.1	2.1
TPC-H Q4	10	30.3	0.9	3.0	15.6	0.1
	30	105.2	2.6	8.7	56.6	0.2
	100	417.7	9.1	29.5	222.6	0.7

6.6 Database Queries

Our final test involves running SQL queries over existing DBMS 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 $\mathcal{G}$ 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 – table A containing (src, out-degree) pairs and table B consisting of (src, dst) out-edges. All numbers are 4-byte integers, the dataset size is 2 or 8 GB, and the load phase is excluded from timing. The SQL query for this task is given by:

```
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 DBMS.

REFERENCES

[1] Arif Arman and Dmitri Loguinov. 2021. Origami Souce Code. Retrieved December 13, 2021 from https://github.com/arif-arman/origami-sort

[2] Kenneth E. Batcher. 1968. Sorting networks and their applications. In *Spring Joint Computer Conference*. 307–314.

[3] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. 1991. A comparison of sorting algorithms for the connection machine CM-2. In *ACM SPAA*. 3–16.

[4] Bronislava Brejová. 2001. Analyzing variants of Shellsort. *Inform. Process. Lett.* 79, 5 (2001), 223–227.

[5] Jing-Chao Chen. 2006. Efficient sample sort and the average case analysis of PEsort. *Theoretical computer science* 369, 1-3 (2006), 44–66.

[6] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. *VLDB Endowment* 1, 2 (2008), 1313–1324.

[7] Y.B. Chiang. 2001. Sorting networks using k-comparators. (2001).

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 1990. *Introduction to Algorithms* (1st ed.). The MIT Press.

[9] Intel Corporation. 2016. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[10] Intel Corporation. 2016. Intel VTune Profiler. Retrieved October 15, 2021 from https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html

[11] Intel Corporation. 2021. Intel Memory Latency Checker. Retrieved October 15, 2021 from https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html

[12] Amr Elmasry, Jyrki Katajainen, and Max Stenmark. 2012. Branch mispredictions don't affect mergesort. In *International Symposium on Experimental Algorithms*. Springer, 160–171.

[13] Oded Green. 2014. When merging and branch predictors collide. In *IA3@ SC*. 33–40.

[14] Kaixi Hou, Hao Wang, and Wu-chun Feng. 2015. Aspas: A framework for automatic simdization of parallel sorting on x86-based many-core processors. In *ACM on International Conference on Supercomputing*. 383–392.

[15] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. 2007. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 189–198.

[16] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. 2014. Faster set intersection with SIMD instructions by reducing branch mispredictions. *VLDB Endowment* 8, 3 (2014), 293–304.

[17] Hiroshi Inoue and Kenjiro Taura. 2015. SIMD-and cache-friendly algorithm for sorting an array of structures. *VLDB Endowment* 8, 11 (2015), 1274–1285.

[18] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *VLDB Endowment* 2, 2 (2009), 1378–1389.

[19] Donald E. Knuth. 1998. *The Art of Computer Programming, Vol. III* (2nd ed.). Addison-Wesley.

[20] Hsin-Tsang Lee, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov. 2008. IRLbot: Scaling to 6 Billion Pages and Beyond. In *WWW*. 427–436.

[21] Kathy J. Liszka and Kenneth E. Batcher. 1992. A Modulo merge sorting network. In *Symposium on the Frontiers of Massively Parallel Computation*. 164–165.

[22] M. Douglas McIlroy. 1999. A killer adversary for quicksort. *Software: Practice and Experience* 29, 4 (1999), 341–344.

[23] Daniel G. O'connor and Raymond J. Nelson. 1962. Sorting system with nu-line sorting switch. US Patent 3,029,413.

[24] Tim Peters. 2021. Timsort. Retrieved October 15, 2021 from https://en.wikipedia.org/wiki/Timsort

[25] Orestis Polychroniou and Kenneth A Ross. 2014. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *ACM SIGMOD*. 755–766.

[26] Siddharth Santurkar. 2016. Speed up Sort in Peloton using AVX2. Retrieved October 15, 2021 from https://github.com/sid1607/avx2-merge-sort

[27] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *ACM SIGMOD*. 351–362.

[28] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *ACM SIGSAC Conference on Computer and Communications Security*. 753–768.

[29] Paul Vitanyi. 2007. Analysis of sorting algorithms by Kolmogorov complexity (A survey). In *Entropy, Search, Complexity*. Springer, 209–232.

[30] Alex Watkins and Oded Green. 2018. A Fast and Simple Approach to Merge and Merge Sort using Wide Vector Instructions. In *IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 37–44.

[31] Wikipedia. 2021. Median of Medians. Retrieved October 15, 2021 from https://en.wikipedia.org/wiki/Median_of_medians

[32] Tian Xiaochen, Kamil Rocki, and Reiji Suda. 2013. Register level sort algorithm on multi-core SIMD processors. In *Workshop on Irregular Applications: Architectures and Algorithms*. 1–8.

[33] Zekun Yin, Tianyu Zhang, André Müller, Hui Liu, Yanjie Wei, Bertil Schmidt, and Weiguo Liu. 2019. Efficient Parallel Sort on AVX-512-Based Multi-Core and Many-Core Architectures. In *IEEE HPCC/SmartCity/DSS*. 168–176.