SimdMinimizers: Computing random minimizers, *fast*

Ragnar Groot Koerkamp
ETH Zurich, Zurich, Switzerland

Igor Martayan
Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, Lille, France

Abstract

Motivation. Because of the rapidly-growing amount of sequencing data, computing *sketches* of large textual datasets has become an essential preprocessing task. These sketches are typically much smaller than the input sequences, but preserve sufficient information for downstream analysis. *Minimizers* are an especially popular sketching technique and used in a wide variety of applications. They sample at least one out of every w consecutive k-mers. As DNA sequencers are getting more accurate, some applications can afford to use a larger w and hence sparser and smaller sketches. And as sketches get smaller, their analysis becomes faster, so the time spent sketching the full-sized input becomes more of a bottleneck.

Methods. Our library `simd-minimizers` implements a random minimizer algorithm using SIMD instructions. It supports both AVX2 and NEON architectures. Its main novelty is two-fold. First, it splits the input into 8 chunks that are streamed over in parallel through all steps of the algorithm. This is enabled by using the completely deterministic *two-stacks* sliding window minimum algorithm, which seems not to have been used before for finding minimizers.

Results. Our library is up to 9.5× faster than a scalar implementation of the *rescan* method when $w = 5$ is small, and 4.5× faster for larger $w = 19$. Computing *canonical* minimizers is only around 50% slower than computing forward minimizers, and around 16× faster than the existing implementation in the `minimizer-iter` crate. Our library finds all (canonical) minimizers of a 3.2Gbp human genome in 4.1 (resp. 6.0) seconds.

2012 ACM Subject Classification Theory of computation → Sketching and sampling; Applied computing → Bioinformatics

Keywords and phrases Minimizers; Randomized algorithms; Sketching; Hashing

Supplementary Material *Software*: https://github.com/rust-seq/simd-minimizers

Funding *Ragnar Groot Koerkamp*: ETH Research Grant ETH-1721-1 to Gunnar Rätsch.
Igor Martayan: ENS Rennes Doctoral Grant.

1 Introduction

Minimizers were simultaneously introduced by [27] and [26] as a method to sample short strings of fixed length k, called k-mers or k-grams, for the purpose of fingerprinting and comparing large textual documents such as genomic sequences. This sampling method plays a central role in bioinformatics for the high-throughput analysis of DNA sequencing data and is a fundamental building block for many related tasks such as indexing [25, 18], counting [3, 20], aligning [16, 11], or assembling [4, 1] genomic sequences.

Minimizers are defined as follows: given a window W of w consecutive k-mers, the minimizer of W is the smallest k-mer according to some order. In practice, the order is often pseudo-random by hashing the k-mers. The *density* of minimizer schemes is the fraction of sampled k-mers, and in recent years there have been a number of papers on methods with lower density than random minimizers [30, 24, 8, 7, 6]. While these contributions have narrowed the gap to an optimal-density sampling scheme [14], none of them focused on improving the computation time of minimizers.

Problem statement. We aim to solve the following problem as fast as possible: given a bitpacked representation of a sequence of ACGT DNA characters, compute the positions of all (canonical) random minimizers.

Contributions. This work introduces a carefully optimized algorithm based on SIMD instructions to compute the minimizers a genomic sequence, as well as a method to preserve consistency with the reverse-complement of the sequence while maintaining a high throughput. This algorithm is built around a few main building blocks that each are vectorized to process $L = 8$ lanes in parallel using AVX2 or NEON instructions. See Figure 1 for an overview. Each of the parts corresponds to a subsection of Section 3.

1. A method for iterating the characters of L chunks of a sequence in parallel.
2. An SIMD implementation of ntHash [22, 13], a pseudo-random rolling hash function for k-mers.
3. Computing the minimum in a sliding window, based on the *two-stacks* method [9, 28].
4. Canonical minimizers based on *refined minimizers* [23] to decide the *strand* of a sequence.
5. Collection and deduplication of the L parallel streams into unique minimizer positions.

Results. Our method is 4.5× (for large $w = 19$) to 9.5× (for small $w = 5$) times faster than the fastest non-SIMD algorithm for computing forward minimizers. For canonical minimizers, we only compare against a simple implementation and find a 16× speedup. As a result, we can compute the minimizers of a human genome in 4.1 seconds, and the canonical minimizers in 6.0 seconds. We also adapt our method to support generic plain-text ASCII input ($|\Sigma| = 256$), which is slightly (37%) slower due to the larger input characters.

Software. A Rust implementation of our method is publicly available at https://github.com/rust-seq/simd-minimizers.

2 Preliminaries

Bitpacking In this work, we assume that the input sequence is over the DNA alphabet $\Sigma = \mathsf{ACTG}$ and that each letter is encoded using two bits: $\mathsf{A} = \mathtt{00}, \mathsf{C} = \mathtt{01}, \mathsf{T} = \mathtt{10}, \mathsf{G} = \mathtt{11}$. This encoding can easily be obtained from the ASCII representation by applying a mask: `(c >> 1) & 3`. Additionally, we assume that the whole sequence is bitpacked using this 2-bit encoding, which can be done as a preprocessing step on the input if necessary. Non-ACTG characters have to be handled during this preprocessing as well, and could be skipped,

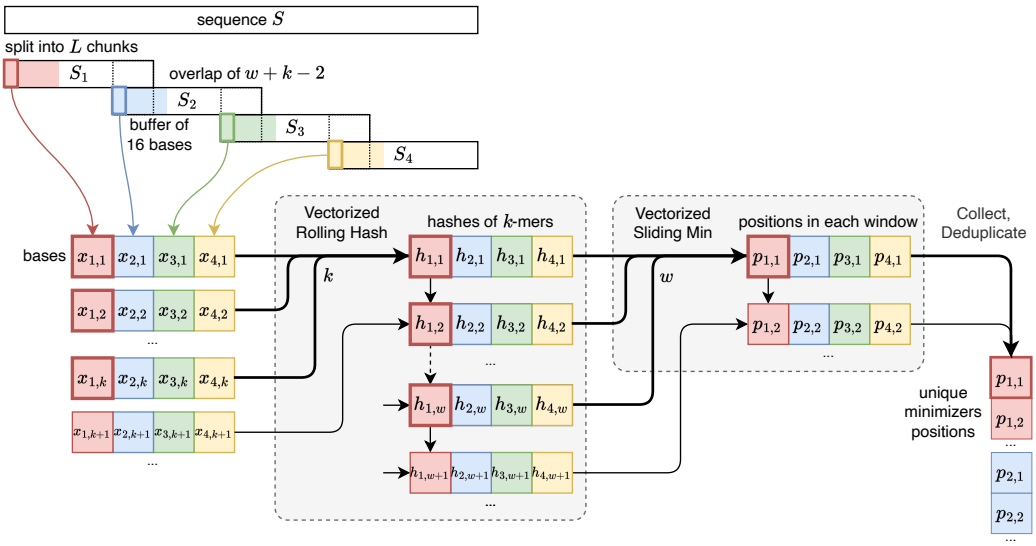

Figure 1 High-level view of the vectorized computation of minimizers. The sequence is first split into L chunks (in this example $L = 4$) that are processed in parallel. The bases of each chunk are gathered in a SIMD register (one lane for each chunk) and passed to a vectorized rolling hash function that computes a hash for the k-mers in each lane. The position of the minimum is then computed over a sliding window of w vectors of hashes. The positions are finally reordered to match the order of the original sequence and deduplicated to have a unique occurrence of each position. A bold red outline indicates the bases of the first k-mer and the hashes in the first window.

converted to A, or the input sequence could be split at these points. We assume that the hardware is little-endian, and that the integer value of a sequence $x_0x_1x_2\ldots x_{k-1}$ is given by $\sum_{i=0}^{k-1} x_i \cdot 4^i$.

Minimizers. Given parameters w and k, a *window* W of length $\ell = w + k - 1$ contains w consecutive k-mers. The *minimizer* of the window is the smallest k-mer in the window. For *random* minimizers, k-mers are ordered by a pseudo-random order, usually given by comparing hashes of the k-mers. In case of ties, the leftmost smallest k-mer is chosen.

Our goal is to compute the absolute position of the minimizer of every window W in the input text. Since adjacent windows often have the same k-mer as minimizer, we only want each position to be listed once in the output.

Canonical minimizers. Because DNA is double-stranded and most sequencing technologies do not distinguish these two strands, genomic sequences have an additional constraint: a sequence and its *reverse-complement* (the reversed sequence of complementary bases A $\leftrightarrow$ T and C $\leftrightarrow$ G) should be considered identical. To satisfy this constraint, *canonical minimizers* should return the same set of k-mers regardless of the strandedness of the input. Specifically, if the canonical minimizer of a window W is at position p, then the canonical minimizer of the reverse-complement $\overline{W}^r$ of W should be at position $|W| - k - p = w - 1 - p$. In practice, canonical minimizers are often overlooked as an implementation detail and most existing methods simply compare canonical k-mers, computed as $x^c = \min(x, \overline{x}^r)$, which gives a weaker guarantee [21].

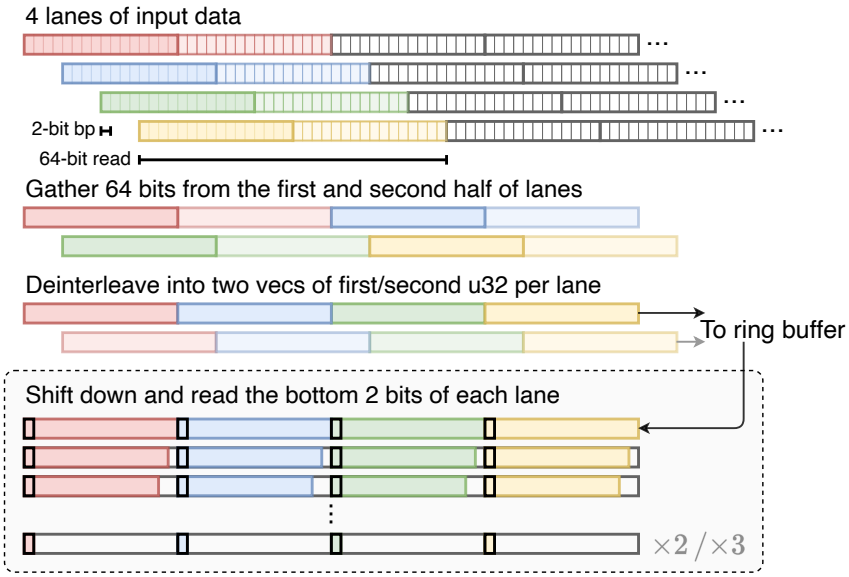

Figure 2 A visualization of the gathering of bases. Fat outlined boxes indicate 32-bit integers, while thin boxes are 2-bit basepairs (bp) in little-endian order with least-significant digits on the left. For the purposes of this and following visualizations, each SIMD register is shown as a horizontal bar with $L = 4$ 32-bit lanes, as opposed the $L = 8$ lanes in actual AVX2 code. Progress over time goes from top to bottom, while memory at each point in time is shown as staggered SIMD registers. The input is L chunks of memory, each shown in its own color. The highlighted box is instantiated two (or three) times, once for the character entering the k-mer, once for the character leaving the k-mer, and optionally once for the character leaving the window.

3 Vectorized SIMD algorithm

SIMD. The goal of our method is to apply SIMD instructions to speed up the minimizer computation. With 256-bit AVX2 instructions for example, we can process $L = 8$ *lanes* of 32-bit values at a time. At first, one could try to use this to compute the hash of L consecutive minimizer values at the same time, and then to compute the minimizer of the L new windows all at once. Unfortunately, this is tricky due to the sequential nature of rolling hashing and sliding window minima.

Chunks. Instead of processing consecutive k-mers in parallel, we choose to split the input sequence into L equally long *chunks* and we process those chunks in parallel. This way, we compute one hash of each chunk in parallel, and then compute one minimizer position of a window of each chunk in parallel as well. We let adjacent chunks overlap by $\ell - 1$ characters, so that each window is fully contained in exactly one chunk. The total number of windows may not be divisible by L. In that case, we process the remaining windows in a scalar (non-vectorized) loop. For simplicity, we omit that case from the code snippets.

Overview. On a high level, the method consists of a few steps, each explained in detail in its own subsection. 1) First, we iterate the 2-bit bases of each chunk in parallel. These are returned as a SIMD vector of 32-bit values with the lower 2 bits indicating the value of the base in each chunk. 2) We then use these bases to compute the ntHash rolling hash. In each step, there is a base *entering* the k-mer and also a base *leaving* the k-mer. Thus, we modify the iterator over bases to return *two* streams at a time, where one has the bases entering each k-mer and the other is delayed by $k - 1$ steps and contains the k-mer leaving each k-mer.

```rust
fn iter_bp_in_out(k: usize, w: usize, delay: usize, seq: &[u8]) -> impl Iterator<(u32x8, u32x8)> {
    let overlap = k + w - 2; // Overlap between adjacent lanes.
    let num_windows = 4 * seq.len() - overlap;
    let n = (num_windows / 8) / 4 * 4; // Windows per lane, rounded down to multiple of 4.
    let byte_len = n / 4; // Number of bytes per lane after packing.
    let offsets_0_4: u64x4 = [0 * byte_len, 1 * byte_len, 2 * byte_len, 3 * byte_len];
    let offsets_4_8: u64x4 = [4 * byte_len, 5 * byte_len, 6 * byte_len, 7 * byte_len];
    // Each buffer entry is a u32 with 16bp. Rounded to power of 2 for efficiency.
    let buf_len = (delay / 16 + 2).next_power_of_two();
    let buf_mask = buf_len - 1;
    let mut buf = vec![S::ZERO; buf_len];
    let mut cur_in  = S::ZERO;
    let mut cur_out = S::ZERO;
    let mut write_idx = 0;                                // Write idx of 'in' stream.
    let mut read_idx = (buf_len - delay / 16) % buf_len; // Read idx of 'out' stream.
    (0..lane_length + overlap).map(|i| {
        if i % 16 == 0 {
            // Every 32 iterations, read the next 64bits = 32bp from each lane.
            if i % 32 == 0 {
                // `splat` copies `i/4` to each lane.
                let idx_0_4 = offsets_0_4 + u64x4::splat(i / 4); // Indices to read.
                let idx_4_8 = offsets_4_8 + u64x4::splat(i / 4);
                let u64_0_4: u32x8 = intrinsics::gather(seq, idx_0_4); // Gather the vals.
                let u64_4_8: u32x8 = intrinsics::gather(seq, idx_4_8);
                // Deinterleave into the current 16bp, and buffer the next 16 bp.
                (buf[write_idx], buf[write_idx + 1]) = intrinsics::deinterleave(u64_0_4, u64_4_8);
                cur_in = buf[write_idx];
            } else {
                cur_in = buf[write_idx + 1];
                write_idx = (write_idx + 2) % buf_mask;
            }
        }
        if i % 16 == delay % 16 {
            cur_out = buf[read_idx];
            read_idx = (read_idx + 1) & buf_mask;
        }
        let bps_in  = cur_in  & u32x8::splat(3); // Extract the low 2 bits of each lane.
        let bps_out = cur_out & u32x8::splat(3);
        cur_in  = cur_in  >> u32x8::splat(2)   ; // Shift remaining bits to the right.
        cur_out = cur_out >> u32x8::splat(2);
        (bps_in, bps_out)                        // Yield the 2-bit basepairs.
    })
}
```

Listing 1 Code to that returns an iterator over the bases of 8 chunks of the input sequence. Also returns a *delayed* stream by `delay=k-1` positions with the character leaving the current k-mer.

3) We then compute the sliding window minimum of the k-mer hashes. In order to efficiently break ties and return positions, we only use the upper 16 bits of each hash, and store the position in the lower 16 bits. 4) The algorithm can optionally be extended to return *canonical* minimizers. In that case, we compute both a forward and reverse-complement minimizer, and choose the one to return based on the fraction of GT characters in the window. This requires the iterator over the input to return an additional stream delayed by $\ell - 1$ positions, to decrease the GT count as needed. 5) Lastly, the minimizers from all chunks are collected L at a time, deduplicated, and written to per-chunk output vectors, that are finally merged into a single vector of unique minimizer positions.

```
// `in_out` iterates over a pair of 2-bit bp. `in` is k-1 positions ahead of `out`.
fn nthash(k: usize, in_out: impl Iterator<(u32x8, u32x8)>) -> impl Iterator<u32x8> {
    // Each 128-bit half has a copy of the 4 32-bit hashes.
    let table: u32x8 = [F[0], F[1], F[2], F[3], F[0], F[1], F[2], F[3]];
    let table_rot: u32x8 = table.map(|h| h.rotate_left(k - 1));
    let mut h = u32x8::ZERO;
    in_out.by_ref().take(k - 1).for_each(|(in_bp, _)| {
        h = ((h << 1) | (h >> 31)) ^ intrinsics::table_lookup(table, in_bp);
    });
    in_out.map(|(in_bp, out_bp)| {
        let h_out = ((h << 1) | (h >> 31)) ^ intrinsics::table_lookup(table, in_bp);
        h = h_out ^ intrinsics::table_lookup(table_rot, out_bp);
        h_out // Yield the hash.
    })
}
```

Listing 2 Vectorized implementation of ntHash. The input is an iterator over pairs of characters being added and removed from the k-mer in each chunk as they slide over the input string. Output is an iterator over the ntHashes of all k-mers. The code first processes the first $k-1$ characters to initialize the rolling hash, and then repeatedly adds and removes on character at a time.

3.1 Gathering bases

The first step of our algorithm is to read the actual bases. Since memory access instructions are relatively slow[1] compared to the SIMD operations we do on them, we cannot afford to read from each chunk one base or one byte at a time. Instead, we read 64 bits at a time from each chunk, spread over two SIMD `gather` instructions that each read four chunks, as shown in Figure 2 and Listing 1. These 64-bit reads are then shuffled into two SIMD vectors that contain the upcoming 32 bits in each chunk, and the next 32 bits, and these are written to a buffer that contains at least k bases per chunk (or ℓ for canonical minimizers).[2] The `gather(seq, idx)` function collects 64-bit elements starting at bytes `seq[idx[0]]` to `seq[idx[3]]`. It can be implemented using `_mm256_i64gather_epi64` in AVX2, but has no equivalent instruction in NEON.

To get the individual characters, the SIMD element with the next 32 bits for each lane is read from the buffer. Then, for 16 iterations, the bottom 2 bits are extracted using a simple bit mask, and the remainder is shifted down two bits.

To get the bases that are leaving the k-mer (and window) trailing behind by $k-1$ (and $\ell-1$) positions, this shifting down process is repeated with a delay of $k-1$ (and $\ell-1$) iterations. By storing the `gather`ed elements in a small ring buffer, these delayed iterations can directly read from there instead of having to read from memory again.

3.2 Rolling Hash

The next step is the computation of a hash for each k-mer. We adapt ntHash [22, 13], a popular rolling hash function for DNA sequences based on cyclic polynomials [2] as an improvement over Karp-Rabin hashing [12]. This rolling hash function can be summarized as follows: each of the 4 bases is associated to a fixed random value, denoted $f(x)$, and the

[1] https://uops.info/html-instr/VPGATHERDD_YMM_VSIB_YMM_YMM.html

[2] We use Rust instead of pseudocode to be more precise and better show the types of all variables. `&[u8]` indicates a reference to a *slice* of bytes. `impl Iterator<Item = T>` (which we simplify to `impl Iterator<T>` for brevity) is some type that iterates over values of type `T`. `u32x8` is a SIMD vector holding eight 32-bit values, and `u32x8::splat(1)` creates a SIMD vector of eight 1 values. Also for succinctness, some type casts, assertions, and handling of edge cases are omitted.

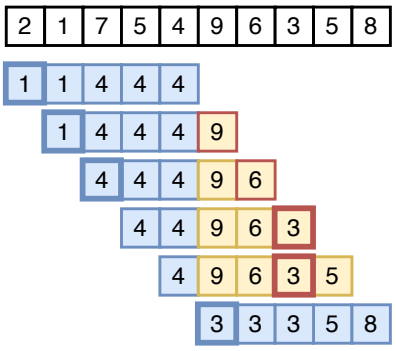

Figure 3 A visualization of six iterations of the (scalar) two-stacks sliding window minimum, for windows of size $w = 5$. The sequence of input values is shown on top in black. Blue cells show the shrinking stack of suffix-minima of each chunk of 5 values. Yellow cells show the growing stack of prefixes of the second chunk, with highlighted in red the prefix minimum. At the start, and after five iterations, the prefix values are converted into suffix minima of the new chunk. The minimum of each window has a bold outline and equals either the suffix-minimum (left-most blue cell) or rolling prefix minimum (red outlined cell). In memory, the two stacks fill a single buffer of w cells, with the yellow stack filling up the space freed up by the shrinking blue stack.

hash of a k-mer $u = x_0 \dots x_{k-1}$ is computed as $h(u) = \bigoplus_{i=0}^{k-1} \mathrm{rot}^{k-1-i}(f(x_i))$, where rot^i denotes a cyclic rotation by i bits to the left and $\oplus$ denotes xor. In practice, each x_i is a value in $\{0, 1, 2, 3\}$ and $f(x_i)$ is a simple table lookup.

Using a rolling hash has a twofold advantage in our use case: first, we only need the first and last bases of each k-mer to update its hash based on the previous one, so that we do not have to store the k-mer itself, and second, we are not limited to k-mers that fit in a constant number of words.

Code is given in Listing 2, where most scalar operations work on `u32x8` SIMD registers containing 8 32-bit values. The `table_lookup` function computes the values of $f(x)$ of the L nucleotides stored in `in_bp` or `out_bp` by looking up their value in `table` or `table_rot`. It can be implemented using `_mm256_permutevar_ps` in AVX2 or `vqtbl1q_u8` in NEON.

MulHash. A drawback of ntHash is that the efficient SIMD table lookup only works because it uses an alphabet of size 4. This means it does not work for general ASCII input. We introduce an alternative that we call *mulHash*, which replaces the table lookup of ntHash by using $f(x) = C \cdot x$, where C is a fixed random constant and the multiplication is over 32-bit integers. Although slightly slower, this can be easily computed for any input value x.

3.3 Sliding Minimum

The third building block of our algorithm computes the position of the minimum in each sliding window of w values of 32bit ntHash values. A number of different approaches can be used for this, and scalar code for each of the methods discussed can be found in Listing 5 in Section A.

Naive. The simplest approach is to simply loop over the values in each window independently. This takes $O(wn)$ time, but can still be quite efficient when w is small when using vectorized instructions.

Monotone queue. A better approach is to use a *monotone queue*, which stores a non-decreasing sequence of at most w values and their positions. Every time the window slides

```
fn sliding_min(w: usize, len: usize, vals: impl Iterator<u32x8>)
  -> impl Iterator<u32x8> {
    let val_mask = u32x8::splat(0xffff_0000);  // Use high 16 bits of each value.
    let pos_mask = u32x8::splat(0x0000_ffff);  // Low 16 bits store positions.
    let mut pos = [0*len, 1*len, ..., 7*len] as u32x8;
    let mut prefix_min = u32x8::MAX;
    let mut suffix_min = vec![u32x8::MAX; w];
    let mut idx = 0;
    vals.map(move |val| {
        let new_vals = (val & val_mask) | pos;
        pos += u32x8::ONE;
        prefix_min = min(prefix_min, new_vals);
        suffix_min[idx] = new_vals;
        idx += 1;
        if idx == w { // When the buffer is full, recompute suffix minima.
            idx = 0;
            *prefix_min = u32x8::MAX; // Reset prefix min.
            for i in (0..w - 1).rev() {
                suffix_min[i] = min(suffix_min[i], suffix_min[i+1]);
            }
        }
        min(prefix_min, suffix_min[idx]) & pos_mask // Yield the position of the min.
    }).skip(w - 1) // Skip the first w-1 incomplete windows.
}
```

Listing 3 Vectorized sliding minimum computation using a method based on the *two-stacks* algorithm. The input is an iterator over hash values of the k-mers in each of the chunks, and the output is an iterator over the position of the leftmost minimum hash in each window of w hashes.

one to the right and we are about to push a new k-mer hash onto the right of the queue, we first remove any values larger than it, as they can never be minimal anymore. The minimum of the window is then always the leftmost queue element. This data structure guarantees an amortized constant time update, but has many unpredictable branches when discarding values, which makes it costly in practice.

Rescan. Another popular approach in bioinformatics is to only keep track of the minimum value and rescan the entire window of w values when the current minimum goes out of scope [16, 17]. While this algorithm does not guarantee a worst-case constant time update, it only branches when the minimum goes out of scope and hence is more predictable. This makes it more efficient in practice, especially since minimizers typically have a density of $O(1/w)$ so that the $O(w)$ rescan step takes amortized constant time per element.

To the best of our knowledge, most existing methods in bioinformatics use either a monotone queue or a rescan approach [10, 16, 17].

Two-stacks. Since our goal here is to compute L minima at the same time using vectorized instructions, we want to avoid any kind of data-dependent branches to ensure that the code path is the same for each chunk. A method for *online* sliding minima where elements may be added and removed at varying rates is the *two-stacks* method [9, 28], that is well-known in the competitive programming community[3]. The branches and code-path in this method are completely data-independent and only depend on the index, and indeed, since we add and remove elements at the same time, the method becomes nearly branch-free.

Because elements are added and removed in sync, the total number of elements in the two stacks remains constant. It turns out they can actually be represented as a single buffer

[3] https://codeforces.com/blog/entry/71687

```
// `in_out` iterates over a pair of 2-bit bp. `in` is l-1 positions ahead of `out`.
fn canonical(k: usize, w: usize, in_out: impl Iterator<(u32x8, u32x8)>)
  -> impl Iterator<i32x8> {
    let l = k + w - 1;
    assert!(l % 2 == 1, "Window length must be odd to guarantee canonicity.");
    // Count #TG - #AC, by starting at -l and adding/subtracting 2 for each T or G.
    let mut cnt = i32x8::splat(-(l as i32));
    let two = i32x8::splat(2);
    in_out.by_ref().take(l-1).map(|(in_bp, _)| {
        cnt += in_bp & two;
    });
    in_out.map(|(in_bp, out_bp)| {
        let full_cnt = cnt + (in_bp & two);
        cnt = full_cnt - (out_bp & two);
        full_cnt.cmp_gt(i32x8::ZERO) // Yield whether the count is > 0.
    })
}
```

Listing 4 Vectorized computation to count #GT − #AC. When ℓ is odd, this can never be 0, and a window is canonical when the value is positive.

of constant size, where a growing prefix and shrinking suffix represent the two stacks, as shown in Figure 3.

Conceptually, our simplification splits the sequence of input values into *chunks* of w values. Then, every *window* of w values intersects with one or two chunks. Its minimum can be found via the suffix minimum of the first overlapping chunk and the prefix minimum of the second chunk. In code (Listing 3) we keep a rolling prefix minimum and a buffer of suffix minima of the preceding chunk. Upon entering a new chunk, the prefix minimum is reset and the suffix minima of the preceding chunk are computed.

16-bit hashes. To easily return the (leftmost) position of the minimum, instead of the minimum itself, we only use the upper 16 bits of each hash value and store the position in the lower 16 bits. After taking the minimum, we mask out the high bits to obtain its position. Strings longer than 2^{16} characters can be processed in chunks of 2^{16}. We note here that while using a 16-bit hash is usually not sufficient for k-mer indexing purposes, it is fine for selecting the minimum of a window with a length that is usually well below 1000. Our library intentionally does not expose this hash to the user, and a second larger hash can be computed afterwards for indexing purposes if needed.

3.4 Canonical minimizers

One problem that arises when using minimizers in practice is that the *strand* of a sequence is often unknown. Thus, we do not know whether we are reading the *forward* (sense) or *reverse-complement* (antisense) strand. When given a sequence, we would like to select the same minimizers regardless of its strand.

Canonical strand. We define the *canonical* strand for each window of length ℓ as the strand where the count of GT bases (values 2 and 3) is the highest [23], as shown in Listing 4. (Any pair of bases can be chosen, as long as they are not complementary.) This count is in $[0, \ell]$ and when ℓ is odd there can be no tie between the two strands. In code, we instead compute #GT − #AC, which is in $[-\ell, \ell]$, so that count > 0 defines the canonical strand. Then, we select the leftmost forward minimizer when the input window is canonical, and the rightmost reverse-complement minimizer when the input window is not canonical.

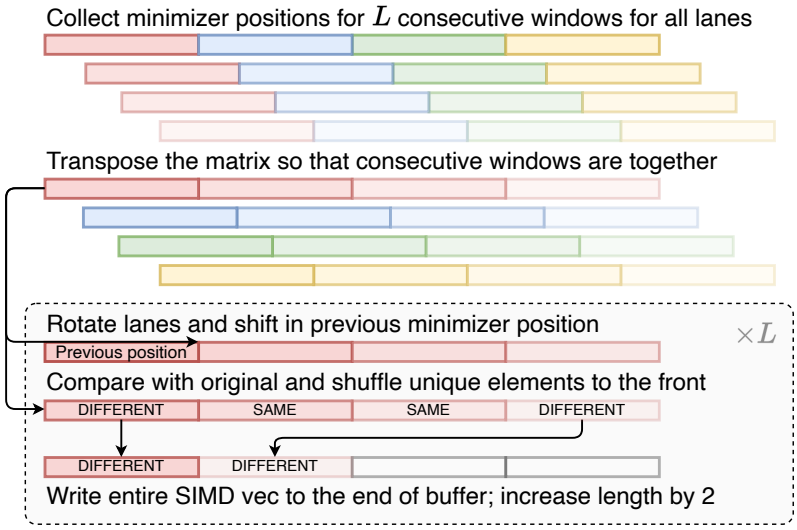

Figure 4 The sliding window minimim algorithm produces minimizer positions for L chunks at a time. In the end, we want to return a single 'flat' vector. This means we have to 'deinterleave' the L lanes. First, we collect the next L minimizer positions of every chunk. Then, we transpose this matrix, so that each SIMD vector has minimizer positions of consecutive windows. These are then compared with their preceding element, and distinct elements are shuffled to the front. These positions are accumulated in a separate buffer for each chunk, that is finally concatenated into a single flat vector.

The benefit of this method over, say, determining the strand via the middle character (assuming again that ℓ is odd), is that the GT count is more stable across consecutive windows, since it varies by ± 1. This way, the strandedness and thus the chosen minimizer is less likely to flip between adjacent windows.

Canonical ntHash. NtHash can be easily modified to compute both the forward and reverse-complement hash of each k-mer at the same time. Then, we can duplicate the sliding minimum algorithm to find the minimum of $(h_{\text{fwd}}(kmer[i]), i)$ and the *maximum* of $(-h_{\text{rc}}(kmer[i]), i)$ over each window. This way, ties in the forward direction are broken towards small i and ties in the reverse-complement direction are broken towards large i. To avoid selecting a new minimizer whenever the strand changes, we use a strand-independent *canonical* version of ntHash for both strands, defined as $h_{\text{c}} = h_{\text{fwd}} + h_{\text{rc}}$ [13], and we do not distinguish which strand a minimizer k-mer was chosen from.

While this process does not assume k to be odd, this may be a useful additional assumption for further processing of the minimizers, so that the canonical representation of each k-mer itself can be indexed. Alternatively, k-mers could all be processed in the direction of the strand of the window they minimize, but this requires additional bookkeeping to store this direction, and would require duplicating k-mers when they minimize both forward and reverse-complement strands.

Non-forward. One small theoretical drawback of this scheme is that it is not *forward*. Suppose a long window has many occurrences of the smallest minimizer, and that shifting the window by one changes its canonical strand. Then the position of the sampled minimizer could jump backwards from sampling the rightmost minimizer to sampling the leftmost minimizer. In practice, this does not seem to be a major limitation, both because it is rare and because downstream methods usually work fine on non-forward schemes anyway.

3.5 Collecting and deduplicating positions

A drawback of our methods so far is that they stream over the positions of the (canonical) minimizer of $L = 8$ independent chunks at a time. Furthermore, when adjacent windows have the same minimizer position, this position is returned multiple times. In practice, all that is usually needed as output is the deduplicated list of unique minimizer positions. Here we solve this problem, as shown in Figure 4 and Listing 6 in the Appendix.

Most vectorized deduplication methods work best on a single stream of integers, but currently we have $L = 8$ parallel streams. To fix this, we collect the next 8 minimizer positions from each lane of the input. This then gives an 8×8 matrix of minimizer positions that can be *transposed*[4] into 8 SIMD vectors containing the next 8 minimizer positions for each lane. We deduplicate each lane using the technique of [15]. This compares each element to the previous one, and compares the first element to the last minimizer of the window before. The distinct elements are then *shuffled* to the front of the SIMD vector using a lookup table and appended to a buffer for each lane. We end by concatenating all the per-lane buffers into a single vector of minimizer positions, and make sure to avoid duplicates between the end and start of adjacent lanes.

Super-*k*-mers. The deduplication can be amended to also find super-k-mers, which are sequences of consecutive windows sharing the same minimizer position. After comparing adjacent minimizer positions, we obtain a mask that determines the shuffle instruction to apply. Normally we shuffle the 32-bit minimizer positions directly. Instead, we can mask out the upper 16 bits and store there the index of its window. If we then shuffle those values, we obtain for each minimizer its position in the input text, and the position of the first window where this k-mer became a minimizer. This information is sufficient to recover all super-k-mers, and sequences longer than 2^{16}bp can be either split-up, or else it is easy to detect manually when the values wrapped.

4 Experimental Evaluation

Our code is available in the `simd-minimizers` crate that can be found at https://github.com/rust-seq/simd-minimizers. It supports both AVX2 and NEON instruction sets, and we will now look at its performance. The code to reproduce the experiments is also available there.

The experiments were run on an Intel Core i7-10750H with 6 cores with AVX2 running at a fixed frequency of 2.6GHz with hyperthreading disabled and cache sizes of 32KiB (L1), 256KiB (L2), and 12MiB shared L3. As input we use a fixed random string of 10^8 bases, depending on the method encoded either as ASCII or as packed representation. Reported timings are the median of five runs and shown in nanoseconds per base.

Tables 3 and 4 in Section B show equivalent results for the NEON architecture.

Incremental time usage. Table 1 shows the time usage for various incremental subsets of our method. To start, iterating the 8 chunks of the input and summing all bases takes 0.19 ns/bp. Appending all `u32x8` SIMD elements containing the bases to a vector takes 0.32 ns/bp, indicating that writing to memory induces some overhead. Collecting the second $k-1$-delayed stream of characters that leave the k-mer has no additional overhead. Computing ntHash only takes 0.05ns/bp extra. The sliding window computation nearly doubles the total time.

[4] https://stackoverflow.com/questions/25622745/transpose-an-8x8-float-using-avx-avx2

Table 1 Time per base taken when adding steps of the implementation, for $(w, k) = (11, 21)$.

Part	ns/bp
Gather and sum all bases	0.19
+ collect to vector	0.32
+ collect the delayed bases	0.32
+ ntHash	0.37
+ sliding window min	0.71
+ collect	1.36
+ dedup	**1.33**
+ canonical nthash	0.86
+ canonical strand	1.23
+ collect	1.94
+ dedup	**1.87**

Table 2 Comparison of our `simd-minimizers` implementation against `minimizer-iter` [19] and a rescan implementation based on [17]. Times in ns/bp are shown for both forward and canonical minimizers (where supported), and for various (w, k) tuples. For our library, we test both ntHash and mulHash with multiple encodings of the input DNA: 1) 2-bit packed, 2) ASCII-ACGT that is packed on-the-fly, and 3) plain ASCII (for mulHash only).

(w, k):	$(5, 31)$		$(11, 21)$		$(19, 19)$	
Method	fwd.	cano.	fwd.	cano.	fwd.	cano.
`minimizer-iter`	25.87	32.69	26.89	33.72	26.96	33.54
Rescan ntHash	11.72	-	7.41	-	5.58	-
`simd-minimizers` ntHash						
- packed input	1.39	2.01	1.33	1.87	1.34	1.86
- on-the-fly packing	1.76	2.37	1.65	2.23	1.64	2.23
Rescan mulHash	11.54	-	6.46	-	5.67	-
`simd-minimizers` mulHash						
- packed input	1.58	2.18	1.50	2.18	1.46	2.13
- on-the-fly packing	1.95	2.54	1.83	2.54	1.79	2.50
- ASCII input	1.95	2.49	1.82	2.47	1.78	2.48

Collecting the minimizer positions to a linear vector (i.e., transposing matrices and writing output for each of the 8 chunks) nearly doubles the time again, but deduplicating them actually *saves* some time, likely because it reduces the amount of output.

Going back a step, using canonical ntHash instead of forward ntHash takes 0.15 ns/bp extra, and determining the canonical strand (via a third ℓ-delayed stream and counting GT bases) takes another 0.37 ns/bp. As before, collecting and deduplicating are slow and add around 0.70ns/bp.

In conclusion, we see that iterating the chunks of the input and determining minimizers is quite fast, but that a lot of time must then be spent to "deinterleave" the output into a linear stream. As can be expected, canonical minimizers are slower to compute than forward minimizers, but the overhead is less than 50%, which seems quite low given that the ntHash and sliding window minimum computation are duplicated and a canonical-strand computation is added.

Full comparison. We compare against the `minimizer-iter` crate (v1.2.1) [19], which implements a queue-based sliding window minimum using `wyhash` [29] and also supports canonical minimizers. For an additional comparison, we optimized an implementation of the remap method with ntHash based on a code snippet by Daniel Liu [17].

Results are in Table 2. `minimizer-iter` takes around 26ns/bp for forward and 33ns/bp for canonical minimizers, and its runtime does not depend much on w and k, because the popping from the queue is unpredictable regardless of w. Rescan starts out at 11.7 ns/bp for $w = 5$ and gets significantly faster as w increases, converging to around 5 ns/bp for $w \gg 100$. This is explained by the fact that rescan has a branch miss every time the current minimizer falls out of the window, which happens for roughly half the minimizers at a rate of $1/(w + 1)$. Thus, as w increases, the method becomes more predictable and branch misses go down. Our method, `simd-minimizers`, runs around 1.33 ns/bp for forward and 1.87 ns/bp for canonical minimizers when given packed input, and therefore is 4.5× to 9.5×

faster than the rescan method. Its performance is mostly independent of k and w since it is mostly data-independent. Only for small w it is slightly slower due to the larger number of minimizers and hence larger size of the output.

As we use SIMD with 8 lanes, we could in theory expect up to 8× speedup. In practice this is hard to reach because of constant overhead and because the code needs to be modified to work well with SIMD in the first place. In particular for large w, rescan benefits from very predictable and simple code and only outputs unique minimizer positions, making it very efficient. In SIMD, on the other hand, we use a data-independent algorithm, and output the minimizer position for every single window, which then has to be deduplicated. Thus, it is nice to see that our method is over 4× faster, despite this overhead.

ASCII input and mulHash. Apart from taking bit-packed input, `simd-minimizers` also works on ASCII-encoded DNA sequences of ACTG characters directly, which are then packed into values $\{0, 1, 2, 3\}$ for ntHash during iteration. This is around 0.35ns/bp slower, mostly because of the larger size of the unpacked input.

The mulHash variant is around 0.20ns/bp slower again, but works for any ASCII input. Performance on 100MB of the Pizza&Chili corpus [5] English and Sources datasets is nearly identical to performance on the random DNA shown in Table 2.

Human genome. The measured time per base extrapolates directly to a human genome of 3.2Gbp (T2T-CHM13v2.0), where forward minimizers take 4.1 seconds, and canonical minimizers take 6.0 seconds for $(w, k) = (11, 21)$.

5 Conclusions and Future Work

Our library `simd-minimizers` computes minimizer positions 4.5× to 9.5× faster than other methods. Using the library, only a single function call is needed to obtain the list of (canonical) minimizer positions, taking as input the (packed) DNA sequence and the parameters k and w. General ASCII input are also supported, allowing use cases such as sketching protein sequences. We hope that the community will adopt `simd-minimizers` as the standard library to compute random minimizers.

Future work. We chose to use the data-independent two-stacks method as the core of our algorithm, that returns the minimizer position of every window and then requires deduplication. Given the promising performance of rescan for large w in Table 2, an interesting alternative could be to go the opposite way and speed up the rescan step. This is particularly relevant when minimizers are sparse, as the number of branches may be small enough for linear scans to benefit from SIMD. Accelerating linear scans could also be useful for smaller inputs such as short reads, where splitting into 8 chunks may not be very efficient.

Another approach would be to use 512-bit AVX512 instructions and process 16 lanes in parallel. In theory that could be another 2× faster, but in practice the collecting and deduplicating of values may become an even larger bottleneck.

Additionally, implementing low-density schemes like the open-closed mod-minimizer [7, 8] would be a valuable extension.

References

1 Gaëtan Benoit, Sébastien Raguideau, Robert James, Adam M Phillippy, Rayan Chikhi, and Christopher Quince. High-quality metagenome assembly from long accurate reads with metamdbg. *Nature Biotechnology*, pages 1–6, 2024. doi:10.1038/s41587-023-01983-6.

2 Jonathan D. Cohen. Recursive hashing functions for n-grams. *ACM Trans. Inf. Syst.*, 15(3):291–320, July 1997. doi:10.1145/256163.256168.

3 Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, January 2015. doi:10.1093/bioinformatics/btv022.

4 Barış Ekim, Bonnie Berger, and Rayan Chikhi. Minimizer-space de bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer. *Cell Systems*, 12(10):958–968.e6, 2021. URL: https://www.sciencedirect.com/science/article/pii/S240547122100332X, doi:10.1016/j.cels.2021.08.009.

5 Paolo Ferragina and Gonzalo Navarro. The Pizza&Chili Corpus. https://pizzachili.dcc.uchile.cl/texts.html, 2007.

6 Shay Golan, Ido Tziony, Matan Kraus, Yaron Orenstein, and Arseny Shur. Generating low-density minimizers. *bioRxiv*, November 2024. doi:10.1101/2024.10.28.620726.

7 Ragnar Groot Koerkamp, Daniel Liu, and Giulio Ermanno Pibiri. The open-closed mod-minimizer algorithm. *bioRxiv*, 2024. doi:10.1101/2024.11.02.621600.

8 Ragnar Groot Koerkamp and Giulio Ermanno Pibiri. The mod-minimizer: A Simple and Efficient Sampling Algorithm for Long k-mers. In Solon P. Pissis and Wing-Kin Sung, editors, *24th International Workshop on Algorithms in Bioinformatics (WABI 2024)*, volume 312 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:23, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.WABI.2024.11, doi:10.4230/LIPIcs.WABI.2024.11.

9 Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. Sliding-window aggregation algorithms: Tutorial. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*, pages 11–14. ACM, 2017. doi:10.1145/3093742.3095107.

10 Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de bruijn graphs. *Genome biology*, 21:1–20, 2020.

11 Chirag Jain, Arang Rhie, Nancy F Hansen, Sergey Koren, and Adam M Phillippy. Long-read mapping to repetitive reference sequences using winnowmap2. *Nature Methods*, 19(6):705–710, 2022. doi:10.1038/s41592-022-01457-8.

12 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

13 Parham Kazemi, Johnathan Wong, Vladimir Nikolić, Hamid Mohamadi, René L Warren, and Inanç Birol. nthash2: recursive spaced seed hashing for nucleotide sequences. *Bioinformatics*, 38(20):4812–4813, August 2022. doi:10.1093/bioinformatics/btac564.

14 Bryce Kille, Ragnar Groot Koerkamp, Drake McAdams, Alan Liu, and Todd J Treangen. A near-tight lower bound on the density of forward sampling schemes. *Bioinformatics*, 41(1):btae736, December 2024. doi:10.1093/bioinformatics/btae736.

15 Daniel Lemire. Removing duplicates from lists quickly. https://lemire.me/blog/2017/04/10/removing-duplicates-from-lists-quickly/, April 2017.

16 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, May 2018. doi:10.1093/bioinformatics/bty191.

17 Daniel Liu. minimizer.rs re-scan implementation. https://gist.github.com/Daniel-Liu-c0deb0t/7078ebca04569068f15507aa856be6e8, July 2023.

18 Camille Marchet, Mael Kerbiriou, and Antoine Limasset. Blight: efficient exact associative structure for k-mers. *Bioinformatics*, 37(18):2858–2865, April 2021. doi:10.1093/bioinformatics/btab217.

19 Igor Martayan. minimizer-iter: Iterate over minimizers of a DNA sequence. https://github.com/rust-seq/minimizer-iter.

20 Igor Martayan, Lucas Robidou, Yoshihiro Shibuya, and Antoine Limasset. Hyper-k-mers: efficient streaming k-mers representation. *bioRxiv*, November 2024. doi:10.1101/2024.11.06.620789.

21 Guillaume Marçais, C S Elder, and Carl Kingsford. k-nonical space: sketching with reverse complements. *Bioinformatics*, 40(11):btae629, October 2024. doi:10.1093/bioinformatics/btae629.

22 Hamid Mohamadi, Justin Chu, Benjamin P. Vandervalk, and Inanc Birol. nthash: recursive nucleotide hashing. *Bioinformatics*, 32(22):3492–3494, July 2016. doi:10.1093/bioinformatics/btw397.

23 Chenxu Pan and Knut Reinert. A simple refined dna minimizer operator enables 2-fold faster computation. *Bioinformatics*, 40(2):btae045, January 2024. doi:10.1093/bioinformatics/btae045.

24 David Pellow, Lianrong Pu, Bariş Ekim, Lior Kotlar, Bonnie Berger, Ron Shamir, and Yaron Orenstein. Efficient minimizer orders for large values of k using minimum decycling sets. *Genome Research*, 33(7):1154–1161, 2023. URL: https://www.genome.org/cgi/doi/10.1101/gr.277644.123, doi:10.1101/gr.277644.123.

25 Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *Bioinformatics*, 38(Supplement_1):i185–i194, June 2022. doi:10.1093/bioinformatics/btac245.

26 Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, July 2004. doi:10.1093/bioinformatics/bth408.

27 Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 76–85, New York, NY, USA, June 2003. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/872757.872770, doi:10.1145/872757.872770.

28 Georgios Theodorakis, Alexandros Koliousis, Peter R. Pietzuch, and Holger Pirk. Hammer slide: Work- and cpu-efficient streaming window aggregation. In Rajesh Bordawekar and Tirthankar Lahiri, editors, *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2018, Rio de Janeiro, Brazil, August 27, 2018*, pages 34–41, 2018. URL: http://www.adms-conf.org/2018-camera-ready/SIMDWindowPaper_ADMS%2718.pdf.

29 Wang Yi and Diego Barrios Romero. wyhash-rs, fast portable non-cryptographic hashing algorithm. https://github.com/eldruin/wyhash-rs.

30 Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Improved design and analysis of practical minimizers. *Bioinformatics*, 36(Supplement_1):i119–i127, July 2020. doi:10.1093/bioinformatics/btaa472.

A Additional code listings

```
fn naive(w: usize, vals: &[u32]) -> impl Iterator<Item = usize> {
    vals.windows(w).enumerate().map(|(pos, window)| {
        pos + window.iter().position_min().unwrap()
    })
}
fn queue(w: usize, vals: &[u32]) -> impl Iterator<Item = usize> {
    // A monotone queue that is always increasing from front (small) to back (large).
    let mut queue = VecDeque::<(u32, usize)>::with_capacity(w);
    vals.iter().enumerate().map(move |(pos, &val)| {
        if queue.front().unwrap().1 + w <= pos {
            queue.pop_front();          // Drop values not in the window anymore.
        }
        while !queue.is_empty() && queue.back().unwrap().0 > val {
            queue.pop_back();           // Drop values larger than the current value.
        }
        queue.push_back((val, pos));    // Push the new value to the back.
        queue.front().unwrap().1        // Read the position of the smallest element.
    }).skip(w-1)                        // The first w-1 windows are not full.
}
fn rescan(w: usize, vals: &[u32]) -> impl Iterator<Item = usize> {
    let mut min = u32::MAX;
    let mut min_pos = 0;
    vals.iter().enumerate().map(move |(pos, &val)| {
        if val < min {
            (min, min_pos) = (val, pos);
        }
        if min_pos + w <= pos {
            min_pos = pos - w + 1 + vals[pos-w+1..=pos].iter().position_min().unwrap();
            min = vals[min_pos];
        }
        min_pos
    }).skip(w-1)                        // The first w-1 windows are not full.
}
fn two_stacks(w: usize, vals: &[u32]) -> impl Iterator<Item = usize> {
    let mut prefix_min = (u32::MAX, 0);          // Rolling prefix minimum.
    let mut suffix_min = vec![(u32::MAX, 0); w]; // Buffer for suffix minima.
    let mut idx = 0;
    vals.iter().enumerate().map(move |(pos, &val)| {
        prefix_min = min(prefix_min, (val, pos));
        suffix_min[idx] = (val, pos);
        idx += 1;
        if idx == w {
            idx = 0;
            prefix_min = (u32::MAX, 0); // Reset prefix min.
            for i in (0..w - 1).rev() { // Compute suffix minima.
                suffix_min[i] = min(suffix_min[i], suffix_min[i + 1]);
            }
        }
        min(prefix_min, suffix_min[idx]).1
    }).skip(w-1)                        // The first w-1 windows are not full.
}
```

Listing 5 Simple scalar implementations of the naive, queue, rescan, and two-stacks sliding minimum methods.

```rust
// Input: Iterator of 8 lanes of positions.
// Output: Single flattened vector of all unique positions.
fn collect_and_dedup(positions: impl Iterator<u32x8>) -> Vec<u32> {
    let mut uniq = [vec![0; BUF_SIZE]; 8]; // Sufficiently large buffer for each lane.
    let mut idx = [0; 8]; // Active index in each output vector.

    // Buffer to store the last 8 elements seen in each lane.
    let mut last = [u32x8::ZERO; 8];

    let mut m = [u32x8::ZERO; 8];
    par_head.enumerate().for_each(|(i, x)| {
        // Collect 8 values from each lane.
        m[i % 8] = x;
        if i % 8 == 7 {
            // Transpose the 8x8 u32 matrix based on:
            // stackoverflow.com/questions/25622745/transpose-an-8x8-float-using-avx-avx2
            let t: [u32x8; 8] = transpose(m);
            for j in 0..8 {
                // Deduplicate `t` against itself and `last` and append to `uniq` at `idx`.
                dedup_and_write(last[j], t[j], &mut uniq[j], &mut idx[j]);
                last[j] = t[j];
            }
        }
    });
    // Omitted: the remainder when the input length is not a multiple of 8.

    // Flatten into output vector.
    let mut out_vec = vec![];
    for j in 0..8 {
        let mut lane = &uniq[j][0..idx[j]];
        // Prevent duplicating the last element of the previous lane.
        if lane[0] == out_vec.last() {
            lane = &lane[1..];
        }
        out_vec.extend_from_slice(lane);
    }
    out_vec
}

// Based on lemire.me/blog/2017/04/10/removing-duplicates-from-lists-quickly
fn dedup_and_write(prev: u32x8, new: u32x8, out: &mut [u32], idx: &mut usize) {
    // Mix the last element of `prev` into `new`.
    let comp = _mm256_blend_epi32(prev, new, 0b01111111);
    // Rotate comp 1 lane.
    let comp = _mm256_permutevar8x32_epi32(comp, [6, 5, 4, 3, 2, 1, 0, 7]);

    // Compare `new` against `comp`.
    let mask = _mm256_movemask_ps(_mm256_cmpeq_epi32(comp, new));
    // Shuffle the unique values to the front.
    // UNIQSHUF is a 256 long array of a permutation for each comparison result.
    let uniq = _mm256_permutevar8x32_epi32(new, UNIQSHUF[mask]);
    out[*idx..*idx + 8] = uniq; // append them to `out`,
    *idx += WIDTH - mask.count_ones(); // and increase the write index.
}
```

Listing 6 The `collect_and_dedup` function takes as input a stream of `u32x8` containing the minimizer position for each lane. It collects 8 values from each lane and then deduplicates these 8 values one lane at a time. The `dedup_and_write` function takes a list of positions `new` and collects the unique ones, also taking into account the previous value. These are then written to `out` starting at index `idx`. At the end, the temporary buffers for each lane are flattened into a single vector.

B Results for NEON architecture

The NEON experiments are run on a performance core on an Apple M1 chip with 4 efficiency and 4 performance cores. The performance core runs at 3.2GHz and has 128KiB of L1 cache, 12MiB of shared L2 cache (for the performance cores) and 8MiB of shared L3 cache (for the whole system).

Table 3 Time per base taken when adding steps of the implementation, for $(w, k) = (11, 21)$ on NEON architecture.

Part	ns/bp
Gather and sum all bases	0.94
+ collect to vector	0.85
+ collect the delayed bases	0.90
+ ntHash	0.99
+ sliding window min	1.25
+ collect	1.51
+ dedup	**1.82**
+ canonical nthash	1.45
+ canonical strand	1.77
+ collect	1.98
+ dedup	**2.28**

Table 4 Comparison of our `simd-minimizers` implementation against `minimizer-iter` [19] and a rescan implementation based on [17]. Times in ns/bp are shown for both forward and canonical minimizers (where supported), and for various (w, k) tuples, on NEON architecture.

(w, k):	$(5, 31)$		$(11, 21)$		$(19, 19)$	
Method	fwd.	cano.	fwd.	cano.	fwd.	cano.
`minimizer-iter`	13.00	16.80	14.56	18.27	14.30	18.11
Rescan	7.19	-	4.58	-	3.27	-
`simd-minimizers`	1.84	2.26	1.81	2.30	1.83	2.26