# Accel-Align: A Fast Sequence Mapper and Aligner based on the Seed–Embed–Extend Method

**Yiqing Yan [1] Nimisha Chaturvedi [2] and Raja Appuswamy [1,*]**

[1] Data Science Department, EURECOM, Biot, 06410, France and
[2] ACCELOM, Antibes, 06600, France.

* To whom correspondence should be addressed.

## Abstract

**Motivation:** Improvements in sequencing technology continue to drive sequencing cost towards 100$ per genome. However, mapping sequenced data to a reference genome remains a computationally-intensive task due to the dependence on edit distance for dealing with indels and mismatches introduced by sequencing. All modern aligners use seed–filter–extend (SFE) methodology and rely on filtration heuristics to reduce the overhead of edit distance computation. However, filtering makes assumptions about error patterns, has inherent performance–accuracy trade-offs, and requires careful hand tuning to match the dataset.

**Results:** Motivated by algorithmic advances in randomized low-distortion embeddings, we introduce *seed–embed–extend* (SEE), a new design methodology for developing sequence mappers and aligners. While SFE focuses on eliminating sub-optimal candidates, SEE focuses instead on identifying optimal candidates. To do so, SEE transforms the read and reference strings from edit distance regime to the Hamming regime by embedding them using a randomized algorithm, and uses Hamming distance over the embedded set to identify optimal candidates. To show that SEE perfoms well in practice, we present Accel-Align, an SEE-based short-read sequence mapper and aligner that is 3-12× faster than state-of-the-art aligners on commodity CPUs, without any special-purpose hardware, while providing comparable accuracy.

**Availability:** https://github.com/raja-appuswamy/accel-align-release

**Contact:** raja.appuswamy@eurecom.fr

## 1 Introduction

Over the last decade, DNA sequencing technology has achieved dramatic improvements in both cost and throughput. With the $100-per-genome sequencing goal emerging as a realistic target in the near future, the amount of genomic data generated by sequencing is only poised to grow faster. The first, and often one of the most time consuming steps, in analyzing genomic datasets is sequence alignment–the process of determining the location in the reference genome of each sequencing read. Sequence alignment can be boiled down to a string matching problem. Given a string $G$ as the reference genome, and a set of substrings $R$ as the sequencing reads, the task of read alignment is to find the origin location of each read $r \epsilon R$ in $G$. However, due to sequencing errors, and differences between the reference genome and the sequenced organism, a read might not exactly align at any candidate location in the reference genome. Thus, an aligner has to perform approximate string matching to tolerate errors.

The error-handling capability of a string matching algorithm is related to the distance metric used for comparing strings. The two popular string distance metrics are Hamming distance which only counts the number of position-wise mismatches between two strings, and edit distance or Levenshtein distance, which counts the difference between two strings while allowing for characters to be inserted or deleted. Aligners can be classified into *ungapped* or *gapped* depending on whether they use Hamming or Levenshtein distance for computing mismatch between reads and the reference (Canzar and Salzberg (2017)). As modern sequencers can produce both substitution and indel errors, gapped aligners are preferred in practice over their ungapped counterparts.

Modern gapped read aligners, like Bowtie2, BWA-MEM, and Minimap2, can map thousands of reads per second to the reference genome. However, as sequencing datasets continue to grow at a rapid pace, even these state-of-the-art aligners face scalability bottlenecks due to a crucial design aspect that is universal across all gapped read aligners–the use of edit-distance as a string comparison metric. Computing edit distance between two sequences is a computationally-expensive task that takes approximately quadratic time in the length of the input sequences. Given that sub-quadratic computation of edit distance is extremely unlikely (Backurs and Indyk (2015)), the brute force approach of trying to align a read at each position in the reference is infeasible even for a single read due to sheer number of edit-distance computations that would be required. Thus, all modern aligners focus on minimizing the number of such computations using a *seed-filter-extend* (SFE) strategy for performing alignment (Canzar and Salzberg (2017)).

SFE strategy works by first indexing the reference genome and storing the occurrence locations of short string sequences, which are also referred to as seeds or k-mers, typically in a hash table. Each read is processed in three steps. First, seeds are extracted from the read and the hash table is used to look up potential mapping locations in the reference genome. Second, filtering techniques are used to eliminate as many candidate locations as possible to minimize the overhead of the extension stage, during which the entire read is aligned at each of unfiltered candidate locations using the edit-distance-based approximate string matching algorithms.

Although state-of-the-art filtering techniques provide an order of magnitude improvement over unfiltered extension, they all suffer from two major limitations. First, all current filtering techniques are based on *elimination*. Their goal is to improve performance by eliminating some candidate locations without forwarding them to the extension stage. They cannot provide *selection* which is able to identify a candidate that is likely to be the actual match, or *ordering* which sorts candidates in the order of likelihood. This is particularly problematic for seed sequences that inevitably occur at hundreds or thousands of different locations in the reference sequence due to tandem repeats, or transposon-induced duplication. For such sequences, filtering techniques are less effective, as they can neither identify optimal candidates, nor eliminate candidates without significantly increasing the probability of misalignment due to accidental elimination of a true match. The second problem with filtering is that the thresholds and parameters used by these filtering techniques is often manually picked based on empirical analysis, as they are dependent on the type and pattern of errors introduced by the sequencing technology. Hence, filtering techniques are inherently non-portable heuristics and not technology-independent design principles.

In this paper, we introduce a new design principle for constructing sequence mappers and aligners, henceforth referred to as *Seed-Embed-Extend* (SEE). SEE builds on recent theoretical advances in the design of randomized algorithms that can perform embedding from edit distance into Hamming distance with very low distortion (Chakraborty *et al.* (2016)). These algorithms provide a one-to-one mapping $f$ that can be used to transform a set of strings $S$ into another set of strings $S'$, such that the worst case ratio between Hamming distance of any two strings $f(x)$ and $f(y)$ in $S'$, and the edit distance of their equivalent strings $x$ and $y$ in $S$ is very low. SEE uses seeding to identify candidate locations similar to SFE aligners. However, instead of using filtering to eliminate candidates, SEE uses randomized algorithms to embed the reference string at each candidate location. SEE then uses Hamming distance between the embedded candidates and the embedded read to rank the candidates based on likelihood of being an actual alignment target, and chooses candidates with the highest rank for extension. Thus, instead of eliminating many bad candidates, SEE focuses on quickly selecting a few good ones.

To show that SEE works well in practice, we present *Accel-Align*– an SEE-based short-read sequence mapper and aligner that can provide both extension-free mapping and base-to-base alignment with CIGAR and MAPQ. In doing so, we show that a naive implementation of SEE will result in the embedding step becoming a computational bottleneck, and describe several optimizations that Accel-Align uses to implement SEE-based alignment effectively. Using experimental results from both simulated and real datasets, we show that embedding is capable of picking locations that are likely to be the correct alignment targets with very high accuracy. Using the SEE-approach to sequence alignment, Accel-Align can align 100,000 100bp reads per second on a commodity quad-core CPU, and is up to 9× faster than BWA-MEM, 12× faster than Bowtie2, and 3× faster than Minimap2, while providing comparable accuracy without using any special purpose hardware. We believe that SEE specifically, and embedding in general, is a robust technique that opens up new optimization opportunities not only for sequencing alignment, but also for several other computational biology problems that rely on edit distance.

## 2 Methods

Accel-Align adopts the SEE technique for sequence alignment. During alignment, Accel-Align processes each read by first extracting seeds to find candidate locations similar to SFE aligners. However, the similarity with SFE ends with seeding. After seeding, Accel-Align embeds both the read string and the reference string at each candidate location found by seeding. After embedding, Accel-Align calculates the Hamming distance between each embedded candidate reference and the read, and selects the two best candidates with the lowest Hamming distance for extension and scoring. In the rest of this section, we will introduce each of the three stages of SEE, and their implementation in Accel-Align, in detail.

### 2.1 Building the index and seeding

#### 2.1.1 Building the index

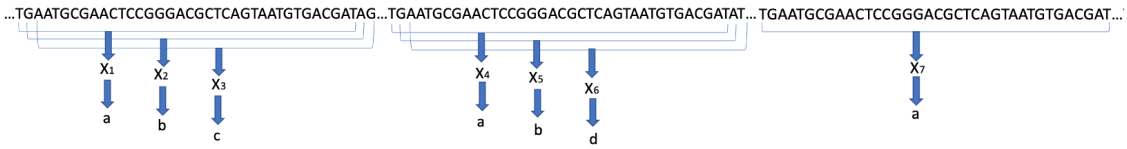
As Accel-Align uses seeding, it requires the reference genome to be indexed before execution. Similar to other aligners, we construct an index over the reference genome in a separate, offline phase. The index is a hash table of key-value pairs, where the key and value are both 32-bit unsigned integers. In order to populate the hash table, we extract k-mers from each position of the reference genome. As the reference sequence usually contains only 4 characters, namely A, T, C and G, we convert each character in the extracted k-mer into a two-bit equivalent representation. Any k-mers that contain 'N' characters are not added to the index. The k-mer length is a configurable parameter in Accel-Align, but we set it to 32 to enable a k-mer to fit in a single 64-bit integer. We hash the k-mer to generate the key by using a simple modulo-based hash function that maps the 64-bit integer into one of $M$ buckets, where $M$ is a large prime number that fits in a 32-bit integer. The 32-bit reference location offset from where the k-mer was extracted is the value associated with the key.

As the hash table is repeatedly used for looking up candidates during alignment, it is important to physically store these key–value pairs efficiently. We do this by using a chained hash table implementation based on two flat 32-bit integer arrays. With our construction, there are at most $M$ different keys and $N - k + 1$ different values, where $N$ is the length of reference genome. As multiple k-mers can hash to the same key, each key can correspond to multiple position values. We gather all position values for each key, sort them individually, and store all such sorted values together, in key order, in a single *position* array. We represent the keys implicitly by an offset in a separate 4GB *key* array, and in each key-array entry, we store the cumulative count of candidate positions for all keys smaller than that key. Thus, as the position array is ordered by key, all the candidate locations indexed between the offsets $K$ and $K + 1$ in position table belong to the key $K$. The process is illustrated in Figure 1. Thus, the entire index, represented using the key and position arrays, is saved in a single file on disk, and loaded in memory whenever alignment starts.

#### 2.1.2 Seeding

During seeding phase of alignment, we extract all non-overlapping k-mers of each read. Then for each k-mer, we compute the key, and use the key to extract the list of candidate positions. The positions are adjusted using the offset of the k-mer into the read to get normalized candidate positions. Then, we merge the candidate lists across k-mers to produce the final list of normalized positions that does not have any duplicates. One way of performing this merging is to gather all candidates in an array, sort it, and then find unique elements. However, such an approach would take $O(N_l \lg N_l)$ time if $N_l$ is the total number of candidates across all k-mers.

We avoid sorting by exploiting the physical organization of our hash table index. The position array of the hash table contains a key-ordered list of candidate positions, where each key's candidates are stored in sorted order. Thus, during seeding, the candidates retrieved for each k-mer will also be in sorted order. We maintain a min-priority queue of size $N_k$,




Key table

| Index | ... | a | a+1 | ... | b | b+1 | ... | c | ... | d | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Accumulated frenquency | | $n_1$ | $n_1 + 3$ | | $n_2$ | $n_2 + 2$ | | $n_3$ | | $n_4$ | |

Value table

| Index | ... | $n_1$ | $n_1$+1 | $n_1$+2 | ... | $n_2$ | $n_2$+1 | ... | $n_3$ | ... | $n_4$ | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Location | | $l_1$ | $l_2$ | $l_3$ | | $l_4$ | $l_5$ | | $l_6$ | | $l_7$ | |

**Fig. 1.** An example to build index. $X_i$ represents a 32-mer extracted from reference genome. $a, b, c, d$ are the keys calculated from $X_i$. $n_i$ represents the number of keys smaller than the corresponding key. For example, $n_1$ is the number of keys smaller than $a$. Specifically, $X_1$, $X_4$ and $X_7$ have the same key $a$. So there are $n_1 + 3$ keys smaller than $a + 1$. Thus, $l_1, l_2$ and $l_3$ indexed with $n_1, n_1 + 1, n_1 + 2$ in value table are the locations for $X_1$, $X_4$ and $X_7$.

where $N_k = readlength/k$ is the number of k-mers in each read, and initialize it with the first candidate location of each k-mer. Then, we pop the minimum value from the queue and push the next candidate from the same k-mer as the popped one into the queue. We repeat the pop–and–push steps until the all candidates have been processed. This approach allows us to process all candidates in $O(N_l)$ time without sorting as long as the number of k-mers is small enough to keep the overhead of min-priority-queue negligible, which we found to be the case for short-read alignment using an empirical evaluation.

In the case of single-end reads, we don't apply any filtering to the merged candidate lists. Thus, all candidates are passed to the embedding stage. For paired-end reads, however, we use a configurable pairwise-distance threshold for identifying candidates from one read that have a matching pair within the specified distance in the other read. All such candidate pairs are passed to the embedding stage.

## 2.2 Embedding

After candidate locations are identified by seeding, Accel-Align moves to embedding, the second stage of SEE. The goal of embedding is to transform both the candidate strings from the reference genome, and the query string which is the read, into different strings such that the edit distance between the original strings can be approximated using the Hamming distance between new strings. We have implemented two randomized embedding algorithms in Accel-Align.

### 2.2.1 $3N$-embedding

The first algorithm was proposed by Chakraborty *et al.* (2016) who showed that given two strings $x, y$ of length $N$ taken from an alphabet $\sum$ such that $d_E(x, y)$, the edit distance between $x$ and $y$, is less than $K$, there exists an embedding function $f\colon \sum^N \rightarrow \sum^{3N}$, such that the distortion $D(x, y) = d_H(f(x), f(y))/d_E(x, y)$ lies in $[1, O(K)]$ with at least 0.99 probability, where $d_H(x, y)$ is the Hamming distance between the embedded strings. In other words, Chakraborty *et al.* (2016) proposed a randomized algorithm that can embed strings of length $N$ into strings of length $3N$ such that Hamming distance of embedded strings is at most square of the edit distance between original strings. Recent studies have demonstrated that this algorithm, which we henceforth refer to as *3N-embedding (3NE)*, works well in practice for performing edit similarity joins for even relatively large edit distances(Zhang and Zhang (2017).)

Accel-Align uses 3NE for embedding both candidate strings from the reference and the read itself. Listing 1 shows the pseudo-code for the embedding algorithm. The input string is a DNA sequence of length $N$ consisting of four possible characters (A,C,G,T). The output is an embedding string of length $3N$ consisting of the four characters and possibly multiple repeats of a pad character (P). In each iteration, the algorithm appends a character from the input string, or the pad if it runs out of the input string, to the output string. Then, it uses a random binary bit string to decide if the input index should be advanced. The net effect of this algorithm is that some input characters appear uniquely in the output string, while others are randomly repeated multiple times. Using the theory of simple random walks, Chakraborty *et al.* (2016) established that the randomization in this algorithm will result in strings that differ by a small edit distance converging quickly to produce embedded strings that have a small Hamming distance.

**Algorithm 1** $3N$-embedding

**Input:** A string $S \in \{A, C, G, T\}^N$, and a random string $r \in \{0, 1\}^{3N}$
**Output:** The embedded string $S' \in \sum^{3N}$

```
i ← 0
for j = 0 → 3N − 1 do
    if i < N then
        S'_j ← S_i
    else
        S'_j ← P
    end if
    i ← i + r_j
end for
return S'
```

### 2.2.2 $2N$-embedding

An initial implementation of 3NE in Accel-Align showed us early on that despite the simplicity of the algorithm, it was computationally intensive to embed billions of candidate locations across millions of reads. We describe several optimizations later in Section 2.4 that reduced the overhead of embedding, but one of the first optimizations we designed was a variant of the embedding proposed by Chakraborty *et al.* (2016), which we refer to as *2N-Embedding (2NE)*. Listing 2 shows the pseudocode for 2NE which is conceptually similar to 3NE with the exception that each character in the input string is copied to the output string at most two times. Thus, 2NE implements a mapping $f\colon \sum^N \rightarrow \sum^{2N}$ instead of $\sum^N \rightarrow \sum^{3N}$.

After implementing 2NE in Accel-Align, we found that Zhang *et al.* (2019) had also developed it in parallel, performed a theoretical analysis of its optimality, and used it for the edit similarity join application. As we show in our evaluation, we found 2NE to be functionally comparable to 3NE in terms of accuracy, and slightly better in terms of performance as it reduces the embedding time due to a reduction in the embedded string length from $3N$ to $2N$.

**Algorithm 2** $2N$-embedding

**Input:** A string $S \in \{A, C, G, T\}^N$, and a random string $r \in \{0, 1\}^N$
**Output:** The embedded string $S^{'} \in \sum^{2N}$

```
j ← 0
for i = 0 → N − 1 do
    S'_j ← S_i
    j++
    if r_i = 1 then
        S'_{j+1} ← S_i
        j++
    end if
end for
for j = j + 1 → 2N − 1 do
    S'_j ← P
end for
return S'
```

#### 2.2.3 Candidate Selection

We use one of the two embedding algorithms described above to embed all candidates and the read. Then, we compute the Hamming distance between each embedded candidate and the embedded read. We refer to this distance as the *embedding distance*. While doing so, we dynamically keep track of the top two candidates with least embedding distance and forward them to the third phase for further extension, scoring, and mapping quality computation. This step is the most important difference between SEE and SFE techniques. While SFE focuses on heuristics for eliminating candidates, SEE provides a way to rank candidates and directly select the most likely ones based on Hamming distance, which a scalable metric that can be computed in linear time.

### 2.3 Extension and MAPQ Computation

Accel-Align can be configured to run in alignment-free mapping mode where only the identified candidate location is reported, or full-alignment mode where base-by-base extension is performed and the CIGAR string is reported. For the mapping mode, we simply pick the best candidate, which is the one with the least embedding distance, and report its normalized location. For the full-alignment mode, we extend the best candidate with the traditional Smith-Waterman (Smith *et al.* (1981)) algorithm. We use lib-ksw (Suzuki and Kasahara (2018)) with match(1), mismatch(4), gap-open(6), and gap-extension(1) penalties configured to match the settings used by BWA-MEM for computing the alignment score and CIGAR.

In addition to the CIGAR, popular SFE aligners also report a mapping quality (MAPQ) that represents the degree of confidence in the alignment for each read. The basic idea behind MAPQ is that if multiple candidates align to the read with comparable alignment score, the aligner cannot be confident in its final choice. However, if alignment scores between the chosen candidate and others are very different, the aligner can be reasonably confident in its choice. Thus, MAPQ computation typically requires SFE aligners to extend at least two candidates so that alignment scores can be compared. SEE, in contrast, makes it possible to avoid such extensions as we can exploit embedding distance once again to compute MAPQ. If two candidates are spaced apart with respect to edit distance from a read, then the embedded candidates will also be spaced apart with respect to Hamming distance from the embedded read. Thus, Accel-Align uses Bowtie2's MAPQ estimation procedure (Langmead and Salzberg (2012)) adjusted to use the embedded length, instead of the read length, and embedding distance of the top two candidates, instead of their alignment scores produced from Smith-Waterman extension, in order to produce a MAPQ value between 0 and 42.

### 2.4 Optimizations

A naive SEE implementation would perform seeding, embedding, and extension as described so far in sequence. However, during initial experimentation Accel-Align, we found that while embedding reduced the overhead of extension, the computational task of embedding and Hamming distance computation added non-negligible overhead. Thus, in addition to the 2NE algorithm described in Section 2.2.2, we implemented three other optimizations, namely, *pipelining*, *early-stop*, and *prioritizing*, that reduced the overhead of embedding without any change in functionality or accuracy.

**Pipelining.** Instead of embedding all candidates and then computing the Hamming distance, our first optimization is to pipeline these steps. We do this by modifying the embedding step so that the read is embedded first. Then each candidate location is embedded one by one, and the embedding algorithm simply updates the embedding distance in each iteration by comparing the output character of the candidate generated in that iteration with the corresponding character in the embedded read. This pipelining of embedding and distance computation provides three major benefits. First, as embedded strings are no longer generated in their entirety, it reduces memory consumption and associated overheads of allocating and freeing memory for storing embedded candidates. Second, it reduces the overhead caused by a needless second loop over the embedded candidates to calculate the Hamming distance. Third, as the output character generated by the algorithm is used immediately for distance computation, it improves processor cache utilization.

**Early-stop.** Using pipelining to produce the embedding distance for each candidate enables us to apply the second optimization based on the observation that only the top-two candidates with the least embedding distance are selected for further extension. Thus, if we have already encountered a candidate with a very low embedding distance, there is no point in continuing the embedding process for another candidate whose distance has already exceeded the previously observed minimum. Thus, we parameterize the embedding algorithm with a threshold such that the algorithm stops embedding a candidate as soon its embedding distance exceeds the threshold. Instead of storing the embedded distance of all candidates, we dynamically track the lowest and second-lowest distances, and simply use the latter as the threshold parameter.

**Prioritizing.** Our third optimization is a policy that drives pipelining and early-stop mechanisms. It is based on the intuition that if candidates with low embedding distance are prioritized before others, the overall cost of embedding will be low. This is due to the fact that the threshold will be set to a relatively low value during the early stages of embedding. As a result, early-stop will be applied to most candidates. However, the embedding distance of candidates is not known to us in advance. Therefore, we use candidate counting, a technique used by SFE aligners for count filtering (Liao *et al.* (2013)), to prioritize candidates based on the assumption that candidates with higher counts or votes are more likely to have lower embedding distance, and more likely to be picked as the best candidate. Thus, we modify the seeding phase to associate with each candidate location a count of the number of k-mers that produced that location during the hash lookup. During embedding, we first embed the candidate with the highest count followed by all other candidates. It is important to note here

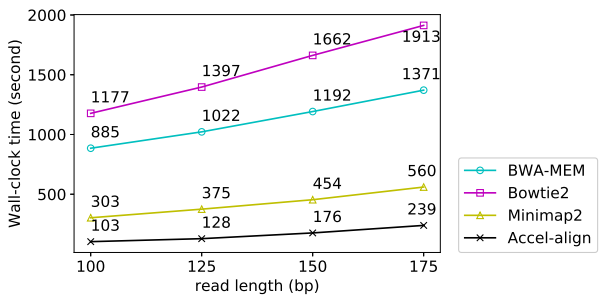



**Fig. 2.** Alignment time for 100bp, 125bp, 150bp and 175bp read

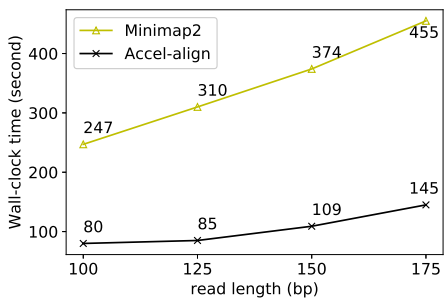



**Fig. 3.** Alignment-free mapping time for 100bp, 125bp, 150bp and 175bp read

that we still embed all candidates, albeit in a different order. Thus, unlike SFE aligners, we do not filter out candidates based on k-mer counting.

## 3 Results

In this section, we present an evaluation of Accel-Align using both simulated and real data to compare its performance and accuracy with respect to three state-of-the-art short-read aligners, namely, BWA-MEM (v0.7.17; Li (2013)), Bowtie2 (v2.3.5; Langmead and Salzberg (2012)), and Minimap2 (v2.17; Li (2018)). Accel-Align is implemented in C++ and is configured to use 2NE algorithm by default as it was found to be faster than 3NE with comparable accuracy (Section 3.1.4). Accel-Align uses Intel Thread Building Blocks for parallelizing both index generation and alignment.

All experiments were run on a server equipped with a quad-core Intel(R) Core(TM) i5-7500 CPU clocked at 3.40GHz, and 32GB RAM. In an offline phase not reported here, we used each aligner to pre-index the reference genome. Then, in each alignment experiment, we run the aligner five times and gather execution statistics. As all aligners read the index from secondary storage, the first run is typically "cold" as data is not in memory. Hence, we ignore the first run. As we found the performance of the last four runs to be stable with all aligners, we only report the average of last four execution times.

### 3.1 Benchmark with simulated short reads

For benchmarking Accel-Align, we used Mason2 (Holtgrewe (2010)) to generate simulated reads from the hg37 reference genome (hs37-1kg) together with an alignment file describing the exact coordinate of each read. We used Accel-Align, BWA-MEM, Bowtie2, and minimap2 (short-read mode) to align the reads and measured the end-to-end wall clock time for alignment. Using the Mason2 generated alignment file as our ground truth, we also evaluated the accuracy of each aligner in terms of the fraction of reads correctly mapped; we consider a read to be correctly mapped if the reported alignment overlaps with the Mason-provided one by at least ten percentage of read length.

#### 3.1.1 Aligner comparison

Table 1 reports the performance and accuracy of the four aligners for a 10M, 100bp, single-end simulated read dataset generated by Mason2. In terms of performance, it can be seen that Accel-Align clearly outperforms the other aligners, as it is 9× faster than Bowtie2, 6× faster than BWA-MEM, and 3× faster than Minimap2.

Table 2 reports the performance and accuracy for a 10M, 100bp, paired-end dataset generated by Mason2. Accel-Align outperforms the other aligners by an even larger margin here, as it is 12× faster than Bowtie2, 9× faster than BWA-MEM, and 3× faster than Minimap2. In terms of accuracy, all four aligners are comparable as a majority of reads are correctly mapped in both the single-end and paired-end datasets. BWA-MEM and Bowtie2 offer marginally better accuracy than Minimap2 and Accel-Align. These results demonstrate that at comparable accuracy, Accel-Align can provide an order of magnitude improvement in performance over some state-of-the-art aligners.

Table 1. Evaluation on simulated single-end data

| | BWA-MEM | Bowtie2 | Minimap2 | Accel-align |
|---|---|---|---|---|
| Exec. time (HH:MM:SS) | 00:06:46 | 00:09:39 | 00:02:53 | 00:01:04 |
| %Correctly mapped | 97.38% | 97.25% | 96.53% | 96.82% |

Table 2. Evaluation on simulated paired-end data

| | BWA-MEM | Bowtie2 | Minimap2 | Accel-align |
|---|---|---|---|---|
| Exec. time (HH:MM:SS) | 00:14:45 | 00:19:37 | 00:05:03 | 00:01:43 |
| %Correctly mapped | 98.58% | 98.55% | 98.17% | 98.39% |

#### 3.1.2 Varying read length

The computational cost of the embedding step is proportional to the read length, as each read and atleast one candidate are converted from length $N$ into a new string of length $2N$. To test the sensitivity of performance with respect to read length, we used Mason2 to generate three more paired-end datasets with 10M reads, where each dataset was configured with a read length of either 125bp, 150bp, or 175bp. Figure 2 shows the performance of the four aligners under various read lengths. We found that all aligners correctly mapped 98.5% of reads. Clearly, Accel-Align outperforms the other aligners at all read lengths as it provides 8–12× improvement over Bowtie2, 6–9× over BWA-MEM, 2.3–3× over Minimap2 across a range of read lengths.

#### 3.1.3 Alignment-free Mapping

Both Accel-Align and Minimap2 can be configured to run in alignment-free mapping mode where they report the position without the CIGAR string. The mapping mode completely eliminates the overhead of edit-distance computation. Although such mapping is useful in several applications that do not require base-by-base alignment, we use it in this context to isolate the benefit of embedding. To compare Accel-Align with Minimap2 in mapping mode, we used the two aligners to perform alignment-free mapping of the four paired-end Mason2 datasets. Figure 3 shows the execution time for various read lengths. Comparing Figures 2 and 3, we can make two observations. First, alignment-free mapping provides a further 1.3× reduction in time over base-to-base alignment. Accel-Align maps 10M, 100bp, paired-end reads in 80 seconds,

thus, mapping more than 250,000 reads per second on a simple quad-core processor without requiring any special hardware. These results demonstrate the benefit of using embedding in sequence mapping. Second, Accel-Align is 3-3.6× faster than Minimap2 at all read lengths (Figure 3) with alignment-free mapping while offering slightly better accuracy (Table 2). This demonstrates that our optimizations have eliminated any the computational overheads associated with embedding, making Accel-Align a competitive alternate to state-of-the-art sequence mappers.

#### 3.1.4 Impact of embedding

To further isolate and understand the benefit of embedding in Accel-Align, we modified Accel-Align to run in three modes: (i) a *no-embed* mode where embedding is not used, and all candidate locations identified by seeding are directly forwarded for extension, (ii) the default mode using 2NE, and (iii) using 3NE instead of 2NE. Table 3 shows the performance and accuracy results for these three modes under the 10M, 150bp, Mason2 pair-end, simulated-read dataset.

Table 3. Comparison of $2N$-embedding and $3N$-embedding

| | No-embedding | $2N$-embedding | $3N$-embedding |
|---|---|---|---|
| Exec. time (HH:MM:SS) | 00:29:24 | 00:02:56 | 00:03:16 |
| not mapped | 0.000536 | 0.0005361 | 0.0005365 |
| correctly mapped | 98.78% | 98.56% | 98.54% |

Comparing 2NE and 3NE cases, we can see that 2NE provides a 11% improvement in performance with no discernible difference in accuracy. Comparing 2NE and the no embedding cases, we see that embedding provides a 10× reduction in execution time as it is able to identify the optimal candidate location without relying on edit distance computations at a marginal 0.2% lower accuracy.

### 3.2 Benchmark with real short reads

To evaluate the accuracy of Accel-Align on real data, we used the human whole-exome sequencing dataset NA12878 (accession No.: SRR098401). We built a pipeline similar to prior work (Kumaran (2019)) to detect variants using GATK HaplotypeCaller (v4.1.0) (DePristo *et al.* (2011)). We used the four aligners to align 85M paired-end reads in NA12878 to the hg37 reference genome. Then, we used the SureSelect Human All Exon v2 target captured kit bed file (ELID: S0293689) for capturing variant locations, and took high confidence variant calls (v2.19) from Genome in a Bottle (GiaB) consortium for validation.

Table 4. Evaluation on real data

| | BWA-MEM | Bowtie2 | Minimap2 | AA-32-mer | AA-25-mer |
|---|---|---|---|---|---|
| Execution time | 1:32:02 | 1:48:09 | 00:31:24 | 00:10:02 | 00:14:38 |
| Ti/Tv | 2.71 | 2.73 | 2.73 | 2.85 | 2.78 |
| Precision | 0.961 | 0.967 | 0.96,7 | 0.981 | 0.972 |
| Recall | 0.952 | 0.948 | 0.952 | 0.923 | 0.936 |
| F-score | 0.956 | 0.957 | 0.959 | 0.952 | 0.954 |
| Fraction Mapped | 94.53% | 95.51% | 94.90% | 90.27% | 92.99% |

We compare the aligners with respect to several metrics as shown in Table 4. The results for Accel-Align using the default parameters is shown in the column AA-32-mer. The execution time reports the wall-clock time taken by various aligners for aligning 85M paired-end reads (or 170M reads in total). Accel-Align provides a speedup of 10× over Bowtie2, 8× over BWA-MEM, and 3× over Minimap2, similar to the Mason2 dataset. The second metric is transition-to-transversion ratio (Ti/Tv), which is a key metric in detecting SNVs and should fall between 2.6–3.3 for this dataset, which is the case with all aligners. The last three metrics report precision, recall, and F-score values based on the GiaB truth set contains 23,686 SNVs and 1,258 InDels contributing to a total of 24,944 variants for the NA12878 exome. The overall F-score of all four aligners are comparable. Accel-Align offers a slightly higher precision than the other aligners as most variants detected with Accel-Align are true variants. However, Accel-Align provides slightly lower recall, as it fails to detect some variants in the truth set.

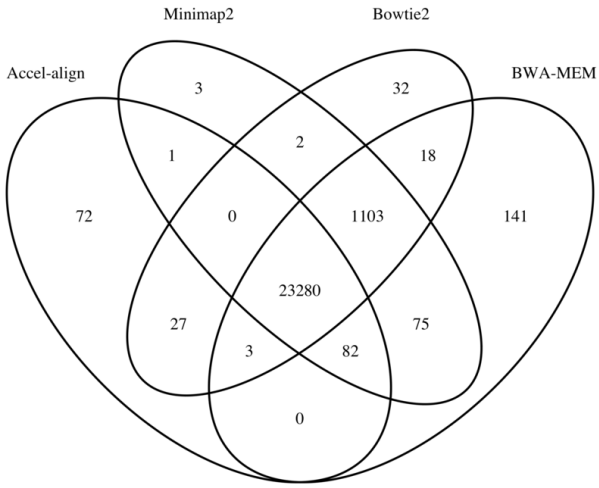



**Fig. 4.** Venn diagram of variants detected by various aligners and Accel-Align (32-mer)

To better understand the difference between detected variants, we show a Venn diagram of variants detected by various aligners in Figure 4. Clearly, 95% of variants are captured by all aligners. We find that there are 1103 variants detected by the other three alignment tools, but not by Accel-Align. Upon further inspection, we found this to be due to two reasons. First, although embedding identifies the correct candidate location in most cases, there are reads for which it chooses the wrong candidate location. We found this to be particularly problematic for reads that map with a low edit distance to multiple locations in the reference. In such cases, we found that the hamming distance of embedded candidates is not spaced apart for embedding to identify a clear target. The second reason is the use of large kmer length (32) in Accel-Align compared to the very short read length in NA12878 (76bp) which resulted in reads with two erroneous k-mers being unmapped. Table 4 shows the percent of mapped reads, and as can be seen, Accel-Align aligns 4% fewer reads.

To understand the impact of k-mer size on overall accuracy, we modified Accel-Align to use 25-mers instead of 32-kmers. Table 4 shows the results obtained using the 25-kmer setting under column AA-25-mer. Comparing it with AA-32-kmer, we can two important observations. First, the overall execution time increases by 45% compared to AA-32-mer case. Figure 5 shows a breakdown of execution time across the three stages when 25-mers or 32-mers are used. Clearly, this increase in time can be attributed almost entirely to seeding as 25-mers produce 4× more candidates than 32-mers. As SEE does not filter out any candidates, the overhead of candidate normalization, counting, and duplicate elimination performed during seeding increases. Despite this, embedding is able to identify the candidates, avoid needless extension, and still provide 2-7×

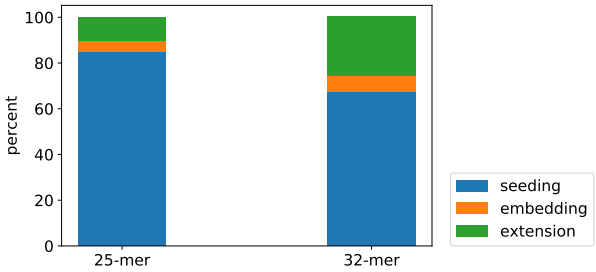



**Fig. 5.** the percent of seeding, embedding and extension over the total processing time.

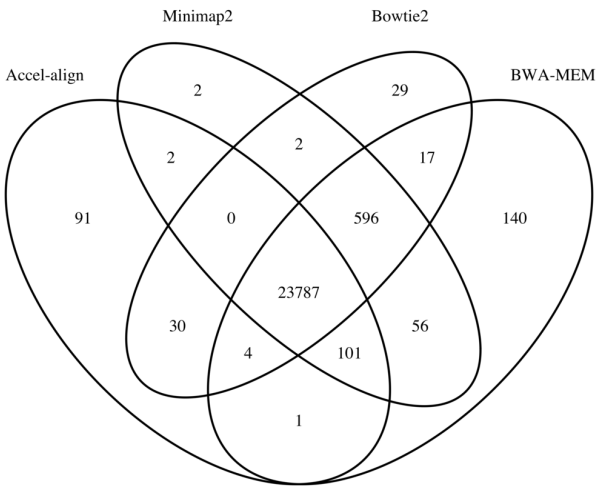



**Fig. 6.** Venn diagram of variants detected by various aligners and Accel-Align (25-mer)

reduction in execution time over other aligners. Second, the fraction of mapped reads increases by 2.8% by using 25-mers instead of 32-mers. Comparing Figures 4 and 6, we can see that this translates to an increase in the number of variants detected by Accel-Align. However, while the 25-mer case has better recall rate than the 32-mer case, it has a slightly lower precision as shown in Table 4. As a result, in terms of the overall F-score, the 25-mer case provides similar accuracy to the 32-mer case.

## 4 Discussion

As sequencing technologies continue to increase read length while improving throughput and accuracy, we believe that randomized embeddings open up new avenues for optimization that cannot be achieved by using edit distance. Accel-Align clearly demonstrates the potential of using embedding for short-read sequence alignment. The SEE method used by Accel-Align is a design principle that is parameter free, applicable to any sequencing technology, and can deal with both indels and mismatches without requiring any customization. Further, the benefit of SEE methodology and low-distortion embedding are not limited to short-read alignment; any computational biology problem that is limited by the overhead of edit distance can benefit from embedding. As embedding transforms strings from edit to Hamming regime, computational tools like Locality Sensitive Hashing can also be applied on the resulting strings. Thus, the techniques presented in this paper have a much broader scope as they can be used for other applications like spliced RNA-seq and bi-sulfite alignment, multiple sequence alignment, and even sequence assembly.

## 5 Conclusion and Future Work

In this work, we presented Accel-Align–a fast sequence mapper and aligner that uses the SEE method for quickly identifying optimal candidates based on randomized embedding. Using an extensive evaluation, we showed that Accel-Align specifically, and SEE more generally, can provide an order of magnitude improvement over some state-of-the-art aligners on commodity CPUs. A natural extension of this work that we are pursuing is investigating the effectiveness of SEE for long-read alignment. Research on string-similarity join has demonstrated that these randomized embeddings excel when applied to long strings with even up to 20% mismatch (Zhang and Zhang (2017)). This suggests that SEE can also be used for aligning error-prone long reads generated by Nanopore or PacBio sequencers. Another line of work that we are exploring involves exploiting hardware-acceleration techniques for implementing SEE across CPUs and GPUs. Looking at Figure 5, we can see that seeding dominates overall execution time at low k-mer sizes and accounts for a substantial portion of time with large k-mers. In a recent microarchitectural study (Appuswamy *et al.* (2018)) of various sequence aligners, we demonstrated that the seeding stage is a prime candidate for acceleration by GPUs due to its data-parallel nature, unlike the extension stage which is difficult to parallelize due to data dependencies in dynamic programming. Thus, we are developing a parallel version of Accel-Align which can work across CPUs and GPUs and provide better accuracy, and a 10× further improvement in performance over the purely CPU-based Accel-Align.

## References

Appuswamy, R., Fellay, J., and Chaturvedi, N. (2018). Sequence alignment through the looking glass. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*.

Backurs, A. and Indyk, P. (2015). Edit distance cannot be computed in strongly subquadratic time (unless seth is false). In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, page 51–58.

Canzar, S. and Salzberg, S. L. (2017). Short read mapping: An algorithmic tour. *Proceedings of the IEEE*, **105**(3), 436–458.

Chakraborty, D., Goldenberg, E., and Koucký, M. (2016). Streaming algorithms for embedding and computing edit distance in the low distance regime. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 712–725.

DePristo, M. A., Banks, E., Poplin, R., Garimella, K. V., Maguire, J. R., Hartl, C., Philippakis, A. A., Del Angel, G., Rivas, M. A., Hanna, M., *et al.* (2011). A framework for variation discovery and genotyping using next-generation dna sequencing data. *Nature genetics*, **43**(5), 491.

Holtgrewe, M. (2010). Mason: a read simulator for second generation sequencing data.

Kumaran, M., S. U. . D. B. (2019). Performance assessment of variant calling pipelines using human whole exome sequencing and simulated data. *BMC bioinformatics*, **20**(342).

Langmead, B. and Salzberg, S. L. (2012). Fast gapped-read alignment with bowtie 2. *Nature methods*, **9**(4), 357.

Li, H. (2013). Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv preprint arXiv:1303.3997*.

Li, H. (2018). Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, **34**(18), 3094–3100.

Liao, Y., Smyth, G. K., and Shi, W. (2013). The Subread aligner: fast, accurate and scalable read mapping by seed-and-vote. *Nucleic Acids Research*, **41**(10).

Smith, T. F., Waterman, M. S., *et al.* (1981). Identification of common molecular subsequences. *Journal of molecular biology*, **147**(1), 195–197.

Suzuki, H. and Kasahara, M. (2018). Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC bioinformatics*, **19**(45).

Zhang, H. and Zhang, Q. (2017). Embedjoin: Efficient edit similarity joins via embeddings. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 585–594.

Zhang, X., Yuan, Y., and Indyk, P. (2019). Neural embeddings for nearest neighbor search under edit distance.