

## Sequence Alignment

# SLIDER: Fast and Efficient Computation of Banded Sequence Alignment

Mohammed Alser<sup>1,2,3,\*</sup>, Hasan Hassan<sup>2</sup>, Akash Kumar<sup>3</sup>, Onur Mutlu<sup>2,1,\*</sup>, and Can Alkan<sup>1,\*</sup>

<sup>1</sup>Computer Engineering Department, Bilkent University, 06800 Bilkent, Ankara, Turkey, <sup>2</sup>Computer Science Department, ETH Zürich, 8092 Zürich, Switzerland, <sup>3</sup>Institute for Computer Engineering, CFAED, Technische Universität Dresden, Germany

\*To whom correspondence should be addressed.

Associate Editor: XXXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

### Abstract

**Motivation:** The ability to generate massive amounts of sequencing data continues to overwhelm the processing capacity of existing algorithms and compute infrastructures. In this work, we explore the use of hardware/software co-design and hardware acceleration to significantly reduce the execution time of short sequence alignment, a crucial step in analyzing sequenced genomes. We introduce SLIDER, a highly parallel and accurate pre-alignment filter that remarkably reduces the need for computationally-costly dynamic programming algorithms. The first key idea of our proposed pre-alignment filter is to provide high filtering accuracy by correctly detecting all common subsequences shared between two given sequences. The second key idea is to design a hardware accelerator design that adopts modern FPGA (field-programmable gate array) architectures to further boost the performance of our algorithm.

**Results:** SLIDER significantly improves the accuracy of pre-alignment filtering by up to two orders of magnitude compared to the state-of-the-art pre-alignment filters, GateKeeper and SHD. Our FPGA accelerator is up to three orders of magnitude faster than the equivalent CPU implementation of SLIDER. Using a single FPGA chip, we benchmark the benefits of integrating SLIDER with five state-of-the-art sequence aligners, designed for different computing platforms. The addition of SLIDER as a pre-alignment step reduces the execution time of five state-of-the-art sequence aligners by up to 18.8x. SLIDER can be adopted for any bioinformatics pipeline that performs sequence alignment for verification. Unlike most existing methods that aim to accelerate sequence alignment, SLIDER does *not* sacrifice any of the aligner capabilities, as it does *not* modify or replace the alignment step.

**Availability:** <https://github.com/BilkentCompGen/SLIDER>

**Contact:** mohammed.alser@inf.ethz.ch, onur.mutlu@inf.ethz.ch, calkan@cs.bilkent.edu.tr

**Supplementary information:** Supplementary data are available at *Bioinformatics* online.

## 1 Introduction

One of the most fundamental computational steps in most bioinformatics analyses is the detection of the differences/similarities between two genomic sequences. Edit distance and pairwise alignment are two approaches to achieve this step, formulated as *approximate string matching* (Navarro, 2001). *Edit distance* approach is a measure of how much the sequences differ. It calculates the minimum number of edits needed to convert one sequence into the other. The higher is the distance, the more

different are the sequences from one another. Commonly-allowed edit operations include deletion, insertion, and substitution of characters in one or both sequences. *Pairwise alignment* is a measure of how much the sequences are alike. It calculates the alignment that is an ordered list of characters representing possible edit operations and matches required to change one of the two given sequences into the other. As any two sequences can have several different arrangements of the edit operations and matches (and hence different alignments), alignment algorithm usually involves a backtracking step. This step finds the alignment that has the highest *alignment score* (called *optimal alignment*). The alignment score is the

sum of the scores of all edits and matches along the alignment implied by a user-defined *scoring function*. The edit distance and pairwise alignment approaches are *non-additive measures* (Calude, et al., 2002). This means that if we divide the sequence pair into two consecutive subsequence pairs, the edit distance of the entire sequence pair is not necessarily equivalent to the sum of the edit distances of the shorter pairs. Instead, we need to examine all possible *prefixes* of the two input sequences and keep track of the pairs of prefixes that provide an optimal solution. Enumerating all possible prefixes is necessary for tolerating edits that result from both sequencing errors (Fox, et al., 2014) and genetic variations (McKernan, et al., 2009). Therefore, the edit distance and pairwise alignment approaches are typically implemented as dynamic programming algorithms to avoid re-examining the same prefixes many times. These implementations, such as Levenshtein distance [40], Smith-Waterman [28], and Needleman-Wunsch [41], are inefficient as they have quadratic time and space complexity (i.e.,  $O(m^2)$  for a sequence length of  $m$ ). Many attempts were made to boost the performance of existing sequence aligners. Despite more than three decades of attempts, the fastest known edit distance algorithm (Masek and Paterson, 1980) has a running time of  $O(m^2/\log^2 m)$  for sequences of length  $m$ , which is still nearly quadratic (Backurs and Indyk, 2017). Therefore, more recent works tend to follow one of two key new directions to boost the performance of sequence alignment and edit distance implementations: (1) Accelerating the dynamic programming algorithms using hardware accelerators. (2) Developing *filtering heuristics* that reduce the need for the dynamic programming algorithms, given an edit distance threshold.

**Hardware accelerators are becoming increasingly popular for speeding up the computationally-expensive alignment and edit distance algorithms** (Al Kawam, et al., 2017; Aluru and Jammula, 2014; Ng, et al., 2017; Sandes, et al., 2016). Hardware accelerators include multi-core and SIMD (single instruction multiple data) capable central processing units (CPUs), graphics processing units (GPUs), and field-programmable gate arrays (FPGAs). The classical dynamic programming algorithms are typically accelerated by computing *only* the necessary regions (i.e., diagonal vectors) of the dynamic programming matrix rather than the entire matrix, as proposed in Ukkonen’s banded algorithm (Ukkonen, 1985). The number of the diagonal bands required for computing the dynamic programming matrix is  $2E+1$ , where  $E$  is a user-defined edit distance threshold. The banded algorithm is still beneficial even with its recent sequential implementations as in Edlib (Šošić and Šikić, 2017). The Edlib algorithm is implemented in C for standard CPUs and it calculates the banded Levenshtein distance. Parasail (Daily, 2016) exploits both Ukkonen’s banded algorithm and SIMD-capable CPUs to compute a *banded alignment* for a sequence pair with a user-defined scoring function. SIMD instructions offer significant parallelism to the matrix computation by executing the same vector operation on *multiple operands* at once. Multi-core architecture of CPUs and GPUs provides the ability to compute alignments of *many sequence pairs* independently and concurrently (Georganas, et al., 2015; Liu and Schmidt, 2015). GSWABE (Liu and Schmidt, 2015) exploits GPUs (Tesla K40) for a highly-parallel computation of global alignment with a user-defined scoring function. CUDASW++ 3.0 (Liu, et al., 2013) exploits the SIMD capability of both CPUs and GPUs (GTX690) to accelerate the computation of the Smith-Waterman algorithm with a user-defined scoring function. CUDASW++ 3.0 provides only the optimal score, not the optimal alignment (i.e., no backtracking step). Other designs, for instance FPGASW (Fei, et al., 2018), exploit the very large number of hardware execution units in FPGAs (Xilinx VC707) to form a linear systolic array (Kung, 1982). Each execution unit in the systolic array is responsible for computing the value of a single entry of the dynamic programming matrix. The systolic array

computes a single vector of the matrix at a time. The data dependencies between the entries restrict the systolic array to computing the vectors sequentially (e.g., top-to-bottom, left-to-right, or in an anti-diagonal manner). FPGA accelerators seem to yield the highest performance gain compared to the other hardware accelerators (Banerjee, et al., 2018; Chen, et al., 2016; Fei, et al., 2018; Waidyasooriya and Hariyama, 2015). However, many of these efforts either *simplify* the scoring function, or only take into account accelerating the computation of the dynamic programming matrix *without* providing the optimal alignment as in (Chen, et al., 2014; Liu, et al., 2013; Nishimura, et al., 2017). Different and more sophisticated scoring functions are typically needed to better quantify the similarity between two sequences (Henikoff and Henikoff, 1992; Wang, et al., 2011). The backtracking step required for the optimal alignment computation involves unpredictable and irregular memory access patterns, which poses a difficult challenge for efficient hardware implementation.

**Pre-alignment filtering heuristics aim to quickly eliminate some of the dissimilar sequences before using the computationally-expensive optimal alignment algorithms.** There are a few existing filtering techniques such as the Adjacency Filter (Xin, et al., 2013), which is implemented for standard CPUs as part of FastHASH (Xin, et al., 2013). SHD (Xin, et al., 2015) is a SIMD-friendly bit-vector filter that provides higher filtering accuracy compared to the Adjacency Filter. GRIM-Filter (Kim, et al., 2018) exploits the high memory bandwidth and the logic layer of 3D-stacked memory to perform highly-parallel filtering in the DRAM chip itself. GateKeeper (Alser, et al., 2017) is designed to utilize the large amounts of parallelism offered by FPGA architectures. MAGNET (Alser, et al., July 2017) is the most accurate filtering algorithm but its current implementation is much slower than that of SHD or GateKeeper. GateKeeper (Alser, et al., 2017) is thus the best-performing pre-alignment filter in terms of accuracy and execution time.

**Our goal** in this work is to significantly reduce the time spent on calculating the *optimal alignment* of short sequences. To this end, we introduce SLIDER, a new, fast, and very accurate pre-alignment filter. SLIDER is based on two key ideas: (1) A new filtering algorithm that remarkably accelerates the computation of the banded optimal alignment by *rapidly excluding dissimilar sequences from the optimal alignment calculation*. (2) Judicious use of the parallelism-friendly architecture of modern FPGAs to greatly speed up this new filtering algorithm.

The contributions of this paper are as follows:

- We introduce SLIDER, a highly-parallel and highly-accurate pre-alignment filter, which uses a *sliding search window approach* to quickly identify dissimilar sequences *without* the need for computationally-expensive alignment algorithms. We overcome the implementation limitations of the MAGNET (Alser, et al., July 2017). We build two hardware accelerator designs that adopt modern FPGA architectures to boost the performance of both SLIDER and MAGNET.
- We provide a comprehensive analysis of the run time and space complexity of SLIDER and MAGNET algorithms. SLIDER and MAGNET are asymptotically *inexpensive* and run in linear time with respect to the sequence length and the edit distance threshold.
- We demonstrate that SLIDER and MAGNET significantly improve the accuracy of pre-alignment filtering by up to two and four orders of magnitude, respectively, compared to GateKeeper and SHD.
- We demonstrate that our FPGA implementations of MAGNET and SLIDER are two to three orders of magnitude faster than their CPU implementations. We demonstrate that integrating SLIDER with five state-of-the-art aligners reduces execution time of the sequence aligner by up to 18.8x.

## 2 METHODS

### 2.1 Overview

Our primary purpose is to reject the dissimilar sequences accurately and quickly such that we reduce the need for the computationally-expensive alignment step. We propose the SLIDER algorithm to achieve highly-accurate filtering. We then accelerate SLIDER by taking advantage of the capabilities and parallelism of FPGAs to achieve fast filtering operations. The key filtering strategy of SLIDER is inspired by the *pigeonhole principle*, which states that if  $E$  items are distributed into  $E+1$  boxes, then one or more boxes would be empty. In the context of pre-alignment filtering, this principle provides the following key observation: If two sequences differ by  $E$  edits, then the two sequences should share *at least* a single common subsequence (i.e., free of edits) and *at most*  $E+1$  non-overlapping common subsequences, where  $E$  is the edit distance threshold. With the existence of at most  $E$  edits, the total length of these non-overlapping common subsequences should *not* be less than  $m-E$ , where  $m$  is the sequence length. SLIDER employs the pigeonhole principle to decide whether or not two sequences are potentially similar. SLIDER finds all the non-overlapping subsequences that exist in both sequences. If the total length of these common subsequences is less than  $m-E$ , then there exist more edits than the allowed edit distance threshold and hence SLIDER filters out the two given sequences. Otherwise, SLIDER preserves the two sequences. Next, we discuss the details of SLIDER.

### 2.2 SLIDER Pre-alignment Filter

SLIDER identifies the dissimilar sequences, without calculating the optimal alignment, in three main steps. (1) The first step is to construct what we call a *neighborhood map* that visualizes the pairwise matches and mismatches between two sequences given an edit distance threshold of  $E$  characters. (2) The second step is to find all the non-overlapping common subsequences in the neighborhood map using a sliding search window approach. (3) The last step is to accept or reject the given sequence pairs based on the length of the found matches. If the length of the found matches is small, then SLIDER rejects the input sequence pair.

#### 2.2.1 Building the Neighborhood Map

The neighborhood map,  $N$ , is a binary  $m$  by  $m$  matrix, where  $m$  is the sequence length. Given a text sequence  $T[1...m]$ , a pattern sequence  $P[1...m]$ , and an edit distance threshold  $E$ , the neighborhood map represents the comparison result of the  $i^{th}$  character of  $P$  with the  $j^{th}$  character of  $T$ , where  $i$  and  $j$  satisfy  $1 \leq i \leq m$  and  $i-E \leq j \leq i+E$ . The entry  $N[i, j]$  of the neighborhood map can be calculated as follows:

$$N[i, j] = \begin{cases} 0, & \text{if } P[i] = T[j] \\ 1, & \text{if } P[i] \neq T[j] \end{cases} \quad (1)$$

We present in Fig. 1 an example of a neighborhood map for two sequences, where a pattern  $P$  differs from a text  $T$  by three edits. The entry  $N[i, j]$  is set to zero if the  $i^{th}$  character of the pattern matches the  $j^{th}$  character of the text. Otherwise, it is set to one. The way we build our neighborhood map ensures that computing each of its entries is independent of every other and thus the entire map can be computed all at once in a parallel fashion. Hence, our neighborhood map is well suited for highly-parallel computing platforms (Alser, et al., 2017; Seshadri, et al., 2017). Notice that in sequence alignment algorithms, computing each entry of the dynamic programming matrix depends on the values of the immediate left, upper left, and upper entries of its own.

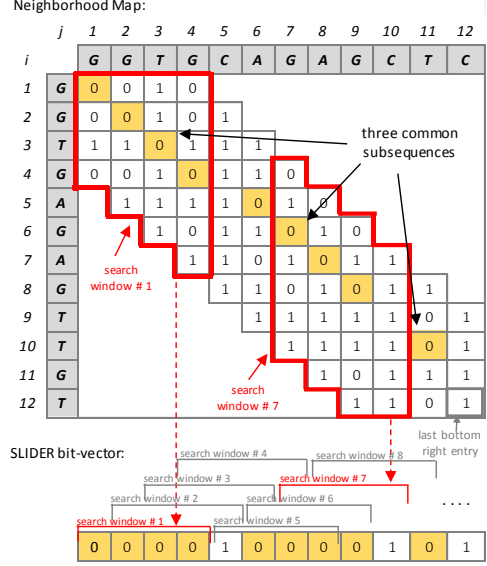


Fig. 1: Neighborhood map ( $N$ ) and the SLIDER bit-vector, for text  $T = \text{GGTGAGAGCTC}$ , and pattern  $P = \text{GGTGAGAGTGT}$  for  $E=3$ . The three common subsequences (i.e., GGTG, AGAG, and T) are highlighted in yellow. We use a search window of size 4 columns (two examples of which are highlighted in red) with a step size of a single column. SLIDER searches diagonally within each search window for the 4-bit vector that has the largest number of zeros. Once found, SLIDER examines if the found 4-bit vector maximizes the number of zeros at the corresponding location of the 4-bit vector in the SLIDER bit-vector. If so then SLIDER stores this 4-bit vector in the SLIDER bit-vector at its corresponding location.

Different from "dot plot" or "dot matrix" (visual representation of the similarities between two closely similar genomic sequences) that is used in FASTA/FASTP (Lipman and Pearson, 1985), our neighborhood map computes *only* some necessary diagonals near the main diagonal of the matrix (e.g., seven diagonals shown in Fig. 1).

#### 2.2.2 Identifying the Diagonally-Consecutive Matches

The key goal of this step is to find accurately all the non-overlapping common subsequences shared between a pair of sequences. The accuracy of finding these subsequences is crucial for the overall filtering accuracy, as the filtering decision is made solely based on their total length. With the existence of  $E$  edits, there are *at most*  $E+1$  non-overlapping common subsequences (based on the pigeonhole principle) shared between a pair of sequences. Each non-overlapping common subsequence is represented as a streak of diagonally-consecutive zeros in the neighborhood map (as highlighted in yellow in Fig. 1). These streaks of diagonally-consecutive zeros are distributed along the diagonals of the neighborhood map without any prior information about their length or number. One way of finding these common subsequences is to use a brute-force approach, which examines all the streaks of diagonally-consecutive zeros that start at the first column and selects the streak that has the largest number of zeros as the first common subsequences. It then iterates over the remaining part of the neighborhood map to find the other common subsequences. However, this brute-force approach is infeasible for highly-optimized hardware implementation as the search space is unknown at design time.

SLIDER overcomes this issue by dividing the neighborhood map into equal-size parts. We call each part a *search window*. Limiting the size of the search space from the entire neighborhood map to a search window has three key benefits: (1) It helps to provide a scalable architecture that can be implemented for any sequence length and edit distance threshold. (2) Downsizing the search space into a reasonably small sub-matrix with a known dimension at design time limits the number of all possible permutations of each bit-vector to  $2^n$ , where  $n$  is the search window size. This reduces the size of the look-up tables (LUTs) required to design the hardware architecture and simplifies the overall design. (3) Each search window is considered as a smaller sub-problem that can be solved independently and rapidly with high parallelism. SLIDER uses a search window of 4 columns wide as we illustrate in Fig. 1. We need  $m$  search windows for processing two sequences, each of which is of length  $m$  characters. Each search window overlaps with its next neighboring search window by 3 columns. This ensures covering the entire neighborhood map and finding all the common subsequences regardless of their starting location. We select the width of each search window to be 4 columns to guarantee finding the shortest possible common subsequence, which is a single match located between two mismatches (i.e., ‘101’). However, we observe that the bit pattern ‘101’ is *not* always necessarily a part of the correct alignment (or the common subsequences). For example, the bit pattern ‘101’ exists once as a part of the correct alignment in Fig.1, but it also appears six times in other different locations that are *not* included in the correct alignment. To avoid confusing SLIDER and improve the accuracy of finding the diagonally-consecutive matches, we increase the length of the diagonal vector to be examined to four bits. We also experimentally evaluate different search window sizes in Supplementary Materials, Section 6.1. We find that a search window size of 4 columns provides the highest filtering accuracy without falsely-rejecting similar sequences.

SLIDER finds the diagonally-consecutive matches that are part of the common subsequences in the neighborhood map in two main steps. **Step 1:** For each search window, SLIDER finds a 4-bit diagonal vector that has the largest number of zeros. SLIDER greedily considers this vector as a part of the common subsequence as it has the least possible number of edits (i.e., 1’s). Finding always the maximum number of matches is necessary to avoid overestimating the actual number of edits and eventually preserving all similar sequences. SLIDER achieves this step by comparing the 4 bits of each of the  $2E+1$  diagonal vectors within a search window and selects the 4-bit vector that has the largest number of zeros. In case of two 4-bit subsequences have the same number of zeros, SLIDER breaks the ties by selecting the one that has a leading zero. SLIDER then slides the search window by a single column (i.e., step size = 1 column) towards the last bottom right entry of the neighborhood map and repeats the previous computations. Thus, SLIDER performs “Step 1”  $m$  times using  $m$  search windows, where  $m$  is the sequence length. **Step 2:** The last step is to gather the results found for each search window (i.e., 4-bit vector that has the largest number of zeros) and construct back all the diagonally-consecutive matches. For this purpose, SLIDER maintains a *SLIDER bit-vector* of length  $m$  that stores all the zeros found in the neighborhood map as we illustrate in Fig. 1. For each sliding search window, SLIDER examines if the selected 4-bit vector maximizes the number of zeros in the SLIDER bit-vector at the same corresponding location. If so, SLIDER stores the selected 4-bit vector in the SLIDER bit-vector at the same corresponding location. This is necessary to avoid overestimating the number of edits between two given sequences. The common subsequences are represented as streaks of consecutive zeros in the SLIDER bit-vector.

### 2.2.3 Filtering out Dissimilar Sequences

The last step of SLIDER is to calculate the total length of the common subsequences. SLIDER examines if the total number of zeros in the SLIDER bit-vector is less than  $m-E$ . If so, SLIDER excludes the two sequences from the optimal alignment calculation. Otherwise, SLIDER considers the two sequences similar within the allowed edit distance threshold and allows their optimal alignment to be computed using optimal alignment algorithms. We provide the pseudocode of SLIDER and discuss its computational complexity in Supplementary Materials, Section 6.2. We also present two examples of applying SLIDER filtering algorithm in Supplementary Materials, Section 8.

## 2.3 Accelerator Architecture

Our second aim is to substantially accelerate SLIDER, by leveraging the capabilities and parallelism of FPGAs. In this section, we present our hardware accelerator that is designed to exploit the large amounts of parallelism offered by modern FPGA architectures (Aluru and Jammula, 2014; Herbordt, et al., 2007; Trimberger, 2015). We then outline the implementation of SLIDER to be used in our accelerator design. Fig. 2 shows the hardware architecture of the accelerator. It contains a user-configurable number of filtering units. Each filtering unit provides pre-alignment filtering independently from other units. The workflow of the accelerator starts with transmitting the sequence pair to the FPGA through the fastest communication medium available on the FPGA board (i.e., PCIe). The sequence controller manages and provides the necessary input signals for each filtering unit in the accelerator. Each filtering unit requires two sequences of the same length and an edit distance threshold. The result controller gathers the output result (i.e., a single bit of value ‘1’ for similar sequences and ‘0’ for dissimilar sequences) of each filtering unit and transmits them back to the host side in the same order as their sequences are transmitted to the FPGAs.

The host-FPGA communication is achieved using RIFFA 2.2 (Jacobsen, et al., 2015). To make the best use of the available resources in the FPGA chip, our algorithm utilizes the operations that are easily supported on an FPGA, such as bitwise operations, bit shifts, and bit count. To build the neighborhood map on the FPGA, we use the observation that the main diagonal can be implemented using a bitwise XOR operation between the two given sequences. The upper  $E$  diagonals can be implemented by gradually shifting the pattern ( $P$ ) to the right-hand direction and then performing bitwise XOR with the text ( $T$ ). This allows each character of  $P$  to be compared with the right-hand neighbor characters (up to  $E$  characters) of its corresponding character of  $T$ . The lower  $E$  diagonals can be implemented in a way similar to the upper  $E$  diagonals, but here the shift operation is performed to the left-hand direction. This ensures that each character of  $P$  is compared with the left-hand neighbor characters (up to  $E$  characters) of its corresponding character of  $T$ .

We also build an efficient hardware architecture for each search window of SLIDER algorithm. It quickly finds the number of zeros in each 4-bit vector using a hardware look-up table that stores the 16 possible permutations of a 4-bit vector along with the number of zeros for each permutation. We present the block diagram of the search window architecture in Supplementary Materials, Section 6.3. Our hardware implementation of the SLIDER filtering unit is independent from specific FPGA-platform as it does not rely on any vendor-specific computing elements (e.g., intellectual property cores). However, each FPGA board has different features and hardware capabilities that can directly or indirectly affect the performance and the data throughput of the design. In fact, the number of filtering units is determined by the maximum data throughput from main memory and the available FPGA resources.

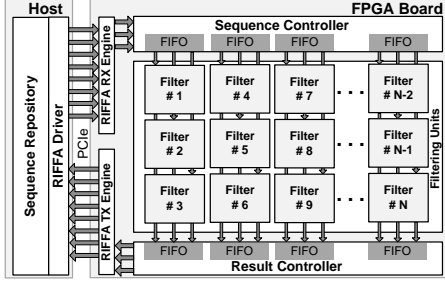


Fig. 2: Overview of our hardware accelerator architecture. The filtering units can be replicated as many times as possible according to the resources available on the FPGA fabric.

### 3 RESULTS

In this section, we evaluate (1) the filtering accuracy, (2) the FPGA resource utilization, (3) the execution time of SLIDER, our hardware implementation of MAGNET (Alser, et al., July 2017), GateKeeper (Alser, et al., 2017), and SHD (Xin, et al., 2015), and (4) the benefits of the pre-alignment filters together with state-of-the-art aligners. As we mention in Section 1, MAGNET is a highly-accurate filtering algorithm but suffers from poor performance. We comprehensively explore this algorithm and provide an efficient and fast hardware implementation of MAGNET in Supplementary Materials, Section 7. We run all experiments using a 3.6 GHz Intel i7-3820 CPU with 8 GB RAM. We use a Xilinx Virtex 7 VC709 board (Xilinx, 2014) to implement our accelerator architecture (for both SLIDER and MAGNET). We build the FPGA design using Vivado 2015.4 in synthesizable Verilog.

#### 3.1 Dataset Description

Our experimental evaluation uses 12 different real datasets. Each dataset contains 30 million real sequence pairs. We obtain three different read sets (ERR240727\_1, SRR826460\_1, and SRR826471\_1) of the whole human genome that include three different read lengths (100 bp, 150 bp, and 250 bp). We download these three read sets from EMBL-ENA (www.ebi.ac.uk/ena). We map each read set to the human reference genome (GRCh37) using the mrFAST (Alkan, et al., 2009) mapper. We obtain the human reference genome from the 1000 Genomes Project (Consortium, 2012). For each read set, we use four different maximum numbers of allowed edits using the *-e* parameter of mrFAST to generate four real datasets. This enables us to measure the effectiveness of the filters over a wide range edit distance thresholds. We summarize the details of these 12 datasets in Supplementary Materials, Section 9. For the reader’s convenience, when referring to these datasets, we number them from 1 to 12 (e.g., set\_1 to set\_12). We use Edlib (Šošić and Šikić, 2017) to generate the ground truth edit distance value for each sequence pair.

#### 3.2 Filtering Accuracy

We first assess the *false accept rate* of SLIDER, MAGNET (Alser, et al., July 2017), SHD (Xin, et al., 2015), and GateKeeper (Alser, et al., 2017) across different edit distance thresholds and datasets. The false accept rate (or false positive rate) is the ratio of the number of dissimilar sequences that are falsely-accepted by the filter and the number of dissimilar sequences that are rejected by optimal sequence alignment algorithm. We

aim to minimize the false accept rate to maximize that number of dissimilar sequences that are eliminated. In Fig.3, we provide the false accept rate of the four filters across our 12 datasets and edit distance thresholds of 0% to 10% of the sequence length.

Based on Fig. 3, we make four key observations. (1) We observe that SLIDER, MAGNET, SHD, and GateKeeper are less accurate in examining the low-edit sequences (i.e., datasets 1, 2, 5, 6, 9, and 10) than the high-edit sequences (i.e., datasets 3, 4, 7, 8, 11, and 12). (2) SHD (Xin, et al., 2015) and GateKeeper (Alser, et al., 2017) become ineffective for edit distance thresholds of greater than 8% ( $E=8$ ), 5% ( $E=7$ ), and 3% ( $E=7$ ) for sequence lengths of 100, 150, and 250 characters, respectively. This causes them to examine each sequence pair unnecessarily twice (i.e., once by GateKeeper or SHD and once by the alignment algorithm). (3) For high-edit datasets, SLIDER provides up to 17.2x, 73x, and 467x (2.4x, 2.7x, and 38x for low-edit datasets) less false accept rate compared to GateKeeper and SHD for sequence lengths of 100, 150, and 250 characters, respectively. (4) MAGNET shows up to 1577x, 3550x, and 25552x for high-edit datasets (3.5x, 14.7x, and 135x for low-edit datasets) less false accept rate compared to GateKeeper and SHD for sequence lengths of 100, 150, and 250 characters, respectively. MAGNET also shows up to 205x, 951x, and 16760x for high-edit datasets (2.7x, 10x, and 88x for low-edit datasets) less false accept rate over SLIDER for sequence lengths of 100, 150, and 250 characters, respectively.

Second, we assess the false reject rate in Supplementary Materials, Section 10. We demonstrate that SLIDER and GateKeeper have a 0% false reject rate. We also observe that SHD and MAGNET falsely-reject correct sequence pairs, which is *unacceptable* for a reliable filter. We conclude that SLIDER and MAGNET are very effective and superior to state-of-the-art pre-alignment filters, SHD (Xin, et al., 2015) and GateKeeper (Alser, et al., 2017) for both low-edit and high-edit datasets. SLIDER and MAGNET maintain a very low rate of falsely-accepted dissimilar sequences and they significantly improve the accuracy of pre-alignment filtering by up to two and four orders of magnitude compared to GateKeeper/SHD, respectively. Unlike MAGNET and SHD, SLIDER preserves all sequence pairs that differ by less than or equal to the user-defined edit distance threshold. Hence, we conclude that SLIDER is the most effective filter, with a low false accept rate and a zero false reject rate.

#### 3.3 Data Throughput and Resource Analysis

The operating frequency of our FPGA accelerator is 250 MHz. At this frequency, we observe a data throughput of nearly 3.3 GB/s, which corresponds to ~13.3 billion bases per second. This nearly reaches the peak throughput of 3.64 GB/s provided by the RIFFA (Jacobsen, et al., 2015) communication channel that feeds data into the FPGA using Gen3 4-lane PCIe. We examine the FPGA resource utilization of SLIDER, MAGNET, and GateKeeper (Alser, et al., 2017) filters. SHD (Xin, et al., 2015) is implemented in C with Intel SSE instructions and *cannot* be directly implemented on an FPGA. We provide several hardware designs for two commonly used edit distance thresholds, 2% and 5% of the sequence length, as reported in (Ahmadi, et al., 2012; Alser, et al., 2017; Hatem, et al., 2013; Xin, et al., 2015). The VC709 FPGA chip contains 433,200 slice LUTs (look-up tables) and 866,400 slice registers (flip-flops). Table 1 lists the FPGA resource utilization for a single filtering unit. We make three main observations. (1) The design for a single MAGNET filtering unit requires about 10.5% and 37.8% of the available LUTs for edit distance thresholds of 2 and 5, respectively.

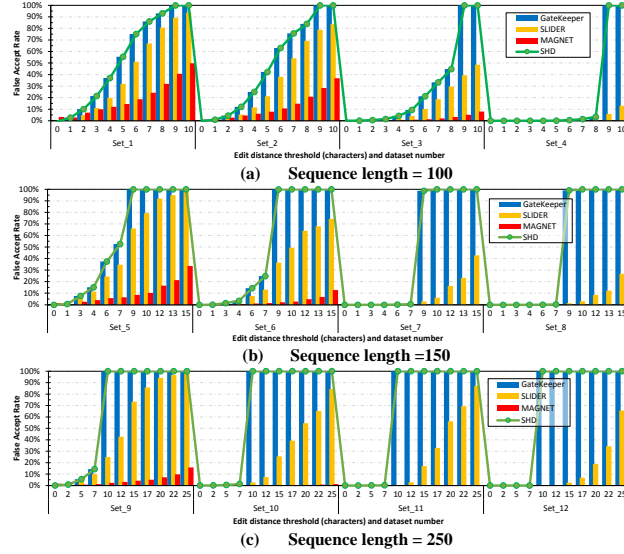


Fig. 3: The false accept rate of SLIDER, MAGNET, SHD and GateKeeper across 12 real datasets. We use a wide range of edit distance thresholds (0%-10% of the sequence length) for sequence lengths of (a) 100, (b) 150, and (c) 250.

Hence, MAGNET can process 8 and 2 sequence pairs concurrently for edit distance thresholds of 2 and 5, respectively, without violating the timing constraints of our accelerator. (2) The design for a single SLIDER filtering unit requires about 15x-21.9x less LUTs compared to MAGNET. This enables SLIDER to achieve more parallelism over MAGNET design as it can have 16 filtering units within the same FPGA chip. (3) GateKeeper requires about 26.9x-53x and 1.7x-2.4x less LUTs compared to MAGNET and SLIDER, respectively. GateKeeper can also examine 16 sequence pairs at the same time.

We conclude that the FPGA resource usage is correlated with the filtering accuracy. For example, the least accurate filter, GateKeeper, occupies the least FPGA resources. We also conclude that the less the logic utilization of a single filtering unit, the more the number of filtering units that can be integrated into the FPGA.

Table 1: FPGA resource usage for a single filtering unit of SLIDER, MAGNET, and GateKeeper, for a sequence length of 100 and under different edit distance thresholds.

Filter	$E$	Single Filtering Unit		Max. No. of Filtering Units
		Slice LUT	Slice Register	
SLIDER	2	0.69%	0.01%	16
	5	1.72%	0.01%	16
MAGNET	2	10.50%	0.8%	8
	5	37.80%	2.30%	2
GateKeeper	2	0.39%	0.01%	16
	5	0.71%	0.01%	16

### 3.4 Effects of Pre-Alignment Filtering on Sequence Alignment

We analyze the execution time of MAGNET and SLIDER compared to SHD (Xin, et al., 2015) and GateKeeper (Alser, et al., 2017). We evaluate GateKeeper, MAGNET, and SLIDER using a single FPGA chip and run SHD using a single CPU core. SHD supports a sequence length of up to

only 128 characters (due to the SIMD register size). To ensure as fair a comparison as possible, we allow SHD to divide the long sequences into batches of 128 characters, examine each batch individually, and then sum up the results. In Table 2, we provide the execution time of the four pre-alignment filters using 120 million sequence pairs under sequence lengths of 100 and 250 characters.

Table 2: Execution time (in seconds) of FPGA-based GateKeeper, MAGNET, SLIDER, and CPU-based SHD under different edit distance thresholds and sequence lengths. We use set\_1 to set\_4 for a sequence length of 100 and set\_9 to set\_12 for a sequence length of 250. We provide the performance results for both a single filtering unit and the maximum number of filtering units (in parentheses).

$E$	GateKeeper	MAGNET	SLIDER	SHD
<i>Sequence Length = 100</i>				
2	2.89 <sup>a</sup> (0.18 <sup>b</sup> , 16 <sup>c</sup> )	2.89 (0.36, 8)	2.89 (0.18, 16)	60.33
5	2.89 (0.18, 16)	2.89 (1.45, 2)	2.89 (0.18, 16)	67.92
<i>Sequence Length = 250</i>				
5	5.78 (0.72, 8)	5.78 (2.89 <sup>d</sup> , 2)	5.78 (0.72 <sup>d</sup> , 8)	141.09
15	5.78 (0.72, 8)	5.78 (5.78 <sup>d</sup> , 1)	5.78 (0.72 <sup>d</sup> , 8)	163.82

<sup>a</sup> Execution time, in seconds, for a single filtering unit.

<sup>b</sup> Execution time, in seconds, for maximum filtering units.

<sup>c</sup> The number of filtering units.

<sup>d</sup> Theoretical results based on the resource utilization and data throughput.

We make four key observations. (1) SLIDER’s execution time is as low as that of GateKeeper (Alser, et al., 2017), and 2x-8x lower than that of MAGNET. This observation is in accord with our expectation and can be explained by the fact that MAGNET has more resource overhead that limits the number of filtering units on an FPGA. Yet SLIDER is up to two orders of magnitude more accurate than GateKeeper (as we show earlier in Section 3.2). (2) SLIDER is up to 28x and 335x faster than SHD using one and 16 filtering units, respectively. (3) MAGNET is up to 28x and 167.5x faster than SHD using one and 8 filtering units, respectively. As we present in Supplementary Materials, Section 11, the hardware-accelerated versions of SLIDER and MAGNET provide up to three orders of magnitude of speedup over their functionally-equivalent CPU implementations.

We conclude that SLIDER is extremely fast and accurate. SLIDER’s performance also scales very well over a wide range of both edit distance thresholds and sequence lengths.

### 3.5 Effects of Pre-Alignment Filtering on Sequence Alignment

We analyze the benefits of integrating our proposed pre-alignment filter (and other filters) with state-of-the-art aligners. Table 3 presents the effect of different pre-alignment filters on overall alignment time. We select five best-performing aligners, each of which is designed for a different type of computing platform. We use a total of 120 million real sequence pairs from our previously-described four datasets (set\_1 to set\_4) in this analysis. We evaluate the actual execution time of Edlib (Šošić and Šikić, 2017) and Parasail (Daily, 2016) on our machine. However, FPGASW (Fei, et al., 2018), CUDASW++ 3.0 (Liu, et al., 2013), and GSWABE (Liu and Schmidt, 2015) are *not* open-source and not available to us. Therefore, we scale the reported number of computed entries of the dynamic programming matrix in a second (i.e., GCUPS) as follows: 120,000,000 / (GCUPS / 100<sup>2</sup>).

We make three key observations. (1) The execution time of Edlib (Šošić and Šikić, 2017) reduces by up to 18.8x, 16.5x, 13.9x, and 5.2x after the addition of SLIDER, MAGNET, GateKeeper, and SHD, respectively, as a pre-alignment filtering step. We also observe a very similar trend for Parasail (Daily, 2016) combined with each of the four pre-alignment filters. (2) Aligners designed for FPGAs and GPUs follow a different trend than that we observe in the CPU aligners. We observe that FPGASW (Fei, et al., 2018), CUDASW++ 3.0 (Liu, et al., 2013), and GSWABE (Liu and Schmidt, 2015) are *faster* alone than with SHD (Xin, et al., 2015) incorporated as the pre-alignment filtering step. SLIDER, MAGNET, and GateKeeper (Alser, et al., 2017) still significantly reduce the overall execution time of both FPGA and GPU based aligners. SLIDER reduces the overall alignment time of FPGASW (Fei, et al., 2018), CUDASW++ 3.0 (Liu, et al., 2013), and GSWABE (Liu and Schmidt, 2015) by factors of up to 14.5x, 14.2x, and 17.9x, respectively. This is up to 1.35x, 1.4x, and 85x more than the effect of MAGNET, GateKeeper, and SHD on the end-to-end alignment time. (3) We observe that if the execution time of the aligner is much larger than that of the pre-alignment filter (which is the case for Edlib and Parasail for  $E=5$  characters), then MAGNET provides up to 1.3x more end-to-end speedup over SLIDER. This is expected as MAGNET produces a smaller false accept rate compared to SLIDER. However, unlike MAGNET, SLIDER provides a 0% false reject rate.

We conclude that among the four pre-alignment filters, SLIDER is the best-performing pre-alignment filter in terms of both speed and accuracy. Integrating SLIDER with an aligner leads to strongly positive benefits and reduces the aligner’s total execution time by up to 18.8x.

**Table 3: End-to-end execution time (in seconds) for several state-of-the-art sequence alignment algorithms, with and without pre-alignment filters (SLIDER, MAGNET, GateKeeper, and SHD) and across different edit distance thresholds.**

<i>E</i>	Edlib	w/ SLIDER	w/ MAGNET	w/ GateKeeper	w/ SHD
2	506.66	26.86	30.69	36.39	96.54
5	632.95	147.20	106.80	208.77	276.51
<i>E</i>	Parasail	w/ SLIDER	w/ MAGNET	w/ GateKeeper	w/ SHD
2	1310.96	69.21	78.83	93.87	154.02
5	2044.58	475.08	341.77	673.99	741.73
<i>E</i>	FPGASW	w/ SLIDER	w/ MAGNET	w/ GateKeeper	w/ SHD
2	11.33	0.78	1.04	0.99	61.14
5	11.33	2.81	3.34	3.91	71.65
<i>E</i>	CUDASW++ 3.0	w/ SLIDER	w/ MAGNET	w/ GateKeeper	w/ SHD
2	10.08	0.71	0.96	0.90	61.05
5	10.08	2.52	3.13	3.50	71.24
<i>E</i>	GSWABE	w/ SLIDER	w/ MAGNET	w/ GateKeeper	w/ SHD
2	61.86	3.44	4.06	4.60	64.75
5	61.86	14.55	11.75	20.57	88.31

## 4 DISCUSSION AND FUTURE WORK

We demonstrate that the concept of pre-alignment filtering provides substantial benefits to the existing and future sequence alignment algorithms. Accelerated sequence aligners are frequently introduced that offer different strengths and features. Many of these efforts either simplify the scoring function, or only take into account accelerating the computation of the dynamic programming matrix *without* supporting the backtracking step. SLIDER offers the ability to make the best use of existing aligners *without* sacrificing any of their capabilities, as it does *not* modify or replace the

alignment step. As such, we hope that it catalyzes the adoption of specialized pre-alignment accelerators in genome sequence analysis. However, the use of specialized hardware chips may discourage users who are not necessarily fluent in FPGAs. This concern can be alleviated in at least two ways. First, the SLIDER accelerator can be integrated more closely *inside* the sequencing machines to perform real-time pre-alignment filtering concurrently with sequencing (Lindner, et al., 2016). This allows a significant reduction in total genome analysis time. Second, cloud computing offers access to a large number of advanced FPGA chips that can be used concurrently via a simple user-friendly interface. However, such a scenario requires the development of privacy-preserving pre-alignment filters due to privacy and legal concerns (Salinas and Li, 2017). Our next efforts will focus on exploring privacy-preserving real-time pre-alignment filtering.

Another potential target of our research is to explore the possibility of accelerating optimal alignment calculations for longer sequences (few tens of thousands of characters) (Senol, et al., 2018) using pre-alignment filtering. Longer sequences pose two challenges. First, we need to transfer more data to the FPGA chip to be able process a single pair of sequences which is mainly limited by the data transfer rate of the communication link (i.e., PCIe). Second, typical edit distance threshold used for sequence alignment is 5% of the sequence length. For considerably long sequences, edit distance threshold is around few hundreds of characters. For a large edit distance threshold, each character of a given sequence is compared to a large number of neighboring characters of the other given sequence. This makes the short matches (e.g., a single zero or two consecutive zeros) to occur more frequently in the diagonal vectors, which negatively affect the accuracy of SLIDER. We will investigate this effect and explore new pre-alignment filtering approaches for the sequencing data produced by third-generation sequence machines.

## 5 CONCLUSION

In this work, we propose SLIDER, a highly-parallel and accurate pre-alignment filtering algorithm accelerated on a specialized hardware platform. The key idea of SLIDER is to rapidly and accurately eliminate dissimilar sequences *without* calculating banded optimal alignment. Our hardware-accelerated version of SLIDER provides, on average, three orders of magnitude speedup over its functionally-equivalent CPU implementation. SLIDER improves the accuracy of pre-alignment filtering by up to two orders of magnitude compared to the best-performing existing pre-alignment filter, GateKeeper. The addition of SLIDER as a pre-alignment step significantly reduces the alignment time of state-of-the-art aligners by up to 18.8x, leading to the fastest alignment mechanism that we know of.

## Acknowledgments

We thank Tuan Duy Anh Nguyen for his valuable comments on the hardware design.

## Funding

This work is supported in part by the NIH Grant (HG006004 to O. Mutlu and C. Alkan) and the EMBO Installation Grant (IG-2521) to C. Alkan. M. Alser is supported by the HiPEAC collaboration grant and TUBITAK-2215 graduate fellowship from the Scientific and Technological Research Council of Turkey.

*Conflict of Interest:* none declared.



## References

- Ahmadi, A., et al. (2012) Hobbes: optimized gram-based methods for efficient read alignment, *Nucleic acids research*, **40**, e41-e41.
- Al Kawam, A., Khatri, S. and Datta, A. (2017) A Survey of Software and Hardware Approaches to Performing Read Alignment in Next Generation Sequencing, *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, **14**, 1202-1213.
- Alkan, C., et al. (2009) Personalized copy number and segmental duplication maps using next-generation sequencing, *Nature genetics*, **41**, 1061-1067.
- Alser, M., et al. (2017) GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping, *Bioinformatics*, **33**, 3355-3363.
- Alser, M., Mutlu, O. and Alkan, C. (July 2017) Magnet: Understanding and improving the accuracy of genome pre-alignment filtering, *Transactions on Internet Research* **13**.
- Aluru, S. and Jammula, N. (2014) A review of hardware acceleration for computational genomics, *Design & Test, IEEE*, **31**, 19-30.
- Backurs, A. and Indyk, P. (2017) Edit Distance Cannot Be Computed in Strongly Subquadratic Time (unless SETH is false), *arXiv preprint arXiv:1412.0348v4*
- Banerjee, S.S., et al. (2018) ASAP: Accelerated Short-Read Alignment on Programmable Hardware, *arXiv preprint arXiv:1803.02657*.
- Calude, C., Salomaa, K. and Yu, S. (2002) Additive distances and quasi-distances between words, *Journal of Universal Computer Science*, **8**, 141-152.
- Chen, P., et al. (2014) Accelerating the next generation long read mapping with the FPGA-based system, *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, **11**, 840-852.
- Chen, Y.-T., et al. (2016) When spark meets FPGAs: a case study for next-generation DNA sequencing acceleration. *Field-Programmable Custom Computing Machines (FCCM)*, 2016 IEEE 24th Annual International Symposium on. IEEE, pp. 29-29.
- Consortium, G.P. (2012) An integrated map of genetic variation from 1,092 human genomes, *Nature*, **491**, 56-65.
- Daily, J. (2016) Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments, *BMC bioinformatics*, **17**, 81.
- Fei, X., et al. (2018) FPGASW: Accelerating Large-Scale Smith–Waterman Sequence Alignment Application with Backtracking on FPGA Linear Systolic Array, *Interdisciplinary Sciences: Computational Life Sciences*, **10**, 176-188.
- Fox, E.J., et al. (2014) Accuracy of next generation sequencing platforms, *Next generation, sequencing & applications*, **1**.
- Georganas, E., et al. (2015) meraligner: A fully parallel sequence aligner. *Parallel and Distributed Processing Symposium (IPDPS)*, 2015 IEEE International. IEEE, pp. 561-570.
- Hatem, A., et al. (2013) Benchmarking short sequence mapping tools, *BMC bioinformatics*, **14**, 184.
- Henikoff, S. and Henikoff, J.G. (1992) Amino acid substitution matrices from protein blocks, *Proceedings of the National Academy of Sciences*, **89**, 10915-10919.
- Herbordt, M.C., et al. (2007) Achieving high performance with FPGA-based computing, *Computer*, **40**, 50.
- Jacobsen, M., et al. (2015) RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators, *ACM Trans. Reconfigurable Technol. Syst.*, **8**, 1-23.
- Kim, J.S., et al. (2018) GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies, *BMC genomics*, **19**, 89.
- Kung, H.-T. (1982) Why systolic architectures?, *IEEE computer*, **15**, 37-46.
- Lindner, M.S., et al. (2016) HiLive–Real-Time Mapping of Illumina Reads while Sequencing, *Bioinformatics*, **32**, 659.
- Lipman, D.J. and Pearson, W.R. (1985) Rapid and sensitive protein similarity searches, *Science*, **227**, 1435-1441.
- Liu, Y. and Schmidt, B. (2015) GSWABE: faster GPU-accelerated sequence alignment with optimal alignment retrieval for short DNA sequences, *Concurrency and Computation: Practice and Experience*, **27**, 958-972.
- Liu, Y., Wirawan, A. and Schmidt, B. (2013) CUDASW++ 3.0: accelerating Smith–Waterman protein database search by coupling CPU and GPU SIMD instructions, *BMC bioinformatics*, **14**, 117.
- Masek, W.J. and Paterson, M.S. (1980) A faster algorithm computing string edit distances, *Journal of Computer and System Sciences*, **20**, 18-31.
- McKernan, K.J., et al. (2009) Sequence and structural variation in a human genome uncovered by short-read, massively parallel ligation sequencing using two-base encoding, *Genome research*, **19**, 1527-1541.
- Navarro, G. (2001) A guided tour to approximate string matching, *ACM computing surveys (CSUR)*, **33**, 31-88.
- Ng, H.-C., Liu, S. and Luk, W. (2017) Reconfigurable acceleration of genetic sequence alignment: A survey of two decades of efforts. *Field Programmable Logic and Applications (FPL)*, 2017 27th International Conference on. IEEE, pp. 1-8.
- Nishimura, T., et al. (2017) Accelerating the Smith–Waterman Algorithm Using Bitwise Parallel Bulk Computation Technique on GPU. *Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017 IEEE International. IEEE, pp. 932-941.
- Salinas, S. and Li, P. (2017) Secure Cloud Computing for Pairwise Sequence Alignment. *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*. ACM, pp. 178-183.
- Sandes, E.F.D.O., Boukerche, A. and Melo, A.C.M.A.D. (2016) Parallel optimal pairwise biological sequence comparison: Algorithms, platforms, and classification, *ACM Computing Surveys (CSUR)*, **48**, 63.
- Senol, C.D., et al. (2018) Nanopore sequencing technology and tools for genome assembly: computational analysis of the current state, bottlenecks and future directions, *Briefings in bioinformatics*.
- Seshadri, V., et al. (2017) Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, pp. 273-287.
- Šošić, M. and Šikić, M. (2017) Edlib: a C/C++ library for fast, exact sequence alignment using edit distance, *Bioinformatics*, **33**, 1394-1395.
- Trimberger, S.M. (2015) Three ages of FPGAs: a retrospective on the first thirty years of FPGA technology, *Proceedings of the IEEE*, **103**, 318-331.
- Ukkonen, E. (1985) Algorithms for approximate string matching, *Information and control*, **64**, 100-118.
- Waidyasooriya, H. and Hariyama, M. (2015) Hardware-Acceleration of Short-read Alignment Based on the Burrows–Wheeler Transform, *Parallel and Distributed Systems, IEEE Transactions on*, **PP**, 1-1.
- Wang, C., et al. (2011) Comparison of linear gap penalties and profile-based variable gap penalties in profile–profile alignments, *Computational biology and chemistry*, **35**, 308-318.
- Xilinx (2014) Virtex-7 XT VC709 Connectivity Kit, Getting Started Guide, **UG966 (v3.0.1) June 30, 2014**.
- Xin, H., et al. (2015) Shifted Hamming Distance: A Fast and Accurate SIMD-Friendly Filter to Accelerate Alignment Verification in Read Mapping, *Bioinformatics*, **31**, 1553-1560.
- Xin, H., et al. (2013) Accelerating read mapping with FastHASH, *BMC genomics*, **14**, S13.



## Supplementary Materials

### 6 SLIDER Filter

#### 6.1 Examining the effect of different window sizes on the accuracy of the SLIDER algorithm.

In Fig. 4, we experimentally evaluate the effect of different window sizes on the false accept rate of SLIDER. We observe that as we increase the window size, the rate of dissimilar sequences that are accepted by SLIDER decreases. This is because individual matches (i.e., single zeros) are usually useless and they are not necessarily part of the common subsequences. As we increase the search window size, we are ignoring these individual matches and instead we only look for longer streaks of consecutive zeros. We also observe that a window size of 4 columns provides the lowest false accept rate (i.e., the highest accuracy).

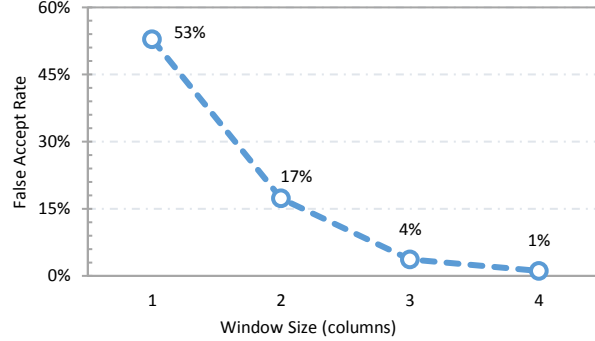


Fig. 4: The effect of the window size on the rate of the falsely accepted sequences (i.e., dissimilar sequences that are considered as similar ones by SLIDER filter). We observe that a window width of 4 columns provides the highest accuracy. We also observe that as window size increases beyond 4 columns, more similar sequences are rejected by SLIDER, which should be avoided.

#### 6.2 The SLIDER Algorithm and Its Analysis

We provide the SLIDER algorithm along with analysis of its computational complexity (asymptotic run time and space complexity). SLIDER divides the problem of finding the common subsequences into at most  $m$  subproblems, as described in Algorithm 1 (line 9). Each subproblem examines each of the  $2E+1$  bit-vectors and finds the 4-bit subsequence that has the largest number of zeros within the sliding window (line 13 to line 23). Once found, SLIDER also compares the found subsequence with its corresponding subsequence in the SLIDER bit-vector and stores the one that has more zeros in the SLIDER bit-vector (line 24). Now, let  $c$  be a constant representing the run time of examining a subsequence of 4 bits long. Then the time complexity of SLIDER algorithm is as follows:

$$T_{SLIDER}(m) = c.m.(2E+2) \quad (2)$$

This demonstrates that SLIDER algorithm runs in linear time with respect to the sequence length and edit distance threshold. SLIDER algorithm maintains  $2E+1$  diagonal bit-vectors and an additional auxiliary bit-vector (i.e., SLIDER bit-vector) for each two given sequences. The space complexity of SLIDER algorithm is as follows:

$$D_{SLIDER}(m) = m.(2E+2) \quad (3)$$

Hence, SLIDER algorithm requires linear space with respect to the sequence length and edit distance threshold. Next, we describe the hardware implementation details of SLIDER filter.

#### 6.3 Hardware Implementation

We present the FPGA chip layout for our hardware accelerator in Fig. 5. As we illustrated in the main manuscript, Section 2.3, we implement the first step of our SLIDER algorithm, building neighborhood map, using shift registers and bitwise XOR operations. The second step of SLIDER algorithm is identifying the diagonally-consecutive matches. This key step involves finding the 4-bit vector that has the largest number of zeros. For each search window, there are  $2E+1$  diagonal bit-vectors and an additional SLIDER bit-vector. To enable the computation to be performed in a parallel fashion, we build  $2E+2$  counters. As presented in Fig. 5, each counter counts the number of zeros in a single bit-vector. The counter takes four bits as input and generates three bits that represents the number of zeros within the window. Each counter requires three 4-input LUTs, as each LUT has a single output signal. In total, we need  $6E+6$  4-input LUTs to build a single search window. All bits of the counter output are generated at the same time, as the propagation delay through an FPGA look-up table is independent of the implemented function (Xilinx, November 17, 2014). The comparator is responsible for selecting the 4-bit subsequence that maximizes the number of consecutive matches based on the output of each counter and the SLIDER bit-vector. Finally, the selected 4-bit subsequence is then stored in the SLIDER bit-vector at the same corresponding location.

Algorithm 1: SLIDER	Comments
<b>Input:</b> text ( $T$ ), pattern ( $P$ ), edit distance threshold ( $E$ ). <b>Output:</b> 1 (Similar/Alignment is needed) / 0 (Dissimilar/Alignment is not needed).	
1: $m \leftarrow \text{length}(T)$ ; 2: <b>for</b> $i \leftarrow 1$ to $m$ <b>do</b> 3: <b>for</b> $j \leftarrow i-E$ to $i+E$ <b>do</b> 4: <b>if</b> $T[i] == P[j]$ <b>then</b> 5: $N[i,j] \leftarrow 0$ ; 6: <b>else</b> $N[i,j] \leftarrow 1$ ; 7: <b>for</b> $i \leftarrow 1$ to $m$ <b>do</b> $\text{SLIDER}[i] \leftarrow 1$ ; //initializing SLIDER bit-vector to 1's 8: $Z \leftarrow [0000]$ ; // $Z$ is 4-bit vector that stores the longest streak of diagonally-consecutive zeros 9: <b>for</b> $i \leftarrow 1$ to $m$ <b>do</b> // slide the search window by a single step 10: <b>for</b> $j \leftarrow 1$ to $E$ <b>do</b> // iterate over the diagonals 11:         // function $\text{CZ}(D)$ counts the occurrence of zeros in its input bit-vector $D$ 12:         // Compare $j^{\text{th}}$ lower diagonal with $j^{\text{th}}$ upper diagonal 13: <b>if</b> $\text{CZ}(N[i+j:i+3+j, i:i+3]) > \text{CZ}(N[i:i+3, i+j:i+3+j])$ <b>then</b> 14: $Z \leftarrow N[i+j:i+3+j, i:i+3]$ ; 15:         // If $j^{\text{th}}$ lower and $j^{\text{th}}$ upper diagonals have the same number of 16:         // zeros then selects the diagonal that starts with zeros 17: <b>else if</b> $\text{CZ}(N[i+j:i+3+j, i:i+3]) == \text{CZ}(N[i:i+3, i+j:i+3+j])$ <b>then</b> 18: <b>if</b> $N[i+j, i] == 0$ <b>then</b> $Z \leftarrow N[i+j:i+3+j, i:i+3]$ ; 19: <b>else if</b> $N[i, i+j] == 0$ <b>then</b> $Z \leftarrow N[i:i+3, i+j:i+3+j]$ ; 20:         // Compare $Z$ with the $j^{\text{th}}$ upper diagonal 21: <b>else</b> $Z \leftarrow N[i:i+3, i+j:i+3+j]$ ; 22:         // Compare $Z$ with main diagonal and SLIDER bit-vector 23: <b>if</b> $\text{CZ}(N[i:i+3, i:i+3]) > \text{CZ}(Z)$ <b>then</b> $Z \leftarrow N[i:i+3, i:i+3]$ ; 24: <b>if</b> $\text{CZ}(Z) > \text{CZ}(\text{SLIDER}[i:i+3])$ <b>then</b> $\text{SLIDER}[i:i+3] \leftarrow Z$ ; 25: <b>if</b> $\text{CZ}(\text{SLIDER}) \geq m-E$ <b>then return</b> 1; 26: <b>else return</b> 0;	<p><b>Step 1:</b> Building neighborhood map (<math>N</math>)</p> <p>Output: <math>2E+1</math> diagonal bit-vectors</p> <p><b>Step 2:</b> Identifying the Diagonally-Consecutive Matches</p> <p><b>Step 3:</b> Filtering out Dissimilar Sequences</p>

<b>Algorithm 2:</b> CZ function	
<b>Function:</b> CZ() counts the number of occurrences of zeros.	
<b>Input:</b> bit-vector $D$ .	
<b>Output:</b> number of occurrences of zeros.	
1: $\text{count} \leftarrow 0$ ; 2: <b>for</b> $i \leftarrow 1$ to $\text{length}(D)$ <b>do</b> 3: <b>if</b> $D[i] == 0$ <b>then</b> 4: $\text{count} \leftarrow \text{count} + 1$ ; 5: <b>return</b> $\text{count}$ ;	

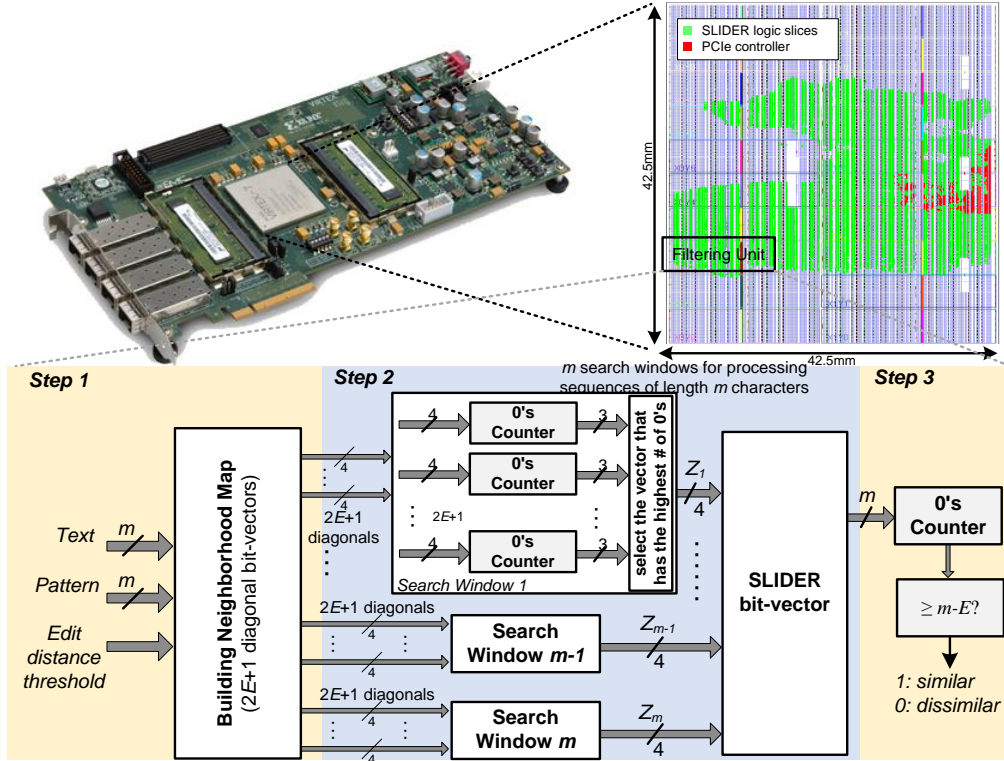


Fig. 5: FPGA chip layout for SLIDER and block diagram of the search window scheme implemented in a Xilinx VC709 FPGA for a single filtering unit.

## 7 MAGNET Filter

First, we provide the MAGNET (Alser, et al., July 2017) algorithm and describe its main filtering mechanism. Second, we analyze the computational complexity of MAGNET algorithm. Third, we provide details about the hardware implementation of the MAGNET algorithm.

### 7.1 Overview

MAGNET (Alser, et al., July 2017) is another filter that uses a divide-and-conquer technique to find all the  $E+1$  common subsequences, if any, and sum up their length. By calculating their total length, we can estimate the total number of edits between the two given sequences. If the total length of the  $E+1$  common subsequences is less than  $m-E$ , then there exist more common subsequences than  $E+1$  that are associated with more edits than allowed. If so, then MAGNET excludes the two given sequences from optimal alignment calculation. We present the algorithm of MAGNET in Algorithm 3. Finding the common subsequences involves four main steps. (1) **Building the neighborhood map**. Similar to SLIDER, MAGNET starts with building the  $2E+1$  diagonal bit-vectors of the neighborhood map for the two given sequences (Algorithm 3, lines 2-6). (2) **Extraction**. Each diagonal bit-vector nominates its local longest subsequence of consecutive zeros. Among all nominated subsequences, a single subsequence is selected as a global longest subsequence based on its length (Algorithm 4, lines 2-11). MAGNET evaluates if the length of the global longest subsequence is less than  $\lceil (m-E)/(E+1) \rceil$ , then the two sequences contain more edits than allowed, which cause the common subsequences to be shorter (i.e., each edit results in dividing the sequence pair into more common subsequences). If so, then the two sequences are rejected (Algorithm 4, lines 12-13). Otherwise, MAGNET stores its length to be used towards calculating the total length of all  $E+1$  common subsequences. The lower bound equality occurs when all edits are equispaced and all  $E+1$  subsequences are of the same length.

(3) **Encapsulation**. The next step is essential to preserve the original edit (or edits) that causes a single common sequence to be divided into smaller subsequences. MAGNET penalizes the found subsequence by two edits (one for each side). This is achieved by excluding from the search space of all bit-vectors the indices of the found subsequence in addition to the index of the surrounding single bit from both left and right sides (Algorithm 4, lines 14-17). (4) **Divide-and-Conquer Recursion**. In order to locate the other  $E$  non-overlapping subsequences, MAGNET applies a divide-and-conquer technique where we decompose the problem of finding the non-overlapping common subsequences into two subproblems. While the first subproblem focuses on finding the next long subsequence that is located on the right-hand side of the previously found subsequence in the first *extraction* step (Algorithm 4, line 15), the second subproblem focuses on the other side of the found subsequence (Algorithm 4, line 17). Each subproblem is solved by recursively repeating all the three steps mentioned above, but without evaluating again the length of the longest subsequence. MAGNET applies two early termination methods that aim at reducing the execution time of the filter. The first method is evaluating the length of the longest subsequence in the first recursion call (Algorithm 4, lines 12-13). The second method is limiting the number of the subsequences to be found to at most  $E+1$ , regardless of their

actual number for the given sequence pair (Algorithm 4, line 1). (5) **Filtering out Dissimilar Sequences.** Once after the termination, if the total length of all found common subsequences is less than  $m-E$  then the two sequences are rejected. Otherwise, they are considered to be similar and the alignment can be measured using sophisticated alignment algorithms.

Algorithm 3: MAGNET	Comments
<b>Input:</b> text ( $T$ ), pattern ( $P$ ), edit distance threshold ( $E$ ). <b>Output:</b> 1 (Similar/Alignment is needed) / 0 (Dissimilar/Alignment is not needed). 1: $m \leftarrow \text{length}(T)$ ; 2: <b>for</b> $i \leftarrow 1$ to $m$ <b>do</b> 3: <b>for</b> $j \leftarrow i-E$ to $i+E$ <b>do</b> 4: <b>if</b> $T[i] == P[j]$ <b>then</b> 5: $N[i,j] \leftarrow 0$ ; 6: <b>else</b> $N[i,j] \leftarrow 1$ ; 7: <b>for</b> $i \leftarrow 1$ to $m$ <b>do</b> 8: $\text{MAGNET}[i] \leftarrow 1$ ; // Initializing MAGNET bit-vector 9: $[\text{MAGNET}, \text{calls}] \leftarrow \text{EXEN}(N, 1, m, E, \text{MAGNET}, 1)$ ; 10: // Function CZ() returns number of zeros 11: <b>if</b> $\text{CZ}(\text{MAGNET}) \geq m-E$ <b>then return 1</b> ; <b>else return 0</b> ; 	<p><b>Step 1:</b> Building neighborhood map (<math>N</math>)</p> <p>Output: <math>2E+1</math> diagonal bit-vectors</p> <p><b>Step 2 - Step 4</b></p> <p><b>Step 5:</b> Filtering out Dissimilar Sequences</p>
<b>Algorithm 4:</b> EXEN function	Comments
<b>Function:</b> EXEN() extracts the longest subsequence of consecutive zeros and generate two subproblems. <b>Input:</b> Neighborhood map ( $N$ ), start index ( $SI$ ), end index ( $E$ ), $E$ , MAGNET bit-vector, number of recursion calls. <b>Output:</b> updated MAGNET bit-vector, updated number of calls. 1: <b>if</b> ( $SI \leq EI$ and $\text{calls} \leq E+1$ ) <b>then</b> // Early termination condition 2:     // Function CCZ() returns number and indices of longest 3:     // subsequence of diagonally consecutive zeros 4: <b>for</b> $j \leftarrow 1$ to $E$ <b>do</b> //Extraction 5: $[X, s1, e1] \leftarrow \text{CCZ}(N[SI+j, SI], EI)$ ; // Lower diagonal 6: $[Y, s2, e2] \leftarrow \text{CCZ}(N[SI, SI+j], EI)$ ; // Upper diagonal 7: <b>if</b> $X > Y$ <b>then</b> $s \leftarrow s1$ ; $e \leftarrow e1$ ; 8: <b>else</b> $s \leftarrow s2$ ; $e \leftarrow e2$ ; 9: $[X, s1, e1] \leftarrow \text{CCZ}(N[SI, SI], EI)$ ; 10: <b>if</b> $X > (e-s+1)$ <b>then</b> 11: $s \leftarrow s1$ ; $e \leftarrow e1$ ; 12: <b>if</b> ( $\text{calls}=1$ and $(e-s+1) < [(m-E)/(E+1)]$ ) <b>then</b> 13: <b>return</b> $[\text{MAGNET}, 0]$ ; 14:     // Right subproblem with encapsulation 15: $[\text{MAGNET}, \text{calls}] \leftarrow \text{EXEN}(N, e+2, EI, E, \text{MAGNET}, \text{calls}+1)$ ; 16:     // Left subproblem with encapsulation 17: $[\text{MAGNET}, \text{calls}] \leftarrow \text{EXEN}(N, SI, s-2, E, \text{MAGNET}, \text{calls}+1)$ ; 18: <b>return</b> $[\text{MAGNET}, \text{calls}]$ ; 19: <b>else return</b> $[\text{MAGNET}, \text{calls}-1]$ ; 	<p><b>Step 2:</b> Extracting the longest subsequence of consecutive zeros</p> <p>Early termination condition (only in first call)</p> <p><b>Step 3:</b> Encapsulating the found longest subsequence and <b>Step 4:</b> Divide-and-Conquer Recursion</p>

## 7.2 Analysis of MAGNET Algorithm

We analyze the asymptotic run time and space complexity of the MAGNET algorithm. MAGNET applies a divide-and-conquer technique that divides the problem of finding the common subsequences into two subproblems in each recursion call. In the first recursion call, the extracted common subsequence is of length at least  $a = \lceil (m - E)/(E + 1) \rceil$  bases. This reduces the problem of finding the common subsequences from  $m$  to at most  $m - a$ , which is further divided into two subproblems: a left subproblem and a right subproblem. For the sake of simplicity, we assume that the size of the left and the right subproblems decreases by a factor of  $b$  and  $c$ , respectively, as follows:

$$m = a + 2 + m/b + m/c \quad (4)$$

The addition of 2 bases is for the encapsulation bits added at each recursion call. Now, let  $T_{MAGNET}(m)$  be the time complexity of MAGNET algorithm, for identifying non-overlapping subsequences. If it takes  $O(km)$  time to find the global longest subsequence and divide the problem into two subproblems, where  $k = 2E+I$  is the number of bit-vectors, we get the following recurrence equation:

$$T_{\text{MAGNET}}(m) = T_{\text{MAGNET}}(m/b) + T_{\text{MAGNET}}(m/c) + O(km) \quad (5)$$

Given that the early termination condition of MAGNET algorithm restricts the recursion depth as follows:

$$\text{Recursion tree depth} = \lceil \log_2(E + 1) \rceil - 1 \quad (6)$$

Solving the recurrence in (5) using (4) and (6) by applying the recursion-tree method provides a loose upper-bound to the time complexity as follows:

$$T_{\text{MAGNET}}(m) = O(km) \cdot \sum_{x=0}^{\lceil \log_2(E+1) \rceil - 1} \left( \frac{1}{b} + \frac{1}{c} \right)^x \approx O(fkm) \quad (7)$$

Where  $f$  is a fraction number satisfies the following range:  $1 \leq f < 2$ . This in turn demonstrates that the MAGNET algorithm runs in linear time with respect to the sequence length and edit distance threshold and hence it is computationally inexpensive. The space complexity of MAGNET algorithm is as follows:

$$\begin{aligned} D_{\text{MAGNET}}(m) &= D_{\text{MAGNET}}(m/b) + D_{\text{MAGNET}}(m/c) + (km+m) \\ &\approx O(fkm + fm) \end{aligned} \quad (8)$$

Hence, MAGNET algorithm requires linear space with respect to the read length and edit distance threshold. Next, we describe the hardware implementation details of MAGNET filter.

### 7.3 Hardware Implementation

We outline the challenges that are encountered in implementing MAGNET filter to be used in our accelerator design. Implementing the MAGNET algorithm on an FPGA is more challenging than implementing the SLIDER algorithm due to the random location and variable length of each of the  $E+1$  common subsequences. Verilog-2011 imposes two challenges on our architecture as it does not support variable-size partial selection and indexing of a group of bits from a vector (McNamara, 2001). In particular, the first challenge lies in excluding the extracted common subsequence along with its encapsulation bits from the search space of the next recursion call. The second challenge lies in dividing the problem into two subproblems, each of which has an unknown size at design time. To address these limitations and tackle the two design challenges, we keep the problem size fixed and at each recursion call. We exclude the longest found subsequence from the search space by amending all bits of all  $2E+1$  bit-vectors that are located within the indices (locations) of the encapsulation bits to '1's. This ensures that we exclude the longest found subsequence and its corresponding location in all other bit-vectors during the subsequent recursion calls. We build the MAGNET accelerator using the same FPGA board as that used for SLIDER for a fair comparison.

## 8 Examples of Applying SLIDER and MAGNET algorithms

In this section, we provide two examples of applying SLIDER and MAGNET filtering algorithms to different sequence pairs. In Fig. 6, we set the edit distance threshold to 4 in both examples. The diagonal vectors of the neighborhood map are horizontally presented in the same order of the diagonal vectors for a better illustration. In the two examples, we observe that MAGNET is highly accurate in providing the exact location of the edits in the MAGNET bit-vector. This is due to two main reasons. First, MAGNET finds the exact length of each common subsequence by performing multiple individual iteration for each common subsequence. Second, it manually encapsulates each found longest subsequence of consecutive zeros by ones, which ensures to maintain the edits in the MAGNET bit-vector. On the contrary, SLIDER uses overlapping search windows to detect segments of consecutive zeros. If two segments of consecutive zeros are overlapped within a single search window, then the edit between the two segments is sometimes eliminated by the overlapping zeros of the two segments as shown in Fig. 6(a).

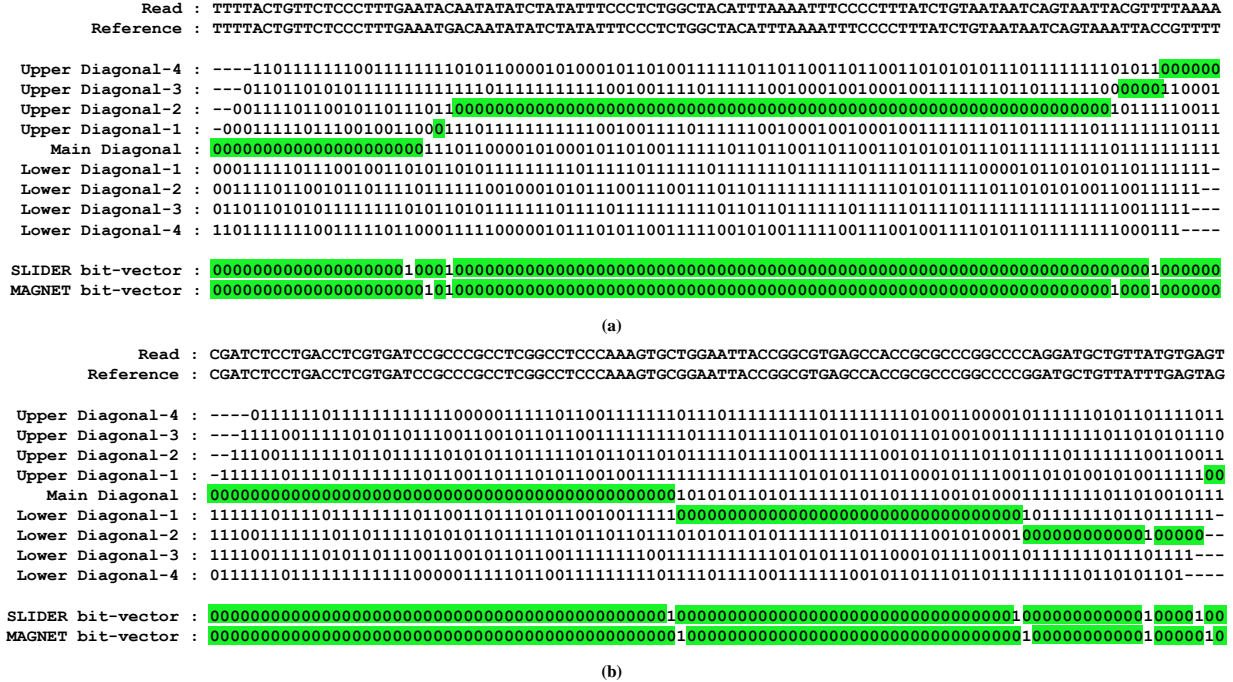


Fig. 6: Examples of applying SLIDER and MAGNET filtering algorithms to two different sequence pairs, where the edit distance threshold is set to 4. We present the content of the neighborhood map along with the SLIDER and MAGNET bit-vectors. In both examples, we apply SLIDER and MAGNET algorithms starting from the leftmost column towards the rightmost column.

## 9 Dataset Description

Table 4 provides the configuration used for the  $-e$  parameter of mrFAST (Alkan, et al., 2009) for each of the 12 datasets. We use Edlib (Šošić and Šikić, 2017) to assess the number of similar (i.e., having edits fewer than or equal to the edit distance threshold) and dissimilar (i.e., having more edits than the edit distance threshold) pairs for each of the 12 datasets across different user-defined edit distance thresholds. We provide these details for set 1, set 2, set 3, and set 4 in Table 5. We provide the same details for set 5, set 6, set 7, and set 8 in Table 6 and for set 9, set 10, set 11, and set 12 in Table 7.

**Table 4: Benchmark illumina-like datasets (read-reference pairs). We map each read set to the human reference genome in order to generate four datasets using different mappers’ edit distance thresholds (using the  $-e$  parameter).**

Accession no.	ERR240727_1				SRR826460_1				SRR826471_1			
Sequence Length	100				150				250			
HTS	Illumina HiSeq 2000				Illumina HiSeq 2000				Illumina HiSeq 2000			
Dataset	Set_1	Set_2	Set_3	Set_4	Set_5	Set_6	Set_7	Set_8	Set_9	Set_10	Set_11	Set_12
mrFAST $-e$	2	3	5	40	4	6	10	70	8	12	15	100
Amount of Edits	Low-edit		High-edit		Low-edit		High-edit		Low-edit		High-edit	

**Table 5: Details of our first four datasets (set 1, set 2, set 3, and set 4). We use Edlib to benchmark the accepted and the rejected pairs for edit distance thresholds of  $E=0$  up to  $E=10$  edits.**

Dataset	Set_1		Set_2		Set_3		Set_4	
$E$	Accepted	Rejected	Accepted	Rejected	Accepted	Rejected	Accepted	Rejected
0	381,901	29,618,099	124,531	29,875,469	11,989	29,988,011	11	29,999,989
1	1,345,842	28,654,158	441,927	29,558,073	44,565	29,955,435	18	29,999,982
2	3,266,455	26,733,545	1,073,808	28,926,192	108,979	29,891,021	24	29,999,976
3	5,595,596	24,404,404	2,053,181	27,946,819	206,903	29,793,097	27	29,999,973
4	7,825,272	22,174,728	3,235,057	26,764,943	334,712	29,665,288	29	29,999,971
5	9,821,308	20,178,692	4,481,341	25,518,659	490,670	29,509,330	34	29,999,966
6	11,650,490	18,349,510	5,756,432	24,243,568	675,357	29,324,643	83	29,999,917
7	13,407,801	16,592,199	7,091,373	22,908,627	891,447	29,108,553	177	29,999,823
8	15,152,501	14,847,499	8,531,811	21,468,189	1,151,447	28,848,553	333	29,999,667
9	16,894,680	13,105,320	10,102,726	19,897,274	1,469,996	28,530,004	711	29,999,289
10	18,610,897	11,389,103	11,807,488	18,192,512	1,868,827	28,131,173	1,627	29,998,373

**Table 6: Details of our second four datasets (set\_5, set\_6, set\_7, and set\_8). We report the accepted and the rejected pairs for edit distance thresholds of  $E=0$  up to  $E=15$  edits.**

Dataset	Set_5		Set_6		Set_7		Set_8	
$E$	Accepted	Rejected	Accepted	Rejected	Accepted	Rejected	Accepted	Rejected
0	1,440,497	28,559,503	248,920	29,751,080	444	29,999,556	201	29,999,799
1	1,868,909	28,131,091	324,056	29,675,944	695	29,999,305	327	29,999,673
3	2,734,841	27,265,159	481,724	29,518,276	927	29,999,073	444	29,999,556
4	3,457,975	26,542,025	612,747	29,387,253	994	29,999,006	475	29,999,525
6	5,320,713	24,679,287	991,606	29,008,394	1,097	29,998,903	529	29,999,471
7	6,261,628	23,738,372	1,226,695	28,773,305	1,136	29,998,864	546	29,999,454
9	7,916,882	22,083,118	1,740,067	28,259,933	1,221	29,998,779	587	29,999,413
10	8,658,021	21,341,979	2,009,835	27,990,165	1,274	29,998,726	612	29,999,388
12	10,131,849	19,868,151	2,591,299	27,408,701	1,701	29,998,299	710	29,999,290
13	10,917,472	19,082,528	2,923,699	27,076,301	2,146	29,997,854	796	29,999,204
15	12,646,165	17,353,835	3,730,089	26,269,911	3,921	29,996,079	1,153	29,998,847





## 11 FPGA Acceleration of SLIDER and MAGNET

We analyze the benefits of accelerating the CPU implementation of our pre-alignment filters SLIDER and MAGNET using FPGA hardware. As we show in Table 8, our hardware accelerators are two to three orders of magnitude faster than the equivalent CPU implementations of SLIDER and MAGNET.

**Table 8: Execution time (in seconds) of the CPU implementations of SLIDER and MAGNET filters and that of their hardware-accelerated versions (using a single filtering unit).**

<i>E</i>	SLIDER-CPU	SLIDER-FPGA	Speedup	MAGNET-CPU	MAGNET-FPGA	Speedup
<i>Sequence Length = 100</i>						
2	474.27	2.89	164.11x	632.02	2.89	218.69x
5	1,305.15	2.89	451.61x	1,641.57	2.89	568.02x
<i>Sequence Length = 250</i>						
2	1,689.09	2.89*	584.46x	5,567.62	2.89*	1,926.51x
5	6,096.61	2.89*	2,109.55x	14,328.28	2.89*	4,957.88x

\* Estimated based on the resource utilization and data throughput

## 12 Edlib, Parasail, and SHD Configurations

In Table 9, we list the software packages that we cover in our performance evaluation, including their version numbers and function calls used.

**Table 9: Read aligners and pre-alignment filters used in our performance evaluations.**

<b>Edlib: November 5 2017</b>
Edit Distance Mode: EdlibAlignResult resultEdlib = edlibAlign(RefSeq, ReadLength, ReadSeq, ReadLength, edlibDefaultAlignConfig()); Accepted = (resultEdlib.editDistance <= ErrorThreshold); edlibFreeAlignResult(resultEdlib);
Levenshtein Distance with backtracking: EdlibAlignResult resultEdlib = edlibAlign(RefSeq, ReadLength, ReadSeq, ReadLength, edlibNewAlignConfig(ErrorThreshold, EDLIB_MODE_NW, EDLIB_TASK_PATH, NULL, 0)); char* cigar = edlibAlignmentToCigar(resultEdlib.alignment, resultEdlib.alignmentLength, EDLIB_CIGAR_STANDARD); free(cigar); edlibFreeAlignResult(resultEdlib);
<b>Parasail: January 7 2018</b>
function = parasail_lookup_function("nw_banded"); result = function(RefSeq, ReadLength, ReadSeq, ReadLength, 10, 1, ErrorThreshold, &parasail_blosum62); if (parasail_result_is_trace(result) == 1) { parasail_traceback_generic(RefSeq, ReadLength, ReadSeq, ReadLength, "Query:", "Target:", &parasail_blosum62, result, ' ', ':', ':', 50, 14, 0); if (result->score != 0) { cigar2 = parasail_result_get_cigar(result, RefSeq, ReadLength, ReadSeq, ReadLength, &parasail_blosum62); parasail_cigar_free(cigar2); } } }
<b>SHD: November 7 2017</b>
for (k=1; k<=1+(ReadLength/128); k++) totalEdits = totalEdits + (bit_vec_filter_sse1(read_t, ref_t, length, ErrorThreshold));

## REFERENCES

- Alkan, C., *et al.* (2009) Personalized copy number and segmental duplication maps using next-generation sequencing, *Nature genetics*, **41**, 1061-1067.
- Alser, M., *et al.* (2017) GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping, *Bioinformatics*, **33**, 3355-3363.
- Alser, M., Mutlu, O. and Alkan, C. (July 2017) Magnet: Understanding and improving the accuracy of genome pre-alignment filtering, *Transactions on Internet Research* **13**.
- Hamming, R.W. (1950) Error detecting and error correcting codes, *Bell System technical journal*, **29**, 147-160.
- McNamara, M. (2001) IEEE Standard Verilog Hardware Description Language. The Institute of Electrical and Electronics Engineers, Inc. *IEEE Std*, 1364-2001.
- Šošić, M. and Šikić, M. (2017) Edlib: a C/C++ library for fast, exact sequence alignment using edit distance, *Bioinformatics*, **33**, 1394-1395.
- Xilinx (November 17, 2014) 7 Series FPGAs Configurable Logic Block User Guide, Xilinx.
- Xin, H., *et al.* (2015) Shifted Hamming Distance: A Fast and Accurate SIMD-Friendly Filter to Accelerate Alignment Verification in Read Mapping, *Bioinformatics*, **31**, 1553-1560.