Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Manuscript Category

OXFORD

Subject Section

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

Mohammed Alser [1,2,*], Taha Shahroodi [1], Juan Gómez-Luna [1,2], Can Alkan [4,*], and Onur Mutlu [1,2,3,4,*]

[1]Department of Computer Science, ETH Zurich, Zurich 8006, Switzerland
[2]Department of Information Technology and Electrical Engineering, ETH Zurich, Zurich 8006, Switzerland
[3]Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh 15213, PA, USA
[4]Department 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 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: https://github.com/CMU-SAFARI/SneakySnake
Contact: alserm@inf.ethz.ch, calkan@cs.bilkent.edu.tr, omutlu@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* (Alser *et al.*, 2020b; Senol Cali *et al.*, 2019). 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 (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): (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), 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 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: 1) slow and they suffer from a limited sequence length ($\leq 128bp$), 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). 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. 1) 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). 2) 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 1) correct and optimal in solving the SNR problem and 2) 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× 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. We also demonstrate, using long sequences, that SneakySnake accelerates Parasail by up to 979× (276.9× 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× and 91.7× using short and long sequences, respectively.

2 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 Fig. 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:

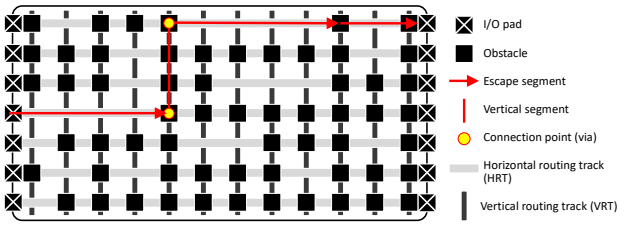


Fig. 1. Chip layout with processing elements and two layers of metal routing tracks. In this example, the chip layout has 7 horizontal routing tracks (HRTs) located on the first layer and another 12 vertical routing tracks (VRTs) located on the second layer. The optimal signal net that is calculated 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 $3 \times t_{obstacle}$.

- 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).
- 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 can not 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 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 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 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 $R[1 \dots m]$ 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:

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

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 Fig. 2(a). 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 'GE**N**OME' shifts the last three characters to the left direction, making it 'GE**OME**'). 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 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 (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.

2.4 Solving the Single Net Routing 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: 1) achieving the lowest possible latency by finding the minimum number of escape segments that are

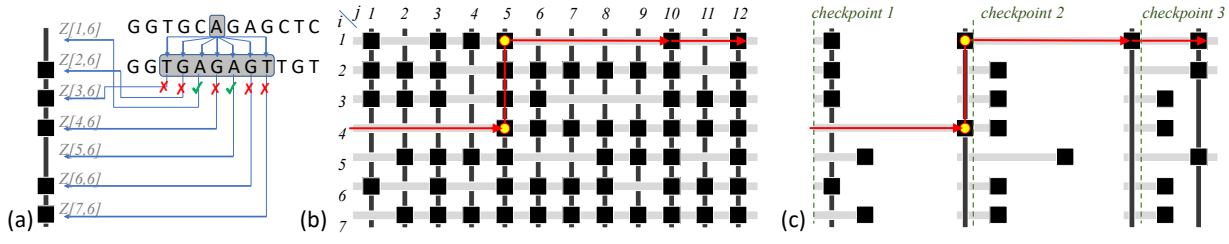


Fig. 2. (a) An example of how we build the 6^{th} column of the chip maze, Z, using Equation 1 for a reference sequence R = 'GGTGCAGAGCTC', a query sequence Q = 'GGTGAGAGTTGT', and an edit distance threshold (E) of 3. The 6^{th} character of R is compared with each of its corresponding $2E + 1$ characters 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 3 obstacles (each of which is located at the end of each escape segment) and hence sequence alignment is needed, as the number of differences $\leq E$.

sufficient to link the source terminal to the destination terminal and 2) 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. (1) 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. (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., $E \times t_{obstacle}$). 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. (5) 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 favourite 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 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 5.

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 Theorem 1 in Supplementary Materials, Section 6.1. 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 Theorem 2 in Supplementary Materials, Section 6.2.

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 1) does not require knowing the location and the length of common subsequences between the two input sequences in advance, 2) 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 3) 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: (1)

allows sequence alignment to be calculated only for similar (or nearly similar) sequences and (2) 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. (1) 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. (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 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 Materials, Section 8.

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 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 9.

3 Results

We evaluate 1) filtering accuracy, 2) filtering time, and 3) 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 10 Kbp long, while the `100Kbp` dataset has 74,687 sequence pairs, each of which is 100 Kbp long. Supplementary Materials, Section 10.1 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 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 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) 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 `250bp_2`, E= 10%), Shouji (using `250bp_2`, E= 10%), and MAGNET (using `100bp_1`, E= 1%), respectively. (2) MAGNET provides the second lowest false accept rate. It provides up to 25552× and 16760× less number of falsely-accepted sequences compared to GateKeeper/SHD (using `250bp_2`, E= 10%) and Shouji

(using 250bp_2, E= 10%), respectively. (3) All five pre-alignment filters are less accurate in examining 100bp_1 and 250bp_1 than the other datasets, 100bp_2 and 250bp_2. This is expected as the actual number of edits of most of the sequence pairs in 100bp_1 and 250bp_1 datasets is very close to the edit distance threshold (Supplementary Materials, Table 4) and hence any underestimation in calculating the edit distance can lead to falsely-accepted sequence pairs (i.e., estimated edit distance $\leq$ E). (4) 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.

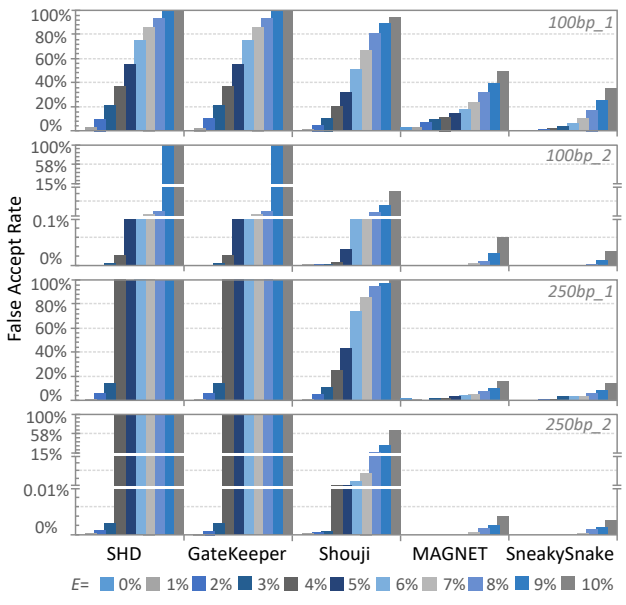


Fig. 3. False accept rates of SHD, GateKeeper, Shouji, MAGNET, and SneakySnake across 4 real datasets of short sequences. We use a wide range of edit distance thresholds ($0\% - 10\%$ of the sequence length) for sequence lengths of 100 and 250 bp.

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. 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) The addition of SneakySnake as a pre-alignment filtering step significantly reduces the execution time of Edlib and Parasail by up to 37.7× (using 250bp_2, E= 0%) and 43.9× (using 250bp_2, 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 Materials, Section 10.2. 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. (2) SneakySnake is up to 43× (using 250bp_1, E= 0%) and 47.2× (using 250bp_1, E= 2%) faster than Edlib and Parasail, respectively, in examining the sequence pairs. (3) 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). (4) 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 100bp_2, E = 0%) and 34.9× (using 250bp_2, 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.

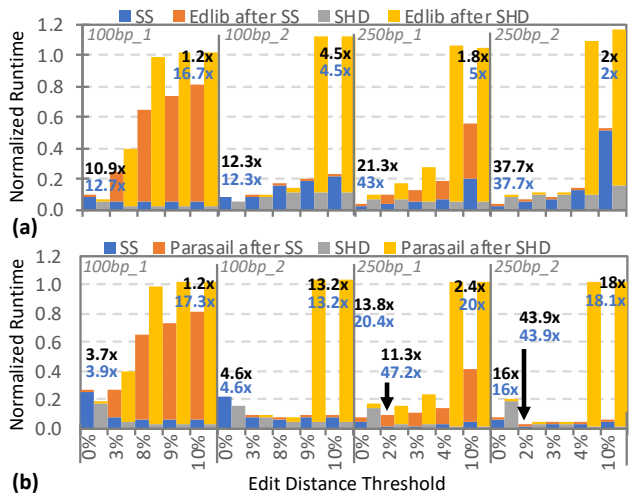


Fig. 4. Normalized end-to-end execution time of SneakySnake and SHD, each combined with (a) Edlib and (b) Parasail. The execution time values in (a) and (b) are normalized to that of Edlib and Parasail, respectively, without pre-alignment filtering. We use four datasets over a wide range of edit distance thresholds (E= 0%-10% of the sequence length) for sequence lengths (m) of 100 bp (100bp_1 and 100bp_2) and 250 bp (250bp_1 and 250bp_2). 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.

3.4 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 Fig. 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/100^2). We design the hardware architecture of Snake-on-Chip for a sub-maze's width of 8 VRTs (t=8) and 3 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 SneakySnake GitHub page).

Based on Fig. 5, we make two key observations. (1) The execution time of Edlib and Parasail reduces by up to 321× (using `100bp_2` and E = 5%) and 536× (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× (using `100bp_2` and E = 5%) and 689× (using `100bp_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 than that provided by adding SneakySnake as a pre-alignment filter, using `100bp_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 `100bp_1` and E=5% for Snake-on-Chip and using `100bp_2` and E=5% for Snake-on-GPU. (2) Snake-on-GPU provides up to 27.7× (using `100bp_2` and E = 5%) and 5.1× (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× more speedup than that provided by Snake-on-Chip, using `100bp_2`. That is also up to 1.7× 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 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.

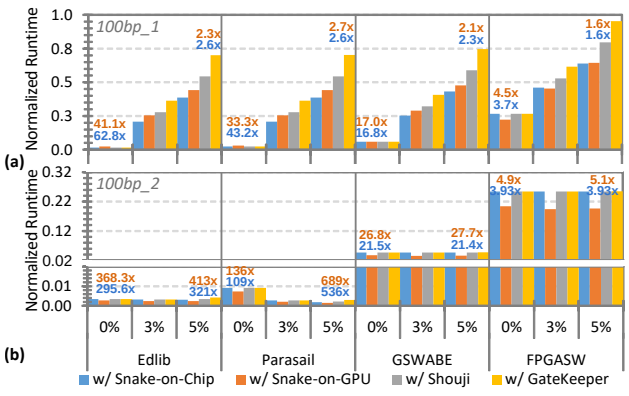


Fig. 5. Normalized end-to-end execution time of a pre-alignment filter (Snake-on-Chip, Snake-on-GPU, Shouji, and GateKeeper) combined with a sequence aligner (Edlib, Parasail, GSWABE, and FPGASW). Each execution time value is normalized to that of the corresponding sequence aligner without pre-alignment 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.

3.5 Effect of SneakySnake on Long Sequence Alignment

We examine the benefits of integrating SneakySnake with Parasail (Daily, 2016) and KSW2 (Suzuki and Kasahara, 2018; Li, 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. (1) SneakySnake accelerates Parasail and KSW2 by 50.9-979× and 3.8-91.7×, respectively, even at high edit distance thresholds (up to E=5010 (5%), which 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, 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 Materials, Section 10.3. We observe that SneakySnake accelerates Parasail and KSW2 by 276.9× and 31.7× 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.

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 (`100Kbp`).

E	Parasail	SS+Parasail	KSW2	SS+KSW2	SS Accept Rate
0.01%	84.0	**0.23**	1380.2	**15.1**	0%
0.3%	2,756.3	**2.8**	8,215.5	**135.4**	0%
5.0%	37,492.3	**736.5**	100,178.3	**26,261.4**	0%
10.7%	81,881.6	**49,322.1**	204,135.3	**184,312.5**	57%
10.8%	82,646.1	**63,756.0**	206,041.4	225,815.2	73%
11.0%	84,098.7	**83,437.5**	209,662.8	287,206.8	94%
12.0%	91,744.1	95,533.6	228,723.1	325,966.0	100%
20.0%	152,906.8	157,982.0	381,205.1	544,282.1	100%

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. (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 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. (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 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 (Mutlu *et al.*, 2019; Ghose *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 is 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.].

References

Alser, M., Hassan, H., Xin, H., Ergin, O., Mutlu, O., and Alkan, C. (2017a). GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping. *Bioinformatics*, **33**(21), 3355–3363.

Alser, M., Mutlu, O., and Alkan, C. (2017b). MAGNET: Understanding and improving the accuracy of genome pre-alignment filtering. *Transactions on Internet Research*, **13**(2), 33–42.

Alser, M., Hassan, H., Kumar, A., Mutlu, O., and Alkan, C. (2019). Shouji: a fast and efficient pre-alignment filter for sequence alignment. *Bioinformatics*, **35**(21), 4255–4263.

Alser, M., Bingöl, Z., Cali, D. S., Kim, J., Ghose, S., Alkan, C., and Mutlu, O. (2020a). Accelerating Genome Analysis: A Primer on an Ongoing Journey. *IEEE Micro*, **40**(5), 65–75.

Alser, M., Rotman, J., Taraszka, K., Shi, H., Baykal, P. I., Yang, H. T., Xue, V., Knyazev, S., Singer, B. D., Balliu, B., *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**(1), 238.

Chakraborty, D., Das, D., Goldenberg, E., Koucky, M., and Saks, M. (2018). Approximating edit distance within constant factor in truly sub-quadratic time. In *IEEE Annual Symp. on Foundations of Computer Science (FOCS)*, pages 979–990.

Charikar, M., Geri, O., Kim, M. P., and Kuszmaul, W. (2018). On Estimating Edit Distance: Alignment, Dimension Reduction, and Embeddings. In *45th International Colloquium on Automata, Languages, and Programming (ICALP)*.

Chen, P., Wang, C., Li, X., and Zhou, X. (2014). Accelerating the next generation long read mapping with the FPGA-based system. *IEEE/ACM transactions on computational biology and bioinformatics*, **11**(5), 840–852.

Consortium, . G. P. *et al.* (2015). A global reference for human genetic variation. *Nature*, **526**(7571), 68–74.

Daily, J. (2016). Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC bioinformatics*, **17**(1), 81.

Eddy, S. R. (2004). What is dynamic programming? *Nature biotechnology*, **22**(7), 909.

Fei, X., Dan, Z., Lina, L., Xin, M., and Chunlei, Z. (2018). FPGASW: Accelerating Large-Scale Smith–Waterman Sequence Alignment Application with Backtracking on FPGA Linear Systolic Array. *Interdisciplinary Sciences: Computational Life Sciences*, **10**(1), 176–188.

Firtina, C., Kim, J. S., Alser, M., Senol Cali, D., Cicek, A. E., Alkan, C., and Mutlu, O. (2020). Apollo: a sequencing-technology-independent, scalable and accurate assembly polishing algorithm. *Bioinformatics*, **36**(12), 3669–3679.

Ghose, S., Boroumand, A., Kim, J. S., Gómez-Luna, J., and Mutlu, O. (2019). Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, **63**(6), 3–1.

Kim, J. S., Cali, D. S., Xin, H., Lee, D., Ghose, S., Alser, M., Hassan, H., Ergin, O., Alkan, C., and Mutlu, O. (2018). GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies. *BMC Genomics*, **19**(2), 89.

Lee, J., Bose, N., and Hwang, F. (1976). Use of Steiner's problem in suboptimal routing in rectilinear metric. *IEEE Transactions on Circuits and Systems*, **23**(7), 470–476.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics-Doklady*, volume 10, pages 707–710.

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

Liu, Y. and Schmidt, B. (2015). GSWABE: faster GPU-accelerated sequence alignment with optimal alignment retrieval for short DNA sequences. *Concurrency and Computation: Practice and Experience*, **27**(4), 958–972.

Mutlu, O., Ghose, S., Gómez-Luna, J., and Ausavarungnirun, R. (2019). Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems*, **67**, 28–41.

Myers, G. (1999). A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, **46**(3), 395–415.

Navarro, G. (2001). A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, **33**(1), 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. *Journal of molecular biology*, **48**(3), 443–453.

NVIDIA (2019a). CUDA C Programming Guide.

NVIDIA (2019b). NVIDIA GeForce RTX 2080 Ti User Guide.

Senol Cali, D., Kim, J. S., Ghose, S., Alkan, C., and Mutlu, O. (2019). Nanopore sequencing technology and tools for genome assembly: computational analysis of the current state, bottlenecks and future directions. *Briefings in bioinformatics*, **20**(4), 1542–1559.

Senol Cali, D., Kalsi, G. S., Bingöl, Z., Firtina, C., Subramanian, L., Kim, J. S., Ausavarungnirun, R., Alser, M., Luna, J. G., Boroumand, A., Nori, A., Scibisz, A., Subramoney, S., Alkan, C., Ghose, S., and Mutlu, O. (2020). GenASM: A High Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis. In *MICRO*.

Seshadri, V., Lee, D., Mullins, T., Hassan, H., Boroumand, A., Kim, J., Kozuch, M. A., Mutlu, O., Gibbons, P. B., and Mowry, T. C. (2017). Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *MICRO*.

Šošić, M. and Šikić, M. (2017). Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, **33**(9), 1394–1395.

Suzuki, H. and Kasahara, M. (2018). Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC bioinformatics*, **19**(1), 33–47.

Wang, C., Yan, R.-X., Wang, X.-F., Si, J.-N., and Zhang, Z. (2011). Comparison of linear gap penalties and profile-based variable gap penalties in profile–profile alignments. *Computational biology and chemistry*, **35**(5), 308–318.

Xilinx (2013). Virtex-7 XT VC709 Connectivity Kit.

Xin, H., Lee, D., Hormozdiari, F., Yedkar, S., Mutlu, O., and Alkan, C. (2013). Accelerating read mapping with FastHASH. In *BMC genomics*, volume 14, page S13.

Xin, H., Greth, J., Emmons, J., Pekhimenko, G., Kingsford, C., Alkan, C., and Mutlu, O. (2015). Shifted Hamming distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping. *Bioinformatics*, **31**(10), 1553–1560.

Supplementary Materials

5. Run Time and Space Complexity Analysis of the SneakySnake Algorithm

We analyze the asymptotic run time and space complexity of the SneakySnake algorithm. We provide the pseudocode of SneakySnake in Algorithm 1. The SneakySnake algorithm builds the chip maze on-the-fly by partially constructing each horizontal routing track starting from each new checkpoint until it reaches an obstacle in each horizontal routing track. The SneakySnake algorithm does not necessarily construct the entire chip maze. At each new checkpoint, the SneakySnake algorithm examines if the signal net 1) does not reach the destination terminal or 2) does not exceed the allowed propagation delay before the SneakySnake algorithm continues calculating the horizontal routing tracks (as we explain in Algorithm 1, line 4). It then uses the function *UpperHRT*() (Algorithm 2) to construct the first escape segment, after the current checkpoint, of each of the upper HRTs (as we explain in Algorithm 1, line 6). After constructing the escape segments, it computes their length and returns the length of the longest escape segment. Note that during the first iteration of the SneakySnake algorithm, the function *UpperHRT*() (Algorithm 2) returns a value of 1, which is the length of a single obstacle. This is because all upper HRTs start with an obstacle. The SneakySnake algorithm performs the same steps as in the function *UpperHRT*() for the main HRT (Algorithm 1, line 7) and the lower HRTs (Algorithm 1, line 12), by calling the two functions: *MainHRT*() (Algorithm 3) and *LowerHRT*() (Algorithm 4). Finally, we update the position of the checkpoint and the current propagation delay of the found signal net through Algorithm 1, lines 15-18. Once the signal net exceeds the allowed propagation delay, the SneakySnake algorithm terminates (as we show in Algorithm 1, line 4 and lines 19-20). Otherwise, the SneakySnake algorithm allows computationally expensive edit distance or pairwise alignment algorithms to compute their output based on the user-defined parameters (as we show in Algorithm 1, lines 21-22).

On the one hand, the lower-bound on the time complexity of the SneakySnake algorithm is $O(m)$, which is achieved when the SneakySnake algorithm reaches the destination terminal of the maze without facing any obstacle along the signal net. For example, when a query sequence matches exactly a reference sequence, the SneakySnake algorithm traverses only through the $E+1^{th}$ HRT (i.e., main HRT) and then allows the edit distance or alignment algorithm to perform its computation.

On the other hand, the upper-bound on the run time complexity of the SneakySnake algorithm is reached when the algorithm has to construct the *entire* chip maze, which is the worst case. As we have $2E+1$ horizontal routing tracks, each of which is m characters long, the upper-bound run time complexity is $O((2E+1)m)$. However, it is unrealistic to construct the entire chip maze, as in this case, all the horizontal routing tracks should be identical in terms of the number and the location of all obstacles. Consider a pair of query and reference sequences, where each character is generated completely randomly (having 1/4 probability of being either A, C, G, or T). The probability that a character of the query sequence does not match any neighboring character of the reference sequence during the construction of any of the $2E+1$ horizontal routing tracks is $(3/4)^{2E+1}$, which decreases exponentially as E increases. Therefore, this upper-bound on the run time complexity is still loose.

Algorithm 1: SneakySnake

Input: query (*Q*), reference (*R*), and edit distance threshold (*E*)
Output: -1 for dissimilar sequences / *EditDistance*() or *Alignment*()
Functions: *UpperHRT*(), *MainHRT*(), *LowerHRT*() construct the first escape segment of each of the *E* upper, main, and *E* lower horizontal routing tracks, respectively, and returns the length of the longest escape segment
Pseudocode:

```
 1: checkpoint = 0
 2: PropagationDelay = 0
 3: m = length(Q)
 4: while checkpoint < m and PropagationDelay <= E do
 5:     count = 0
 6:     longest_es = UpperHRT(Q[checkpoint:m-1], R[checkpoint:m-1], E)
 7:     count = MainHRT(Q[checkpoint:m-1], R[checkpoint:m-1])
 8:     if count == m then
 9:         return = EditDistance() or Alignment()
10:     if count > longest_es then
11:         longest_es = count
12:     count = LowerHRT(Q[checkpoint:m-1], R[checkpoint:m-1], E)
13:     if count > longest_es then
14:         longest_es = count
15:     checkpoint = checkpoint + longest_es
16:     if checkpoint < m then
17:         PropagationDelay++
18:         checkpoint++
19: if PropagationDelay > E then
20:     return -1
21: else
22:     return EditDistance() or Alignment()   //depends on user's requirement
```

Algorithm 2: UpperHRT

Input: query (*Q[checkpoint:m-1]*), reference (*R[checkpoint:m-1]*), and edit distance threshold (*E*)
Output: length of the longest escape segment of the upper horizontal routing tracks
Pseudocode:

```
 1: longest_es = 0
 2: for r = E to 1 do
 3:     count = 0
 4:     for n = checkpoint to length(Q)-1 do
 5:         if n < r then
 6:             goto EXIT
 7:         else if Q[n-r] != R[n] then
 8:             goto EXIT
 9:         else if Q[n-r] == R[n] then
10:             count++
11: EXIT:
12:     if count > longest_es
13:         longest_es = count
14: return longest_es
```

Algorithm 3: MainHRT

Input: query (*Q[checkpoint:m-1]*) and reference (*R[checkpoint:m-1]*)
Output: length of the longest escape segment of the main horizontal routing track
Pseudocode:

```
1:  longest_es = 0
2:  for n = checkpoint to length(Q)-1 do
3:      if Q[n] != R[n] then
4:          return longest_es
5:      else if Q[n] == R[n] then
6:          longest_es = longest_es + 1
7:  return longest_es
```

Algorithm 4: LowerHRT

Input: query (*Q[checkpoint:m-1]*), reference (*R[checkpoint:m-1]*), and edit distance threshold (*E*)
Output: length of the longest escape segment of the lower horizontal routing tracks
Pseudocode:

```
 1:  longest_es = 0
 2:  for r = 1 to E do
 3:      count = 0
 4:      for n = checkpoint to length(Q)-1 do
 5:          if n > m-r-1 then
 6:              goto EXIT
 7:          else if Q[n+r] != R[n] then
 8:              goto EXIT
 9:          else if Q[n+r] == R[n] then
10:              count++
11:  EXIT:
12:      if count > longest_es
13:          longest_es = count
14:  return longest_es
```

6. Proofs of the Correctness and Optimality of the SneakySnake Algorithm

As the propagation delay of a signal net is mainly affected by the number of obstacles that are considered in the horizontal escape segments of the selected path, for simplicity, we do not consider the vertical segments in our proof.

6.1. Correctness proof

PROOF. We prove Theorem 1 by contradiction. Let $A = \{s_1, s_2, \ldots, s_n\}$ be the signal net that connects the source terminal to the destination terminal using n escape segments that are part of the horizontal routing tracks within a routing region. The escape segments are sorted by their start position (i.e., s_1 starts before s_2 and ends at s_2). Assume that the SneakySnake algorithm is not able to find this signal net A that reaches the

destination terminal. This means that the SneakySnake algorithm finds an escape segment, s_k, but it fails to find the next escape segment, s_{k+1}. Since there is a signal net that connects s_1 to s_n, there exists an escape segment that starts before s_{k+1} and ends at s_{k+1}. This escape segment is not reachable from s_k (as we assume that the SneakySnake algorithm terminates the solution after finding s_k), so it should be reachable from another escape segment, s_t, where $t < k$. This indicates that s_{k+1} is not reachable from s_k and s_k is not reachable from s_t. This contradicts the assumption that s_{k+1} is reachable and it is part of the solution. Thus, our assumption that the SneakySnake algorithm is not able to find a signal net is wrong. ■

6.2. Optimality proof

PROOF. We prove Theorem 2 by induction. Suppose you have a set of n candidate horizontal segments {1, 2, …, n} that are part of the horizontal routing tracks within a routing region. Each horizontal segment has a pair of start and end positions ($s(i)$, $f(i)$). The SneakySnake algorithm determines a signal net with the minimum total propagation delay by repeatedly selecting from the available horizontal segments the one that starts at the current location and has the farthest end location, and removing all overlapping horizontal segments from the set. Let $A = \{x_1, x_2, \ldots, x_k\}$ be the solution (set of escape segments) to SNR problem provided by the SneakySnake algorithm. The escape segments are sorted by their start position (i.e., x_1 starts before x_2 and ends at x_2). Let $B = \{y_1, y_2, \ldots, y_m\}$ be the optimal solution for the same SNR problem. Let $k = |A|$ and $m = |B|$ denote the number of escape segments in A and B, respectively. The proof is by *induction* on the number of escape segments. We will compare A and B by their segments' end positions. We will show that for all $r \leq k$, $f(x_r) \geq f(y_r)$.

As the base case, we take $k = m = 1$. Since SneakySnake and the optimal algorithm select the longest escape segment that start at the beginning of a horizontal routing track, it certainly must be the case that $f(x_1) \geq f(y_1)$.

For $r > 1$, assume the statement $f(x_{r-1}) \geq f(y_{r-1})$ is true for $r - 1$ and we will prove it for r. The induction hypothesis states that $f(x_{r-1}) \geq f(y_{r-1})$, and so any horizontal segment that is not overlapping with the first $r - 1$ escape segments in the optimal solution is certainly not overlapping with the first $r - 1$ escape segments of the SneakySnake algorithm. Therefore, we can add y_r to the SneakySnake solution, and since the SneakySnake algorithm always considers the longest escape segments, it must be the case that $f(x_r) \geq f(y_r)$. So we have that for all $r \leq k$, $f(x_r) \geq f(y_r)$. In particular, $f(x_k) \geq f(y_k)$. If A is not optimal, then it must be the case that $m < k$, and so there is an escape segment x_{m+1} in A that is not in B. This escape segment must start after A's m^{th} escape segment ends, and hence after $f(y_m)$. But then the segment x_{m+1} is not overlapping with all the escape segments in B, and so it should be part of the solution in B. This contradicts the assumption that $m<k$, and thus A has as many elements as B. So the SneakySnake algorithm always produces an optimal solution. ■

7. Similarities and Differences Between the SNR Problem in VLSI CAD and the SNR Problem for Pre-alignment Filtering

We use the SNR problem as a simple example that can explain/visualize the pre-alignment filtering problem (Alser *et al.*, 2020a; Alser *et al.*, 2020b). We believe that the SNR problem and the pre-alignment filtering problem are very similar. There are three main similarities. 1) Both problems aim to find the net (a set of non-overlapping matching segments) that provides the minimal propagation delay (number of edits). 2) Both problems have normally a free choice of pin assignment. That is, the source and destination nodes can be any of the IO pads around the chip. 3) Both problems consider the presence of obstacles (edits) and some constraints. Fig. 6 provides a 3-dimensional top-view and a side-view of the chip maze in Fig. 1 to clearly illustrate how the different metal layers (routing tracks) are connected.

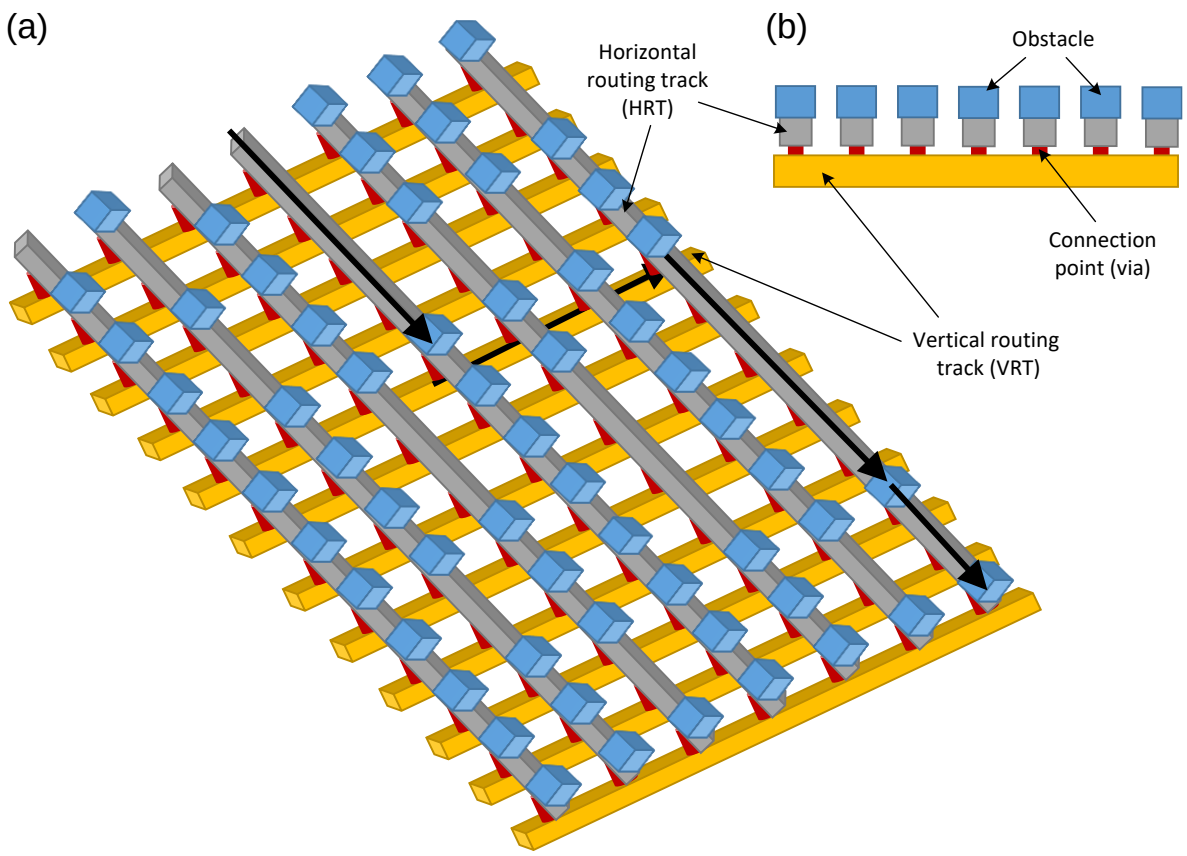


Fig. 6: (a) A 3-dimensional top-view and (b) side-view of the same chip maze presented in Fig. 1. The signal net has 3 obstacles, each of which is located at the end of each escape segment (a black arrow on the horizontal routing track).

However, we also highlight four key differences that make our SNR problem slightly different (a special case) from what we normally have in VLSI CAD, as we summarize in Table 2. These four differences can render the existing general algorithms that solve the SNR problem in VLSI CAD, e.g., (Roy and Markov, 2008; Chu and Wong, 2007) inefficient at directly solving our SNR problem. Instead, SneakySnake provides a new efficient algorithm that does not require building the entire chip maze in advance (as we illustrate in Fig. 2(c)), while it considers the propagation delay of each obstacle faced throughout the signal net.

Table 2: A summary of the four key differences between the SNR problem in VLSI CAD and the SNR problem for pre-alignment filtering.

	SNR problem in VLSI CAD	**SNR problem for pre-alignment filtering**
The size and location of an obstacle	The obstacles (modules such as caches, embedded memories, and intellectual property (IP) cores) in the VLSI chip do not necessarily have the same shape, area, nor fixed locations.	In our chip maze, all obstacles are assumed to be 1x1 objects and can be placed anywhere in the grid.
Optimal solution	The obstacle that is faced by the optimal net is typically avoided in VLSI routing, using a new metal layer or another track, if available, on the same metal layer.	The obstacle that is faced throughout the optimal net cannot be avoided and it contributes to the total delay of the optimal net.
Pre-processing requirements	Prior to solving the SNR problem, a step called *chip planning* should be carried out to optimize the location and the aspect ratio of each individual obstacle.	No pre-processing steps are required.
Building a complete chip maze	After performing chip planning, we build a complete graph that represents the chip maze and then apply one of the state-of-the-art algorithms (e.g., Dijkstra's algorithm (Dijkstra, 1959) and A* (Hart *et al*., 1968) to solve the SNR problem. These algorithms typically require building the entire chip maze and calculating the distance between every two nodes before applying the algorithm itself. A detailed summary of these algorithms is presented in (Roy and Markov, 2008; Chu and Wong, 2007).	SneakySnake builds only the portion of the chip maze that is absolutely needed to provide an optimal solution to the SNR problem for pre-alignment filtering.

8. Snake-on-Chip Hardware Architecture

Next, we present the details of our hardware architecture of Snake-on-Chip in four key steps.

(1) Snake-on-Chip constructs the *entire* chip maze of each subproblem. Each chip maze has 2*E*+1 bit-vectors (rows) and each bit-vector is *t* bits long. This is different from the CPU implementation of the SneakySnake algorithm, as the number of entries computed in each row is no longer limited to the entries that are located only between a checkpoint and the first following obstacle. This is due to the fundamental difference between a CPU core (sequential execution) and an FPGA chip (parallel processing). We want to concurrently compute all bits of all bit-vectors beforehand so that we can exploit massive bitwise parallelism provided by an FPGA and perform computations on all bit-vectors in a parallel fashion.

(2) It computes the length of the first horizontal segment of consecutive zeros for each bit-vector (i.e., each HRT) using a leading-zero counter (LZC). Snake-on-Chip uses the LZC design proposed in (Dimitrakopoulos *et al.*, 2008) as it requires a low number of both logic gates and logic levels. It counts the number of leading consecutive zeros that appear in a *t*-bit input vector.

(3) Snake-on-Chip finds the bit-vector (i.e., HRT) that has the largest number of leading zeros. Snake-on-Chip implements a hierarchical comparator structure with $\lceil log_2(2E+1) \rceil$ levels. Each comparator compares the output of two LZCs and finds the largest value. That is, we need 2*E*+2 comparators, each of which is a $(\lfloor log_2 t \rfloor + 1)$-bit comparator, for comparing the leading zero counts of 2*E*+1 *t*-bit LZCs and finding the largest leading zero count. Consider that we choose *t*, *E*, and *m* to be 8 columns, 5 edits (i.e., 11 rows), and 100 characters, respectively. This results in partitioning the chip maze of size 11 × 100 into 13 (i.e., *m*/*t*) subproblems, each of size 11 × 8. We need 11 LZCs and 12 comparators. We arrange the 12 LZC comparators into 4 levels: the first level of LZC comparators that is directly connected to the LZCs has 6 LZC comparators, the second level has 3 LZC comparators, the third level has 2 LZC comparators, and the last level has a single LZC comparator. This hierarchical comparator structure compares the 11 escape segments of a subproblem and produces the length of the longest escape segment (*x*). We provide the overall architecture of the 4-level LZC comparator tree including the 11 LZC block diagrams in Fig. 7.

(4) After computing the length of the longest segment (i.e., the largest leading-zero count), Snake-on-Chip creates a new checkpoint to iterate over the HRTs once again to find the next optimal escape segment. Snake-on-Chip achieves this by shifting the bits of each row (i.e., HRT) to the right-hand direction (assuming the least significant bit starts from the right-hand side). The shift amount is equal to *x* bits, where *x* is the length of the found longest escape segment of the consecutive zeros calculated in the third step. To skip the obstacle that exists at the end of the longest escape segment, Snake-on-Chip shifts the bits of each row by an additional single step to the right-hand direction. This guarantees to exclude the previously-found longest escape segment along with a single obstacle from the new search round.

(5) Snake-on-Chip repeats the previous three steps (steps 2, 3, and 4) to find the next optimal escape segment starting from the least significant bit (i.e., the new checkpoint) all the way to the most significant bit. Repeating the previous three steps for each iteration is achieved by building a new module instance for the architecture design of all the three previous steps. The 2*E*+1 output bit-vectors calculated by the fourth step are the 2*E*+1 input bit-vectors to the new hardware instance. The number of iterations (*y*, i.e., hardware instances) needed depends on the desired accuracy of the SneakySnake algorithm (as we experimentally

evaluate the effect of choosing different values of y on the accuracy of Snake-on-Chip in https://github.com/CMU-SAFARI/SneakySnake/tree/master/Evaluation%20Results). If our target is to find an optimal signal net that has at most a single obstacle within each subproblem built in the first step, then we need to build two hardware instances, each of which performs the previous three steps (steps 2, 3, and 4). For example, let D, one of the $2E+1$ bit-vectors that is also the optimal signal net, be "00010000", where $t = 8$. The first hardware instance computes the value of x (the length of the longest escape segment calculated in the third step) as four zeros, updates the bits of D to "11111000", and passes the updated D to the second hardware instance. The second hardware instance computes the value of x as three zeros and updates the bits of D to "11111111".

(6) The last step is to calculate the total number of obstacles faced along the entire optimal signal net in each subproblem. For each subproblem, Snake-on-Chip calculates the total number of obstacles as follows:

$$min\ (y, t - \sum_{k=1}^{y} x_k) \quad (2)$$

where y is the total number of hardware instances included in the architecture of Snake-on-Chip, t is the width of the chip maze of each subproblem, and x_k is the length of the longest segment of consecutive zeros found by the hardware instance of index k. Hence, the total number of obstacles for the original problem of size $(2E+1) \times m$ is simply the summation of the total number of obstacles (calculated in Equation 2) faced along the optimal signal net of all subproblems.

Snake-on-Chip makes the following technical contributions:

1) We introduce the approach of dividing a single SNR problem into several subproblems that can be solved concurrently and independently. FPGAs typically provide parallelism in two main ways: 1) providing a large number (typically few millions) of look-up tables (LUTs) that can form a large number of hardware compute units to perform computation in a parallel fashion and 2) providing massive bitwise parallelism for each compute unit. To build Snake-on-Chip, we need to decide on 1) the size and the number of compute units (we call them *filtering units*) that can be integrated within the FPGA chip and 2) custom-tailored operations to the SNR problem that leverage bitwise operations. A filtering unit that occupies a large number of LUTs can have a large critical path delay, which directly affects the maximum operating frequency and hence it affects the filtering speed. The approach of dividing the SNR problem into several SNR subproblems provides three key benefits that can reduce the LUT requirement of each filtering unit, as we list in Section 2.5 in the main manuscript.
2) We comprehensively analyze and evaluate different design choices for the size of each filtering unit of Snake-on-Chip (as we experimentally evaluate in "Effect of y & t on SneakySnake" Excel sheet in https://github.com/CMU-SAFARI/SneakySnake/tree/master/Evaluation%20Results). This analysis helps us to build an efficient hardware architecture that has a very small LUT requirement. This allows integrating a large number of these hardware filtering units within the FPGA chip, where they all operate concurrently and independently.
3) We build a modular hardware architecture that is scalable with both sequence length and edit distance threshold.

4) We introduce an efficient FPGA-friendly implementation with a low FPGA resource utilization (less than 1.5% of the total number of FPGA LUTs for a single filtering unit, as we show in Section 10.4 in the Supplementary Materials). We make both the hardware architecture of Snake-on-Chip and the complete software/hardware co-design FPGA project publicly available at: https://github.com/CMU-SAFARI/SneakySnake/tree/master/Snake-on-Chip

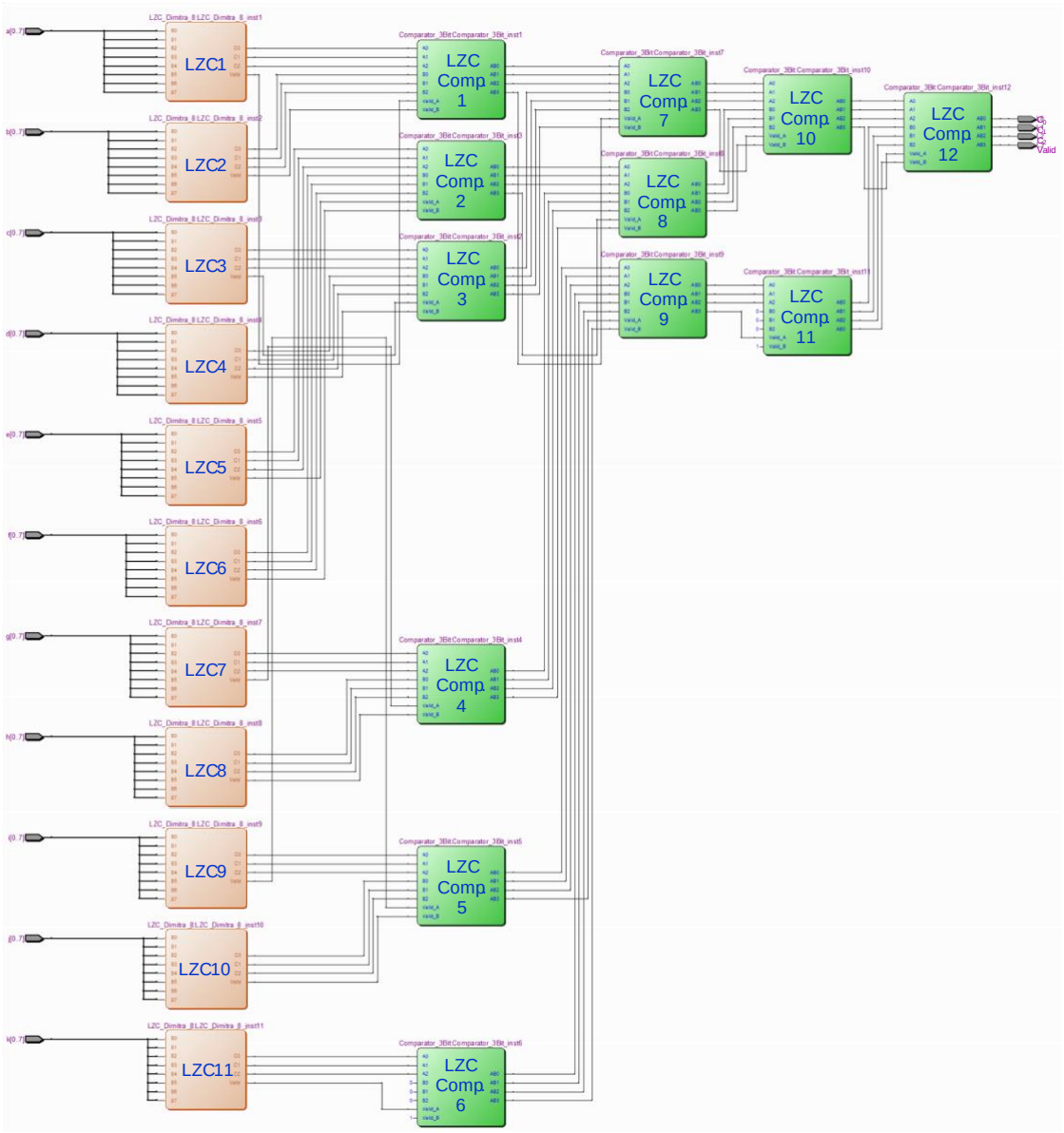


Fig. 7: Block diagram of the 11 LZCs (highlighted in orange color) and the hierarchical LZC comparator tree (highlighted in green color) for computing the largest number of leading zeros in 11 rows.

9. Snake-on-GPU Parallel Implementation

Snake-on-GPU makes three key assumptions that help with providing an efficient GPU implementation. (1) The entire input dataset of query and reference sequences fits in the GPU global memory, which is the off-chip DRAM memory of a GPU (NVIDIA, 2019a) and it typically fits several GB of data (e.g., NVIDIA GeForce RTX 2080Ti card, which is used for Snake-on-GPU implementation, has a global memory of 11 GB). (2) We copy the entire input dataset from the CPU main memory to the GPU global memory before the GPU kernel execution starts. This enables massively-parallel computation by making a large number of input sequences available in the GPU global memory. (3) We copy back the pre-alignment filtering results from the GPU global memory to the CPU main memory only after the GPU kernel completes the computation. If the size of the input dataset exceeds the size of the GPU global memory, we divide the dataset into independent smaller datasets, each of which can fit the capacity of the GPU global memory. This approach also helps us to overlap the computation performed on one small dataset with the transfer of another small dataset between the CPU memory and GPU memory (Gómez-Luna *et al.*, 2012).

Given the large size of the input dataset that the GPU threads need to access from the GPU global memory, we carefully design Snake-on-GPU to efficiently use the on-chip register file to store the query and the reference sequences and avoid unnecessary accesses to the off-chip global memory. The workflow of Snake-on-GPU includes two key steps, as we show in Fig. 8. 1) Each thread copies a single reference sequence and another single query sequence from global memory to the on-chip registers. Assuming the maximum length of a query (or reference) sequence is m (i.e., the maximum number of VRTs), we need $2m$ bits to encode each character of the query (or reference) sequence into a unique binary representation. Since the size of a register is 4 bytes (32 bits), each thread needs $R = \left\lceil \frac{2m}{32} \right\rceil$ registers to store an entire query/reference sequence. For example, for a maximum length of m = 128, R = 8. This way, 16 registers are enough to store both query and reference sequences. This number is much lower than the maximum of 256 registers that each thread can use in current NVIDIA GPUs. Thus, the resources of a GPU core are not exhausted and more threads can run concurrently. 2) Each thread solves the complete SNR problem for a single query sequence and a single reference sequence. Each GPU thread applies the same computation of the SneakySnake algorithm to solve the SNR problem.

Snake-on-GPU makes the following two technical contributions:

1) We provide a theoretical analysis of the available resources (on-chip register file and off-chip global memory) of typical modern GPUs and how they affect the performance of Snake-on-GPU in Section 9. Based on this analysis, Snake-on-GPU uses one single GPU thread to solve one SNR problem. This design choice provides three key benefits: 1) it maximizes the utilization of the on-chip registers as they provide fast data access, 2) it minimizes the utilization of the off-chip global memory as off-chip communication is expensive, i.e., time-consuming and energy inefficient (Mutlu *et al.*, 2019; Ghose *et al.*, 2019), and it can affect the number of threads that operate concurrently (NVIDIA, 2019a), and 3) it avoids the need for synchronizing several threads working on the same SNR problem. These benefits lead to achieving a high degree of parallelism.
2) We introduce an efficient fully-configurable GPU implementation where users can change the edit distance threshold value at run time without the need to change the implementation. We make our parallel GPU implementation, Snake-on-GPU, *publicly available* at: https://github.com/CMU-SAFARI/SneakySnake/blob/master/Snake-on-GPU

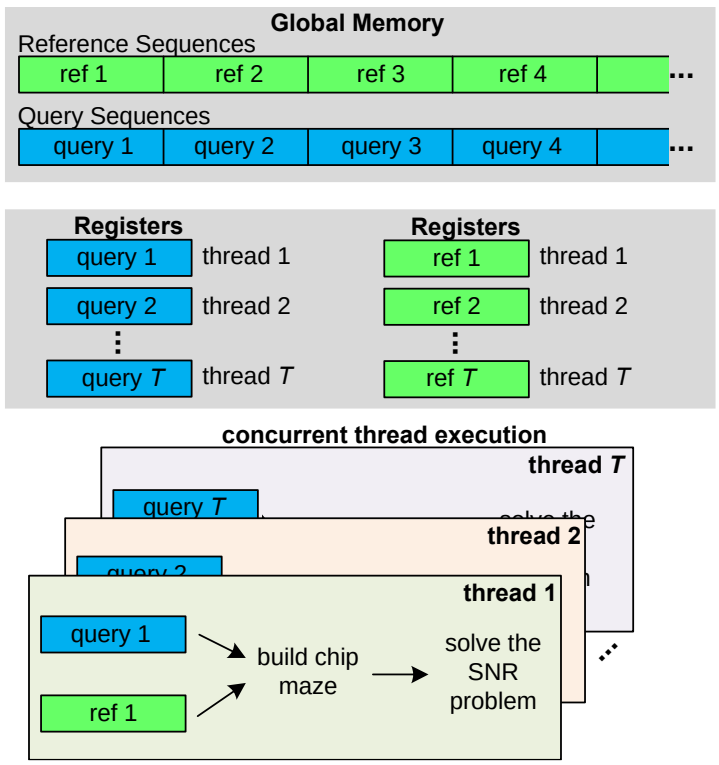


Fig. 8: Workflow of Snake-on-GPU. It includes two key steps: (1) each GPU thread loads a single reference sequence and a single query sequence into registers, (2) the assigned thread solves a single SNR problem for the two sequences.

10. Supplementary Evaluation

10.1. Dataset Descriptions

We have two key approaches to generating sequence pairs for testing the performance of pre-alignment filters. 1) We can use existing read mappers to find reference segments that might be similar or dissimilar to real reads. We use the reference segments that are generated by read mapper before applying the DP-based pairwise alignment step to ensure that we obtain both similar and dissimilar (i.e., that are usually filtered out by the pairwise alignment step) generated pairs (a read sequence and its reference segment). 2) We can also use available read simulators to generate sequence pairs. The read simulators allow controlling the type of edits, the number of edits, and their distribution over a sequence. We follow both approaches, as they both are still widely-used in evaluating existing algorithms (Li, 2018). 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 SRR826471_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 the mrFAST mapper (Alkan *et al.*, 2009) and observe all potential mapping locations of every read. We obtain the human reference genome from the 1000 Genomes Project (1000 Genomes Project Consortium, 2015). Before mapping the reads, we disable the DP-based pairwise alignment algorithm of the 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 m =100 bp and 8 and 100 for m = 250 bp) using the e parameter of mrFAST to generate four real datasets in total. Each dataset contains the sequence pairs that are generated by the mrFAST mapper before the read alignment step of mrFAST, such that we allow each dataset to contain both similar (i.e., having edits fewer than or equal to the edit distance threshold) and dissimilar (i.e., having more edits than the edit distance threshold) sequences over a wide range of edit distance thresholds. For the reader's convenience, we refer to these datasets as `100bp_1`, `100bp_2`, `250bp_1`, and `250bp_2`. We summarize the details of these four datasets in Table 3. We provide the source used to obtain the read sets, the read length in each read set, and the configuration used for the e parameter of mrFAST (Alkan *et al.*, 2009) for our real 4 datasets. We use Edlib (Šošić and Šikić, 2017) to assess the number of similar (i.e., having edits fewer than or equal to the edit distance threshold) and dissimilar (i.e., having more edits than the edit distance threshold) pairs for each of the 4 datasets across different user-defined edit distance thresholds. We provide these details for `100bp_1`, `100bp_2`, `250bp_1`, and `250bp_2` in Table 4.

Simulated datasets. We generate two sets (we refer to them as `10Kbp` and `100Kbp`) of long sequence pairs using PBSIM (Ono *et al.*, 2013). We choose this simulator as it provides pairs of two sequences, the original segment of the reference (not only the location as in some read simulators) and its simulated segment. This helps us to directly obtain sequence pairs that can be used to evaluate the performance of sequence aligners and pre-alignment filters. We use the first Human chromosome sequence (GRCh38.p13 assembly, downloaded from https://www.ncbi.nlm.nih.gov/nuccore/NC_000001.11) for the input reference sequence in PBSIM. We generate `10Kbp` to have 100,000 sequence pairs, each of which is 10 Kbp long, at 30× genome coverage. `100Kbp` has 74,687 sequence pairs, each of which is 100 Kbp long, at 30× genome coverage. For both sets (`10Kbp` and `100Kbp`), we use the default error profile for continuous long reads (CLR) in PBSIM.

Table 3: Benchmark Illumina datasets (read-reference pairs). We map each read set to the human reference genome to generate four datasets of sequence pairs (read sequence and reference segment) using different edit distance thresholds (using the *e* parameter).

Accession no.	**ERR240727_1**		**SRR826471_1**	
Source	https://www.ebi.ac.uk/ena/data/view/ERR240727		https://www.ebi.ac.uk/ena/data/view/SRR826471	
Sequence Length	100		250	
Sequencing Platform	Illumina HiSeq 2000		Illumina HiSeq 2000	
Dataset	`100bp_1`	`100bp_2`	`250bp_1`	`250bp_2`
mrFAST *e*	2	40	8	100
Amount of Edits	Low-edit	High-edit	Low-edit	High-edit

Table 4: Details of evaluating the number of similar and dissimilar sequences in each of our four datasets using Edlib over a wide range of edit distance thresholds of *E*= 0% up to *E*= 10% of the sequence length. Each dataset contains 30 million sequence pairs.

	100bp_1		100bp_2			250bp_1		250bp_2	
***E* (%)**	**Similar**	**Dissimilar**	**Similar**	**Dissimilar**	***E* (%)**	**Similar**	**Dissimilar**	**Similar**	**Dissimilar**
0	381,901	29,618,099	11	29,999,989	**0**	707,517	29,292,483	49	29,999,951
1	1,345,842	28,654,158	18	29,999,982	**1**	1,462,242	28,537,758	163	29,999,837
2	3,266,455	26,733,545	24	29,999,976	**2**	1,973,835	28,026,165	301	29,999,699
3	5,595,596	24,404,404	27	29,999,973	**3**	2,361,418	27,638,582	375	29,999,625
4	7,825,272	22,174,728	29	29,999,971	**4**	3,183,271	26,816,729	472	29,999,528
5	9,821,308	20,178,692	34	29,999,966	**5**	3,862,776	26,137,224	520	29,999,480
6	11,650,490	18,349,510	83	29,999,917	**6**	4,915,346	25,084,654	575	29,999,425
7	13,407,801	16,592,199	177	29,999,823	**7**	5,550,869	24,449,131	623	29,999,377
8	15,152,501	14,847,499	333	29,999,667	**8**	6,404,832	23,595,168	718	29,999,282
9	16,894,680	13,105,320	711	29,999,289	**9**	6,959,616	23,040,384	842	29,999,158
10	18,610,897	11,389,103	1,627	29,998,373	**10**	7,857,750	22,142,250	1,133	29,998,867

10.2. Effect of Multithreading on Filtering and Alignment Time

We examine the execution time of SneakySnake, Parasail (Daily, 2016), and SneakySnake integrated with Parasail as the number of threads increases from 1 to 40, as we show in Fig. 9. We run this experiment using a 2.3 GHz Intel Xeon Gold 5118 CPU with up to 48 threads and 192 GB RAM. We choose SneakySnake as it is the only pre-alignment filter that supports multithreading, compared to Shouji (Alser *et al.*, 2019), MAGNET (Alser *et al*., 2017b), GateKeeper (Alser *et al*., 2017a), and SHD (Xin *et al*., 2015). We choose Parasail (*parasail_nw_banded*) as it supports both multithreading and configurable scoring function. We make three key observations based on Fig. 9. (1) SneakySnake is always faster than Parasail over a wide range of both number of threads and datasets. SneakySnake is 9.3× (using `100bp_2` and 24 threads) to 30× (using `100bp_1` and a single thread) faster than Parasail in examining the sequence pairs, when the edit distance threshold is set to 10% of the sequence length. (2) The addition of SneakySnake as a pre-alignment filtering step reduces the execution time of Parasail by 1.2× (using `100bp_1` and 40 threads) to 28.2× (using `250bp_2` and a single thread). (3) Both SneakySnake and Parasail scale very well as the number of threads increases. We provide the exact values of all evaluation results in https://github.com/CMU-SAFARI/SneakySnake/tree/master/Evaluation%20Results.

We conclude that SneakySnake efficiently supports multithreading. Integrating SneakySnake with a state-of-the-art sequence alignment algorithm is always beneficial and reduces the end-to-end execution time by up to an order of magnitude even when using a large number of threads for both tools.

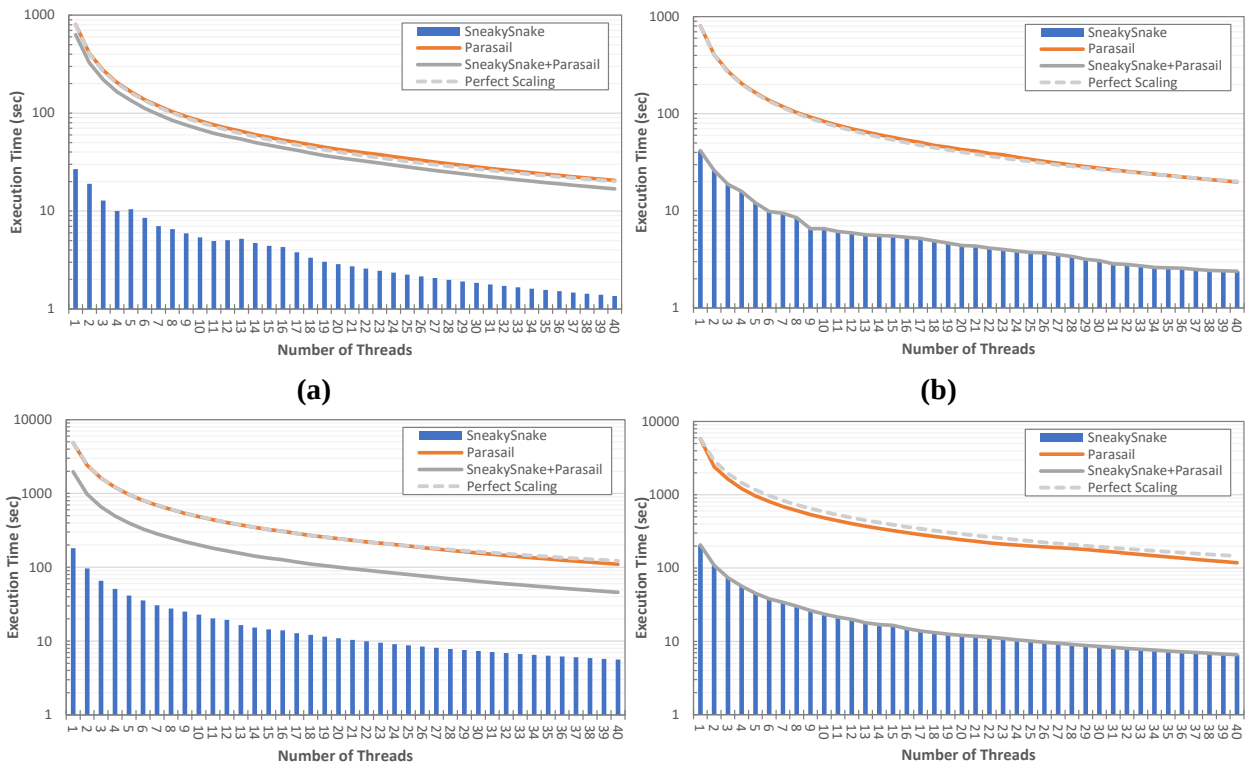


Fig. 9: The effect of multithreading on the execution time of SneakySnake, Parasail, and SneakySnake integrated with Parasail. All y-axes are on a logarithmic scale. We use four datasets: 100bp_1 in (a), 100bp_2 in (b), 250bp_1 in (c), and 250bp_2 in (d). 100bp_1 and 100bp_2 use a sequence length (*m*) of 100 bp, while 250bp_1 and 250bp_2 use a sequence length (*m*) of 250 bp. We set the edit distance threshold (*E*) to 10% of the sequence length (i.e., *E*=10 in (a) and (b) and *E*=25 in (c) and (d)). We also provide a theoretical linear scaling of Parasail's execution time, referred to as *perfect scaling*.

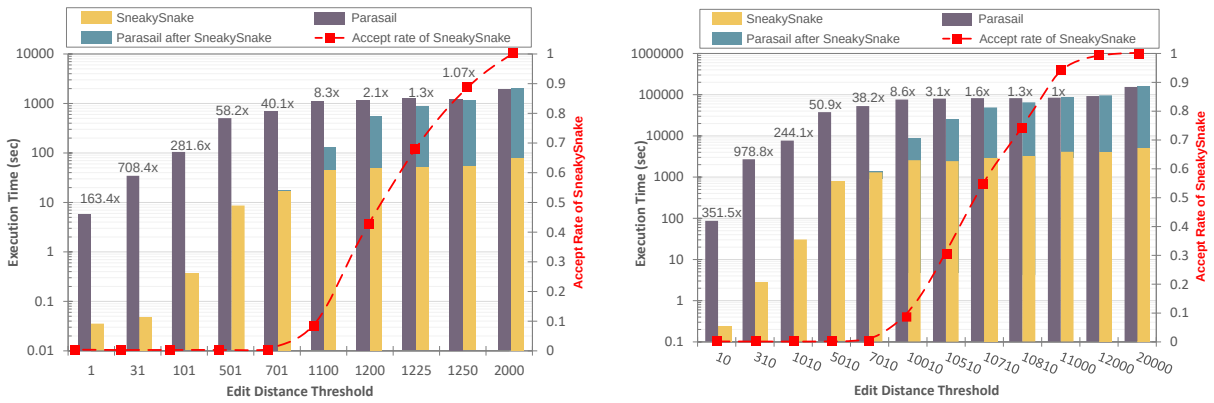
10.3. Evaluating the Execution Time of Filtering and Alignment Using Long Sequences

We examine the execution time of SneakySnake, Parasail, and SneakySnake integrated with Parasail using long sequences, as we show in Fig. 10. We run both SneakySnake and Parasail using two sets (10Kbp and 100Kbp) of long sequences and 40 CPU threads. We run SneakySnake with $t = y = (E+500)$, where t is the width of the chip maze of each subproblem, y is the number of iterations performed to solve each subproblem, and E is the edit distance threshold. We choose the values of t and y to be less than the sequence length to prevent SneakySnake from examining the entire chip maze, which helps to achieve fast filtering at the cost of a slight increase in the number of falsely-accepted pairs (with a 0% false reject rate). We also choose the values of t and y to be more than E to prevent the chip maze from having complete rows of obstacles based on Equation 1 in the main paper. We experimentally evaluate the effect of varying the values of t and y on both the accuracy and execution time of SneakySnake in https://github.com/CMU-SAFARI/SneakySnake/tree/master/Evaluation%20Results. We generate the two sets of long sequence pairs

using PBSIM (Ono *et al.*, 2013). We use Human chromosome 1 sequence (GRCh38.p13 assembly, downloaded from https://www.ncbi.nlm.nih.gov/nuccore/NC_000001.11) for the input reference sequence in PBSIM. We generate 10Kbp to have 100,000 sequence pairs, each of which is 10 Kbp long, at 30× genome coverage. 100Kbp has 74,687 sequence pairs, each of which is 100 Kbp long, at 30× genome coverage. For both sets (10Kbp and 100Kbp), we use the default error profile for the continuous long reads (CLR) in PBSIM. We use a wide range of edit distance thresholds, up to 20% of the sequence length.

Based on Fig. 10, we make two key observations. (1) Using 10Kbp and 100Kbp, SneakySnake makes Parasail significantly faster (by 58.2-708.4× and by 50.9-978.8×, respectively) than Parasail alone in detecting dissimilar pairs of long sequences, even at high edit distance thresholds (up to E=501 for 10Kbp and up to E=5010 for 100Kbp, which results in building and examining 1003 and 10021 rows, respectively, for each chip maze of the SneakySnake algorithm). (2) As the number of similar sequence pairs increases (at E > 501 for 10Kbp and at E > 5010 for 100Kbp), the benefit of integrating SneakySnake with Parasail in reducing the end-to-end execution time reduces. When Parasail examines 89% and 94% of the input sequence pairs (SneakySnake filters out the rest of the sequence pairs) of 10Kbp and 100Kbp datasets, 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 Parasail). We provide the exact values of all evaluation results in https://github.com/CMU-SAFARI/SneakySnake/tree/master/Evaluation%20Results.

We conclude that SneakySnake supports multithreaded filtering for long sequences. Integrating SneakySnake with a state-of-the-art sequence alignment algorithm that supports multithreading is also beneficial and sometimes reduces the end-to-end execution time by up to two orders of magnitude.

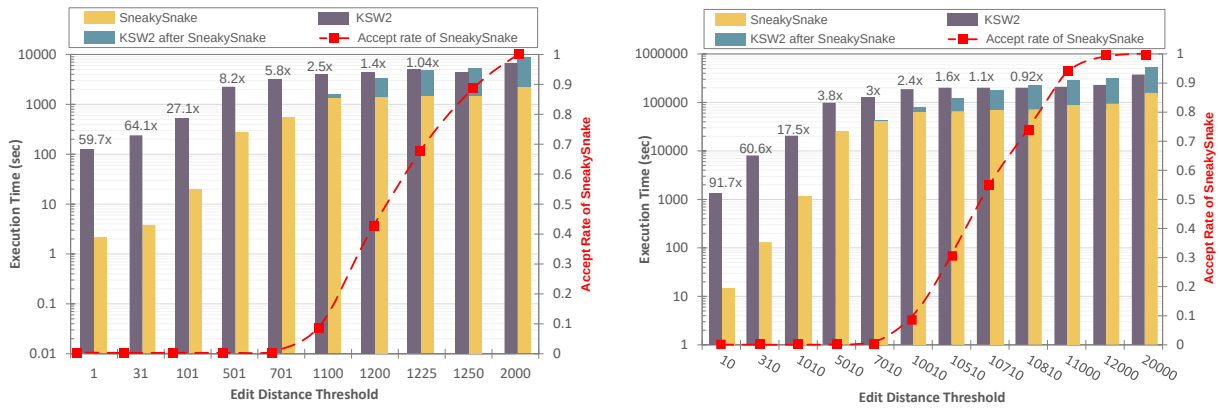


(a) **(b)**

Fig. 10: The execution time of SneakySnake, Parasail, and SneakySnake integrated with Parasail using long sequences, (a) 10Kbp and (b) 100Kbp, and 40 CPU threads. The left y-axes of (a) and (b) are on a logarithmic scale. For each edit distance threshold value, we provide in the right y-axes of (a) and (b) the rate of accepted pairs (out of 100,000 pairs for 10Kbp and out of 74,687 pairs for 100Kbp) by SneakySnake that are passed to Parasail. We present the end-to-end speedup values obtained by integrating SneakySnake with Parasail.

We examine the execution time of SneakySnake, KSW2, and SneakySnake integrated with KSW2 using long sequences, as we show in Fig. 11. KSW2 is a sequence aligner used in minimap2 (Li, 2018), a widely-used read mapper. We run KSW2 as *extz2_sse*, a global alignment implementation that is parallelized using the Intel SSE instructions. KSW2 uses the Z-drop heuristic (Suzuki and Kasahara, 2018) to improve the alignment time. We run both SneakySnake and KSW2 using a single CPU thread (as KSW2 does not support multithreading) and two datasets (10Kbp and 100Kbp). We run SneakySnake with $t = y = (E+500)$. Based on Fig. 11, we make two key observations. (1) Using 10Kbp and 100Kbp, SneakySnake is beneficial even for KSW2, a parallelized sequence aligner that uses heuristics. SneakySnake makes KSW2 significantly faster (by 8.2-64.1× and by 3.8-60.6×, respectively) than KSW2 alone in detecting dissimilar pairs of long sequences. (2) As the number of input sequence pairs passing SneakySnake increases up to 68% and 73% of the input sequence pairs of 10Kbp and 100Kbp, respectively, the benefits of integrating SneakySnake with KSW2 in reducing the end-to-end execution time reduces.

We conclude that SneakySnake supports filtering long sequence pairs and its performance scales well over a wide range of edit distance thresholds and sequence lengths.



(a) **(b)**

Fig. 11: The execution time of SneakySnake, KSW2, and SneakySnake integrated with KSW2 using long sequences, (a) 10Kbp and (b) 100Kbp, and a single CPU thread. The left y-axes of (a) and (b) are on a logarithmic scale. For each edit distance threshold value, we provide in the right y-axes of (a) and (b) the rate of accepted pairs (out of 100,000 pairs for 10Kbp and out of 74,687 pairs for 100Kbp) by SneakySnake that are passed to KSW2. We present the end-to-end speedup values obtained by integrating SneakySnake with KSW2.

10.4. Evaluating Accuracy, Resource Analysis, and Execution Time of Snake-on-Chip

We examine 1) the number of sequence pairs that are accepted/rejected by Snake-on-Chip using 100bp_1 and 100bp_2 datasets, 2) the FPGA resource utilization for the hardware implementation of Snake-on-Chip, and 3) the execution time of Snake-on-GPU.

We build the FPGA implementation of Snake-on-Chip using a sub-matrix width of 8 columns (t=8) and we include 3 module instances in the design. Table 5 lists the number of accepted and rejected sequence pairs by Snake-on-Chip using the 100bp_1 and 100bp_2 datasets. We observe that Snake-on-Chip filters out 16.3% (using 100bp_1 and E=10) to 99.99% (using 100bp_2 and E=0) of input sequence pairs. This leads to a significant savings in sequence alignment time, as we show in Section 3.4. We comprehensively analyze and evaluate different sub-matrix widths in "*Effect of y & t on SneakySnake*" Excel sheet in https://github.com/CMU-SAFARI/SneakySnake/tree/master/Evaluation%20Results).

Table 5: Number of accepted and rejected sequence pairs by Snake-on-Chip for a sequence length of 100 and under edit distance thresholds (E) of E= 0% up to E= 10% of the sequence length. We use 100bp_1 and 100bp_2 datasets.

	100bp_1			100bp_2		
***E* (%)**	**Accepted**	**Rejected**	**Filtering Rate (%)**	**Accepted**	**Rejected**	**Filtering Rate (%)**
0	381'901	29'618'099	98.7270	11	29'999'989	99.9999
1	1'388'240	28'611'760	95.3725	20	29'999'980	99.9999
2	3'491'611	26'508'389	88.3613	25	29'999'975	99.9999
3	6'187'022	23'812'978	79.3766	29	29'999'971	99.9999
4	8'926'539	21'073'461	70.2449	40	29'999'960	99.9999
5	11'542'855	18'457'145	61.5238	126	29'999'874	99.9996
6	14'266'733	15'733'267	52.4442	480	29'999'520	99.9984
7	17'056'251	12'943'749	43.1458	1'805	29'998'195	99.9940
8	20'023'178	9'976'822	33.2561	6'078	29'993'922	99.9797
9	22'763'290	7'236'710	24.1224	17'109	29'982'891	99.9430
10	25'091'831	4'908'169	16.3606	40'697	29'959'303	99.8643

We examine the FPGA resource utilization for the hardware implementation of GateKeeper, Shouji, MAGNET, and Snake-on-Chip pre-alignment filters. We evaluate our four pre-alignment filters using a single FPGA chip, the Xilinx VC709 (Xilinx, 2013). We use 60 million sequence pairs, each of which is 100 bp long, from 100bp_1 and 100bp_2. We provide several hardware designs for two commonly used edit distance thresholds, 2 bp and 5 bp, for a sequence length of 100 bp. The VC709 FPGA chip contains 433,200 slice LUTs (look-up tables) and 866,400 slice registers (flip-flops). Table 6 lists the FPGA resource utilization for a single filtering unit. We make five main observations. (1) The design for a single MAGNET filtering unit requires about 10.5% and 37.8% of the available LUTs for edit distance thresholds of 2 bp and 5 bp, respectively. Hence, MAGNET can process 8 and 2 sequence pairs concurrently for edit distance thresholds of 2 bp and 5 bp, respectively, without violating the timing constraints of our hardware accelerator. (2) The design for a single Shouji filtering unit requires about 15×-21.9× fewer LUTs compared to MAGNET. This enables Shouji to achieve more parallelism over MAGNET as Shouji can have 16 filtering units within the same FPGA chip. (3) GateKeeper requires about 26.9×-53× and 1.7×-2.4× fewer LUTs compared to MAGNET and Shouji, respectively. GateKeeper can also examine up to 16 sequence

pairs at the same time on the same FPGA chip. (4) Snake-on-Chip requires 15.4×-26.6× fewer LUTs compared to MAGNET. While Snake-on-Chip requires slightly fewer LUTs compared to Shouji, it requires about 2× more LUTs compared to GateKeeper. Snake-on-Chip can also examine up to 16 sequence pairs concurrently on the same FPGA chip. (5) We observe that the hardware implementations of Shouji, MAGNET, and Snake-on-Chip require pipelining the design (i.e., shortening the critical path delay of each processing core by dividing it into stages or smaller tasks) to meet the timing constraints (the operating frequency of the accelerator is 250 MHz) and achieve more parallelism. Although we use at most 16 filtering units for GateKeeper, Shouji, and Snake-on-Chip, the Xilinx VC709 chip can still accommodate more filtering units for these three filters. However, we observe that the number of filtering units is limited by the maximum data throughput that can supply inputs to the filtering units, which is nearly 3.3 GB/s (13.3 billion bases per second) as provided by the RIFFA communication channel that feeds data into the FPGA (Jacobsen *et al*., 2015).

Table 6: FPGA resource usage for a single filtering unit of GateKeeper, Shouji, MAGNET, and Snake-on-Chip for a sequence length of 100 and under different edit distance thresholds (*E*).

	***E* (%)**	**Slice LUT**	**Slice Register**	**No. of Filtering Units**
GateKeeper	2	0.39%	0.01%	16
	5	0.71%	0.01%	16
Shouji	2	0.69%	0.08%	16
	5	1.72%	0.16%	16
MAGNET	2	10.50%	0.80%	8
	5	37.80%	2.30%	2
Snake-on-Chip	2	0.68%	0.16%	16
	5	1.42%	0.34%	16

We also analyze the execution time of our hardware pre-alignment filters, GateKeeper, MAGNET, Shouji, and Snake-on-Chip. For a single filtering unit, each of the four pre-alignment filters takes about 0.7233 seconds to complete examining `100bp_1` and `100bp_2`, regardless the edit distance threshold used (we test it for E = 0% to 5% of the sequence length). This is because these hardware architectures utilize a 250 MHz clock signal that synchronizes the entire computation. That is, increasing the edit distance threshold directly increases the number of HRTs for each SNR subproblem but does not necessarily increase the execution time as the FPGA provides a large number of LUTs that operate in parallel. Increasing the edit distance threshold is only limited by the available FPGA resource and probably the critical path delay. This is clear from the FPGA resource usage that is correlated with the filtering accuracy and the edit distance threshold. For example, the least accurate filter, GateKeeper, occupies the least amount of FPGA resources.

We conclude that Snake-on-Chip requires a reasonably small number of LUTs, which allows us to integrate a large number of filtering units that can examine a large number of sequence pairs in parallel.

10.5. Evaluating Execution Time and Accuracy of Snake-on-GPU

We examine 1) the end-to-end filtering time of Snake-on-GPU and 2) the number of sequence pairs that are accepted/rejected using 100bp_1 and 100bp_2 datasets. We use *cudaEventElapsedTime()* function to measure the total execution time (i.e., end-to-end filtering time), which we provide in Table 7. We make two key observations. 1) Snake-on-GPU filters out 13.3% (using 100bp_1 and E=10) to 99.99% (using 100bp_2 and E=0) of input sequence pairs. This leads to a significant savings in sequence alignment time, as we show in Section 3.4. 2) Host-GPU data transfer (sending the sequence pairs from the host to the GPU and receiving back the filtering results from the GPU) consumes 72% (using 100bp_1 and E=10) to 85% (using 100bp_2 and E=0) of the end-to-end filtering time.

Table 7: Breakdown of Snake-on-GPU end-to-end filtering time (in seconds) and number of accepted and rejected sequence pairs by Snake-on-GPU, using NVIDIA GeForce RTX 2080Ti card, under different edit distance thresholds (E). We use 100bp_1 and 100bp_2 with a sequence length of 100 bp.

Dataset	*E (%)*	Computation Time (sec)	Data Transfer Time (sec)	End-to-End Filtering Time (sec)	Accepted	Rejected	Filtering Rate (%)
100bp_1	**0**	0.0903	0.4818	**0.5722**	653'408	29'346'106	97.8204
	1	0.1004	0.4529	**0.5534**	2'065'683	27'932'871	93.1096
	2	0.1050	0.4530	**0.5581**	4'665'768	25'331'194	84.4373
	3	0.1097	0.4558	**0.5655**	7'601'344	22'393'785	74.6460
	4	0.1173	0.4519	**0.5692**	10'460'264	19'533'122	65.1104
	5	0.1251	0.4529	**0.5781**	13'202'659	16'789'361	55.9645
	6	0.1320	0.4597	**0.5918**	16'029'917	13'960'784	46.5359
	7	0.1579	0.6049	**0.7628**	18'836'982	11'152'303	37.1743
	8	0.1560	0.5354	**0.6914**	21'604'033	8'383'825	27.9461
	9	0.1681	0.4727	**0.6408**	24'019'045	5'967'465	19.8916
	10	0.1815	0.4636	**0.6451**	25'994'473	3'990'988	13.3033
100bp_2	**0**	0.0877	0.4900	**0.5777**	11	29'999'989	99.9999
	1	0.1002	0.4533	**0.5535**	22	29'999'978	99.9999
	2	0.1017	0.4518	**0.5534**	29	29'999'971	99.9999
	3	0.1024	0.4483	**0.5507**	34	29'999'966	99.9999
	4	0.1047	0.4494	**0.5540**	61	29'999'939	99.9998
	5	0.1080	0.4492	**0.5572**	292	29'999'708	99.9990
	6	0.1078	0.4548	**0.5626**	1'287	29'998'713	99.9957
	7	0.1324	0.6449	**0.7773**	4'233	29'995'767	99.9859
	8	0.1233	0.5221	**0.6453**	12'039	29'987'961	99.9599
	9	0.1302	0.4522	**0.5824**	30'176	29'969'824	99.8994
	10	0.1393	0.4537	**0.5931**	68'791	29'931'209	99.7707

10.6. Key Differences Between Snake-on-Chip and Snake-on-GPU

We summarize the differences between Snake-on-Chip and Snake-on-GPU in terms of 1) their ability to configure the parameter values with minimal changes, 2) energy efficiency of FPGA compared to GPU, 3) their portability from implementation on the same FPGA or GPU system architecture to implementation on another FPGA or GPU system with minimal code changes, 4) their scalability with edit distance threshold, 5) typical design effort required, 6) the market cost of a powerful FPGA compared to a powerful GPU. We provide the summary of these six key differences in Table 8. We observe that both Snake-on-Chip and Snake-on-GPU have their unique pros and cons and hence deciding on which hardware accelerator to use is left to the user's preferences and design goals.

Table 8: A summary of the key differences between Snake-on-Chip and Snake-on-GPU.

	Snake-on-Chip (FPGA)	**Snake-on-GPU (GPU)**	**Explanation**
Parameter Configurability	✗	✓	– Snake-on-Chip requires changing the architecture at design time for each different parameter (e.g., edit distance threshold, E, and the width, t, of each subproblem) value. – Snake-on-GPU is fully configurable at compile-time and run-time.
Energy Efficiency	✓	✗	– FPGA is typically more energy-efficient than GPU (Falsafi *et al.*, 2017; Chung *et al.*, 2010; Guo *et al*., 2019).
Portability	✓	✓	– Snake-on-Chip is independent of the specific FPGA-platform as it does not rely on any vendor-specific computing element (e.g., intellectual property cores). – Snake-on-GPU is independent of the specific CUDA-supported device.
Scalability	✓	✓	– The performance of Snake-on-Chip and its filtering units depends only on the clock speed and not the filtering speed (as we show in Section 10.4). For example, increasing the edit distance threshold directly increases the number of HRTs for each SNR subproblem but does not necessarily increase the execution time as the FPGA provides a large number of LUTs that operate in parallel. This makes the scalability of Snake-on-Chip to high edit distance thresholds or long sequences dependent on *only* the available FPGA resources (and probably the critical path delay) that can accommodate more filtering units.

			– The scalability of Snake-on-GPU is determined by the number of threads that can work concurrently. This makes it dependent on the filtering speed (i.e., how early a pair of sequences can be deemed dissimilar) of each thread. – Given that FPGA has a large number (typically few millions) of LUTs and GPU has a large number (typically few thousands) of threads, we can consider both Snake-on-Chip and Snake-on-GPU scalable with edit distance threshold (as we also experimentally evaluate in Sections 10.4 and 10.5).
Design Effort	✗	✓	– Snake-on-Chip requires a longer design time and more design effort than Snake-on-GPU.
Cost	✗	✓	– FPGA is usually more expensive than GPU, for example, Xilinx VC709 (Xilinx 2013) is 3.6x more expensive than NVIDIA GeForce RTX 2080Ti (NVIDIA 2019b).

References:

1000 Genomes Project Consortium. (2015). A global reference for human genetic variation. *Nature*, **526**(7571), 68-74.

Alkan, C., Kidd, J. M., Marques-Bonet, T., Aksay, G., Antonacci, F., Hormozdiari, F., Kitzman, J. O., Baker, C., Malig, M. and Mutlu, O. (2009). Personalized copy number and segmental duplication maps using next-generation sequencing, *Nature genetics*, **41**, 1061-1067.

Alser, M., Hassan, H., Xin, H., Ergin, O., Mutlu, O., and Alkan, C. (2017a). GateKeeper: A new hardware architecture for accelerating pre-alignment in DNAshort read mapping. *Bioinformatics*, **33**(21), 3355–3363.

Alser, M., Mutlu, O., and Alkan, C. (2017b). MAGNET: Understanding and improving the accuracy of genome pre-alignment filtering. *Transactions on Internet Research*, **13**(2), 33–42.

Alser, M., Hassan, H., Kumar, A., Mutlu, O., and Alkan, C. (2019). Shouji: A fast and efficient pre-alignment filter for sequence alignment. *Bioinformatics*, **35**(21), 4255–4263.

Alser, M., Bingöl, Z., Cali, D. S., Kim, J., Ghose, S., Alkan, C., and Mutlu, O. (2020a). Accelerating genome analysis: A primer on an ongoing journey. *IEEE Micro*, **40**(5), 65–75.

Alser, M., Rotman, J., Taraszka, K., Shi, H., Baykal, P. I., Yang, H. T.,Xue, V., Knyazev, S., Singer, B. D., Balliu, B.,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**(1), 238.

Chu, Chris, and Wong, Yiu-Chung (2007). FLUTE: Fast lookup table based rectilinear steiner minimal tree algorithm for VLSI design, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **27**(1), 70-83.

Chung, E. S., Milder, P. A., Hoe, J. C., and Mai, K. (2010). Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?. *In 2010 43rd annual IEEE/ACM international symposium on microarchitecture* (pp. 225-236).

Daily, J. (2016). Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC bioinformatics*, **17**(1), 81.

Dimitrakopoulos, G., Galanopoulos, K., Mavrokefalidis, C. and Nikolos, D. (2008). Low-power leading-zero counting and anticipation logic for high-speed floating point units, *IEEE transactions on very large scale integration (VLSI) systems*, **16**, 837-850.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, **1**(1), 269-271.

Falsafi, B., Dally, B., Singh, D., Chiou, D., Joshua, J. Y., and Sendag, R. (2017). FPGAs versus GPUs in data centers. *IEEE Micro*, **37**(1), 60-72.

Ghose, S., Boroumand, A., Kim, J. S., Gómez-Luna, J., and Mutlu, O. (2019). Processing-in-memory: A workload-driven perspective. *IBM Journal of Researchand Development*, **63**(6), 3–1

Gómez-Luna, J., González-Linares, J. M., Benavides, J. I. and Guil, N. (2012). Performance models for asynchronous data transfers on consumer graphics processing units, *Journal of Parallel and Distributed Computing*, **72**, 1117-1126.

Guo, L., Lau, J., Ruan, Z., Wei, P., and Cong, J. (2019). Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between FPGA and GPU. *In 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 127-135).

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, **4**(2), 100-107.

Jacobsen, M., Richmond, D., Hogains, M., & Kastner, R. (2015). RIFFA 2.1: A reusable integration framework for FPGA accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, **8**(4), 1-23.

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

Mutlu, O., Ghose, S., Gómez-Luna, J., and Ausavarungnirun, R. (2019). Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems*, **67**, 28–41.

NVIDIA (2019a). CUDA C programming guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

NVIDIA (2019b). NVIDIA GeForce RTX 2080 Ti user guide.

Ono, Y., Asai, K., andd Hamada, M. (2013). PBSIM: PacBio reads simulator—toward accurate genome assembly. *Bioinformatics*, **29**(1), 119-121.

Roy, Jarrod A., and Markov, Igor L. (2008). High-performance routing at the nanometer scale, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **27.6**, 1066-1077.

Schmidt, M., Heese, K., and Kutzner, A. (2019). Accurate high throughput alignment via line sweep-based seed processing. *Nature Communications*, **10**(1), 1939

Suzuki, H., and Kasahara, M. (2018). Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC bioinformatics*, **19**(1), 33-47.

Šošić, M. and Šikić, M. (2017). Edlib: A C/C++ library for fast, exact sequence alignment using edit distance, *Bioinformatics*, **33**, 1394-1395.

Xilinx (2013). Virtex-7 XT VC709 connectivity kit.

Xin, H., Greth, J., Emmons, J., Pekhimenko, G., Kingsford, C., Alkan, C., and Mutlu, O. (2015). Shifted Hamming Distance: A fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping. *Bioinformatics*, **31**(10), 1553–1560.