1 SUPPLEMENTARY MATERIALS

1.1 Local Alignment and Filtering in Read Mapping

In read mapping, the *Levenshtein-distance* (*edit-distance*) is used to measure the similarity between read and reference DNA segments at each potential mapping site. The edit-distance is defined as the minimum number of edits (i.e. insertions, deletions, or substitutions) required to make the segments match exactly. If the edit-distance is greater than the user-defined error threshold e, the mapping site is rejected.

For most scientific research, small error thresholds (e) are required, usually less 5% of the read length. In Figure 8, we computed the exact edit-distance between 100 basepair long reads (reads are real data from 1000Genome project) and their corresponding potential locations in the Human reference genome as determined by the popular seed-and-extend based mapper, mrFAST Alkan *et al.* (2009). For $e = 0, 1, \ldots, 5$, more than 98% of the potential locations did *not* meet the error threshold. Moreover, over half of all potential locations contained more than 20 errors. Seed-and-extend based mappers that do not filter these clearly invalid mappings waste much of their time performing computationally expensive edit-distances calculations.

Many mechanisms have been proposed to efficiently calculate the edit-distance of strings and filter out incorrect mappings before computationally expensive methods are applied. These mechanisms can be divided into five main classes: (i) dynamic programming (DP) algorithms, (ii) SIMD implementations of DP algorithms, (iii) bit-vector implementations of DP algorithms, (iv) Hamming distance calculation, and (v) locality based filtering mechanisms. We briefly describe each next.

1.1.1 Dynamic-programming algorithms (DP algorithms) Most mappers rely on dynamic-programming methods to compute edit-distances. The classical approach, proposed by Smith and Waterman Smith and Waterman (1981), is to generate a $(l + 1) \times (l + 1)$ edit-distance matrix, where l is the length of the read and the reference. The edit-distance matrix is recursively defined in a top-to-bottom and left-to-right manner. The first row (0th row) and the first column (0th column) of the matrix are initialized to the sequences $0, 1, 2, \ldots, l$ and $0, 1, 2, \ldots, l$ respectively. The value of each element is dependent on its top neighbor, its left neighbor, its top-left neighbor, and the comparison of basepairs from the read and the reference. This relationship is formally described by Equation 1:

$$m_{i,j} = \min \begin{cases} m_{i-1,j} + 1, \\ m_{i,j-1} + 1, \\ \begin{cases} m_{i-1,j-1} & \text{read[i]} = \text{reference[j]} \\ m_{i-1,j-1} + 1 & \text{otherwise} \end{cases} \end{cases} \tag{1}$$

The value of the matrix element at row i and column j represents the number of edits between read[1..i] and reference[1..j]. The complexity of the Smith-Waterman algorithm is $O(l^2)$. However, in read mapping, computing the precise edit-distance between the read and the reference sequences would be wasteful, as it is only necessary to determine if the two sequences differ by more than e errors. Ukkonen's algorithm Ukkonen (1985) takes advantage of this fact and only determines if two sequences differ by more than the error threshold by calculating a strip of $2e + 1$ diagonal lanes of the matrix, rather than the entire matrix as in the Smith-Waterman approach. The time complexity of Ukkonen's improved DP algorithm is $O((2e + 1) \times l)$.

1.1.2 Bit-vector implementations of DP algorithms Several bit-vector algorithms Hyyro *et al.* (2005); Myers (1999) that exploit bit-parallelism in DP algorithms have been proposed. By making the observation that elements in the edit-distance matrix of DP algorithms differ from their top and left neighbors by at most 1 (± 1), the edit-distance matrix can be transformed into two series of bit-vectors. We choose Gene Myers' bit-vector implementation Myers (1999) as a representative algorithm. In Gene Myers' algorithm, bit-vectors record differences between consecutive columns of the edit-distance matrix. Because each element can either be $+1$, -1, or 0 away from its left neighbor, two separate bit-vectors are used for each column: one indicates rows where the elements are $+1$ away from their left neighbors while the other indicates rows where the elements are -1 from their left neighbors. Gene Myers further proves that the differences between each pair of cells in two consecutive columns can be computed in parallel. A *minimum edit-distance score* for each column is computed as part of this process.

Although the complexity of the algorithm is $O(l^2)$, the runtime of Gene Myers' algorithm is much faster than basic DP algorithms. Each bit-vector is mapped to a few computer registers; therefore, applying an operation to the register is equivalent to applying the same operation to many elements in the edit-distance matrix. If each register has w bits, theoretically Gene Myers' algorithm can provide a $(w/|S|)\times$ speedup over the basic Smith-Waterman DP algorithm, where S is the cardinality of some set of symbols ($S\ of\ |\{\text{A, C, T, G}\}| = 4$ in DNA). Nonetheless, even if the registers are wide enough to store an entire bit-vector ($w \geq l + 1$), Gene Myers' algorithm still requires $O(l)$ bit-wise operations to cascade the computation for l total bit-vectors.

1.1.3 SIMD implementations of dynamic-programming algorithms Another approach to speed up the basic DP algorithm is to efficiently map the DP algorithm to SIMD units Manavski and Valle (2008); Szalkowski *et al.* (2008); Farrar (2007). Many modern computers have SIMD units, such as GPUs and vector units in CPUs. These vector units pack multiple data elements into a single, wide register and apply the same instruction to all of the packed data elements simultaneously. SIMD implementations of DP algorithms, such as swps3, exploit the data parallelism between elements in the edit-distance matrix. In swps3, elements in the edit-distance matrix are mapped to SIMD registers in a striped manner. Data is placed such that within a single register there are no dependencies bewteen elements. Also, elements within a register share dependencies on other registers. Therefore, elements within a SIMD register are synchronized for SIMD operations (either all of them are ready or none of them are ready).

Similar to bit-vector implementations of DP algorithms, SIMD implementations do not reduce the complexity of the algorithm, but speedup the process by exploiting parallelism within the algorithm. Theoretically, a SIMD platform that packs p elements into a single register, can provide up to $p\times$ speedup over basic Smith-Waterman implementation. In practice, however, due to extra computation spent on data mapping and other auxiliary processing, the speedup of SIMD implementations is generally smaller than p. For example,

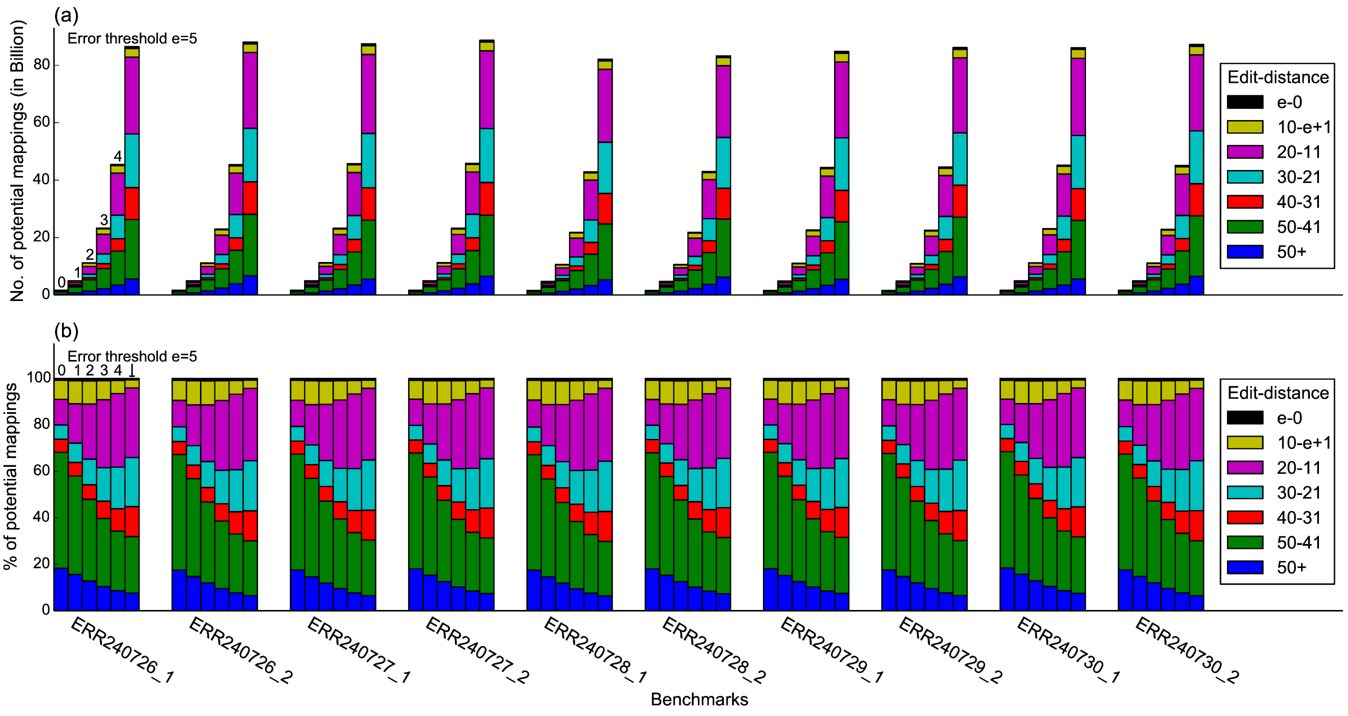

Fig. 8: Figure (a) shows the edit-distance breakdown and Figure (b) shows the edit-distance distribution of all potential mappings evaluated by mrFAST. Both plots show that a vast majority of the potential mappings have far more errors than the threshold e.

swps3 uses Intel SSE, which packs 16 elements into a single register, while providing a maximum speedup of only 8×.

1.1.4 Hamming distance Another method to measure the similarity between the read and the reference is *Hamming distance*. The Hamming distance between a pair of strings is defined as the number of positions at which the corresponding symbols are different. As such, the Hamming distance measures the pure letter-wise differences between strings. Hence, it can only find substitutions.

Hamming distance can be calculated in a bit-parallel fashion. Consider a set of symbols, S ({A, C, T, G} for DNA), which can be represented in $s = log_2(|S|)$ bits ($s = 2$ for DNA). A string of m symbols is decomposed into s bit-vectors, each with m bits. Each bit-vector contains one bit per symbol: the first bit-vector represents the first bit of all symbols; the second bit-vector represents the second bit of all symbols; etc. The *Hamming mask* of two strings is simply a bit-vector representing matches and mismatches between them. In this Hamming mask, a '0' represents a basepair match between the read and the reference while a '1' represents a mismatch. The Hamming distance of two strings is simply to the number of '1's in the Hamming mask. Algorithm 2 provides the pseudocode for calculating the Hamming mask of two strings A and B. In the pseudocode, both strings, A and B, are represented as two sets of s bit-vectors, A[0]...A[s-1] and B[0]...B[s-1]. The Hamming mask of A and B can be calculated using s bit-wise XOR and bit-wise OR operations.

Hamming distance cannot correctly distinguish insertions or deletions from substitutions. Each insertion and deletion can shift multiple trailing symbols in a string and create multiple mismatches in the Hamming mask (as bits of '1's). Therefore, Hamming distance is either strictly greater than or equal to the edit-distance of two strings.

Algorithm 2: ComputeHammingMask

```
Inputs: $A[0]...A[s-1]$, $B[0]...B[s-1]$ (bit-vectors of string A and B)
Outputs: $HMask$ (the Hamming mask of string A and B)
$HMask = 0$;
for $i = 0$ to $s - 1$ do
    $TempMask = A[i]$ ^ $B[i]$;
    $HMask = HMask$ | $TempMask$;
return $HMask$
```

1.1.5 Filtering algorithms exploiting locality among k-mers It is not always necessary to calculate the edit-distance between two strings to verify a potential mapping. Incorrect mappings can also be filtered out with simple searches. Several works Ahmadi *et al.* (2011); Xin *et al.* (2013) have shown that the potential mappings of a read can be verified by checking (searching) relative distances between its k-mers (small subsequences of the read that are used to generate potential mappings). We select Adjacency Filtering (AF) from FastHASH Xin *et al.* (2013) as a representative example. In AF, if a read can be divided into m non-overlapping k-mers, then with e allowed errors, at least $m - e$ k-mers should be located near the mapping site in the reference for correct mapping. For

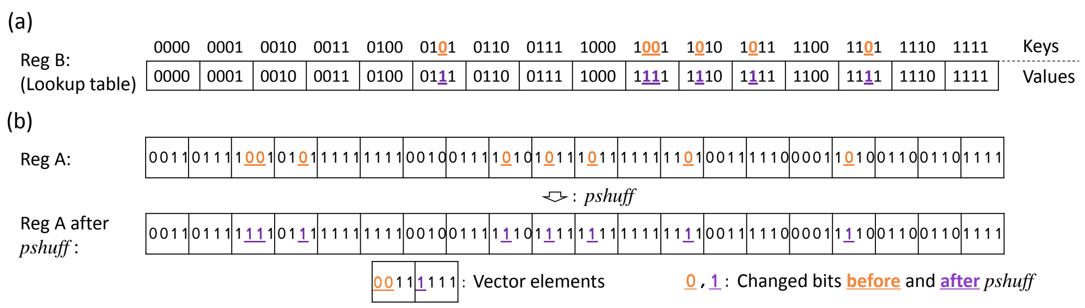

Fig. 9: Figure (a) shows the lookup table that is used to replace short streaks of ‘0’s with amended ‘1’s. In this table, ‘0’s that are bounded by ‘1’s in a binary key are replaced by amended ‘1’s in the value.
Figure (b) shows an example of amending short streaks of ‘0’s using packed shuffle (*pshuff*) operations.

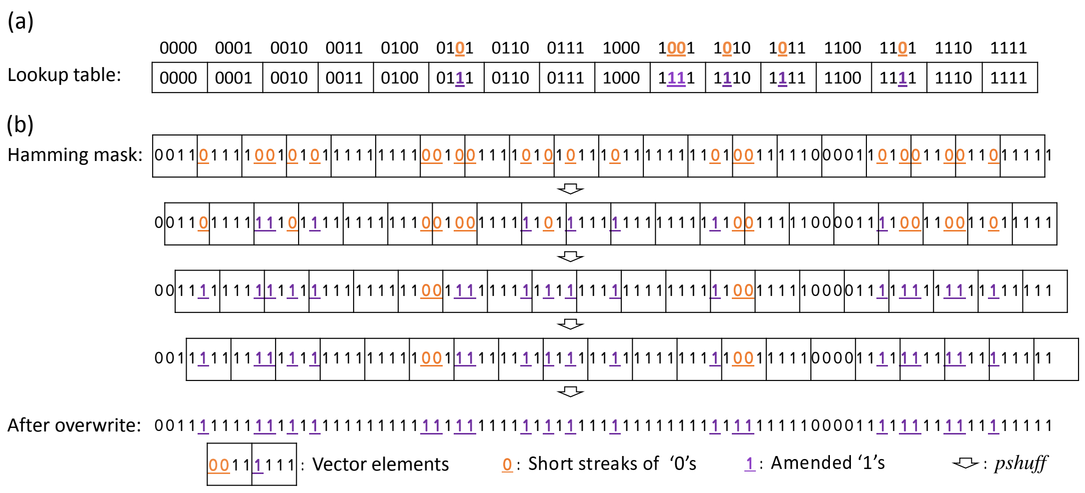

Fig. 10: Figure (a) shows the lookup table that is used to replace short streaks of ‘0’s with amended ‘1’s.
Figure (b) shows the workflow of SRS_amend, according to Algorithm 3. SRS replaces all short streaks of ‘0’s in a Hamming mask using four *right shifts* and *pshuffs*.

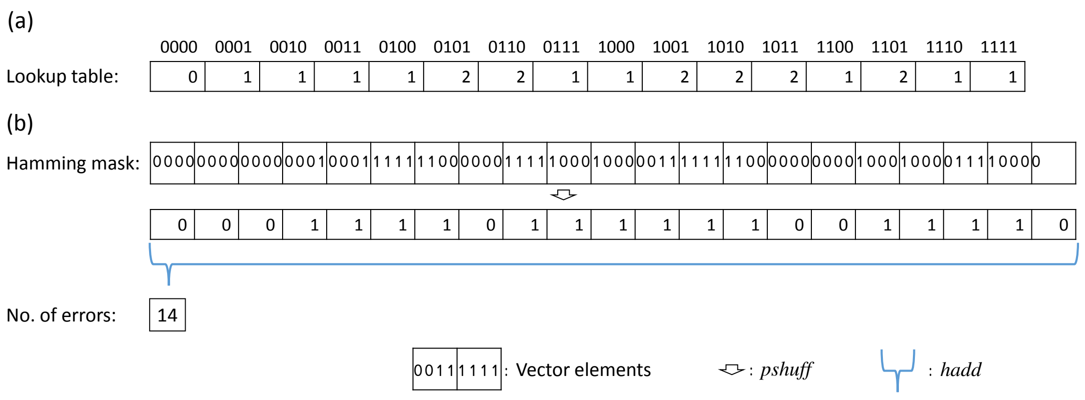

Fig. 11: Figure (a) shows the lookup table that is used to count the number of errors in the final bit-vector of SHM. The value of a key is the minimum number of errors that the key can cover.
Figure (b) shows the workflow of SRS_count, according to Algorithm 4. The minimum number of errors is calculated with a *pshuffs* followed by a *hadd* (horizontal add to sum up all errors).

example, a potential mapping location, loc, is valid for the ith k-mer of the read if there exists a location within the expected range of $[loc + i \cdot k - e, loc + i \cdot k + e]$ in the location list of the k-mer (there is a range of $\pm e$ because of possible insertions/deletions from other k-mers). A potential mapping location passes AF only if more than $m - e$ k-mers pass the locality check.

Filters that are similar to AF work well only when e is very small (e.g., below 1/3rd of m). When e gets larger, their accuracy decreases drastically. There are two reasons for this. First, with a larger e, the range of expected locations ($[loc+i\cdot k-e, loc+i\cdot k+e]$) expands, which causes more potential locations to pass locality checking. Second, the required minimum number of k-mers that exhibit locality around loc, $m - e$, is largely reduced, since m is typically a smaller number of 10 or less. As a result, filtering mechanisms exploiting locality among k-mers are not favorable for large e (e.g., $e \geq 3$).

1.1.6 Conclusion Identifying incorrect mappings quickly is crucial to seed-and-extend based mappers. Previous works are either slow or inaccurate. Calculating the edit-distance of a potential mapping using DP algorithm (Section 1.1.1) or its SIMD (Section 1.1.3) or bit-vector (Section 1.1.2) variations is slow, while Hamming distance (Section 1.1.4) and locality-based filters (Section 1.1.5) are fast but have many false positives (they allow many incorrect mappings to pass). A **high speed**, **low false positive**, and **zero false negative** filter that is effective under moderate error thresholds (e.g., up to 8% of read length) is needed to further improve the performance of seed-and-extend based mappers. In this article, we introduce such a filter.

1.2 SRS SIMD Implementation

In this section, we provide implementation details of SRS_amend and SRS_count, which are essential functions of SHD, as Algorithm 1 shows.

To achieve high efficiency, we implemented both functions using Intel SSE instructions. Specifically, we used the *packed shuffle* operation, a SIMD parallel-table-lookup instruction provided by the Intel SSE instruction set.

Packed shuffle ($pshuff$) takes two vectors of integers $A = [a_0\ a_1\ a_2\ ...]$ and $B = [b_0\ b_1\ b_2\ ...]$ and replaces integers in vector A with integers in vector B with the result $A = [B[a_0]\ B[a_1]\ B[a_2]...]$. Figure 9 gives an example of applying *pshuff* on a single bit-vector. Notice that the values of the integers in vector A must not exceed the maximum size of vector B, otherwise an incorrect result is returned. On Intel platforms, the maximum size of an SSE vector is 16 (with single-byte integers). Therefore, on Intel platforms, the value of the elements in either vectors must not exceed 16 (1111 in binary, four bits).

1.2.1 Implementation of SRS_amend Algorithm 3 shows the pseudocode of SRS_amend. We use packed shuffle to efficiently replace all short streaks of '0's with '1's in parallel. According to SRS (Section 3.2), all streaks of '0's shorter than three (and are bounded by '1's) are turned into streaks of '1's. As a result, bit streams such as 101, 1001 are replaced with 111 and 1111, respectively. Packed shuffle can drastically speed up this replacement process by storing the pre-processed replacement patterns in a lookup table and replacing target bit streams in parallel at runtime. Specifically, the lookup table stores replacement bit streams for binary keys ranging from 0000 to 1111 (16 keys in total), as shown in Figure 9 (a). Among all keys, for those that have '0's in the middle of '1's, such as 0101 or 1010, their values replace the middle '0's with '1's (0101 $\rightarrow$ 0111 and 1010 $\rightarrow$ 1110); for all other keys, their values are simply the keys themselves (hence packed shuffle does not change them). Figure 9 (a) presents the entire lookup table for SRS_amend. Figure 9 (b) presents an example of amending short streaks of '0's using a single packed shuffle operation.

To successfully amend all short streaks of '0's into '1's, however, a single packed shuffle operation is not enough. Short streaks of '0's that span two neighboring keys in the lookup table stays unchanged after the packed shuffle operation (e.g., 0010 0100 $\rightarrow$ 0010 0100). To solve this problem, we incrementally shift the bits in the vector to the right (as shown in Figure 10 (b)) which eventually brings the short streak of '0's into one key (0010 0100 $>>$ 001 0010 0 $>>$ 00 1001 00 $\rightarrow$ 00 1111 00). Figure 10 (b) illustrates SRS_amend (Algorithm 3) amending all short streaks of '0's in a Hamming mask through four *pshuff* operations and shifts.

Algorithm 3: SRS_amend

```
Inputs: HMask (Hamming mask), LTable (lookup table)
SIMD Registers: r1, r2 and r3
Outputs: SRS_HMask (SRS amended Hamming mask)
r1 = HMask;
r2 = LTable;
r3 = pshuff(r2);
r1 = r1 | r3;
for $i = 1$ to $n - 1$ do
    r3 = r1 >> $i$;
    r3 = pshuff(r2);
    r3 = r1 << $i$;
    r1 = r1 | r3;
SRS_HMask = $R1$;
return SRS_HMask;
```

Note that to amend a short streak of '0's using packed shuffle, the entire short streak of '0's and the bounding '1's must fit into the space of a single key at once (e.g., $\underbrace{1001}$ but not $\underbrace{1000}1$). Since the packed shuffle operation on Intel platforms only supports a key length of four bits at maximum, it cannot amend any short streak of '0's longer than two. As a result, we choose an SRS threshold of *three bps* in this paper. A study of the effect of other SRS thresholds on the false positive rate is provided in Section 3.3.

1.2.2 Implementation of SRS_count Similar to SRS_amend, SRS_count is implemented using packed shuffle as well. According to SRS (Section 3.2), a streak of '1's in the final bit-vector of SHM is always assumed to be amended from back-to-back short streaks of '0's. Therefore, the number of errors of a short streak of '1's must be counted as the minimum errors that this streak can cover.

Using packed shuffle, we can quickly provide a lower bound of the number of errors that the final bit-vector contains. In this case, the lookup table of packed shuffle stores the minimum number of errors that each key covers. For example, key 0100 clearly covers only a single error hence, stores a "1" in the table while key 0110 clearly covers two errors hence, stores a "2" in the table. However, for keys such as 1100 or 1111, it is unclear what bits are next to them hence the minimum number of errors that they cover is hard to determine (e.g., in bit-stream [0000 1100] 1100 covers two errors, but in bit-stream [0001 1100] 1100 covers only one error). In order to preserve correctness (such that we do not not over-estimate errors under for any input) while maintaining speed, a lower bound of errors is (always) assumed for such keys. A complete lookup table for SRS_count is provided in Figure 11 (a).

```
Algorithm 4: SRS_count
  Inputs: Final_BV (the final bit-vector of the SHM), LTable
  (lookup Table)
  SIMD Registers: r1 and r2
  Outputs: errorNum (minimum number of errors in the
  bit-vector)
  r1 = Final_BV;
  r2 = LTable;
  r1 = $pshuff$(r2);
  errorNum = $hadd$(r1);
  return errorNum;
```

Algorithm 4 provides the pseudocode of SRS_count. As the pseudocode shows, we first load the pre-processed lookup table into a SIMD register. Then, we count the minimum number of errors of each key in the final bit-vector using packed shuffle. Finally, we sum up all minimum numbers of errors of all keys using a horizontal add (*hadd*). The final sum is a lower bound of the minimum number of errors of the final bit-vector (and the potential mapping). Figure 11 (b) visualizes the entire workflow of Algorithm 4. In this figure, a minimum of 14 errors is counted from the final bit-vector, which indicates that the potential mapping must be erroneous hence must be rejected.

1.2.3 Code Availability Note that we have made the described implementation of SHD available at:
https://github.com/xhongyi/SHD_code.

1.3 Comparison against Hamming Distance Filter

For completeness, we also compared SHD against our homegrown Hamming distance filter. To provide a fair comparison, our homegrown Hamming distance filter also converts both the read and the reference strings into bit-vectors (the same way as SHD does) before calculating the Hamming distance. We also used Intel SSE in our homegrown Hamming distance filter implementation, which includes a vector *XOR* followed by a *popcount* (counting the number of '1's). Both SHD and the Hamming distance filter are benchmarked with the mrFAST front-end. Table 2 presents the average time each program (SHD and Hamming distance filter) spends on verifying 1 million string pairs.

As shown in Table 2, SHD consumes more time on average to verify 1 million string pairs than our homegrown Hamming distance filter. This is because SHD computes multiple Hamming masks in SHM while each Hamming mask is further processed by SRS. As a result, SHD is much more complex than just calculating the Hamming distance. In return, SHD supports handling insertions and deletions whereas Hamming distance does not. Supporting indels is a key feature of SHD.

Error Threshold (bps)	Hamming Distance Filter (s)	SHD (s)
e=0	0.212	0.28
e=1	0.211	0.30
e=2	0.211	0.32
e=3	0.211	0.35
e=4	0.212	0.375
e=5	0.211	0.384

Table 2. The average time SHD and our homegrown Hamming distance filter spend on verifying 1 million string pairs, with error thresholds 0-5.

SHD always spends more time to verify 1 million string pairs when the error threshold increases. This is expected because the complexity of SHD increases linearly with a larger error threshold.

On the contrary, our homegrown Hamming distance filter spends a similar amount of time to verify 1 million string pairs regardless of how large the error threshold is. This is expected as the complexity of the Hamming distance filter remains the same for all error thresholds as Hamming distance has no ability to handle indels (This ability is a key feature of SHD).

Note that both SHD and our homegrown Hamming distance filter have to convert each string pair into bit-vectors before calculating the edit-distance (or Hamming distance). This conversion overhead is included in the execution time of both implementations in Table 2. To measure the overhead of bit-vector conversion, we benchmarked converting 1 million string pairs into bit-vectors. It takes on average 0.209 seconds to convert 1 million string pairs into bit-vectors. This conversion overhead can be drastically reduced in a mapper by pre-calculating the reference genome into bit-vectors and converting each read *only* once (Note that each read can produce many string pairs to be verified). However, such optimization is mapper-specific and will be explored in our future research.

Importantly, even though our homegrown Hamming distance implementation is faster than SHD, it does not support indels as SHD does. Table 3 shows the false negative rate (percentage of incorrectly rejected string pairs, which have edit-distances below the error threshold) of SHD and our homegrown Hamming distance filter.

Error Threshold (bps)	Hamming Distance Filter (%)	SHD (%)
e=0	0	0
e=1	0.98	0
e=2	2.12	0
e=3	3.65	0
e=4	5.47	0
e=5	7.53	0

Table 3. The false negative rate of SHD and our homegrown Hamming distance filter, with error thresholds 0-5.

Compared to SHD, which *always* has 0% false negative rate, the Hamming distance filter has an increasing false negative rate under a greater error threshold. This is because a larger portion of the correct mappings contains indels under a larger error threshold while most of these correct mappings with indels are inaccurately rejected by the Hamming distance filter.

To conclude, although SHD is slower than the Hamming distance filter, as it is more complex, it can handle indels. For applications where tolerating indels is essential, only SHD, but not the Hamming distance filter, provides correct functionality.