


Genome analysis

SneakySnake: a fast and accurate universal genome pre-alignment filter for CPUs, GPUs and FPGAs

Mohammed Alser ^{1,2,*}, Taha Shahroodi¹, Juan Gómez-Luna^{1,2}, Can Alkan^{3,*} and Onur Mutlu^{1,2,3,4,*}

¹Department of Computer Science, ETH Zurich, Zurich 8006, Switzerland, ²Department of Information Technology and Electrical Engineering, ETH Zurich, Zurich 8006, Switzerland, ³Department of Computer Engineering, Bilkent University, Ankara 06800, Turkey and ⁴Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA

*To whom correspondence should be addressed.

Associate Editor: Peter Robinson

Received on March 10, 2020; revised on September 30, 2020; editorial decision on October 21, 2020; accepted on November 24, 2020

Abstract

Motivation: We introduce *SneakySnake*, a highly parallel and highly accurate pre-alignment filter that remarkably reduces the need for computationally costly sequence alignment. 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 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 or not performing sequence alignment is necessary. Reducing the ASM problem into SNR also makes *SneakySnake* efficient to implement on 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. For short sequences, *SneakySnake* accelerates Edlib (state-of-the-art implementation of Myers's bit-vector algorithm) and Parasail (state-of-the-art sequence aligner with a configurable scoring function), by up to 37.7× and 43.9× (>12× on average), respectively, with its CPU implementation, and by up to 413× and 689× (>400× on average), respectively, with FPGA and GPU acceleration. For long sequences, the CPU implementation of *SneakySnake* accelerates Parasail and KSW2 (sequence aligner of minimap2) by up to 979× (276.9× on average) and 91.7× (31.7× on average), respectively. As *SneakySnake* does not replace sequence alignment, users can still obtain *all* capabilities (e.g. configurable scoring functions) of the aligner of their choice, unlike existing acceleration efforts that sacrifice some aligner capabilities.

Availability and implementation: <https://github.com/CMU-SAFARI/SneakySnake>.

Contact: alserm@ethz.ch or calkan@cs.bilkent.edu.tr or omutlu@ethz.ch

Supplementary information: [Supplementary data](#) are available at *Bioinformatics* online.

1 Introduction

One of the most fundamental computational steps in most genomic analyses is *sequence alignment* (Alser *et al.*, 2020b; Senol Cali *et al.*, 2019). This step is formulated as an ASM problem (Navarro, 2001) and it calculates: (i) *edit distance* between two given sequences, (ii) type of each edit (i.e. insertion, deletion or substitution) and (iii) 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 (Firtina *et al.*, 2020) and genetic variations (Consortium *et al.*, 2015). 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 (Alser *et al.*, 2020b; 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, as we comprehensively survey in (Alser *et al.*, 2020a): (i) Accelerating the DP algorithms using hardware accelerators and (ii) 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 (i) 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 (Sošić and Šikić, 2017) is currently the best-performing implementation of Myers's bit-vector algorithm. Other hardware accelerators include (ii) graphics processing units (GPUs), such as GSWABE (Liu and Schmidt, 2015), (iii) field-programmable gate arrays (FPGAs), such as FPGASW (Fei *et al.*, 2018) or (iv) 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 GenASM (Senol Cali *et al.*, 2020). 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 (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: (i) slow and they suffer from a limited sequence length (≤ 128 bp), such as SHD (Xin *et al.*, 2015), or (ii) inaccurate after some edit distance threshold, such as GateKeeper (Alser *et al.*, 2017a) and MAGNET (Alser *et al.*, 2017b). Highly parallel filtering can also be achieved using processing-in-memory architectures, as in GRIM-Filter (Kim *et al.*, 2018). 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 *both short and long 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 a 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 while passing 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. (i) It obviates the need for using computationally costly DP algorithms to keep track of the subpath that provides the optimal solution (i.e. the one with the least possible routing cost). (ii) The independence of 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 (i) correct and optimal in solving the SNR problem and (ii) it runs in linear time with respect to sequence length and 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 knowledge, the *first universal* pre-alignment filter for CPUs, GPUs and FPGAs, by having software as well as software/hardware co-designed versions of SneakySnake.
- We demonstrate, using short sequences, that SneakySnake accelerates Edlib and Parasail by up to 37.7 \times and 43.9 \times ($>12\times$ on average), respectively, with its CPU implementation, and by up to 413 \times and 689 \times ($>400\times$ on average), respectively, with FPGA and GPU acceleration. We also demonstrate, using long sequences, that SneakySnake accelerates Parasail by up to 979 \times (276.9 \times on average).
- We demonstrate that the CPU implementation of SneakySnake accelerates the sequence alignment of minimap2 (Li, 2018), a state-of-the-art read mapper, by up to 6.83 \times and 91.7 \times using short and long sequences, respectively.

2 Materials and methods

2.1 Overview

The primary purpose of SneakySnake is to accelerate sequence alignment calculation by providing fast and accurate pre-alignment filtering. The SneakySnake algorithm quickly examines each sequence pair before applying sequence alignment and decides whether computationally expensive sequence alignment is needed for two genomic sequences. This filtering decision of the SneakySnake algorithm is made based on accurately estimating the number of edits between two given sequences. If two genomic sequences differ 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 while passing through the minimum number of obstacles on a VLSI chip. The number of obstacles faced throughout the found routing path represents a *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.

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 grid graph while respecting constraints. We present an example of a VLSI chip layout in Figure 1. The goal is to find the optimal path—called *signal net*—that connects the source and destination terminals through the chip layout. We describe the special grid graph of the SNR problem and define the 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).

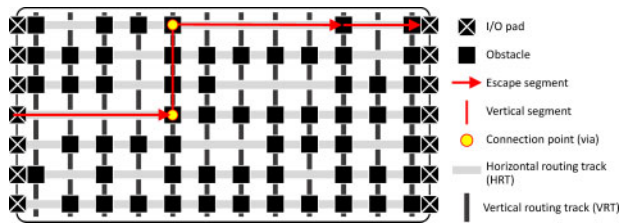


Fig. 1. Chip layout with processing elements and two layers of metal routing tracks. In this example, the chip layout has 7 horizontal routing tracks (HRTs) located on the first layer and another 12 vertical routing tracks (VRTs) located on the second layer. The optimal signal net that is calculated using the SneakySnake algorithm is highlighted in red using three escape segments. The first escape segment is connected to the second escape segment using a VRT through vias. The second escape segment is connected to the third escape segment without passing through a VRT as both escape segments are located on the same HRT. The optimal signal net passes through three obstacles (each of which is located at the end of each escape segment) and hence the signal net has a total delay of $3 \times t_{\text{obstacle}}$

- The horizontal and vertical routing tracks induce a two dimensional uniform grid over the chip layout. Each HRT can be obstructed by some obstacles (e.g. processing elements in the chip). For simplicity, we assume that VRTs cannot be obstructed by obstacles. These obstacles allow the signal to pass horizontally through HRTs, but they induce a signal delay on the passed signal. Each obstacle induces a fixed propagation delay, t_{obstacle} , on the victim signal that passes through the obstacle in the corresponding HRT.
- A signal net often uses a sequence of alternating horizontal and vertical segments that are parts of the routing tracks. Adjacent horizontal and vertical segments in the signal net are connected by an inter-layer *via*. We call a signal net *optimal* if it is both the shortest and the fastest routing path (i.e. passes through the minimum number of obstacles).
- Alternating between horizontal and vertical segments is restricted by passing a single obstacle. Thus, segment alternating strictly delays the signal by t_{obstacle} time.
- The terminals can be any of the I/O pads that are located on the right-hand and left-hand boundaries of the chip layout. The source terminal always lies on the opposite side of the destination terminal.

The general goal of this SNR problem is to find an *optimal* signal net in the grid graph of the chip layout. For the simplicity of developing a solution, we call a horizontal segment that ends with at most an obstacle an *escape segment*. The escape segment can also be a single obstacle only. Also for simplicity, we call the right-hand side of an escape segment a *checkpoint*. Next, we present how we can reduce the ASM problem to the SNR problem.

2.3 Reducing the ASM problem to the 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: (i) replacing the DP table used by the sequence alignment algorithm to a special grid graph called *chip maze* and (ii) 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 in Section 2.4). The chip maze is a $(2E+1) \times m$ grid graph, where E is the edit distance threshold in terms of the number of tolerable character differences, $(2E+1)$ is the number of HRTs, and m is the number of VRTs. The chip maze is an abstract

layout for the VLSI chip layout, as we show in Figure 2b for the same chip layout of Figure 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 $R[1 \dots m]$ and a query sequence $Q[1 \dots m]$, 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:

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

We derive the four cases of Equation 1 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 the corresponding $2E+1$ characters of the query sequence, Q , as we show in Figure 2a. In the first case of Equation 1, we compare the j th character of the reference sequence, R , with the j th character of the query sequence, Q , to detect pairwise matches and substitutions. In the second case of Equation 1, we compare the j th character of the reference sequence with each of the E left-hand neighboring characters of the j th character of the query sequence, to accurately detect deleted characters in the query sequence. In the third case of Equation 1, we compare the j th character of the reference sequence with each of the E right-hand neighboring characters of the j th character of the query sequence, to accurately detect inserted characters in the query sequence. 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, in the second and the third cases of Equation 1, we need to compare a character of the reference sequence with the neighboring characters of its corresponding character of the query sequence to cancel the effect of deletion/insertion and correctly detect the common subsequences between two sequences. In the fourth case of Equation 1, we fill the remaining empty entries of each row with ones (i.e. obstacles) to indicate that there is no match between the corresponding characters. These four cases are essential to accurately detect substituted, deleted and inserted characters in one or both of the sequences. We present in Figure 2b 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 (Seshadri 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 smaller number of escape segments and smaller 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.

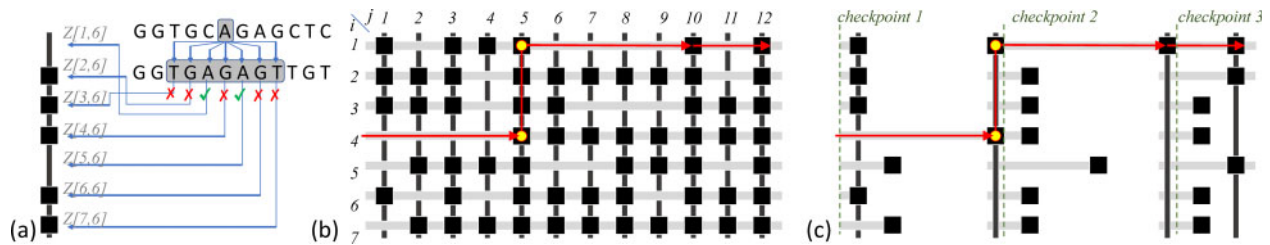


Fig. 2. (a) An example of how we build the 6th column of the chip maze, Z , using Equation 1 for a reference sequence $R = \text{'GGTGCAGAGCTC'}$, a query sequence $Q = \text{'GGTGAGAGTTGT'}$ and an edit distance threshold (E) of 3. The 6th character of R is compared with each of its corresponding $2E + 1$ characters of Q . The order of the results of comparing $R[6]$ with $Q[3]$, $Q[4]$ and $Q[5]$ is reversed to easily derive the second case of Equation 1. (b) The complete chip maze that is calculated using Equation 1, which has $2E + 1$ rows and m (length of Q) columns. (c) The actual chip maze that is calculated using the SneakySnake algorithm. The optimal signal net is highlighted in both chip mazes in red. The signal net has three obstacles (each of which is located at the end of each escape segment) and hence sequence alignment is needed, as the number of differences $\leq E$.

2.4 Solving the SNR problem

The primary purpose of the SneakySnake algorithm is to solve the SNR problem by providing an optimal signal net. Solving the SNR problem requires achieving two key objectives: (i) achieving the lowest possible latency by finding the minimum number of escape segments that are sufficient to link the source terminal to the destination terminal and (ii) achieving the shortest length of the signal net by considering each escape segment just once and in monotonically increasing order of their start index (or end index). The first objective is based on a key observation that a signal net with fewer escape segments always has fewer obstacles, as each escape segment has at most a single obstacle (based on our definition in Section 2.2). This key observation leads to a signal net that has the least possible total propagation delay. The second objective restricts the SneakySnake 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. (i) The SneakySnake algorithm first constructs the chip maze using Equation 1. It then considers the first column of the chip maze as the first checkpoint, where the first iteration starts. (ii) 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. (iii) It creates a new checkpoint after the longest escape segment. (iv) 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. $E \times t_{\text{obstacle}}$). When the two input sequences are different in length, we need to count the number of obstacles more conservatively along the signal net. Doing so ensures a correct reduction of the ASM problem. This means that we need to deduct the total number of leading and trailing obstacles from the total count of edits between two input sequences before making the filtering decision, as such obstacles can be caused by the fourth case of Equation 1. (v) If SneakySnake finds the optimal net using the previous steps, then it indicates that the edit distance between two input sequences is $\leq E$. If so, sequence alignment is needed to know the exact number of edits, type of each edit and location of each edit between the two sequences using user's favorite sequence alignment algorithm. Otherwise, the SneakySnake algorithm terminates without performing computationally expensive sequence alignment, since the differences between sequences is guaranteed to be $> E$.

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 Figure 2c. 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 Section S5.

The SneakySnake algorithm is both correct and optimal in solving the SNR problem. 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 is guaranteed to find a signal net that interconnects the source terminal and the destination terminal when one exists.*

We provide the correctness proof for Supplementary Theorem S1 in Supplementary Section S6.1. The SneakySnake algorithm is also optimal as it is guaranteed 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 Supplementary Theorem S2 in Supplementary Section S6.2.

Different from existing sequence alignment algorithms that are based on DP approaches (Daily, 2016; Xin et al., 2013) or sparse DP (i.e. chaining exact matches between two sequences using DP algorithms) approaches (Chaisson and Tesler, 2012), SneakySnake (i) does not require knowing the location and the length of common subsequences between the two input sequences in advance, (ii) does not consider the vertical distance (i.e. the number of rows) between two escape segments in the calculation of the minimum number of edits and (iii) does not build the entire dynamic programming table; SneakySnake builds only a minimal portion of the chip maze that is needed to provide an optimal solution. The first difference makes SneakySnake independent of any algorithm that aims to calculate sequence alignment, as SneakySnake quickly and efficiently calculates its own data structure (i.e. chip maze) to find all common subsequences. The second difference helps to construct a data dependency-free chip maze and allows for

solving many SNR subproblems in parallel as calculating the routing path after facing an obstacle is independent of the calculated path before this obstacle. The third difference significantly reduces the number of computations needed for the SneakySnake algorithm.

Different from existing edit distance approximation algorithms (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. $O(m^{1.647})$ instead of $O(m^2)$), SneakySnake does not overestimate the edit distance as the calculated optimal signal net has *always* the minimum possible number of obstacles (Theorem 2). We take advantage of the edit distance underestimation of SneakySnake by using our fast computation method as a pre-alignment filter. Doing so ensures two key properties: (i) allows sequence alignment to be calculated only for similar (or nearly similar) sequences and (ii) accelerates the sequence alignment algorithms without changing (or replacing) their algorithmic method and hence preserving all the capabilities of the sequence alignment algorithms.

We next discuss further optimizations and new software/hardware co-designed versions of the SneakySnake algorithm that can 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. (i) Downsizing the search space into a reasonably small grid graph with a known dimension at 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. (ii) 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. (iii) All the smaller subproblems can be solved independently and rapidly with 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 SneakySnake algorithm determines the source and destination terminals for each SNR subproblem based on the optimal signal net of each SNR subproblem. This leads to underestimation of the total number of obstacles found along each signal net of each SNR subproblem. This is still acceptable as long as the SneakySnake algorithm solves the SNR problem quickly and *without overestimating* the number of obstacles compared to the edit distance threshold. We provide the details of our hardware architecture of Snake-on-Chip in [Supplementary Section S8](#).

2.6 Snake-on-GPU parallel implementation

We 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.

Given the large size of the sequence pair dataset that the GPU threads need to access, we carefully design Snake-on-GPU to efficiently (i) 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 (ii) 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 Section S9](#).

3 Results

We evaluate (i) filtering accuracy, (ii) filtering time and (iii) benefits of combining our universal implementation of the SneakySnake algorithm with state-of-the-art aligners. We provide a comprehensive treatment of all evaluation results in the [Supplementary Excel File](#) and on the SneakySnake GitHub page. We compare the performance of SneakySnake, Snake-on-Chip and Snake-on-GPU to four pre-alignment filters, Shouji (Alser et al., 2019), MAGNET (Alser et al., 2017b), GateKeeper (Alser et al., 2017a) and SHD (Xin et al., 2015). We run the experiments that use multithreading and long sequences on a 2.3 GHz Intel Xeon Gold 5118 CPU with up to 48 threads and 192 GB RAM. We run all other experiments on 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 (Shouji, MAGNET and GateKeeper). We build the FPGA design using Vivado 2015.4 in synthesizable Verilog. We use an NVIDIA GeForce RTX 2080Ti card (NVIDIA, 2019b) with a global memory of 11 GB GDDR6 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 Evaluated datasets

Our experimental evaluation uses 4 different real datasets (100bp_1, 100bp_2, 250bp_1 and 250bp_2) and 2 simulated datasets (10Kbp and 100Kbp). Each real dataset contains 30 million real sequence pairs (text and query pairs). 100bp_1 and 100bp_2 have sequences of length 100 bp, while 250bp_1 and 250bp_2 have sequences of length 250 bp. We generate the 10Kbp dataset to have 100 000 sequence pairs, each of which is 10Kbp long, while the 100Kbp dataset has 74 687 sequence pairs, each of which is 100 Kbp long. [Supplementary Section S10.1](#) provides the details of these datasets.

3.2 Filtering accuracy

We evaluate the accuracy of a pre-alignment filter by computing its rate of falsely accepted and falsely rejected sequences before performing 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 to maintain the correctness of the genome analysis pipeline

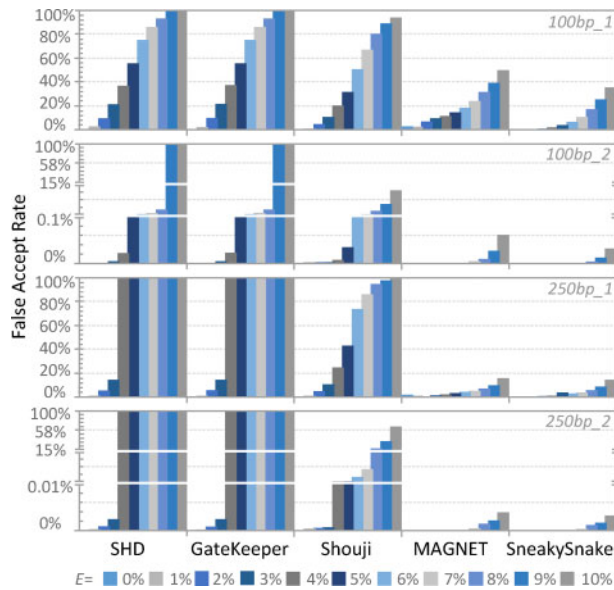


Fig. 3. False accept rates of SHD, GateKeeper, Shouji, MAGNET and SneakySnake across four real datasets of short sequences. We use a wide range of edit distance thresholds (0–10% of the sequence length) for sequence lengths of 100 and 250 bp

and an *as-small-as-possible* false accept rate to maximize the number of dissimilar sequences that are eliminated at low performance overhead.

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 Figure 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 Figure 3, we make four key observations. (i) SneakySnake provides the lowest false accept rate compared to all the four state-of-the-art pre-alignment filters. SneakySnake provides up to 31412 \times , 20603 \times and 64.1 \times less number of falsely accepted sequences compared to GateKeeper/SHD (using 250bp₂, $E = 10\%$), Shouji (using 250bp₂, $E = 10\%$) and MAGNET (using 100bp₁, $E = 1\%$), respectively. (ii) MAGNET provides the second lowest false accept rate. It provides up to 25552 \times and 16760 \times less number of falsely accepted sequences compared to GateKeeper/SHD (using 250bp₂, $E = 10\%$) and Shouji (using 250bp₂, $E = 10\%$), respectively. (iii) All five pre-alignment filters are less accurate in examining 100bp₁ and 250bp₁ than the other datasets, 100bp₂ and 250bp₂. This is expected as the actual number of edits of most of the sequence pairs in 100bp₁ and 250bp₁ datasets is very close to the edit distance threshold (Supplementary Table S4) and hence any underestimation in calculating the edit distance can lead to falsely accepted sequence pairs (i.e. estimated edit distance $\leq E$). (iv) GateKeeper and SHD become ineffective for edit distance thresholds of greater than 8% and 3% for sequence lengths of 100 and 250 characters, respectively, as they accept all the input sequence pairs. This causes a read mapper using them to examine each sequence pair unnecessarily twice (i.e. once by GateKeeper or SHD and once by the sequence alignment algorithm).

Second, we find that SneakySnake has a 0% false reject rate (not plotted). This observation is in accord with our theoretical proof of Theorem 2. 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.

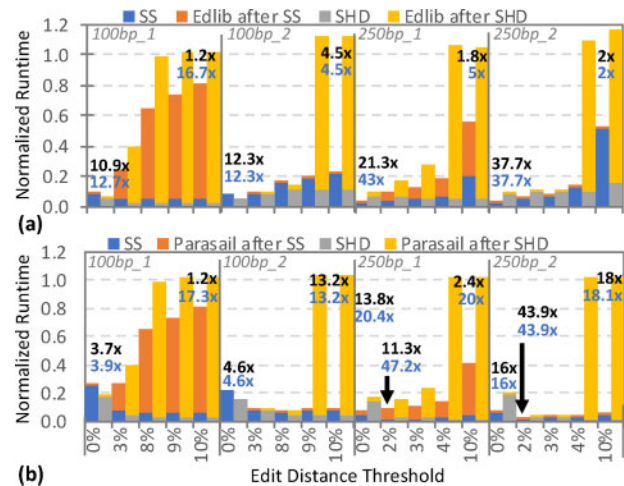


Fig. 4. Normalized end-to-end execution time of SneakySnake and SHD, each combined with (a) Edlib and (b) Parasail. The execution time values in (a) and (b) are normalized to that of Edlib and Parasail, respectively, without pre-alignment filtering. We use four datasets over a wide range of edit distance thresholds ($E = 0$ –10% of the sequence length) for sequence lengths (m) of 100 bp (100bp₁ and 100bp₂) and 250 bp (250bp₁ and 250bp₂). We present two speedup values for $E = 0\%$ and $E = 10\%$ of each dataset and some other E values highlighted by arrows. The top speedup value (in black) represents the end-to-end speedup that is gained from combining the pre-alignment filtering step with the alignment step. It is calculated as $A/(B + C)$, where A is the execution time of the sequence aligner before adding SneakySnake (not plotted in graphs), B is the execution time of SneakySnake and C is the execution time of the sequence aligner after adding SneakySnake. The bottom speedup value (in blue) is calculated as A/B

3.3 Effect of SneakySnake on short 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. Figure 4a 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. (i) The addition of SneakySnake as a pre-alignment filtering step significantly reduces the execution time of Edlib and Parasail by up to 37.7 \times (using 250bp₂, $E = 0\%$) and 43.9 \times (using 250bp₂, $E = 2\%$), respectively. We also observe a similar trend as the number of CPU threads increases from 1 to 40, as we show in Supplementary Section S10.2. To explore the reason for this significant speedup, we need to check how fast SneakySnake examines the sequence pairs compared to sequence alignment, which we observe next. (ii) SneakySnake is up to 43 \times (using 250bp₁, $E = 0\%$) and 47.2 \times (using 250bp₁, $E = 2\%$) faster than Edlib and Parasail, respectively, in examining the sequence pairs. (iii) SneakySnake provides up to 8.9 \times and 40 \times 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). (iv) 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 \times (using 100bp₂, $E = 0\%$) and 34.9 \times (using 250bp₂, $E = 3\%$), 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).

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 short sequences and reduces the end-to-end execution time by up to an order of magnitude without the need for hardware accelerators. We also conclude that SneakySnake's performance scales well over a wide range of edit distance thresholds, number of CPU threads and sequence lengths.

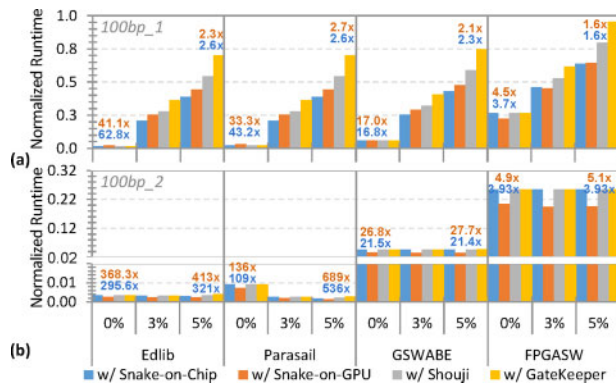


Fig. 5. Normalized end-to-end execution time of a pre-alignment filter (Snake-on-Chip, Snake-on-GPU, Shouji and GateKeeper) combined with a sequence aligner (Edlib, Parasail, GSWABE and FPGASW). Each execution time value is normalized to that of the corresponding sequence aligner without pre-alignment filtering. We use two datasets, (a) 100bp_1 and (b) 100bp_2, over a wide range of edit distance thresholds (0%–10% of the sequence length, 100 bp). We present two end-to-end speedup values for edit distance thresholds of 0% and 5%. The top speedup value (in orange) is the speedup gained from integrating Snake-on-GPU with the corresponding sequence aligner. The bottom speedup value (in blue) represents the speedup gained from integrating Snake-on-Chip with the corresponding sequence aligner

3.4 Effect 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 Figure 5. We compare the effect of combining Snake-on-Chip and Snake-on-GPU with an existing sequence aligner to that of two state-of-the-art FPGA-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 FPGA (FPGASW). We use 100bp_1 and 100bp_2 in this evaluation, as GSWABE, Shouji and GateKeeper work for only short sequences. GSWABE and FPGASW are not open-source and not available to us. Therefore, we scale their reported number of computed entries of the DP matrix per second (i.e. GCUPS) as follows: (number of sequence pairs in 100bp_1 or 100bp_2)/(GCUPS/100²). We design the hardware architecture of Snake-on-Chip for a sub-maze’s width of eight VRTs ($t=8$) and three module instances ($y=3$) per each sub-maze. We select this design choice as it allows for low FPGA resource utilization while maintaining a low false accept rate, based on our analysis of different y and t values on the false accept rate of Snake-on-Chip (these results are reported in the [Supplementary Excel File](#) and on the SnakeySnake GitHub page).

Based on Figure 5, we make two key observations. (i) The execution time of Edlib and Parasail reduces by up to 321 \times (using 100bp_2 and $E=5\%$) and 536 \times (using 100bp_2 and $E=5\%$), respectively, after the addition of Snake-on-Chip as a pre-alignment filtering step and by up to 413 \times (using 100bp_2 and $E=5\%$) and 689 \times (using 100bp_2 and $E=5\%$), respectively, after the addition of Snake-on-GPU as a pre-alignment filtering step. That is 40 \times (321/8) to 51 \times (689/13.39) more speedup than that provided by adding SnakeySnake as a pre-alignment filter, using 100bp_2 and $E=5\%$. It is also up to 2 \times more speedup compared to that provided by adding Shouji and GateKeeper as a pre-alignment filter, using 100bp_1 and $E=5\%$ for Snake-on-Chip and using 100bp_2 and $E=5\%$ for Snake-on-GPU. (ii) Snake-on-GPU provides up to 27.7 \times (using 100bp_2 and $E=5\%$) and 5.1 \times (using 100bp_2 and $E=5\%$) reduction in the end-to-end execution time of GSWABE and FPGASW, respectively. This is up to 1.3 \times more speedup than that provided by Snake-on-Chip, using 100bp_2. That is also up to 1.7 \times more speedup than that provided by adding Shouji and GateKeeper as a pre-alignment filter. The speedup provided by Snake-on-GPU and Snake-on-Chip to GSWABE and FPGASW is

Table 1. The end-to-end execution time (in seconds) of SneakySnake integrated with Parasail (40 CPU threads) and KSW2 (single threaded) using long reads (100 Kbp)

| E | Parasail | SS+Parasail | KSW2 | SS+KSW2 | SS accept rate |
|-------|----------|-------------|----------|----------|----------------|
| 0.01% | 84.0 | 0.23 | 1380.2 | 15.1 | 0% |
| 0.3% | 2756.3 | 2.8 | 8215.5 | 135.4 | 0% |
| 5.0% | 37492.3 | 736.5 | 100178.3 | 26261.4 | 0% |
| 10.7% | 81881.6 | 49322.1 | 204135.3 | 184312.5 | 57% |
| 10.8% | 82646.1 | 63756.0 | 206041.4 | 225815.2 | 73% |
| 11.0% | 84098.7 | 83437.5 | 209662.8 | 287206.8 | 94% |
| 12.0% | 91744.1 | 95533.6 | 228723.1 | 325966.0 | 100% |
| 20.0% | 152906.8 | 157982.0 | 381205.1 | 544282.1 | 100% |

less than that observed in Edlib and Parasail. This is due to the low execution time of hardware accelerated aligners.

We conclude that both Snake-on-Chip and Snake-on-GPU provide the highest speedup (up to two orders of magnitude) when combined with 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 Effect of SneakySnake on long sequence alignment

We examine the benefits of integrating SneakySnake with Parasail (Daily, 2016) and KSW2 (Li, 2018; Suzuki and Kasahara, 2018) for long sequence alignment (100Kbp). We run Parasail as *nw_banded*. We run KSW2 as *extz2_sse*, a global alignment implementation that is parallelized using the Intel SSE instructions. KSW2 uses heuristics (Suzuki and Kasahara, 2018) to improve the alignment time. We run SneakySnake with Parasail using 40 CPU threads. We run SneakySnake with KSW2 using a single CPU thread (as KSW2 does not support multithreading). We use a wide range of edit distance thresholds, up to 20% of the sequence length.

Based on Table 1, we make two key observations. (i) SneakySnake accelerates Parasail and KSW2 by 50.9–979 \times and 3.8–91.7 \times , respectively, even at high edit distance thresholds (up to $E=5010$ (5%), which results in building and examining a chip maze of 10021 rows for each sequence pair). (ii) As the number of similar sequence pairs increases, the performance benefit of integrating SneakySnake with Parasail and KSW2 in reducing the end-to-end execution time reduces. When Parasail and KSW2 examine 94% and 73% of the input sequence pairs (SneakySnake filters out the rest of the sequence pairs), respectively, SneakySnake provides slight or no performance benefit to the end-to-end execution time of the sequence aligner alone. This is expected, as each sequence pair that passes SneakySnake is examined unnecessarily twice (i.e. once by SneakySnake and once by sequence aligner). We provide more details on this evaluation for both 10Kbp and 100Kbp in [Supplementary Section S10.3](#). We observe that SneakySnake accelerates Parasail and KSW2 by 276.9 \times and 31.7 \times on average, respectively, when sequence alignment examines at most 73% of the input sequence pairs.

We conclude that when SneakySnake filters out more than 27% of the input sequence pairs, integrating SneakySnake with long sequence aligners is always beneficial and sometimes reduces the end-to-end execution time by one to two orders of magnitude (depending on the edit distance threshold and how fast the sequence aligner examines the input sequence pairs compared to SneakySnake) without the need for hardware accelerators.

3.6 Effect of SneakySnake on read mapping

After confirming the benefits of the different implementations of the SneakySnake algorithm, we 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. (i) It is a state-of-the-art read mapper that includes efficient methods (i.e. minimizers and seed chaining) for accelerating read mapping. (ii) It utilizes a banded global sequence alignment algorithm (KSW2,

implemented as *extz2_sse*) that is parallelized and accelerated using both the Intel SSE instructions and heuristics (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. (i) SneakySnake and the minimap2's aligner (KSW2) together are at least $6.83\times$ (from 246 to 36 s) and $2.51\times$ (from 338 to 134.67 s) faster than the minimap2's seed chaining and the minimap2's aligner together for edit distance thresholds of 0% and 5%, respectively. (ii) The mapping time of minimap2 reduces by a factor of up to $2.01\times$ (from 418 to 208 s) and $1.66\times$ (from 510 to 306.67 s) after integrating SneakySnake with minimap2 for edit distance thresholds of 0% and 5%, respectively.

We conclude that SneakySnake is very 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 demonstrate that we can convert the approximate string matching problem into an instance of the single net routing problem. We show how to do so and propose a new algorithm that solves the single net routing problem and acts as a new pre-alignment filtering algorithm, called SneakySnake. SneakySnake offers the ability to make the best use of existing aligners without sacrificing any of their capabilities (e.g. configurable scoring functions and backtracking), as it does not modify or replace the alignment step. SneakySnake improves the accuracy of pre-alignment filtering by up to four orders of magnitude compared to three state-of-the-art pre-alignment filters, Shouji, GateKeeper and SHD. The addition of SneakySnake as a pre-alignment filtering step significantly reduces 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 introduce Snake-on-Chip and Snake-on-GPU, efficient and scalable FPGA and GPU based hardware accelerators of SneakySnake, respectively. Snake-on-Chip and Snake-on-GPU achieve up to one order and two orders of magnitude speedup over state-of-the-art CPU- and hardware-based sequence aligners, respectively.

One direction to further improve the performance of Snake-on-Chip is to discover the possibility of performing the SneakySnake calculations near where huge amounts of genomic data resides. Conventional computing requires the movement of genomic sequence pairs from the memory to the CPU processing cores (or to the GPU or FPGA chips), using slow and energy-hungry buses, such that cores can apply sequence alignment algorithm on the sequence pairs. Performing SneakySnake inside modern memory devices via processing in memory (Ghose *et al.*, 2019; Mutlu *et al.*, 2019) can alleviate this high communication cost by enabling simple arithmetic/logic operations very close to where the data resides, with high bandwidth, low latency and low energy. However, this requires re-designing the hardware architecture of Snake-on-Chip to leverage the supported operations in such modern memory devices.

Funding

This work was supported by gifts from Intel [to O.M.]; VMware [to O.M.]; a Semiconductor Research Corporation grant [to O.M.]; and an EMBO Installation Grant [IG-2521 to C.A.].

Conflict of Interest: none declared.

References

- Alser, M. *et al.* (2017a) GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping. *Bioinformatics*, **33**, 3355–3363.
- Alser, M. *et al.* (2017b) MAGNET: understanding and improving the accuracy of genome pre-alignment filtering. *Trans. Internet Res.*, **13**, 33–42.
- Alser, M. *et al.* (2019) Shouji: a fast and efficient pre-alignment filter for sequence alignment. *Bioinformatics*, **35**, 4255–4263.
- Alser, M. *et al.* (2020a) Accelerating genome analysis: a primer on an ongoing journey. *IEEE Micro*, **40**, 65–75.
- Alser, M. *et al.* (2020b) Technology dictates algorithms: recent developments in read alignment. *arXiv Preprint arXiv:2003.00110*.
- Chaisson, M.J. and Tesler, G. (2012) Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. *BMC Bioinformatics*, **13**, 238.
- Chakraborty, D. *et al.* (2018) Approximating edit distance within constant factor in truly sub-quadratic time. *J. ACM.*, **67**, 1–22.
- Charikar, M. *et al.* (2018) On estimating edit distance: alignment, dimension reduction, and embeddings. In: *45th International Colloquium on Automata, Languages, and Programming (ICALP)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, vol. 107, p. 34.
- Chen, P. *et al.* (2014) Accelerating the next generation long read mapping with the FPGA-based system. *IEEE/ACM Trans. Comput. Biol. Bioinf.*, **11**, 840–852.
- Consortium, G.P. *et al.* (2015) A global reference for human genetic variation. *Nature*, **526**, 68–74.
- Daily, J. (2016) Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics*, **17**, 81.
- Eddy, S.R. (2004) What is dynamic programming? *Nat. Biotechnol.*, **22**, 909–910.
- Fei, X. *et al.* (2018) FPGASW: accelerating large-scale smith–waterman sequence alignment application with backtracking on FPGA linear systolic array. *Interdiscip. Sci. Comput. Life Sci.*, **10**, 176–188.
- Firtina, C. *et al.* (2020) Apollo: a sequencing-technology-independent, scalable and accurate assembly polishing algorithm. *Bioinformatics*, **36**, 3669–3679.
- Ghose, S. *et al.* (2019) Processing-in-memory: a workload-driven perspective. *IBM J. Res. Dev.*, **63**, 3:1.
- 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.
- Lee, J. *et al.* (1976) Use of Steiner's problem in suboptimal routing in rectilinear metric. *IEEE Trans. Circuits Syst.*, **23**, 470–476.
- Levenshtein, V.I. (1966) Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics-Doklady*, **10**, 707–710.
- Li, H. (2018) Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, **34**, 3094–3100.
- Liu, Y. and Schmidt, B. (2015) GSWABE: faster GPU-accelerated sequence alignment with optimal alignment retrieval for short DNA sequences. *Concurr. Comput. Pract. Exp.*, **27**, 958–972.
- Mutlu, O. *et al.* (2019) Processing data where it makes sense: enabling in-memory computation. *Microproc. Microsyst.*, **67**, 28–41.
- Myers, G. (1999) A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, **46**, 395–415.
- Navarro, G. (2001) A guided tour to approximate string matching. *ACM Comput. Surv. (CSUR)*, **33**, 31–88.
- Needleman, S.B. and Wunsch, C.D. (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, **48**, 443–453.
- NVIDIA (2019a) CUDA C Programming Guide. NVIDIA Corp. https://docs.nvidia.com/cuda/archive/10.1/pdf/CUDA_C_Programming_Guide.pdf.
- NVIDIA (2019b) NVIDIA GeForce RTX 2080 Ti User Guide. NVIDIA Corp. https://www.nvidia.com/content/geforce-gtx/GEFORCE_RTX_2080Ti_User_Guide.pdf.
- Senol Cali, D. *et al.* (2019) Nanopore sequencing technology and tools for genome assembly: computational analysis of the current state, bottlenecks and future directions. *Brief. Bioinf.*, **20**, 1542–1559.
- Senol Cali, D. *et al.* (2020) GenASM: a high performance, low-power approximate string matching acceleration framework for genome sequence analysis. In: *MICRO*.
- Seshadri, V. *et al.* (2017) In-memory accelerator for bulk bitwise operations using commodity DRAM technology. IN 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, 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.
- Suzuki,H. and Kasahara,M. (2018) Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC Bioinformatics*, **19**, 33–47.
- Wang,C. et al. (2011) Comparison of linear gap penalties and profile-based variable gap penalties in profile–profile alignments. *Comput. Biol. Chem.*, **35**, 308–318.
- Xilinx (2013) Virtex-7 XT VC709 Connectivity Kit. Getting Started. 2013. Citeseer. https://www.xilinx.com/support/documentation/boards_and_kits/vc709/2013_2/ug962-v7-vc709-xt-connectivity-trd-ug.pdf.
- Xin,H. et al. (2013) Accelerating read mapping with FastHASH. *BMC Genomics*, **14**, S13.
- 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.