Subject Section

SneakySnake: A Fast and Accurate Universal Genome Pre-Alignment Filter for CPUs, GPUs, and FPGAs

Mohammed Alser¹,³,∗, Taha Shahroodi¹, Juan Gómez-Luna¹, Can Alkan³,∗, and Onur Mutlu¹,²,³,∗

¹Department of Computer Science, ETH Zurich, Zurich 8006, Switzerland
²Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh 15213, PA, USA
³Department of Computer Engineering, Bilkent University, Ankara 06800, Turkey

∗To whom correspondence should be addressed.
Associate Editor: XXXXXXX
Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: We introduce SneakySnake, a highly parallel and highly accurate pre-alignment filter that remarkably reduces the need for the computationally costly sequence alignment step. The key idea of SneakySnake is to reduce the approximate string matching (ASM) problem to the single net routing (SNR) problem in VLSI chip layout. In the SNR problem, we are interested in only finding the optimal path that connects two terminals with the least routing cost on a special grid layout that contains obstacles. The SneakySnake algorithm quickly solves the SNR problem and uses the found optimal path to decide whether performing sequence alignment is necessary. Reducing the ASM problem into SNR also makes SneakySnake efficient to implement for CPUs, GPUs, and FPGAs.

Results: SneakySnake significantly improves the accuracy of pre-alignment filtering by up to four orders of magnitude compared to the state-of-the-art pre-alignment filters, Shouji, GateKeeper, and SHD. Using short sequences, SneakySnake accelerates Edlib (state-of-the-art implementation of Myers’s bit-vector algorithm) and Parasail (sequence aligner with configurable scoring function), by up to 37.7× and 43.9× (>12× on average), respectively, without requiring hardware acceleration, and by up to 413× and 689× (>600× on average), respectively, using hardware acceleration. Using long sequences, SneakySnake accelerates Parasail by up to 979× (140.1× on average). SneakySnake also accelerates the sequence alignment of minimap2, a state-of-the-art read mapper, by up to 6.83× (4.67× on average) and 64.1× (17.1× on average) using short and long sequences, respectively, and without requiring hardware acceleration. As SneakySnake does not replace sequence alignment, users can still configure the aligner of their choice for different scoring functions, surpassing most existing acceleration efforts.

Availability: https://github.com/CMU-SAFARI/SneakySnake
Contact: alserm@inf.ethz.ch, calkan@cs.bilkent.edu.tr, omutlu@inf.ethz.ch
Supplementary information: Supplementary data is available at Bioinformatics online.

1 Introduction

One of the most fundamental computational steps in most genomic analyses is sequence alignment. This step is formulated as an approximate string matching (ASM) problem (Navarro, 2001) and it calculates: (1) edit distance between two given sequences, (2) type of each edit (i.e., insertion, deletion, or substitution), and (3) location of each edit in one of the two given sequences. Edit distance is defined as the minimum number of edits needed to convert one sequence into the other (Levenshtein, 1966). These edits result from both sequencing errors (Fox et al., 2014) and genetic variations (McKernan et al., 2009). Edits can have different weights, based on a user-defined scoring function, to allow favoring one edit type over another (Wang et al., 2011). Sequence alignment involves a backtracking step, which calculates an ordered list of characters representing the location and type of each possible edit operation 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,
we need to examine all possible prefixes of the two input sequences and keep track of the pairs of prefixes that provide a minimum edit distance. Therefore, sequence alignment approaches are typically implemented as dynamic programming (DP) algorithms to avoid re-examining the same prefixes many times (Eddy, 2004). DP-based sequence alignment algorithms, such as Needleman-Wunsch (Needleman and Wunsch, 1970), are computationally expensive 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. Recent attempts tend to follow one of two key directions: (1) Accelerating the DP algorithms using hardware accelerators and (2) Developing pre-alignment filtering heuristics that reduce the need for the DP algorithms, given an edit distance threshold.

Hardware accelerators include building aligners that use 1) multi-core and SIMD (single instruction multiple data) capable central processing units (CPUs), such as Parasail (Daily, 2016). The classical DP algorithms can also be accelerated by calculating a bit representation of the DP matrix and processing its bit-vectors in parallel, such as Myers’s bit-vector algorithm (Myers, 1999). To our knowledge, Edlib (Šošić and Šikić, 2017) is currently the best-performing implementation of Myers’s bit-vector algorithm. Other hardware accelerators include 2) graphics processing units (GPUs), such as GSWABE (Liu and Schmidt, 2015) and CUDA/SW+ (Liu et al., 2015), 3) field-programmable gate arrays (FPGAs), such as FPGASW (Fei et al., 2018), or 4) processing-in-memory architectures that enable performing computations inside the memory chip and alleviate the need for transferring the data to the CPU cores, such as RADAR (Huangfu et al., 2018). However, many of these efforts either simplify the scoring function as in Edlib, or only take into account accelerating the computation of the DP matrix without performing the backtracking step as in (Liu et al., 2013; Nishimura et al., 2017; Chen et al., 2014). Different and more sophisticated scoring functions are typically needed to better quantify the similarity between two sequences (Wang et al., 2011). The backtracking step involves unpredictable and irregular memory access patterns, which pose 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. Existing pre-alignment filtering techniques are either: 1) slow and they suffer from a limited sequence length (\(\leq 128\) bp), such as SHD (Xin et al., 2015), or 2) inaccurate after some edit distance threshold, such as GateKeeper (Alser et al., 2017a) and MAGNET (Alser et al., 2017b). Shouji (Alser et al., 2019) is currently the best-performing FPGA pre-alignment filter in terms of both accuracy and execution time.

Our goal in this work is to significantly reduce the time spent on calculating the sequence alignment of short sequences using very fast and accurate pre-alignment filtering. To this end, we introduce SneakySnake, a highly parallel and highly accurate pre-alignment filter that works on modern high-performance computing architectures such as CPUs, GPUs, and FPGAs. The key idea of SneakySnake is to provide highly-accurate pre-alignment filtering algorithm by reducing the ASM problem to the single net routing (SNR) problem (Lee et al., 1976). The SNR problem is to find the shortest routing path that interconnects two terminals on the boundaries of VLSI chip layout and passes through the minimum number of obstacles. Solving the SNR problem is faster than solving the ASM problem, as calculating the routing path after facing an obstacle is independent of the calculated path before this obstacle. This provides two key benefits. 1) It obviates the need for using computationally costly DP algorithms to keep track of the subpath that provides optimal (i.e., with the least possible routing cost) solution. 2) The independence between the subpaths allows for solving many SNR subproblems in parallel by judiciously leveraging the parallelism-friendly architecture of modern FPGAs and GPUs to greatly speed up the SneakySnake algorithm.

The contributions of this paper are as follows:

- We introduce SneakySnake, the fastest and most accurate pre-alignment filtering mechanism to date that greatly enables the speeding up of genome sequence alignment while preserving its accuracy. We demonstrate that the SneakySnake algorithm is 1) correct and optimal in solving the SNR problem in linear time with respect to the sequence length and the edit distance threshold.
- We demonstrate that the SneakySnake algorithm significantly improves the accuracy of pre-alignment filtering by up to four orders of magnitude compared to Shouji, GateKeeper, and SHD.
- We provide, to our genomic sequences first filter by more than the edit distance threshold, then the two sequences are identified as dissimilar sequences and hence identifying the location and the type of each edit is not needed. The edit distance estimated by the SneakySnake algorithm should always be less than or equal to the actual edit distance value so that SneakySnake ensures reliable and lossless filtering (preserving all similar sequences). To reliably estimate the edit distance between two sequences, we reduce the ASM problem to the SNR problem. That is, instead of calculating the sequence alignment, the SneakySnake algorithm finds the routing path that interconnects two terminals and passes through the minimum number of obstacles on VLSI chip. The number of obstacles faced throughout the found routing path represents the lower bound on the edit distance between two sequences (Theorem 2, Section 2.4) and hence this number of obstacles can be used for the reliable filtering decision of SneakySnake. SneakySnake treats all obstacles (edits) faced along a path equally (i.e., it does not favor one type of edits over the others). This eliminates the need for examining different possible arrangements of the edit operations, as in DP-based algorithms, and makes solving the SNR problem easier and faster than solving the ASM problem. However, users can still configure the aligner of their choice for their desired scoring function. Next, we explain the SNR problem.

2.2 Single Net Routing (SNR) Problem

The SNR problem in VLSI chip layout refers to the problem of optimally interconnecting two terminals on a special grid graph while respecting constraints. We present an example of a VLSI chip layout in Fig. 1. The goal is to find the optimal path (called signal net) that connects the source and the destination terminals through the chip layout. We describe the special grid graph of the SNR problem and define such optimal signal net as follows:

- The chip layout has two layers of evenly spaced metal routing tracks. While the first layer allows traversing the chip horizontally through dedicated horizontal routing tracks (HRTs), the second layer allows traversing the chip vertically using dedicated vertical routing tracks (VRTs).
2.3 Reducing the Approximate String Matching (ASM) Problem to the Single Net Routing (SNR) Problem

We reduce the problem of finding the similarities and differences between two genomic sequences to that of finding the optimal signal net in a VLSI chip layout. Reducing the ASM problem to the SNR problem requires two key steps: (1) replacing the DP table used by the sequence alignment algorithm to a special grid graph called chip maze and (2) finding the number of differences between two genomic sequences in the chip maze by solving the SNR problem. We replace the \((m + 1) \times (m + 1)\) DP table with our chip maze, \(Z\), where \(m\) is the sequence length (for simplicity, we assume that we have a pair of equal-length sequences but we relax this assumption towards the end of the next section). The chip maze is a \((2E + 1) \times m\) grid graph, where \(E\) is the edit distance threshold, and \(m\) is the number of VRTs. The chip maze is an abstract layout for the VLSI chip layout, as we show in Fig. 2(b) for the same chip layout of Fig. 1. Each entry of the chip maze represents the pairwise comparison result of a character of one sequence with another character of the other sequence. A pairwise mismatch is represented by an obstacle (an entry of value ‘1’) in the chip maze and a pairwise match is represented by an available path (an entry of value ‘0’) in its corresponding HRT. Given two genomic sequences, a reference sequence \(R[1 \ldots m]\) and a query sequence \(Q[1 \ldots m]\), and an edit distance threshold \(E\), we calculate the entry \(Z[i, j]\) of the chip maze, where \(1 \leq i \leq (2E + 1)\) and \(1 \leq j \leq m\), as follows:

\[
Z[i, j] = \begin{cases} 
0, & \text{if } i = E + 1, \ Q[j] = R[i] \\
0, & \text{if } 1 \leq i \leq E, \ Q[j - i] = R[i] \\
0, & \text{if } i > E + 1, \ Q[j + i - E - 1] = R[i] \\
1, & \text{otherwise}
\end{cases}
\]

We derive this equation by considering all possible pairwise matches and mismatches (due to possible edits) between two sequences. That is, each column of the chip maze stores the result of comparing the \(j\)th character of the reference sequence, \(R\), with each of its corresponding \((2E + 1)\) characters of the query sequence, \(Q\). As we show in Fig. 2(a), these \((2E + 1)\) characters of the query sequence, \(Q\), are as follows: the \(j\)th character of the query sequence, \(Q\), the \(E\) right-hand neighboring characters of the \(j\)th character, and the \(E\) left-hand neighboring characters of the \(j\)th character.

This is essential to maintain an accurate detection of substituted, deleted, and inserted characters in one or both given sequences. Each insertion and deletion can shift multiple trailing characters (e.g., deleting the character ‘N’ from ‘GENOME’ shifts the last three characters to the left direction, making it ‘GEOME’). Hence, we need to compute a character of the reference sequence \(R\) with the neighboring characters of its corresponding character of the query sequence, \(Q\), to cancel the effect of deletion/insertion and correctly detect the common subsequences between two sequences. As the substitutions have no shift effect on the alignment of subsequent bases, comparing the \(j\)th character of the reference sequence \(R\) with the \(j\)th character of the query sequence \(Q\) is needed to find such substitutions. We fill the remaining empty entries of each row by ones (i.e., obstacles) to indicate that there is no match between the corresponding characters. We present in Fig. 2(b) an example of the chip maze for two sequences, where a query sequence, \(Q\), differs from a reference sequence, \(R\), by three edits.

The chip maze is a data-dependency free data structure as computing each of its entries is independent of every other and thus the entire grid graph can be computed all at once in a parallel fashion. Hence, our chip maze is well suited for both sequential and highly-parallel computing platforms (Seshadi et al., 2017). The challenge is now calculating the minimum number of edits between two sequences using the chip maze. Considering the chip maze as a chip layout where the rows represent the HRTs and the columns represent the VRTs, we observe that we can reduce the ASM problem to the SNR problem. Now, the problem becomes finding an optimal set (i.e., signal net) of non-overlapping escape segments. As we discuss in Section 2.2, a set of escape segments is optimal if there is no other set that solves the SNR problem and has both less number of escape segments and less number of entries of value ‘1’ (i.e., obstacles). Once we find such an optimal set of escape segments, we can compute the minimum number of edits between two sequences as the total number of obstacles along the computed optimal set. Next, we present an efficient algorithm that solves this SNR problem.
algorithm from ever searching backward for the longest escape segment. This leads to a signal net that has non-overlapping escape segments.

To achieve these two key objectives, the SneakySnake algorithm applies five effective steps. (1) The SneakySnake algorithm first constructs the chip maze as we explain in the previous section. It then considers the first column of the chip maze as the first checkpoint, where the first iteration starts. (2) At each new checkpoint, the SneakySnake algorithm always selects the longest escape segment that allows the signal to travel as far forward as possible until it reaches an obstacle. For each row of the chip maze, it computes the length of the first horizontal segment of consecutive entries of value ‘0’ that starts from a checkpoint and ends at an obstacle or at the end of the current row. The SneakySnake algorithm compares the length of all the $2E + 1$ computed horizontal segments, selects the longest one, and considers it along with its first following obstacle as an escape segment. If the SneakySnake algorithm is unable to find a horizontal segment (i.e., following a checkpoint, all rows start with an obstacle), it considers one of the obstacles as the longest escape segment. It considers the computed escape segment as part of the solution to the SNR problem. (3) It creates a new checkpoint after the longest escape segment. (4) It repeats the second and third steps until either the signal net reaches a destination terminal, or the total propagation delay exceeds the allowed propagation delay threshold (i.e., $E \times 1$). (5) If SneakySnake finds the optimal net using the previous steps, then sequence alignment (e.g., exact number of edits, type of each edit, and location of each edit) between two sequences is calculated using user’s favorite sequence alignment algorithm. Otherwise, the SneakySnake algorithm terminates without performing computationally expensive sequence alignment.

To efficiently implement the SneakySnake algorithm, we use an implicit representation of the chip maze. That is, the SneakySnake algorithm starts computing on-the-fly one entry of the chip maze after another for each row until it faces an obstacle (i.e., $Z[i,j] = 1$) or it reaches the end of the current row. Thus, the entries that are actually calculated for each row of the chip maze are the entries that are located only between each checkpoint and the first obstacle, in each row, following this checkpoint, as we show in Fig. 2(c). This significantly reduces the number of computations needed for the SneakySnake algorithm. We provide the SneakySnake algorithm along with analysis of its computational complexity (asymptotic run time and space complexity) in Supplementary Materials, Section 6.

The SneakySnake algorithm is both correct and optimal. The SneakySnake algorithm is correct as it always provides a signal net (if it exists) that interconnects the source terminal and the destination terminal. In other words, it does not lead to routing failure as signal will eventually reach its destination.

**Theorem 1.** The SneakySnake algorithm guarantees to find a signal net that interconnects the source terminal and the destination terminal when one exists.

We provide the correctness proof for Theorem 1 in Supplementary Materials, Section 5.1. The SneakySnake algorithm is also optimal as it guarantees to find an optimal signal net that links the source terminal to destination terminal when one exists. Such an optimal signal net always ensures that the signal arrives the destination terminal with the least possible total propagation delay.

**Theorem 2.** When a signal net exists between the source terminal and the destination terminal, using the SneakySnake algorithm, a signal from the source terminal reaches the destination terminal with the minimum possible latency.

We provide the optimality proof for Theorem 2 in Supplementary Materials, Section 5.2.

Different from existing edit distance approximation algorithms (Batu et al., 2006; Andoni and Onak, 2012; Chakraborty et al., 2018; Charikar et al., 2018) that sacrifice the optimality of the edit distance solution (i.e., its solution $\geq$ the actual edit distance of each sequence pair) for a reduction in time complexity, (e.g., $O(n^{1.647})$ instead of $O((n^2)$), SneakySnake does not overestimate the edit distance as the calculated optimal signal net has $always$ the minimum possible number of obstacles (Theorem 2). We can justify the edit distance underestimation of SneakySnake by using our fast computation method as a pre-alignment filtering step to decide whether sequence alignment computation is needed. If two sequences have more edits than the edit distance threshold, $E$, then we do not need computationally costly algorithms to conclude that the two sequences have unacceptable number of edits. But if the number of edits is less than or equal the edit distance threshold, then our filtering step should be followed by accurate sequence alignment algorithms, where users can choose different scoring functions. This way ensures achieving two key properties: (1) allowing sequence alignment to be calculated only for similar (or nearly similar) sequences and (2) accelerating the sequence alignment algorithms without changing (or replacing) their algorithmic method and hence preserving all the capabilities of the sequence alignment algorithms.

Sequence alignment can be performed as a global alignment, where two sequences of the same length are aligned end-to-end, or a local alignment, where subsequences of the two given sequences are aligned. It can also be performed as a semi-global alignment, where the entirety of one sequence is aligned to one of the ends of the other sequence. To ensure a correct reduction of the ASM problem, we need to count the number of obstacles more conservatively along the optimal solution set. This means that if an optimal alignment algorithm performs a local alignment, then we need to deduct the total number of leading and trailing obstacles from the total count of edits between two given sequences before making the filtering decision. Similarly for a semi-global alignment, we should not consider the leading or the trailing obstacles. For the rest of the paper, we consider only the global alignment as the general case since it is more challenging and includes more computations, as it examines the similarity end-to-end.

We next discuss further optimizations and new software/hardware co-designed versions of the SneakySnake algorithm that leverage FPGA and GPU architectures for highly-parallel computation.
2.5 Snake-on-Chip Hardware Architecture

We introduce an FPGA-friendly architecture for the SneakySnake algorithm, called Snake-on-Chip. The main idea behind the hardware architecture of Snake-on-Chip is to divide the SNR problem into smaller non-overlapping subproblems. Each subproblem has a width of $t$ VRTs and a height of $2E + 1$ HRTs, where $1 < t \leq m$. We then solve each subproblem independently from the other subproblems. This approach results in three key benefits. (1) Downizing the search space into a reasonably small grid graph with a known dimension at the design time limits the number of all possible solutions for that subproblem. This reduces the size of the look-up tables (LUTs) required to build the architecture and simplifies the overall design. (2) Dividing the SNR problem into subproblems helps to maintain a modular and scalable architecture that can be implemented for any sequence length and edit distance threshold. (3) All the smaller subproblems can be solved independently and rapidly with a high parallelism. This reduces the execution time of the overall algorithm as the SneakySnake algorithm does not need to evaluate the entire chip maze.

However, these three key benefits come at the cost of accuracy degradation. As we demonstrate in Theorem 2, the SneakySnake algorithm guarantees to find an optimal solution to the SNR problem. However, the solution for each subproblem is not necessarily part of the optimal solution for the main problem (with the original size of $(2E + 1) \times m$). This is because the source and destination terminals of these subproblems are not necessarily the same. The source and destination terminals should be located at any of the $2E + 1$ entries of the first and the last VRTs, respectively, of each subproblem, but the SneakySnake architecture determines the exact location of the source and destination terminals for each subproblem based on its individual optimal solution. This causes to underestimate the total number of obstacles found along each signal net of each SNR subproblem. This is still acceptable as long as it solves the SNR problem quickly and without overestimating the number of obstacles. We provide the details of our hardware architecture of Snake-on-Chip in Supplementary Materials, Section 9.

2.6 Snake-on-GPU Parallel Implementation

We now introduce our GPU implementation of the SneakySnake algorithm, called Snake-on-GPU. The main idea of Snake-on-GPU is to exploit the large number (typically few thousands) of GPU threads provided by modern GPUs to solve a large number of SNR problems rapidly and concurrently. In Snake-on-Chip, we explicitly divide the SNR problem into smaller non-overlapping subproblems and then solve all subproblems concurrently and independently using our specialized hardware. In Snake-on-GPU, we follow a different approach than that of Snake-on-Chip by keeping the same size of the original SNR problem and solving a massive number of these SNR problems at the same time. Snake-on-GPU uses one single GPU thread to solve one SNR problem (i.e., comparing one query sequence to one reference sequence at a time). This granularity of computation fits well the amount of resources (e.g., registers) that are available to each GPU thread and avoids the need for synchronizing several threads working on the same SNR problem. GPUs offer more flexibility to the users to change the values of some input parameters of Snake-on-Chip without the need to build a new design as in FPGAs.

Given the large size of the sequence pair dataset that the GPU threads need to access, we carefully design Snake-on-GPU to efficiently 1) copy the input dataset of query and reference sequences into the GPU global memory, which is the off-chip DRAM memory of GPUs (NVidia, 2019a) and it typically fits a few GB of data and 2) allow each thread to store its own query and reference sequences using the on-chip register file to avoid unnecessary accesses to the off-chip global memory. Each thread solves the complete SNR problem for a single query sequence and a single reference sequence. We provide the details of our parallel implementation of Snake-on-GPU in Supplementary Materials, Section 10.

3 Results

We now evaluate 1) the filtering accuracy, 2) the filtering time, and 3) the benefits of combining our universal implementation of the SneakySnake algorithm with state-of-the-art aligners. We provide the exact value of all evaluation results in https://github.com/CMU-SAFARI/SneakySnake. For each experiment, we compare the performance of SneakySnake, Snake-on-Chip, and Snake-on-GPU to the existing state-of-the-art pre-alignment filters, Shouji, MAGNET, GateKeeper, and SHD. We use the experiments that use multithreading and long sequences using a 2.3 GHz Intel Xeon Gold 5118 CPU with up to 48 threads and 192 GB RAM. We run all other experiments using a 3.3 GHz Intel E3-1225 CPU with 32 GB RAM. We use a Xilinx Virtex 7 VC709 board (Xilinx, 2013) to implement Snake-on-Chip and other existing accelerator architectures (for Shouji, MAGNET, and GateKeeper). We build the FPGA design using Vivado 2015.4 in synthesizable Verilog. We use a NVIDIA GeForce RTX 2080 Ti card (NVIDIA, 2019b) with a global memory of 11 GB DDR6 to implement Snake-on-GPU. Both Snake-on-Chip and Snake-on-GPU are independent of the specific FPGA and GPU platforms as they do not rely on any vendor-specific computing elements (e.g., intellectual property cores).

3.1 Dataset Description

We have two key approaches to generate sequence pairs for testing the performance of pre-alignment filters: 1) using existing read mappers to find reference segments that might be similar or dissimilar to real reads, and 2) using available read simulators. These reference segments generated by read mappers are not necessarily reported in their output SAM file, as they are generated before applying DP-based pairwise alignment step to ensure that the edit distance of every generated pair is not necessarily within the edit distance threshold. We follow both approaches, as they are still widely-used in evaluating existing algorithms. Our experimental evaluation uses 4 different real datasets and 2 simulated datasets.

Real datasets. Each real dataset contains 30 million real sequence pairs (text and query pairs). We obtain two different read sets, ERR240727_1 and SRX626471_1, of the whole human genome that include two different read lengths, 100 bp and 250 bp, respectively. We download these two read sets from EMBL-ENA (https://www.ebi.ac.uk/ena). We map each read set to the human reference genome (GRCh37) using mrFAST (Alkan et al., 2009) mapper and observe all potential mapping locations of every read. We obtain the human reference genome from the 1000 Genomes Project (The 1000 Genomes Project Consortium, 2015). Before mapping the reads, we disable the DP-based pairwise alignment algorithm of mrFAST mapper to obtain both aligned and unaligned sequences. For each read set, we use two different maximum numbers of allowed edits (2 and 40 for rs = 100 bp and 8 and 100 for rs = 250 bp) using the $e$ parameter of mrFAST to generate four real datasets in total. We provide the details of these four datasets in the Supplementary Materials, Section 11.1. For the reader’s convenience, we refer to these datasets as Set_1, Set_2, Set_3, and Set_4.

Simulated datasets. We generate two sets (we refer to them as Set_5 and Set_6) of long sequence pairs using PBSIM (Ong et al., 2013). We choose this simulator as it provides pairs of two sequences, the original segment of the reference (or only the location as in some read simulators) and its simulated segment. This helps us to directly obtain sequence pairs that can be used for evaluating the performance of sequence aligners and pre-alignment filters. We use Human chromosome 1 (GRCh38.p13 assembly, downloaded from https://www.ncbi.nlm.nih.gov/assembly/GC_000001.13) sequence for the input reference sequence in PBSIM. We generate Set_5 to have 100,000 sequence pairs, each of which is 10 Kbp long, at 30x genome coverage. Set_6 has 74,687 sequence pairs, each of which is 100 Kbp long, at 30x genome coverage. For both sets (Set_5 and Set_6), we use the default error profile for the continuous long reads (CLR) in PBSIM.

3.2 Filtering Accuracy

We evaluate the accuracy of pre-alignment filter by computing its rate of falsely-accepted and falsely-rejected sequences to performance...
sequence alignment. The false accept 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 the sequence alignment algorithm. The false reject rate is the ratio of the number of similar sequences that are rejected by the filter and the number of similar sequences that are accepted by the sequence alignment algorithm. A reliable pre-alignment filter should always ensure both a 0% false reject rate and minimal false accept rate to maintain the correctness of overall pipeline and maximize the number of dissimilar sequences that are eliminated.

We first assess the false accept rate of SneakySnake, Shouji, MAGNET, GateKeeper, and SHD across different four real datasets and edit distance thresholds of 0% – 10% of the sequence length. In Fig. 3, we provide the false accept rate of each of the five filters. We use Edlib to identify the ground truth truly-accepted sequences for each edit distance threshold. Based on Fig. 3, we make four key observations. (1) We observe that all the five pre-alignment filters are less accurate in examining Set_1 and Set_3 than the other datasets, Set_2 and Set_4. (2) GateKeeper and SHD become ineffective for edit distance thresholds of greater than 6% and 3% for sequence lengths of 100 and 250 characters, respectively, as they accept all the input sequence pairs. This causes them to examine each sequence pair unnecessarily twice (i.e., once by GateKeeper or SHD and once by the sequence alignment algorithm). (3) SneakySnake provides the lowest false accept rate compared to all the four state-of-the-art pre-alignment filters. SneakySnake provides up to 31412 ×, 20603 ×, and 64.1 × less number of falsely-accepted sequences compared to GateKeeper/SHD (using Set_4, E = 10%), Shouji (using Set_4, E = 10%), and MAGNET (using Set_1, E = 1%), respectively. (4) MAGNET provides the second lowest false accept rate. It provides up to 25532 × and 16760 × less number of falsely-accepted sequences compared to GateKeeper/SHD (using Set_4, E = 10%) and Shouji (using Set_4, E = 10%), respectively.

Second, we assess the false reject rates of SneakySnake and observe that SneakySnake has a 0% false reject rate. It is also demonstrated in (Alser et al., 2019) that Shouji and GateKeeper have a 0% false reject rate, while MAGNET can falsely reject some similar sequence pairs.

We conclude that SneakySnake improves the accuracy of pre-alignment filtering by up to four orders of magnitude compared to the state-of-the-art pre-alignment filters. We also conclude that SneakySnake is the most effective pre-alignment filter, with a very low false accept rate and a 0% false reject rate across a wide range of both edit distance thresholds and sequence lengths.

3.3 Effects of SneakySnake on Sequence Alignment

We analyze the benefits of integrating CPU-based pre-alignment filters, SneakySnake and SHD with the state-of-the-art CPU-based sequence aligners, Edlib and Parasail. We evaluate all tools using a single CPU core and single thread environment. Fig. 4(a) and (b) present the normalized end-to-end execution time of SneakySnake and SHD each combined with Edlib and Parasail, using our four real datasets over edit distance thresholds of 0% – 10% of the sequence length. We make four key observations. (1) SneakySnake is up to 43× (using Set_3, E = 0%) and 47.2× (using Set_3, E = 2%) faster than Edlib and Parasail, respectively, in examining the sequence pairs. (2) The addition of SneakySnake as a pre-alignment filtering step reduces significantly the execution time of Edlib and Parasail by up to 37.7 × (using Set_4, E = 0%) and 34.9× (using Set_4, E = 3%), respectively. We also observe a similar trend as the number of CPU threads increases from 1 to 40, as we provide in Supplementary Materials, Section 11.2. (3) The addition of SHD as a pre-alignment step reduces the execution time of Edlib and Parasail for some of the edit distance thresholds by up to 17.2× (using Set_2, E = 0%) and 34.9× (using Set_4, E = 3%), respectively. However, for most of the edit distance thresholds, we observe that Edlib and Parasail are faster alone than with SHD combined as a pre-alignment filtering step. This is expected as SHD becomes ineffective in filtering for E > 8% and E > 3% for m = 100 bp and m = 250 bp, respectively, (as we show earlier in Section 3.2). (4) SneakySnake provides up to 8.9× and 40× more speedup to the end-to-end execution time of Edlib and Parasail compared to SHD. This is expected as SHD produces a high false accept rate (as we show earlier in Section 3.2).

We now examine the benefits of integrating SneakySnake with Parasail using long reads (Set_6). We run SneakySnake and Parasail using 40 CPU threads. We use a wide range of edit distance thresholds, up to 10% of the sequence length. Based on Fig. 4(c), we make two key observations: (1) SneakySnake makes Parasail significantly faster (by up to 5979×, respectively) than Parasail alone in detecting dissimilar pairs of long sequences, even at high edit distance thresholds (up to E = 5010%). This results in building and examining a chip maze of 10,021 rows for each sequence pair. (2) As the number of similar sequence pairs increases at E = 5010%, the benefits of integrating SneakySnake with Parasail in reducing the end-to-end execution time becomes less. As 94% of the input sequence pairs passed SneakySnake to Parasail integrating SneakySnake with Parasail shows almost the same end-to-end execution time of Parasail alone. This is expected, as each sequence pair that passed SneakySnake is examined unnecessarily twice (i.e., once by SneakySnake and once by Parasail). We provide more details on this evaluation for both Set_5 and Set_6 in Supplementary Materials, Section 11.3.

We conclude that SneakySnake is the best-performing CPU-based pre-alignment filter in terms of both speed and accuracy. Integrating SneakySnake with sequence alignment algorithms is always beneficial for both short and long sequences and reduces the end-to-end execution time by up to an order and two orders of magnitude, respectively, without the need for hardware accelerators. We also conclude that SneakySnake’s performance also scales very well over a wide range of edit distance thresholds, number of CPU threads, and sequence lengths.

3.4 Effects of Snake-on-Chip and Snake-on-GPU on Sequence Alignment

We analyze the benefits of integrating Snake-on-Chip and Snake-on-GPU with the state-of-the-art sequence aligners, designed for different computing platforms in Fig. 5. We design the hardware architecture of Snake-on-Chip for a sub-maze’s width of 8 VRTs (t = 8) and 3 replications (p = 3) per each sub-maze. We select this design choice as it allows for low FPGA resource utilization while maintaining low false accept rate, as we analyze the effect of choosing different y and t values on the false accept rate of Snake-on-Chip (these analysis results are reported in the GitHub page). In this analysis, we compare the effect of combining Snake-on-Chip and Snake-on-GPU with existing sequence aligner with that of two state-of-the-art FPGAs-based pre-alignment filters,
Shouji and GateKeeper. We also select four state-of-the-art sequence aligners that are implemented for CPU (Edlib and Parasail), GPU (GSWABE, and FPGASW). We use Set_1 and Set_2 in this analysis. GSWABE and FPGASW are not open-source and not available to us. Therefore, we scale the reported number of compared entries of the DP matrix in a second (i.e., GCUPS) as follows: 60000000/GCUPS/1002.

Based on Fig. 5, we make three key observations. (1) The execution time of Edlib and Parasail reduces by up to 321× (using Set_2 and E = 5%) and 536× (using Set_2 and E = 5%), respectively, after the addition of Snake-on-Chip as a pre-alignment filtering step and by up to 413× (using Set_2 and E = 5%) and 689× (using Set_2 and E = 5%), respectively, after the addition of Snake-on-GPU as a pre-alignment filtering step. That is 40× (321/8) to 51× (689/13.39) more speedup compared to that provided by adding Snake-on-Chip as a pre-alignment filter, using Set_2 and E = 5%. It is also up to 2× more speedup compared to that provided by adding Shouji and GateKeeper as a pre-alignment filter, using Set_1 and E5% for Snake-on-Chip and using Set_2 and E5% for Snake-on-GPU.

(2) FPGAs and GPUs based sequence aligners follow a similar trend that we observe in the CPU implementations. However, the speedup ratios are reduced compared to that observed in the CPU based aligners.

This is due to the low execution time of these hardware accelerated aligners. Snake-on-GPU provides up to 27.7× (using Set_2 and E = 5%) and 5.1× (using Set_2 and E = 5%) reduction in the end-to-end execution time of GSWABE and FPGASW, respectively. Thus is up to 1.3× more speedup compared to that provided by Snake-on-Chip, using Set_2. That is also up to 1.7× more speedup compared to that provided by adding Shouji and GateKeeper as a pre-alignment filter.

We conclude that both Snake-on-Chip and Snake-on-GPU provide the highest speedup ratio (up to two orders of magnitude) compared to the state-of-the-art CPU, FPGA, and GPU based sequence aligners over edit distance thresholds of 0%-5% of the sequence length.

3.5 Effects of SneakySnake on Read Mapping

After confirming the benefits of the different implementations of the SneakySnake algorithm, we now evaluate the overall benefits of integrating SneakySnake with minimap2 (2.17-r974-dirty, 22 January 2020) (Li, 2018). We select minimap2 for two main reasons. (1) It is a state-of-the-art read mapper that includes efficient methods (i.e., minimizers and seed chaining) for accelerating read mapping. (2) It utilizes a banded global sequence alignment algorithm (KSW2, implemented as extz2_sse) that is parallelized and accelerated using the Intel SSE instructions and it uses Z-drop heuristic (Suzuki and Kasahara, 2018) to improve the alignment time. We map all reads from ERR240727_1 (100 bp) to GRCh37 with edit distance thresholds of 0% and 5% of the sequence length. We run minimap2 using --sr mode (short read mapping) and the default parameter values. We replace the seed chaining of minimap2 with SneakySnake. In these experiments, we ensure that we maintain the same reported mappings for both tools. We make two observations. (1) SneakySnake and the minimap2's aligner (KSW2) together are at least 6.83× (from 246 seconds to 36 seconds) and 2.51× (from 338 seconds to 134.67 seconds) faster than the minimap2's seed chaining and the minimap2's aligner together for edit distance thresholds of 0% and 5%, respectively.

(2) The mapping time of minimap2 reduces by a factor of up to 2.01× (from 418 seconds to 208 seconds) and 1.66× (from 510 seconds to 306.67 seconds) after integrating SneakySnake with minimap2 for edit distance thresholds of 0% and 5%, respectively. We also evaluate the effect of integrating SneakySnake with KSW2 using long sequences, Set_5 and Set_6, in Supplementary Materials, Section 11.3. We observe that SneakySnake accelerates KSW2 by up to 64.1× (47.1× on average) and 60.6× (15.4× on average) using Set_5 and Set_6, respectively.

We conclude that SneakySnake is still beneficial even for minimap2, a state-of-the-art read mapper, which uses minimizers, seed chaining, and SIMD-accelerated banded alignment. This promising result motivates us to explore in detail accelerating minimap2 using Snake-on-GPU and Snake-on-Chip in our future research.
4 Discussion and Future Work

We introduce the single net routing problem and we show how to convert an approximate string matching problem into an instance of the single net routing problem. Subsequently, we propose a new algorithm that solves the single net routing problem and acts as a new pre-alignment filtering algorithm, which we call SneakySnake. We demonstrate that the concept of pre-alignment filtering provides substantial benefits to the existing and future sequence alignment algorithms and read mappers for both short and long sequences. SneakySnake offers the ability to make the best use of existing aligners without sacrificing any of their capabilities (e.g., configurable scoring and backtracking), as it does not modify or replace the alignment step. Our algorithm supports CPU multithreading very well and it does not exploit any SIMD-enabled CPU instructions or vendor-specific processor. This makes it attractive and cost-effective given a limited resources environment. SneakySnake improves the accuracy of pre-alignment filtering by up to four orders of magnitude compared to the state-of-the-art pre-alignment filters, Shouji, GateKeeper, and SHD. The addition of SneakySnake as a pre-alignment filtering step reduces significantly the execution time of state-of-the-art CPU-based sequence aligners by up to an order and two orders of magnitude using short and long sequences, respectively. We also explore the use of hardware/software co-design and hardware accelerations to further accelerate our SneakySnake algorithm. We introduce Snake-on-Chip and Snake-on-GPU, efficient and scalable FPGA and GPU based pre-alignment filters, respectively. Snake-on-Chip and Snake-on-GPU achieve up to two orders of magnitude speedup to the state-of-the-art sequence aligners. One direction to further improve the performance of Snake-on-Chip is to discover the possibility of performing the SneakySnake calculations where the huge amount of genomic data resides. Conventional computing requires the movement of genomic sequence pairs from the memory to the CPU processing cores (or to the FPGA chip), using slow and energy-hungry buses, such that cores can apply sequence alignment algorithm on the sequence pairs. Performing SneakySnake inside modern memory devices can alleviate this high communication cost by enabling simple arithmetic/logic operations very close to where the data resides, with high bandwidth and low latency. However, this requires re-designing the hardware architecture of Snake-on-Chip to leverage the supported operations in such modern memory devices. This also encourages us to explore the possibility of accelerating sequence alignment algorithms for longer sequences (few tens of thousands of characters) which is mainly limited by 1) the data transfer rate of the communication link (i.e., PCIe) and 2) large edit distance thresholds.

Funding

This work is supported by gifts from Intel [to O.M.]; VMware [to O.M.]; and an EMBO Installation Grant [IG-2521 to C.A.].

References


