IEEE Access
Multidisciplinary ⁝ Rapid Review ⁝ Open Access Journal

Received January 24, 2022, accepted February 14, 2022, date of publication February 21, 2022, date of current version March 3, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3153032

FPGA Acceleration of Pre-Alignment Filters for Short Read Mapping With HLS

DAVID CASTELLS-RUFAS[1], SANTIAGO MARCO-SOLA[1,2], JUAN CARLOS MOURE[1], QUIM AGUADO[1], AND ANTONIO ESPINOSA[1]

[1]Departament d'Arquitectura de Computadors i Sistemes Operatius, Universitat Autònoma de Barcelona, 08193 Cerdanyola del Vallès, Spain
[2]Barcelona Supercomputing Center, 08034 Barcelona, Spain

Corresponding author: David Castells-Rufas (david.castells@uab.cat)

This research was supported by the European Union Regional Development Fund (ERDF) within the framework of the ERDF Operational Program of Catalonia 2014-2020 with a grant of 50% of the total cost eligible under the Designing RISC-V based Accelerators for next generation computers project (DRAC) [001-P-001723], in part by the Catalan Government under Grant 2017-SGR-313 and Grant 2017-SGR-1624, and by the Spanish Ministry of Science, Innovation and Universities under Grants PID2020-113614RB-C21 and RTI2018-095209-B-C22.

ABSTRACT Pre-alignment filters are useful for reducing the computational requirements of genomic sequence mappers. Most of them are based on estimating or computing the edit distance between sequences and their candidate locations in a reference genome using a subset of the dynamic programming table used to compute Levenshtein distance. Some of their FPGA implementations of use classic HDL toolchains, thus limiting their portability. Currently, most FPGA accelerators offered by heterogeneous cloud providers support C/C++ HLS. In this work, we implement and optimize several state-of-the-art pre-alignment filters using C/C++ based-HLS to expand their portability to a wide range of systems supporting the OpenCL runtime. Moreover, we perform a complete analysis of the performance and accuracy of the filters and analyze the implications of the results. The maximum throughput obtained by an exact filter is 95.1 MPairs/s including memory transfers using 100 bp sequences, which is the highest ever reported for a comparable system and more than two times faster than previous HDL-based results. The best energy efficiency obtained from the accelerator (not considering host CPU) is 2.1 MPairs/J, more than one order of magnitude higher than other accelerator-based comparable approaches from the state of the art.

INDEX TERMS Field programmable gate arrays, hardware, acceleration, OpenCL, bioinformatics, sequence alignment, pre-alignment filters, read mapping.

I. INTRODUCTION

More than a decade after the irruption of Next-Generation Sequencing (NGS, [1], [2]), genetic sequencing has become an indispensable tool in current medical practice and it is expected to be even more important in the future. DNA is present in all living organisms and consists of long sequences of nucleotides which play a fundamental role in biology. Pseudo-living organisms like viruses contain similar RNA sequences which are also fundamental for their replication. The lengths of DNA and RNA sequences range from the 3 kbp (kilo-base pairs) of the RNA from MS2 virus to the 150 Gbp of the DNA from Paris Japonica plant. Human genome is around 3 Gbp.

Current sequencing technologies are not able to extract the complete genome of complex organisms in one sequence but just a large set of small subsequences from them, called reads. Hence, a post-processing step is required to assemble multiple reads from the sequencing of the same sample into a complete genomic sequence. When the expected complete sequence is unknown, this process is called De novo assembly. On the other hand, when there is an existing reference genome, the process consists in mapping the obtained reads to their best matching locations in the reference genome. Bioinformatics applications that solve this problem are called *mappers*. The completion of the first human reference genome [3] in 2003 was a remarkable achievement that triggered the mass adoption of genomic analysis and the continuous demand for faster sequence alignment methods.

The evolution of sequencing technology has derived into two main classes of systems: short-read and long-read sequencing. Short-read sequencers typically produce reads of a few hundred base pairs long and have a very high

The associate editor coordinating the review of this manuscript and approving it for publication was Leonel Sousa.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

throughput, while long-read sequencers are slower but can produce reads longer than thousands of base pairs with often higher sequencing error rates. In this work, we address short read sequencing.

There are some strategies to map sequences into a reference genome [4] based on the fact that all individuals from the same species share almost all the genome except a very small percentage due to mutations and evolution. Read mappers also have to consider sequencing errors introduced by sequencers, which are usually higher in long-read sequencers. Considering the differences between the reads and the reference, the mapping cannot be based on exact string matching. The alternative is to use approximate string matching algorithms to compute an estimation of the distance between the strings being compared (the lower the better) or their alignment or similarity score (the higher the better).

Most frequently used distance, similarity, or alignment algorithms (such as Levenshtein distance [5], Smith & Waterman [6], Needleman & Wunsch [7]) are based on dynamic programming and have a quadratic complexity with respect to the read length. Smith & Waterman gap-affine is often assumed to be the best accurate in biological terms. Other heuristic methods (such as BLAST [8]) are less accurate in presence of insertion and deletion operations. There have been many efforts to reduce the complexity of such algorithms (including our recent optimization [9]), however, the computational cost of approximate matching is still super-linear. Due to the excessive computational cost of a linear approximate search on the whole sequence, a solution combining exact string matching and approximate string matching if often used. The most popular method is seed-and-extend, which is implemented by mappers like Bowtie2 [10], BWA-MEM [11], Minimap2 [12], or GEM [13]. In this approach, some short subsequences, called seeds, are extracted from the reads. The seeds are used to search for exact matches in the reference. Many candidate locations are obtained from this first step, but the number is orders of magnitude smaller than the original possible locations. Then, the whole read is compared against the extended search context around the candidate locations using approximate matching. The locations from all candidates that match with a lower distance than a given threshold are finally selected as possible mappings.

The drawback of this solution is that multiple candidates are evaluated for every read, and almost all of them (except the one finally selected) are discarded, incurring in a considerable computational cost that goes to waste. There are some proposals to reduce part of this excessive computation defining a new sequence processing stage commonly known as *pre-alignment filtering*. These filters quickly discard pairs of sequences that are too different and avoid computing the approximate distance between them.

Given two sequences T and P, pre-alignment filters are based on the fact that computing the equation $distance\,(T, P) > th$ can be significantly simpler than computing $d = distance(T, P)$. Some pre-alignment filters implement an approximated estimation (inexact) of the previous expression, while others provide an exact computation. In any case, the algorithmic simplicity of pre-alignment filters and the data parallelism of the large number of candidates' evaluation makes them very adequate for their implementation in heterogeneous computing accelerators such as GPUs and FPGAs.

The main contributions of this paper are: 1) OpenCL implementions of three state-of-the-art FPGA-based inexact pre-alignment filters (SHD, Shouji, and Sneaky-Snake) and two exact pre-alignment filters (Banded-Lev and Banded-Myers); 2) a frequency-domain analysis of the response of pre-alignment filters; and 3) an analysis of the relationship between accuracy and performance of pre-alignment filters. We present revisions of relevant parts of the algorithms and optimize the filters to several read lengths (100, 150, 300) that are illustrative short-read NGS. All filters are tested in multiple FPGA accelerator boards comparable to those found in many Data Center environments. The results are extensible to other accelerating platforms from different manufacturers as long as they support OpenCL. The resulting designs achieve the highest filtering performance and energy efficiency reported to date.

The paper is organized as follows. In section 2, we describe some basic concepts used along the work. In section 3, we review the more relevant state-of-the-art pre-alignment filters having a FPGA implementation. In section 4, we describe some optimizations, and the implementation of all the filters using HLS. In section 5, we analyze the results of all the kernels, both from the performance and filtering accuracy perspectives. Finally, in section 6, we close with the concluding remarks.

II. BACKGROUND

The enormous interest of genomic analysis, its introduction as part of the regular medical practice, and the continuous price reduction of sequencing machines has produced an enormous increase in the data loads processed by genomic labs. In this context, the acceleration of all the processes involved in the analysis is fundamental to continue the mass deployment of the technology ([14]). Several pre-alignment filters have been proposed during the last decade to accelerate the genomic toolchain ([15]–[23]). They are mostly based on computing an estimation of the Levenshtein distance [5].

Before describing the algorithms in detail, we should introduce the used nomenclature to avoid any possible confusion. We will denote the input pair of base sequences as T and P, for text and pattern respectively. We assume that T will be a part from the reference genome and P a read obtained by NGS. Let the i-th base of sequence T be denoted as $t[i]$. The same concept can be rewritten as $T = \{t\,[1]\,, \ldots t\,[n]\}$. We denote the subsequence of the sequence T from index i to index j as $t\,[i:j] = \{t\,[i]\,, t[i+1], \ldots t\,[j]\}$. We use the operator $\wedge$ for *bitwise and* operation, and $\vee$ for *bitwise or* operation. To describe a *bitwise or* operation on multiple indexed expressions we write $\vee_{i=1}^{n}$ instead of using the sum symbol $\sum_{i=1}^{n}$ to avoid confusing it with the arithmetic sum of bits. We use the operator $\oplus$ to denote the xor function. When used between two sequences, $\oplus$ will denote the base-wise xor function, which is 0 if the two bases are equal and 1 otherwise. We will use the operator $\ll$ to denote a *shift left* operation. To avoid using a different operator for *shift right* we assume that $\ll$ using a negative index is equivalent to a *shift right* operation. The Levenshtein distance between the pair T and P will be denoted by $d_{Lev}(P, T)$.

In sequence alignment there are three possible alignment scopes: global, semi-global, and local. For distance estimation we are interested in the global and semi-global scopes. When using the Levenshtein distance, and assuming T is longer than P, the global scope refers to compute the distance between the two sequences $d_{global}(P, T) = d_{Lev}(P, T)$. On the other hand, the semi-global scope refers to find the substring of T with minimum distance to P, as defined in (1).

$$d_{semi-global}(P, T) = \min_{\forall T' \subset T} d_{Lev}(P, T') \tag{1}$$

One important factor to consider in genome mappers is how the seed extension is performed. As described in [24], different scopes (global, semi-global, or local) can yield different sensitivities. In any case sequence similarity is often preferred over distance estimation although distance and alignment computations are closely related and one can often be reformulated as the other [25]. Most FPGA-accelerated pre-alignment filters work with Levenshtein distance at the global scope, hence comparing strings of the same length.

A. ACCURACY AND PERFORMANCE

The performance and accuracy of the filters have an impact on the global application performance. A careful analysis of the key performance drivers is required because the execution speed of the filter is not the only aspect affecting the execution time of a mapper. The accuracy of the filter also has an impact on the overall execution time. Burkhardt [26] already did some analysis of the performance drivers of the filters. The execution time for read mappers is affected by Amdahl's law [27] and its implications [28]. We will try to shed some light on the question.

The goal of pre-alignment filters is to quickly discard the sequence pairs above a certain error threshold *th*. This requires to classify each pair of sequences in two classes: the pairs with a difference below the threshold and the rest. In a set of M pairs $\mathcal{M} = \{T_i, P_i\} | i \in [1, M]$ a pair is considered positive if the distance between the pair is below the threshold, $\mathcal{P} = \{T_i, P_i\} \in \mathcal{M} | d_{Lev}(T_i, P_i) < th$. Otherwise it is considered negative, $\mathcal{N} = \{T_i, P_i\} \in \mathcal{M} | d_{Lev}(T_i, P_i) \geq th$. Let's denote N as the number of elements of the set $\mathcal{N}$, $N = |\mathcal{N}|$, and P the number of elements of the set $\mathcal{P}$, $P = |\mathcal{P}|$.

If the filter misclassifies a positive pair as a negative one, we get a false negative. Note that a false negative is a pair with an error below the threshold (i.e. a good candidate to be part of the final alignment) that will be discarded. This is usually non acceptable. Since we must not avoid any good candidate, we should aim to have zero false negatives. If the filter function is defined as a distance estimator d_{est} we can formally define false negatives as $\mathcal{FN} = \{T_i, P_i\} \in \mathcal{P} | d_{est}(T_i, P_i) \geq th$, and its count as $FN = |\mathcal{FN}|$. The only way to ensure that $FN = 0$ is ensuring that (2) holds true.

$$d_{est}(T_i, P_i) \leq d_{Lev}(T_i, P_i) \tag{2}$$

If the filter misclassifies a negative pair as a positive we get a false positive, which is a pair with more errors than the threshold that should have been discarded but has been not. This is undesirable but still acceptable because the algorithm will compute the fine-grain pairwise alignment on all selected positives, and these values will end up being filtered out in that next phase. The problem with having false positives is that they will generate useless computation. Again, we can formally define false positives as $\mathcal{FP} = \{T_i, P_i\} \in \mathcal{N} | d_{est}(T_i, P_i) < th$, and its count as $FP = |\mathcal{FP}|$.

To be independent on the number of elements, classifiers are usually characterized by their false positive and false negative rates ($FP_{rate} = FP/N$, $FN_{rate} = FN/P$). The impact in the performance of the filter accelerator should be analyzed as a speedup factor, i.e. the relation between the time taken by the function with and without the accelerator in place. In this case, if no pre-alignment filter is used the execution time of the extension phase of a mapper will be given by equation 3, where T_{pair} is the time to execute a dynamic programing based distance operation.

$$T_{orig} = T_{pair}(P + N) \tag{3}$$

When using a pre-alignment filter, the execution time of the filtering and extension phases of a mapper will be determined by equation 4, where T_{pre} is the time to execute the pre-alignment filter error estimation on a pair of sequences. Basically, T_{acc} considers that all pairs have to be evaluated by the filter, but only the positives and the false positives will be evaluated by the dynamic programming distance operation in the next phase.

$$T_{acc} = (P + N) \cdot T_{pre} + (P + N \cdot FP_{rate}) \cdot T_{pair} \tag{4}$$

The speedup factor provided by the accelerator in the filter and extend phases will be derived from (3) and (4) as (5).

$$SF_{align} = \frac{T_{orig}}{T_{acc}} = \frac{T_{pair}(P + N)}{(P + N) \cdot T_{pre} + (P + N \cdot FP_{rate}) \cdot T_{pair}} \tag{5}$$

The speedup factor achieved by the accelerated version of the pre-alignment filter with respect to a distance computation can be defined as $SF_{pre} = T_{pair}/T_{pre}$. For analysis convenience we can also define the rate of negatives in the input dataset as $N_{rate} = N/(N + P)$. Using both relative terms, we can find another expression (6) which is conveniently independent on the absolute N and P values.

$$SF_{align} = \frac{SF_{pre}}{1 + SF_{pre}((FP_{rate} - 1) N_{rate} + 1)} \tag{6}$$

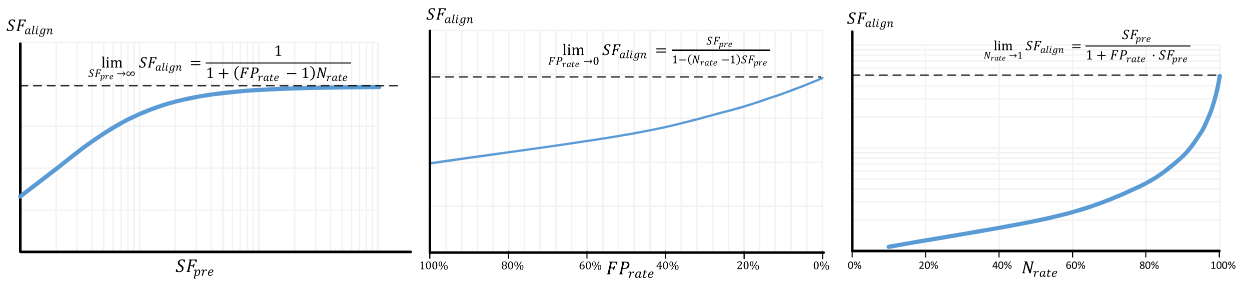


FIGURE 1. Ideal speedu-factor profiles of filter-and-extend phases of read mapping depending on various factors: (left) acceleration of the pre- filtering phase SF_{pre} w.r.t. CPU, (center) false positive rate of the filter FP_{rate}, and (right) rate of negative samples in the input set N_{rate} (right).

Figure 1 illustrates how the three variables (SF_{pre}, FP_{rate}, N_{rate}) affect the final speedup factor. We can see than increasing SF_{pre} has a logarithmic effect on total speedup, this means, that improving the speedup of the filter has a limited effect as we approach the limit imposed by the other variables. The reduction of FP_{rate} has a more linear impact on the speedup. Finally, N_{rate} shows a super-linear speedup; that is, the more negatives the input set contains, the more visible are the benefits of using the filter to discard them.

III. PREVIOUS WORK

The dynamic programming approach to compute the Levenshtein distance is often perceived as too slow and not scalable enough for fast evaluation of the similarity between two strings. However, some implementations of the Levenshtein distance have been successfully implemented in FPGAs achieving high performance like [29] and [30].

One of the simplest methods to compare a pair strings T and P of same length is by using the Hamming distance, which can be defined as $d_{hamming}(T, P) = \sum_{\forall i} t[i] \oplus p[i]$. This distance operator is useful to detect substitution differences and can be executed in one clock cycle, but it overestimates differences caused by insertions and deletions. This limitation is often overcome in some pre-alignment filters (like [16], [17], [20], [22]) by creating shifted replicas of the pattern. This approach is analyzed in more depth in the following subsections. Another technique used in some pre-alignment filters (like [20]) is the Pigeonhole principle. This principle comes from the observation that if the hamming distance between a pair of strings is $d_{hamming}(T, P) = n$, then if we divide the strings into $n+1$ slots, there must be at least one slot that exactly match between the two strings, otherwise we can be sure that the number of errors is higher than n. An alternative approach used in other pre-alignment filters ([15], [19], [21]) is comparing the number of (the 4 possible) bases present in sequence or the existing combinations of bases, known as q-grams, or k-mers.

In the following subsections we review some of the existing algorithms found in the literature that try to get a fast approximation of the Levenshtein distance and were successfully implemented in FPGA.

We do not analyze the closely related kmer based filters GRIM-Filter [19] and QCKer-FPGA [21]. First, because they are used for the seed phase of the mapper algorithm instead of the extend phase, which is our focus. Second, because GRIM-Filter does not provide an FPGA implementation and the implementation provided by QCKer-FPGA is slower that the equivalent software solution.

A. GATEKEEPER

In [16] Xin realized that Hamming Distance is an efficient method to detect substitution errors, and if shifted appropriately could be also used to detect insertion and deletion errors. If we denote XO_i as the bit vector resulting from the xor between T and P, i.e. $XO_i = T \oplus (P \ll i)$ we can see that matching sequences appear as a run of zeros. Substitutions appear as ones, while insertions and deletions appear as jumps from a runs of zeros a bit vector, to a consecutive run of zeros in another bit vector. This is illustrated in Figure 2, where XO_0 starts with a sequence of 3 zeros. When this sequence ends it is followed by a long sequence of zeros in XO_1 caused by an insertion. The following single 1 is caused by a substitution error and finally a delete error returns the stream of zeros to XO_0. The bit vectors can be organized in a bit matrix.

T	CCAGTTTGCTTCGCGTCACAGCGACTGTTA
$P \ll 2$	AAGTTTGCTTCGCGTAACAGCGACTGTA--
$P \ll 1$	CAAGTTTGCTTCGCGTAACAGCGACTGTA-
P	CCAAGTTTGCTTCGCGTAACAGCGACTGTA
$P \gg 1$	-CCAAGTTTGCTTCGCGTAACAGCGACTGT
$P \gg 2$	--CCAAGTTTGCTTCGCGTAACAGCGACTG
XO_2	111100111011111111111111111111
XO_1	010000000000000010000000000011
XO_0	000111111111111111111111111100
XO_{-1}	101111011111100111101101111011
XO_{-2}	111111111010111101101011011101

FIGURE 2. Analysis of the bit vectors of the xor between T and shifted versions of P.

To be able to analyze the differences (errors) introduced by n insert or delete operations the number of shifted (on both directions) versions must be equal to n, so the total number of needed bit vectors will be $2n+1$. Xin realizes that insertions, deletions and substitutions generate columns in the bit matrix with a large number of ones. In most of the cases the values that are not 1 in a column affected by an error are caused by an accidental matching character, not because of a long sequence of zeros. To remove this effect, he proposes to use a short zero removal step that removes short sequence of zeros (1 or 2 consecutive zeros) from the bit vectors. After this step, all columns are *anded* into an accumulator bit vector. Finally, the number of ones in the accumulator bit vector are counted to get an estimate of the number of errors

We reformulate the SHD algorithm as Algorithm 1, and 2.

Algorithm 1 SHD. Shifted Hamming Distance

Input: P as the pattern sequence, T as the text sequence both of length n, th the threshold so that the $d_{Lev}(P, T) > th$ can be quickly discarded.

Output: The estimated minimum number of errors EE, such that $d_{Lev}(P, T) \geq EE$

```
1 $A \leftarrow 2^n - 1$
2 for all $s \in [-th, th]$ do
3     $SP \leftarrow P \ll s$
4     $XO \leftarrow SP \oplus T$
5     $XO \leftarrow$ RemoveShortZeros($XO$)
6     $A \leftarrow A \wedge XO$
7 end for
8 $EE \leftarrow$ CountOnes($A$)
```

Algorithm 2 RemoveShortZeros. Removes the Short Sequences of Zeros (One or 2 Consecutive Zeros) From a Bit Sequence

Input: A sequence X of n bits.

Output: A sequence R of n bits where the short sequences of zeros of X are removed.

```
1 $A \leftarrow \{1, 1, x[1] \ldots x[n], 1, 1\}$
2 $R \leftarrow X$
3 for $i \leftarrow 3$ to $n + 2$ do
4     if $(a[i-1] = 1 \wedge a[i+1] = 1)$
5         $\vee(a[i-1] = 1 \wedge a[i+2] = 1)$
6         $\vee(a[i-1] = 1 \wedge a[i+2] = 1)$ then
7         $r[i-2] \leftarrow 1$
8 end for
```

The architecture of an FPGA is extremely suitable for this kind of algorithms. Shifting by a fixed amount is an inexpensive operation in FPGAs since it is implemented just by wire connections. The rest of operations are bit-wise ands, `ors` and `xors` and a final `CountOnes` operation. If the read length is small enough, all the loops can be unrolled and all the operations can be all implemented using combinational logic. Such design would take just one clock cycle to execute. The long combinational path can limit the maximum frequency of the system, but this can be solved by introducing pipeline stages.

Alser *et al.* [17] provide a FPGA implementation of the SHD algorithm in GateKeeper. As many works from the same research group they provide an Open Source accelerator tested in a Xilinx board. The design is implemented in Verilog and makes use of the Open Source RIFFA accelerator interface to interact with the host processor. They report a f_{max} of 250 Mhz using the Xilinx Virtex-7 FPGA VC709 Connectivity Kit board. The simplicity of the design allows for its easy replication. The authors propose to replicate up to 140 units to improve the performance of the system.

As we present in more detail in Section 6, the algorithm's accuracy is lower than subsequent proposed filters from the same group, especially in the presence of insertions and deletions and when the error threshold is higher than few edits. As the error threshold is increased, the probability of having runs of zeros in the shifted hamming versions increases, reducing the probability of having ones in the accumulator bit vector, thus, missing some edits.

B. SHOUJI

In [18] Alser *et al.* realize that a key to find the best alignment is to find the longest non-overlapping run of zeros in the shifted xored bit vectors used by SHD. A solution is to find the longest run first, and then expand the best matching zero runs from the edges in both directions in an iterative fashion. However, this iterative approach is not adequate for a FPGA implementation as data dependencies between loop iterations prevent unrolling the loop.

Algorithm 3 Shouji

Input: P as the pattern sequence, T as the text sequence both of length n, th the threshold so the $d_{Lev}(P, T) > th$ are quickly discarded. WS is the window size.

Output: The estimated minimum number of errors (EE), such that $d_{Lev}(P, T) \geq EE$

```
1 $A \leftarrow 2^n - 1$
2 for all $i \in [1, n - WS]$ do
3   $maxz \leftarrow$ CountZeros($a[i : i + WS - 1]$)
4   for all $s \in [-th, th]$ do
5     $SP \leftarrow P \ll s$
6     $XO \leftarrow SP \oplus T$
7     $cz \leftarrow$ CountZeros($xo[i : i + WS - 1]$)
8     if $(cz > maxz) \vee (cz = maxz \wedge xo[i] = 0)$ then
9       $maxz \leftarrow cz$
10      $a[i : i + WS - 1] \leftarrow xo[i : i + WS - 1]$
11  end for
12 end for
13 $EE \leftarrow CountOnes(A)$
```

As an alternative, in a following work [20], the same authors propose a new filter design called Shouji. The concept of the first step of the algorithm is very similar to the previous SHD algorithm, to build a matrix with the xored shifted bit vectors. But in this case, the accumulator bit vector is computed differently. A window of few columns is slided along the matrix selecting the row inside the window with more zeros. The selected row is stored in the accumulator bit vector. If several rows have the same number of zeros the one starting with a zero is selected. The paper concludes that the best window size is 4 bits. We rewrite the algorithm described in [20] to unify the notation as Algorithm 3. We obviate the algorithmic description of count ones and count zeros. Note that CountOnes is included in many Instruction Set Architectures (ISAs) as the popcount instruction, and that CountZeros(x) is equivalent to CountOnes(¬x). Regarding performance and accuracy, the same paper describes a successful implementation of the filter in the VC709 Xilinx board achieving the same single-clock as previous SHD design. The accuracy of the filter is reported to increase with respect to SHD.

C. SNEAKY SNAKE

In SneakySnake [22], the authors change the strategy from selecting the rows of the bit matrix with more zeros (as in Shouji) to selecting the longest consecutive runs of zeros. To avoid selecting overlapping runs of zeros, the search for the next run of zeros must start from the next point after the end of the previously found run of zeros, i.e. after the one ending the run of zeros. The first phase of the algorithm is still shared with previous designs: the rows with the xored shifted bit vectors.

The algorithm has a clear sequential nature, since the search of runs of zeros can only start after finding the finishing point of the previous zero stream. To break this dependency SneakySnake applies a windowing approach. The complete sequence pair is split in windows of limited size. The processing of all the windows to identify the number of edit operations in them can be done in parallel and a final step aggregates the results for every window.

Algorithm 4 SneakySnakes

Input: The pattern sequence P, and text sequence T, both of length n. The error threshold th so that the $ED(P,T) > th$ can be quickly discarded. The window size WS.

Output: The estimated minimum number of errors (EE), such that $ED(P,T) \geq EE$

```
1  $EE \leftarrow 0$
2  for all $s \in [-th, th]$ do
3      $SP \leftarrow P$
4      $XO_s \leftarrow SP \oplus T$
5  end for
6  for $i \leftarrow 1$ to $n/WS$ do
7      $k_{st} \leftarrow \mathrm{WS} \cdot (\mathrm{i}-1)+1$
8      $k_e \leftarrow \mathrm{WS} \cdot \mathrm{i}$
9      while $(k_s < k_e)$ do
10         $lz \leftarrow \max_{\forall s \in [-th, th]} \mathrm{CLZ}(xo_s[k_{st} : k_e])$
11         $c \leftarrow k_e - lz$
12         if $(c < k_e)$ then
13             $EE \leftarrow EE + 1$
14         end if
15         $k_{st} \leftarrow k_{st} + lz + 1$
16     end while
17 end for
```

Since the search in one window has a known limit, the algorithm can be implemented as a loop that can be unrolled considering the worst case scenario. This means that the circuit will speculatively compute all the possible zero stream lengths in a window and finally select the existing one. The analysis in the paper suggest that a window size of 8 bits is a good tradeoff between accuracy and hardware resources utilized. We present the algorithm with the previously used common notation as Algorithm 4. Because it detects better the streams of zeros, SneakySnake has a better accuracy than Shouji. Although is uses more resources than SHD it can still be implemented in a single cycle throughput design for short reads as done in [22] for the VC709 Xilinx board.

D. BANDED-LEV

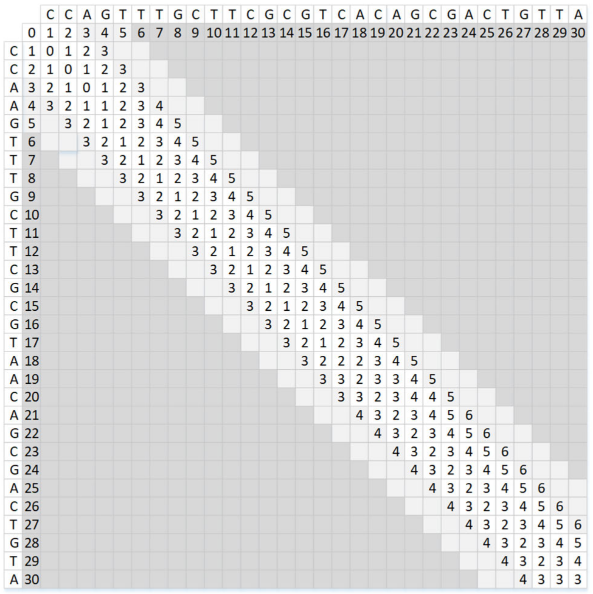
A brute force approach can be used to determine whether the Levenshtein distance of two sequences P and T of the same length n is above a threshold th or not. Using the dynamic programming method, the Levenshtein distance between P and T is given by equation 7 which requires to build a dynamic programming table using equation 8.

$$d_{Lev}(P,T) = D_{n,n} \tag{7}$$

$$D_{i,j} = \begin{cases} i, \text{ if } j = 0 \\ j, \text{ if } i = 0 \\ min \begin{cases} D_{i-1,j} + 1 \\ D_{i,j-1} + 1 \\ D_{i-1,j-1} + (P[i] \oplus T[i]) \end{cases} \end{cases} \tag{8}$$

The implementation of the whole table in a FPGA has usually been dismissed due to the excessive number of resources consumed. However, with the high resource density of current FPGA devices, this is now feasible for small enough sequence lengths. Moreover, additional optimizations can reduce resource usage. One of those optimizations is based on the pruning of cells of the matrix that are irrelevant for the final distance estimation computation. Pre-alignment filters only need to determine whether the edit distance is below or above a certain threshold. Since the dynamic programming matrix is monotonically increasing in the diagonals, once a cell reaches the threshold error the computation of its below diagonal values can be avoided because these values will surely be greater or equal to the threshold. Based on this observation and the known values of row and column 0 it is easily derived that we can prune out the computation of cells that are farther than th from the main diagonal.

But even more cells can be pruned out. When computing the distance between pairs of the same length n, the edit path starts at cell $D_{0,0}$ and must end at cell $D_{n,n}$, which are both part of the main diagonal. Any turn of the edit path in the horizontal axis must be compensated later with a turn in the vertical axis to return to the main diagonal. Similarly, any turn in the vertical axis, must be compensated later with a turn in the horizontal axis. The number of horizontal turns must be equal to the number of vertical turns, and each vertical



		C	C	A	G	T	T	T	G	C	T	T	C	G	C	G	T	C	A	C	A	G	C	G	A	C	T	G	T	T	A
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
C	1	0	1	2	3																										
C	2	1	0	1	2	3																									
A	3	2	1	0	1	2	3																								
A	4	3	2	1	1	2	3	4																							
G	5		3	2	1	2	3	4	5																						
T	6			3	2	1	2	3	4	5																					
T	7				3	2	1	2	3	4	5																				
T	8					3	2	1	2	3	4	5																			
G	9						3	2	1	2	3	4	5																		
C	10							3	2	1	2	3	4	5																	
T	11								3	2	1	2	3	4	5																
T	12									3	2	1	2	3	4	5															
C	13										3	2	1	2	3	4	5														
G	14											3	2	1	2	3	4	5													
C	15												3	2	1	2	3	4	5												
G	16													3	2	1	2	3	4	5											
T	17														3	2	1	2	3	4	5										
A	18															3	2	2	2	3	4	5									
A	19																3	3	2	3	3	4	5								
C	20																	3	3	2	3	4	4	5							
A	21																		4	3	2	3	4	5	6						
G	22																			4	3	2	3	4	5	6					
C	23																				4	3	2	3	4	5	6				
G	24																					4	3	2	3	4	5	6			
A	25																						4	3	2	3	4	5	6		
C	26																							4	3	2	3	4	5	6	
T	27																								4	3	2	3	4	5	6
G	28																									4	3	2	3	4	5
T	29																										4	3	2	3	4
A	30																											4	3	3	3

FIGURE 3. Example of the computation of the Levenshtein distance between two sequences using a banded dynamic programming table. Dark grey cells represent the cells that are higher than the threshold value. Light grey cells represent the cells that are higher than halt of the threshold.

and horizontal turn introduce a cost (error) of 1. Thus, the maximum number of cells that the edit path of a sequence pair P, T with $d_{Lev}(P, T) < th$ can separate from the main diagonal is $\lfloor th/2 \rfloor$. From this observation, we can obtain an analytical expression (equation 9) that determines the number of cells that must be computed.

$$\begin{aligned} c &= (n+1) + 2\sum_{i=1}^{\lfloor th/2 \rfloor}(n+1) - 2i \\ &= n + 2 \cdot \left(\left\lfloor \frac{th}{2} \right\rfloor\right)\left(n - \left\lfloor \frac{th}{2} \right\rfloor\right) \end{aligned} \tag{9}$$

Figure 3 illustrates an example in which we compute the dynamic programming matrix of the Levenshtein distance for two sequences with an error threshold $th = 5$. The computation of all dark grey cells can be avoided since they have a higher value than the threshold. Light grey cells can also be avoided, since going through them and returning to the main diagonal would require a cost higher than the threshold. In this example, the length of the sequences is $n = 30$, so the number of required cells is $c = 30 + 2 \cdot 2 \cdot (30 - 2) = 142$, which is significantly lower than the 961 original cells of the matrix.

Another possible optimization is the reduction of the number of bits of the adders. Once we go above the threshold it is not necessary to keep a good account of the exact value of the error. If the threshold is th, any number above can be used to determine that the pair of sequences are above the threshold. Hence, the dynamic programming table equation can add a saturating term $(th + 1)$ as shown in equation 10.

$$D_{i,j} = \begin{cases} i, \; if \; j = 0 \\ j, \; if \; i = 0 \\ min \begin{cases} D_{i-1,j} + 1 \\ D_{i,j-1} + 1 \\ D_{i-1,j-1} + (P[i] \oplus T[i]) \\ th + 1 \end{cases} \end{cases} \tag{10}$$

E. BANDED-MYERS

The Myers algorithm [31] to compute the Levenshtein distance has been extensively used in parallel implementations including FPGAs [29], [32]. The Myers analysis starts with the observation that a cell from the dynamic programming table can only differ by +1,0,−1 with the preceding cell in the horizontal or vertical axis. In the diagonal axis, the possible differences are only 0, or +1 from the preceding cell of the diagonal. Given this observation the original D table can be expressed with three alternative differences tables: Δv, Δh, Δd, which can be build from the original D table by using the differential equations 11,12, and 13.

$$\Delta v_{i,j} | i > 0 = D_{i,j} - D_{i-1,j} \tag{11}$$
$$\Delta h_{i,j} | j > 0 = D_{i,j} - D_{i,j-1} \tag{12}$$
$$\Delta d_{i,j} | i > 0, j > 0 = D_{i,j} - D_{i-1,j-1} \tag{13}$$

The original D table can easily be reconstructed from any of the three differences tables. The value of any cell $D_{i,j}$ is obtained by taking the first (precomputed) cell of the column, row, or diagonal and aggregating all the elements of the column, row, or diagonal until $D_{i,j}$ is reached. The equation 14 details the three possible ways to obtain a D cell value from any of the differences tables.

$$\begin{aligned} D_{i,j} &= j + \sum_{k=1}^{i} \Delta v_{k,j} = i + \sum_{k=1}^{j} \Delta h_{i,k} \\ &= |i - j| \sum_{k=0}^{\min(i,j)-1} \Delta d_{i-k,j-k} \end{aligned} \tag{14}$$

The great advantage of the Myers algorithm is that it finds an alternative way to build the differences tables (Δv, Δh, or Δd) by using simple boolean expressions without computing D. Once the differences tables are build, a final step is required to recreate one or a small number of cells from the D matrix by using equation 14. The simple binary expressions to create the differences tables are built as follows. First, Myers separates positive and negative increments (equations 15,16, and 17) and then he obtains simple boolean expressions to compute the increments (equations 18,19,20,21, and 22).

$$\Delta v_{i,j} | i > 0 = VP_{i,j} - VN_{i,j} \tag{15}$$
$$\Delta h_{i,j} | j > 0 = HP_{i,j} - HN_{i,j} \tag{16}$$
$$\Delta d_{i,j} | i > 0, j > 0 = 1 - D0_{i,j} \tag{17}$$
$$HN_{i,j} = VP_{i,j-1} \wedge D0_{i,j} \tag{18}$$
$$VN_{i,j} = HP_{i-1,j} \wedge D0_{i,j} \tag{19}$$
$$HP_{i,j} = VN_{i,j-1} \vee \neg(VP_{i,j-1} \vee D0_{i,j}) \tag{20}$$

$$VP_{i,j} = HN_{i-1,j} \vee \neg(HP_{i-1,j} \vee D0_{i,j}) \quad (21)$$

$$D0_{i,j} = \neg(P[i] \oplus T[i]) \vee VN_{i,j-1} \vee HN_{i-1,j} \quad (22)$$

Most software and hardware implementations of the Myers algorithm use a column approach (i.e. computing Δv) to do semi-global matching by taking advantage of vectorized bit vector operations available in many computing systems. For semi-global matching, the first row of the D matrix is initialized to zero (instead of the increasing values used when computing global matching) and all the cells of the last column are computed to find the best alignment (see equation 23).

$$d_{semi-global}(P, T) = \min_{j\in[1,n]} D_{n,j}. \quad (23)$$

Since we address the global matching of sequences of the same length we use a totally different approach. As we proposed in [32], we use the diagonal differences matrix Δd and just aggregate its main diagonal (see equation 24).

$$d_{est}(P, T) = D_{n,n} = \sum_{k=1}^{n} \Delta d_{k,k} \quad (24)$$

Moreover, we know that we can prune out cells that cannot contribute to the final solution. We substitute the previous expressions used to compute $HN_{i,j}$, $VN_{i,j}$, $HP_{i,j}$, and $VP_{i,j}$ by equations 25, 26, 27, and 28. In this step we are hard-coding positive increments in the cells that are far enough from the main diagonal. The motivation of this step is to reduce hardware resources.

$$HN'_{i,j} = \begin{cases} 0, & if\ |i-j| > \lfloor th/2 \rfloor \\ HN_{i,j}, & otherwise \end{cases} \quad (25)$$

$$VN'_{i,j} = \begin{cases} 0, & if\ |i-j| > \lfloor th/2 \rfloor \\ VN_{i,j}, & otherwise \end{cases} \quad (26)$$

$$HP'_{i,j} = \begin{cases} 1, & if\ |i-j| > \lfloor th/2 \rfloor \\ HP_{i,j}, & otherwise \end{cases} \quad (27)$$

$$VP'_{i,j} = \begin{cases} 1, & if\ |i-j| > \lfloor th/2 \rfloor \\ VP_{i,j}, & otherwise \end{cases} \quad (28)$$

Finally, we can also introduce a saturating value in the distance value above which it has no added value to continue computing the exact distance value (equation 29).

$$d_{est}(P, T) = \min(\sum_{k=1}^{n} \Delta d_{k,k}, th + 1) \quad (29)$$

IV. HLS IMPLEMENTATION

In this section we will present the development strategy and the details for all studied filters. Some of the previously analyzed implementations GateKeeper [17], Shouji [20], and SneakySnake [22] are coded in Verilog and tested in the same Xilinx FPGA accelerator. The simplicity of these algorithms and the low level control of the language makes Verilog a good choice to code the filter units. The interface with the host in all implementations is done with RIFFA [33], an Open Source framework to implement the communication infrastructure between hosts and FPGA co-processors. RIFFA was presented in 2015 and is supported in some devices, however its support for future FPGA devices seems unlikely, especially for the systems used by Heterogeneous Cloud providers. On the other hand, Cai [29] already uses an OpenCL approach. Our goal is to create pre-alignment filters that can be integrated into genomic datacenters equipped with FPGA coprocessors. We select OpenCL as a widely available framework for the implementation of FPGA accelerators. The source code of the filters is available at https://github.com/davidcastells/OpenCLPrealignmentFilters

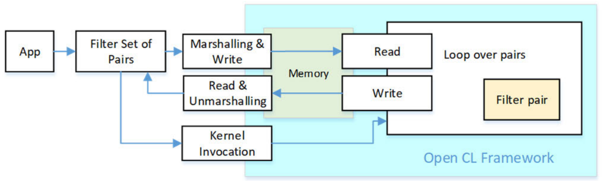


FIGURE 4. Logic diagram of pre-alignment filter accelerators. The same design is common for all filters, only the yellow part (that estimates the difference) is customized for every design.

All pre-alignment filters have the same goal, compare the pattern with a text, and obtain a distance estimation. This can be implemented in C/C++ by functions with the same number of parameters and the same return values. We can take profit of these similarities and implement a common set of functions for all filters both, at the host side and the accelerator side. Figure 4 depicts the logic structure of the approach. Only the filter pair function (in yellow) is changed among different versions of the filters. Many FPGA accelerator boards with OpenCL support provide a 512 bit data bus width to access the device memory. In most cases the device memory is implemented as DDR memory external to the FPGA device, while HBM2 is starting to become available. As we have seen in previous sections pre-alignment filters can work with a single clock throughput. This means that their ideal memory bandwidth requirements would be $512 \times f_{max}$. For instance, a kernel running at 250 MHz would require 16 GBps memory bandwidth. In current systems, an ideal single filtering unit would already saturate the available memory bandwidth of many accelerator boards. However, as higher end FPGA accelerators include multiple memory ports, it might be reasonable to replicate some of them.

FPGA OpenCL kernels are designed in two possible styles: single workitem or NDRange. NDRange kernels resemble GPU kernels, as they are designed to achieve parallelism by running multiple execution threads concurrently. On the other hand, single workitem kernels are based on a single execution thread. In this case parallelism is achieved by unrolling loops or introducing long pipelines. In our case, the regularity of the memory access pattern and the constant execution time of loop iterations makes single workitem style more adequate.

In order to provide a flexible solution that can be used in various applications of a data center, we encapsulate

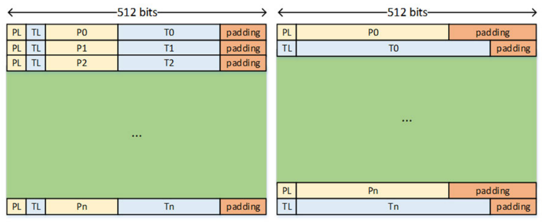


FIGURE 5. Some of the possible organizations of device memory to store the sequence pairs. (left) text and pattern in the same 512 bits word that can be fetched in a single memory transfer operation. (right) text and pattern in consecutive words, requiring two memory transfers to fetch.

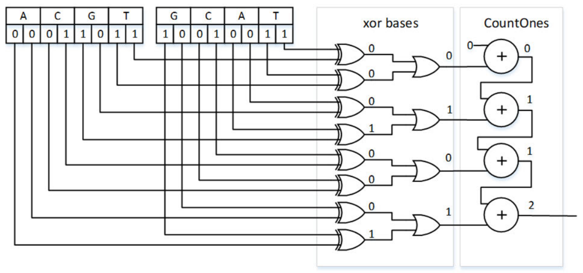


FIGURE 6. Example logic circuit to compute the Hamming distance between two genomic sequences of 4 bases.

the communication between the host software application with the pre-alignment filter executing in the FPGA in a C++ class that is responsible to load the compiled OpenCL binaries, marshal and transfer the sequence pairs to the device memory, invoke the appropriate OpenCL kernels, and collect the results. We use a 2 bit representation of the bases to reduce the amount of memory used, and its associated transfer time. The organization of sequence pairs in the device memory can also have an impact on the filter throughput. A pair of short reads of 100 bases can fit in a single 512 bit word. It seems convenient to place them consecutively so that they can be both fetched in the same clock cycle and processed in a single clock throughput rate. As shown in figure 5, longer reads can be organized in different ways. In any case we consider the use of padding to ensure aligned memory transfers. In addition to the memory organization options, we foresee another source of variability: the implementation can take profit of some significant optimizations if the sequence lengths are known and fixed. On top of that, we can consider the additional parameters used by the specific algorithms. For instance, in the analyzed methods from the literature the threshold value *th* and sequence lengths *n* are parameters that should be fixed which directly influence the resources required by the hardware implementation.

The dimensions of the design space are considerable, and its exploration is expected to be easier with HLS rather than Verilog. One of the reasons for this claim is the existing functional verification that avoids RTL simulation and synthesis to verify the implemented solutions. In this variable scenario, with potential multiple design implementations the flexible circuit loader of OpenCL based on runtime reconfiguration can offer additional benefits. Thus, the application could reconfigure the device to adapt to the different characteristics of the incoming workloads.

A. BASIC OPERATIONS AND ARBITRARY PRECISION INTEGERS

The details of the necessary Load and Store Memory Units (LSUs) that are required to implement the memory access operations are conveniently hidden by HLS frameworks when using the regular C/C++ language pointer semantics. The OpenCL __global, __local, or __private modifiers can be used to specify the use of the DDR memory external to the device, or the on-chip internal memory. Local C variables are usually synthesized as FPGA registers, while local arrays rely on the LLVM optimization phase of the toolchain to decide whether to implement them as on-chip memories or registers. Arbitrary precision integer types are very convenient to define large integers and use them as single variables rather than arrays. This ensures than they will be stored in registers rather than memories, potentially having a big impact on the system performance since LSUs can become a bottleneck that prevent parallel access to the long bit vectors.

Basic operations used by the filters can be described with C/C++ functions and used by the kernels. One fundamental factor to consider is that function invocations are often inlined by the compiler. So, multiple invocations in different parts of the code will end up creating multiple Hardware instances of the equivalent circuit.

The following example illustrates how to implement the hamming distance computation between two sequences of 4 bases. This is a very simple example as the length is very short, but it is still illustrative. Hamming distance between longer strings using OpenCL on FPGA has been addressed in [34]. As depicted in figure 6 the circuit has two phases, the equality check of each base, and the counting of equals. As shown below, the ideal OpenCL implementation of the two phases is straight-forward if using arbitrary precision integer types (`ap_int`).

```
ap_uint<4> xorBases(ap_uint<8> a, ap_uint<8> b)
{
   ap_uint<4> r;
   #pragma unroll
   for (int i=0; i < 4; i++)
   {
         a0 = a.get_bit(i*2);
         a1 = a.get_bit(i*2+1);
         b0 = b.get_bit(i*2);
         b1 = b.get_bit(i*2+1);
         r.set_bit(i, a0^b0 | a1^b1 );
   }
   return r;
}

int countOnes(ap_uint<4> a)
{
   int r = 0;
   #pragma unroll
   for (int i=0; i < 4; i++)
     r += a.get_bit(i);
   return r;
}
```

However, this simple implementation is not possible in some platforms due to the differences in the level of support of the arbitrary precision integer library provided by different FPGA manufactures. While Intel have a very limited support for them, Xilinx only allows their use when using C/C++ kernels instead of the standard OpenCL kernels. There have been some initiatives to overcome these limitations by creating an additional abstraction layer on top of `ap_int` (like HINT [35]) to hide the implementation differences between the different platforms. However, they are not universally supported either in different FPGA families.

To overcome these limitations we use C/C++ kernels in Xilinx platforms and we create our own library of arbitrary precision integers by using metaprogramming [36] for Intel platforms. Instead of using a C pre-processor based metaprogramming approach, we use a syntax similar to Java Server Pages [37] to annotate the parts of the OpenCL source code that must be modified before compilation. For example, the previous countOnes function could be implemented as below.

```
int countOnes(ap_uint<4> a)
{
   int r = 0;
   <%for (int i=0; i < 4; i++){%>
   r += (a ≫  <%printf(``%d'',i);%>) & 1;
   <%}%>
   return r;
}
```

It would be translated to the expanded form (see below) by our metaprogramming infrastructure before compilation.

```
int countOnes(ap_uint<4> a)
{
   int r = 0;
   r += (a ≫ 0) & 1;
   r += (a ≫ 1) & 1;
   r += (a ≫ 2) & 1;
   r += (a ≫ 3) & 1;
   return r;
}
```

In this way, the code can be adapted to an arbitrary number of bit lengths as required. The drawback of this approach is that the readability of the code is seriously affected. We expect that the future evolution of the HLS frameworks makes this step unnecessary.

B. SHIFTED HAMMING DISTANCE

The implementation of the Shifted Hamming Distance design is straight forward in C/C++. Algorithm 1 (page 5) contains a single for loop (line 2) that can be unrolled by using the #pragma unroll clause. Similarly, the removeShortZeros function can also be easily expressed and parallelized by a simple loop unrolling as shown in the following code for a 256 bit vector.

```
ap_uint<256> removeShortZeros(ap_uint<256> x)
{
   ap_uint<256> r;
   #pragma unroll
   for (int i=0; i < 256; i++)
   {
         xm1 = (i>0) ? x.get_bit(i-1): 1;
         xm2 = (i>1) ? x.get_bit(i-2): 1;
         xp1 = (i<254) ? x.get_bit(i+1): 1;
         xp2 = (i<253) ? x.get_bit(i+2): 1;
         xs = x.get_bit(i);
         r.set_bit(xs | (xm1 & xp1) | (xm1 & xp2)
              | (xm2 & xp1));
   }
   return r;
}
```

C. SHOUJI

After an analysis of the algorithm 3 (page 5) we realize that Shouji is working in windows of 4 bits and the values written to the accumulator bit vector (line 10 of Algorithm 1) are also 4 bits vectors. Several iterations of the main loop can update the same accumulator bit vector positions multiple times. We try to generate each output bit independently removing the collisions. Following this idea we realize that every output bit can only depend on the values of the columns of the xored bit matrix ranging from 3 bits before and 3 bits after the computed value.

Algorithm 5 Shouji (Proposed Update)

Input: P as the pattern sequence, T as the text sequence both of length n, th the threshold so that the pairs with $d_{Lev}(P, T) > th$ are quickly discarded. WS is the window size.

Output: The estimated minimum number of errors (EE), such that $d_{Lev}(P, T) \geq EE$

```
 1  $A \leftarrow 2^n - 1$
 2  for all $i \in [1, n]$ do
 3    $maxz \leftarrow 0$
 4    $b \leftarrow 1$
 5    for all j $\in [\max(1, i-WS+1), \min(i, n-WS+1)$
      do
 6      for all $s \in [-th, th]$ do
 7        $SP \leftarrow P \ll s$
 8        $XO \leftarrow SP \oplus T$
 9        $cz \leftarrow$ CountZeros($xo[j : j + WS - 1]$)
10        if $(cz > maxz) \vee (cz = maxz \wedge xo[i] = 0)$ then
11          $maxz \leftarrow cz$
12          $b \leftarrow xo[i]$
13        end if
14      end for
15    end for
16    $a[i] \leftarrow b$
17  end for
18  $EE \leftarrow$ CountOnes($A$)
```

We missed this observation in an initial analysis of the algorithm, but the HLS compiler unveiled it as it was having problems to parallelize the outer loop due to the apparent loop carried dependency in line 10. We recoded the algorithm as Algorithm 5, removing the dependency. Again, the implementation of this algorithm in C/C++ is straight forward with good support for arbitrary precision integers.

D. SNEAKYSNAKES

The implementation of SneakySnakes is a little more complex compared with previous algorithms. The algorithm 4 creates the initial bit matrix in the for loop in line 2. This results in a bidimensional bit-matrix and the OpenCL compiler will tend to place the structure in local memory. A local memory storage of the matrix would become a bottleneck that would prevent the unrolling of the following loops. To solve these issues we compute the matrix values inside the sliding window. The read values is used straight away to compute the necessary intermediate values, removing the need to store the matrix for later use. An additional inconvenient is the use of a `while` loop in line 9. To ease the job of the compiler in trying to unroll the loops we substitute it for a bounded `for` loop.

The original algorithm in [22] uses a tree of comparators and multiplexors to select the longest value from all the parallel `CountLeadingZeros` units working on different rows of the bit matrix window (see figure 7). If we use the bit manipulation instruction `RFILL(x)` to set all the bits that are right to the highest set bit in `x`, we observe that the expression $\max_{\forall s \in [-th, th]} \text{CLZ}(xo_s[k_{st} : k_e])$ in line 10 is equivalent to $\text{CLZ}(\wedge_{s=-th}^{th} \text{RFILL}(xo_s[k_{st} : k_e]))$. In other words, this means that maximum selection of the multiple CLZs can be substituted by a single CLZ operation of the result from doing an and operation among all the elements of the matrix XO_s after setting to one all the bits to the right of their highest set bit. This substitution are expected to reduce resource usage since the comparators used in max consume more resources than the gates required to implement RFILL. The RFILL algorithm is very simple as shown in Algorithm 6 and is already proposed in the RISC-V Bitmanip Extension [38]. Figure 8 depicts an example logic circuit to compute an example $\text{CLZ}(\wedge_{s=-1}^{1} \text{RFILL}(XP_s))$ operation. A single CLZ unit is used, and `RFILL` is implemented with a small number of or gates.

Algorithm 6 RFILL. Bit-Wise Highest Set Bit Fill. Fill With Ones From the Highest Set Bit

Input: An input number (X) of n bits.
Output: A 8 bit output number (R) in which each bit position is 1 if there is a 1 in the same position or a left position in the input number

```
1  $R \leftarrow 0$
2  for $i \leftarrow 1$ to $n$ do
3      $r[i] \leftarrow \vee_{j=1}^{i} x[j]$
4  end for
```

SneakySnakes requires that, after a one is detected in a window, the following search for the longest run of zeros should start at the following bit. Note that the one that ends the run of zeros is a detected error. If the error threshold of the implementation is lower that the window size some optimization can be done at the window level. This is the case of the FPGA implementation done in [22], which limits the units to find the longest streams of zeros to 4 inside a window. We aim to implement a generic implementation, so we replicate all blocks until we cover the whole window. The algorithm with all the discussed optimizations is presented in Algorithm 7.

Algorithm 7 SneakySnakes (Proposed Update)

Input: P as the pattern sequence, T as the text sequence both of length n, th the threshold so that the pairs with $d_{Lev}(P, T) > th$ are quickly discarded. WS is the window size.
Output: The estimated minimum number of errors (EE), such that $d_{Lev}(P, T) \geq EE$

```
1  $EE \leftarrow 0$
2  for $w \leftarrow 1$ to $n/WS$ do
3      $k_{st} \leftarrow WS \cdot (w-1) + 1$
4      $k_e \leftarrow WS \cdot w$
5      for all $s \in [-th, th]$ do
6          $SP \leftarrow P \ll s$
7          $XO_s \leftarrow SP \oplus T$
8          $XP_s \leftarrow xo_s[k_{st} : k_e]$
9      end for
10     for all $i \in [1, WS]$ do
11         $lz_i \leftarrow \text{CLZ}(\wedge_{s=-th}^{th} \text{RFILL}(xp_s[i : WS]))$
12     end for
13     for $i \leftarrow 1$ to $WS$ do
14         $Edits_i = \begin{cases} 0, & \text{if } lz_i = i \\ 1, & \text{if } lz_i = i-1 \\ Edits_{i-lz_i-1} + 1, & \text{otherwise} \end{cases}$
15     end for
16     $EE \leftarrow EE + Edits_{WS}$
17 end for
```

The implementation in C/C++ can easily be derived from the algorithm. A benefit of this approach compared with previous implementation is that no full sequence accumulator bit vector is used. Thus, after the windowed parts extracted from the bit matrix XP_s are obtained (in line 8) all the operations required to compute the number of edits of a window can be described by standard integer C/C++ variables as they only require a small number of bits WS. The number of edits on a window is computed by combining the information from the set of maximum number of leading zeros of every subwindow. The resulting circuit is depicted in figure 9. The final number of edits of the whole pair is obtained by aggregating the values for every window.

E. BANDED-LEV

The computation of the Levenshtein distance with the dynamic programming approach is a well-known basic method. The value of each cell of the table depends on the values of previous column, row, and diagonal cells. Thus, it is not possible to break the data dependency among cells of the same row or column to compute them in parallel. However, it is also well known that cells of anti-diagonals do

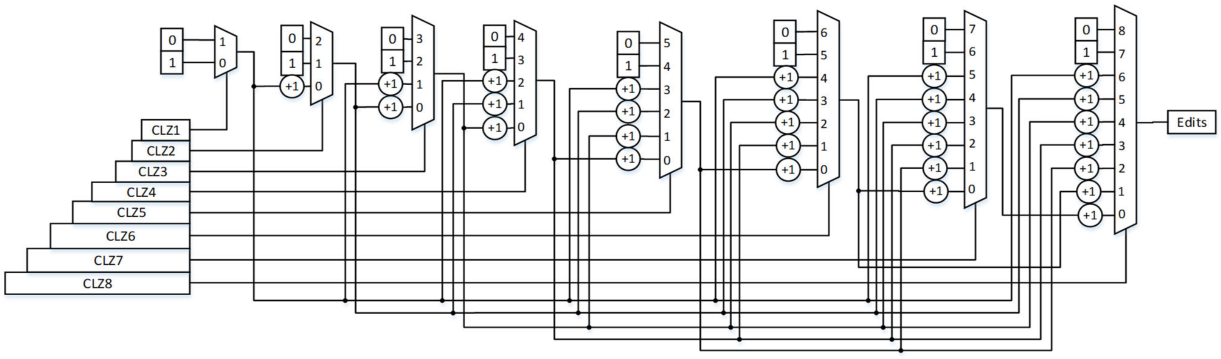


FIGURE 7. Logic diagram of the circuit that computes the number of edits on a 8-bit window (for $WS = 8$) in the SneakySnakes filter. The window is divided in smaller subwindows that allow the detection of runs of zeros at different column offsets of the input matrix. The number of leading zeros from the biggest units (that consider smaller starting column offsets) are used to detect the position of the one finishing the run of zeros and select the following unit to consider (corresponding to the offset of the column following the column from the detected one).

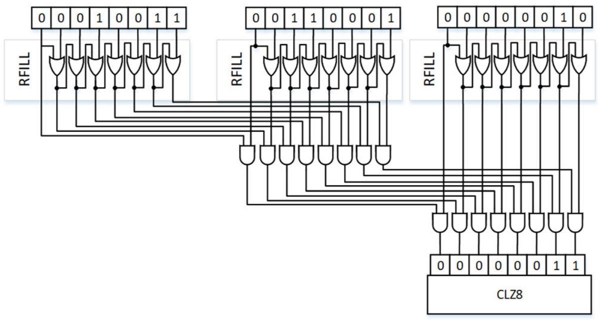


FIGURE 8. Optimized logic circuit to compute the maximum count of leading zeros over three 8 bit inputs. The alternative circuit to do the same computation would consist of three count leading zeros units (CLZ8) and a tree of comparators to select the maximum value.

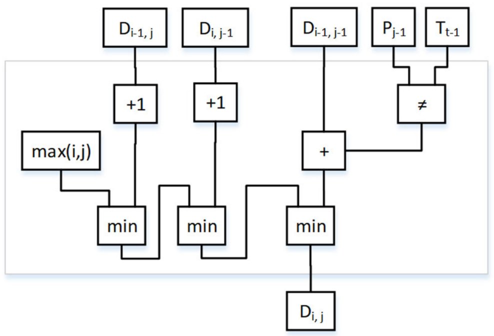


FIGURE 9. Diagram of the processing element used for each cell of the dynamic programming table when using the classical DP approach to compute Levenshtein distance. Notice that the circuit is a direct implementation of equation 10 and the width of the signals must be large enough to accommodate the maximum distance value.

not have data dependencies among them, and therefore, can be computed in parallel.

Hardware implementations (like [39]) tend to compute the cells of the anti-diagonals in parallel. If the allocated processing elements (PEs) are created to address the longest anti-diagonal, the matrix computation takes several cycles and the PEs are be reused. Another alternative is to create a PE for each cell of the matrix and compute the anti-diagonals in pipeline. Unlike classic HDL designs we do not create a specific PE for cell computation. We can still interpret that there is a kind of virtual PE diluted in the OpenCL source code for each matrix cell. The logic diagram of this virtual PE is depicted by figure 9. Different elements from the PE can be pruned out depending on the cell.

Instead of an explicit description of an anti-diagonal pipelined design, we will use an iterative algorithm (Algorithm 8: Banded Lev) to build the elements of the matrix, pruning out the computation of all cells farther than $\lfloor th/2 \rfloor$ from the main diagonal. The direct implementation of the algorithm in C/C++ could lead the compiler to detect the data dependency between the iterations of the loops iterations preventing the unrolling of the for loops (in lines 1 and 2). In our case, we unroll the loops using metaprogramming on Intel platforms. With this step we make it easier for the OpenCL compiler to detect the data dependencies and implement a pipelined design automatically.

F. BANDED-MYERS

Despite the apparent fundamental difference, the Banded-Myers algorithm structure is very similar to the previous Banded-Lev design as both compute the same number of cells from the matrix. However, in this case, the virtual PE does not compute a distance, but a set of bits. The design is totally implemented with binary gates as shown in figure 10.

The algorithm requires a final phase to aggregate the diagonal cells to obtain the distance estimation. Separating the computation of the distance from the cell computation has an important impact on resource utilization.

Despite the number of the considered matrix cells of Banded-Lev and Banded-Myers are the same, a cell of Banded-Lev contains multiple adders and a similar number of circuits to get the minimum from two values. Minimum circuits can be implemented with a comparator (often using a subtract circuit) and a multiplexor. In the worst case, the number of add/sub units in the virtual PE to compute a cell of the matrix will be 6. Therefore, the upper bound of the total number of add/sub units of the Banded-Lev design is

Algorithm 8 Banded Lev

Input: The pattern sequence P, and text sequence T, both of length n. The error threshold th so that pairs with $ED(P, T) > th$ can be quickly discarded.

Output: The estimated minimum number of errors (EE), such that $d_{Lev}(P, T) \geq EE$

```
1  for $x \leftarrow 0$ to $n$ do
2    for $y \leftarrow 0$ to $n$ do
3      if $(|x - y| \leq \lfloor th/2 \rfloor)$ then
4        if $(x = 0 \lor y = 0)$ then
5          $D_{y,x} = \max(x, y)$
6        else
7          $D_{y,x} = \min(th + 1, D_{y-1,x-1} + P[y-1] \oplus T[x-1])$
8          if $(|x - 1 - y| \leq \lfloor th/2 \rfloor)$ then
9            $D_{y,x} = \min(th + 1, D_{y,x}, D_{y,x-1} + 1)$
10         end if
11         if $(|x - y + 1| \leq \lfloor th/2 \rfloor)$ then
12           $D_{y,x} = \min(th + 1, D_{y,x}, D_{y-1,x} + 1)$
13         end if
14       end if
15     end if
16   end for
17 end for
18 $EE \leftarrow D_{n,n}$
```

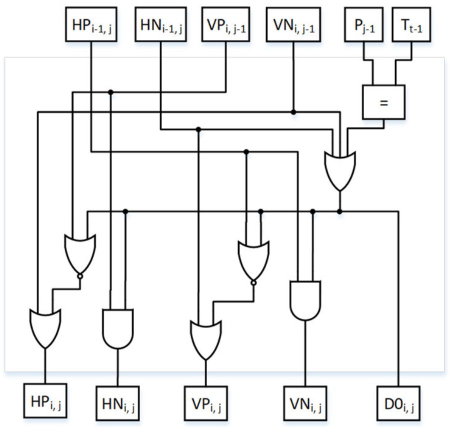


FIGURE 10. Diagram of the processing element used for each cell of the DP table when using a Banded-Myers approach to compute the Levenshtein distance. Notice that the circuit implements equations 18,19,20,21, and 22 and the width of most inputs and outputs is just one bit.

$Res_{add/sub} < 6(n + 2 \cdot (\lfloor th/2 \rfloor)(n - \lfloor th/2 \rfloor))$. On the other hand, the Banded-Myers design only needs n adders.

Similarly to previous design, we use the algorithm 9 to implement the system. Again, the for loops (lines 1 and 2) are unrolled with our metaprogramming framework in Intel platforms so that the OpenCL compiler can easily detect data dependencies and automatically infer an anti-diagonals pipeline. In Xilinx platforms, we use `#pragma HLS array_partition variable=<var> complete` to obtain the same effect.

Algorithm 9 Banded Myers

Input: The pattern sequence P, and text sequence T, both of length n. The error threshold th so that pairs with $ED(P, T) > th$ can be quickly discarded.

Output: The estimated minimum number of errors (EE), such that $d_{Lev}(P, T) \geq EE$

```
1  for $x \leftarrow 0$ to $n$ do
2    for $y \leftarrow 0$ to $n$ do
3      if $(x = 0)$ then
4        $VP_{y,x} = 1$
5        $VN_{y,x} = 0$
6      else if $(y = 0)$ then
7        $HP_{y,x} = 1$
8        $HN_{y,x} = 0$
9      else if $(|x - y| \leq \lfloor th/2 \rfloor)$ then
10       $D0_{y,x} = \neg(P[y] \oplus T[x]) \lor VN_{y,x-1} \lor HN_{y-1,x}$
11       $HN_{y,x} = VP_{y,x-1} \land D0_{y,x}$
12       $VN_{y,x} = HP_{y-1,x} \land D0_{y,x}$
13       $HP_{y,x} = VN_{y,x-1} \lor \neg(VP_{y,x-1} \lor D0_{y,x})$
14       $VP_{y,x} = HN_{y-1,x} \lor \neg(HP_{y-1,x} \lor D0_{y,x})$
15     else if $(|x - y| = \lfloor th/2 \rfloor + 1)$ then
16       if $(x > y)$ then
17         $HN_{y-1,x} = 0$
18         $HP_{y-1,x} = 1$
19       else
20         $VN_{y,x-1} = 0$
21         $VP_{y,x-1} = 1$
22       end if
23     end if
24   end for
25 end for
26 $EE \leftarrow \sum_{i=1}^{n} \neg(D0_{i,i})$
```

V. RESULTS

In this section we first analyze the accuracy of the different filters following a novel frequency-domain interpretation. As we saw in section II.A, accuracy has a direct impact on the expected speedup factor (SF_{align}) achieved when integrating them in read-mappers. Then, we describe the FPGA synthesis results for different platforms, focusing on the resource usage of the different filters and their scalability with respect to error thresholds and read lengths. We measure all filters execution time on different platforms and report the speedup factor with respect to Edlib library [40] executed in a single thread CPU. Finally, we estimate the speedup factors that could be achieved after the integration of filters on seed-and-extend mappers.

A. ACCURACY

We measure the accuracy of the filters with a synthetic benchmark that simulates random edition operations in a controlled way. For every sample, the benchmark creates a random *text* string and generates a *pattern* with a certain number of insert, delete or substitution operations specified

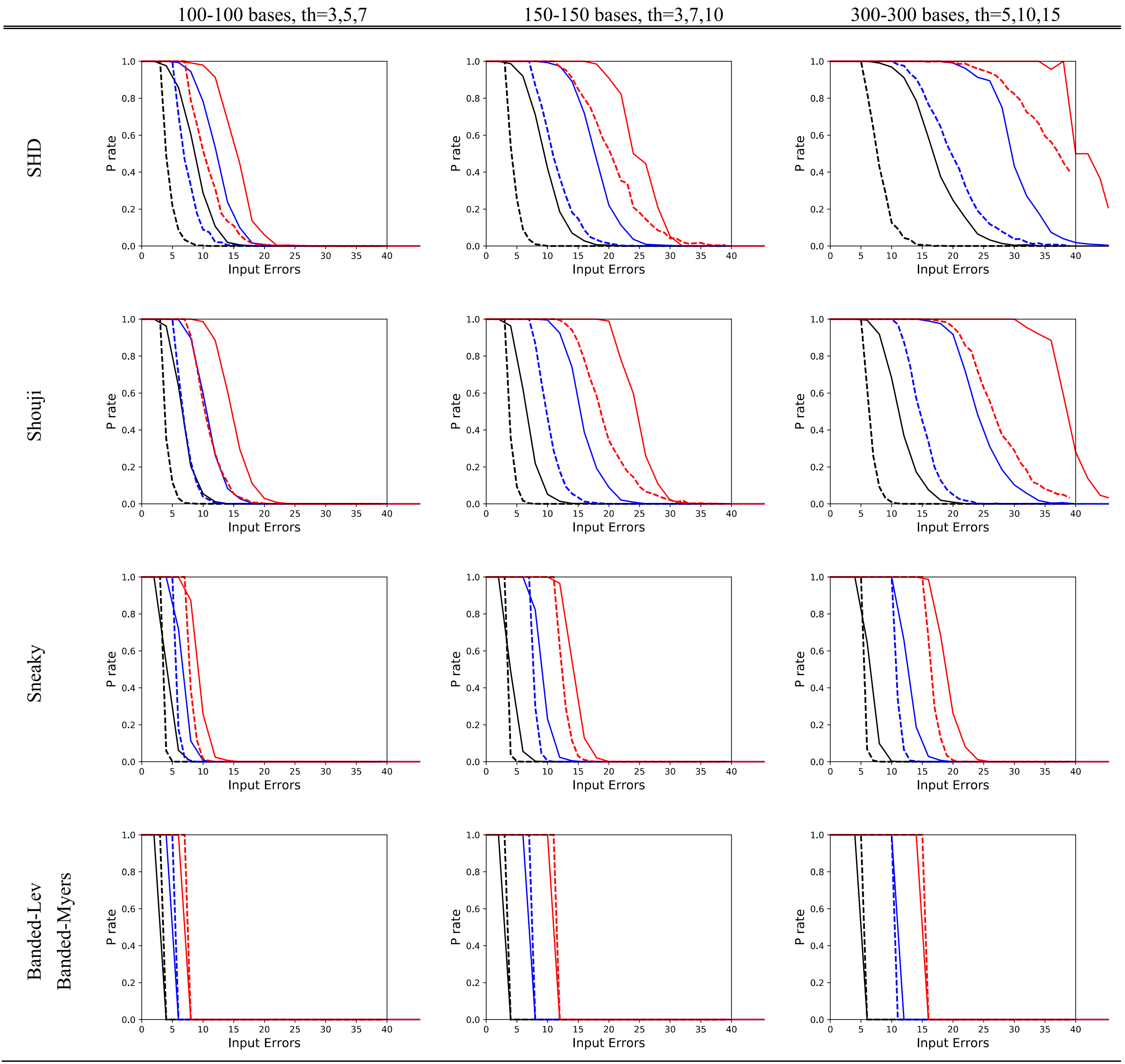


FIGURE 11. False positive (FP) observed values of different filters as the number of errors is increased in the input data set. Dashed lines correspond to the FP rate on substitution errors. Solid lines correspond to the FP rate on ins/del errors. For every sequence pair length we test three different thresholds. Notice that, in inexact filters, the higher the threshold the higher the observed FP rates, which is a non-desirable effect.

by the user. For every pair sample, the filter is executed and each pair is classified as positive or negative.

To analyze the filter response we introduce a frequency-domain analysis method as the filters effectively act as low-pass filters, where the frequency is the frequency of differences (or errors) between the pattern and text pairs. An ideal response a filter would accept all values below the cutoff frequency and reject all values above it in a sharp drop to zero (as depicted in figure 12 first diagram). However, for many filters the false positives rate is often high after the threshold, showing a gradual drop to zero as error frequency increases (as depicted in figure 12 second diagram).

To do the analysis we consider the number of positives rate obtained for a varying number of input errors. Since our synthetic benchmark allows to exactly introduce a specific number of errors we start introducing no errors and keep increasing the number of introduced errors and collecting the filter response. For each value of ''number of errors'' (or error frequency) we run two tests: one introducing substitution errors, and another introducing insertion and deletion errors. Generally, insertion and deletions are harder to detect than substitutions, and they give higher false positive rates. To the best of our knowledge, this is the first time that pre-alignment filters are analyzed by their frequency-domain response.

Figure 11 depicts the response of the filters for various sequence lengths and different threshold values. The vertical axis corresponds to the P_{rate}. The horizontal axis correspond to the number of errors injected to the filter. Dashed

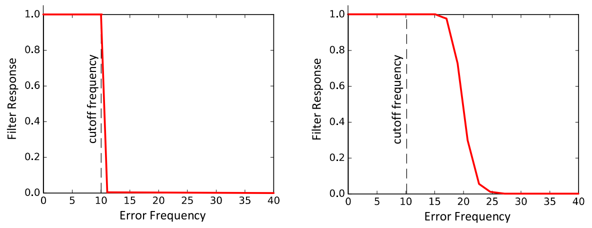


FIGURE 12. Left) Ideal filter response: all positives (P) are detected (100% acceptance), after the cutoff frequency all negatives (N) are rejected (0% acceptance). Right) Possible filter response: all positives (P) are detected (100% acceptance), after the cutoff frequency some negatives are accepted becoming false positives (%FP > 0), but as the error frequency is increased the FP rate drops to zero.

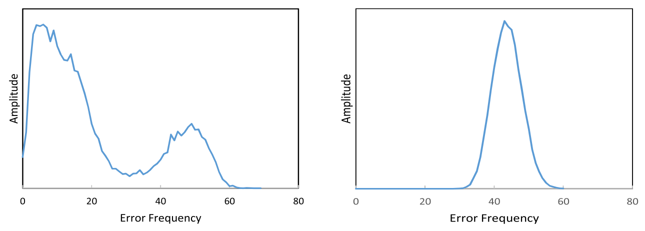


FIGURE 13. Error frequency profiles of some input data sets used in previous work from the literature. The amplitude value (y axis) corresponds to the number of sequences having a specific number of errors (x axis). (Left) Error frequency profile of file ERR240727_1_E2. (Right) Error frequency profile of file ERR240727_1_E40. Notice that correctly filtering the right file (i.e. reducing the number of FP) is less challenging than filtering the left file.

lines correspond to the false positive rate obtained when substitutions are injected. Solid lines correspond to the false positive rate obtained with insertion and deletions. For all sequence lengths and all inexact filters, increasing the error threshold increases the observed FP_{rate}. SneakySnake is the inexact filter less affected by this effect. For all algorithms, increasing the sequence length increases the observed FP_{rate}. Again SneakySnake is the less affected from the inexact filters.

As we studied in section II the performance of a complete system is also affected by the characteristics of the input data. The number of positives and false positives is a determinant factor to the final speedup. As an example we take two dataset files (ERR240727_1_E2, ERR240727_1_40) used in [22] which are generated with the mrFAST mapper [41]. Their error frequency profile is shown in figure 13 diagrams. The important difference between these two datasets is that the first one will generate a large number of false positives if the filter response is like the shown in figure 12 (right). On the other hand the same filter will generate no false positives for the second dataset.

Besides false positive analysis, it is also important to ensure that the false negative rate for any filter is zero, otherwise we would be possibly discarding some potentially matching pairs. Although there are very few of them (see table 1), SHD and Shouji do not completely avoid false negatives.

From our analysis, we conclude that SneakySnakes is the inexact method that better approximates the real Levenshtein distance computation. However, exact filters (Banded-Lev, and Banded Myers) provide the exact computation, and a zero FP_{rate} and FN_{rate}.

TABLE 1. Pre-alignment filters with non-zero false negatives rate.

Algorithm	Threshold	Dels.	Ins.	False Negative Rate
SHD	2	1	1	1.25%
Shouji	2	1	1	0.02%

TABLE 2. FPGA details of target accelerator boards.

Platform	FPGA	kLEs / kCLBs	Int. Mem. Mbits	Ext. Mem.	Ext. Mem. bandwidth
Terasic OpenVino Starter Kit	Intel Cyclone V 5CGTFD9	301	14	1GB DDR3	6.4 GB/s
Terasic DE5Net	Intel Stratix V 5SGXEA7	622	52	8 GB DDR3	12.8 GB/s
HARPv2	Intel Arria10 10AX115U3	1150	55	-	28.5 GB/s
PAC10	Intel Arria10 10AX115N2	1150	55	DDR4	17 GB/s
D5005	Intel Stratix10 1SX280	2800	240	32GB DDR4	4 × 19.2 GB/s
AWS F1	Xilinx UltraScale+ VU9P	2586	345	64 GB DDR4	4 × 17 GB/s
Alveo U50	Xilinx XCU50	1907	251	8 GB HBM	32 × 14.3 GB/s
Alveo U250	Xilinx XCU250	3780	448	64 GB DDR4	4 × 19.2 GB/s

B. FPGA SYNTHESIS AND RESOURCES USAGE

We selected eight FPGA accelerators from the leading manufacturers Intel and Xilinx to synthesize our designs. We include a low-cost accelerator based on Cyclone V FPGA, and seven high performance accelerators based on Arria, Stratix, and Virtex FPGA families. The characteristics of the accelerators are described in tables 2 and 3. An important difference between the HARPv2 accelerator and the other systems is the faster QPI link to access the host memory.

One of the selling points of High Level Synthesis (HLS) is the expected higher productivity. A side result from our work is the assessment of this claim. Given the combination of platforms, filter types, read lengths, and error thresholds we have created 504 different designs.

A small fraction of the compilations were not successful either due to the exhaustion of device resources or by bugs in the compilation toolchain. The lack of resources is the main reason of failure for the smallest device (OSK) but also affects the Banded-Lev filter in most Intel devices except HARPv2. After the analysis of the generated HDL, we observed that the reason of the significant difference on resource consumption for this filter in Intel platforms is the different way that saturating adders are handled. In the

TABLE 3. Additional details of target accelerator boards.

Platform	Toolchain Version	Bus connection	Bus bandwidth	TDP
Terasic OpenVino Starter Kit	17.1	PCIe-2 × 4	2 GB/s	18 W
Terasic DE5Net	17.1	PCIe-3 × 8	7.8 GB/s	24 W
HARPv2	16.0	QPI + PCIe-3 × 8	28.5 GB/s	n/a
PAC10	17.1	PCIe-3 × 8	7.8 GB/s	66 W
D5005	19.2	PCIe-3 × 16	15.7 GB/s	225 W
AWS F1	2021.1	PCIe-3 × 16	15.7 GB/s	225 W
Alveo U50	2020.2	PCIe-3 × 16	15.7 GB/s	75 W
Alveo U250	2020.2	PCIe-3 × 16	15.7 GB/s	225 W

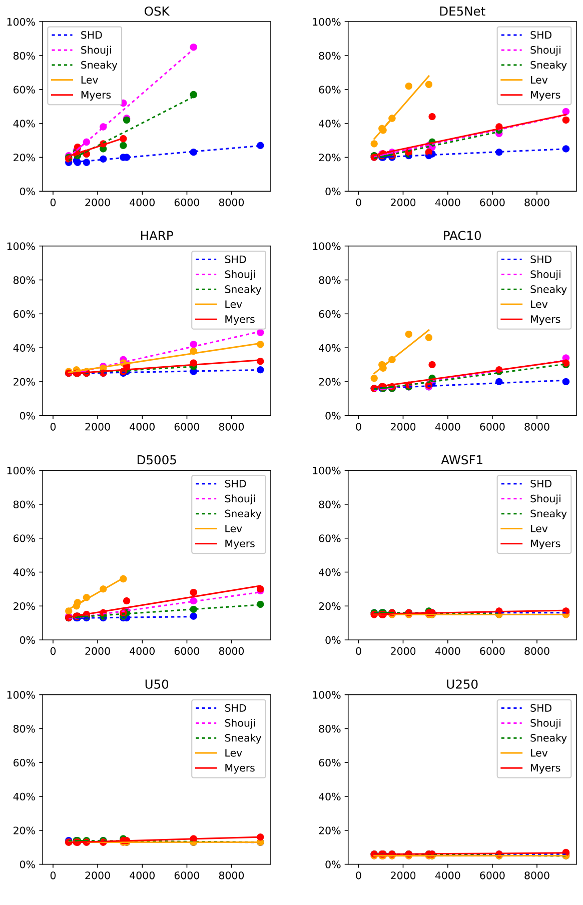


FIGURE 14. Synthesis results for different FPGA accelerators. The plots report resource usage percentage as a function of the equivalent DP table cells used by the designs, i.e. $m \times (2 \times th + 1)$.

worst case, the compiler avoids to remove the unused bits from the adders and introduces additional pipeline registers. Regarding bugs in the compilation toolchain, HDL synthesis is never failing, but an occasional failures happen in the LLVM optimization phase when dealing with very large designs.

The percentage of occupancy for all platforms with respect to the number of computed cells from the DP table $(m \times (2 \times th + 1))$ is depicted in figure 14. The simplicity of SHD results in the lowest resource consumption consistently on all devices. Another consistent behavior is the larger resource usage of Shouji with respect to Sneaky Snake. Among exact filters, Banded-Myers consistently has a lower resource usage than Banded-Lev, as expected. An unexpected result is the rather flat resource occupancy profile in Xilinx devices. Our designs are not able to achieve an initiation interval of 1 clock cycle on Xilinx devices, but start with an initiation interval of 8 and grow up to 64 clock cycles. This reduces the parallelism achieved in the loop iterations and the resource demand. The limits to the scalability of designs with respect to longer read lengths can be estimated by extrapolating the trend lines in figure 14 considering the required number of cells to compute. For instance, read lengths of 500 bp with a 10% of error threshold would require 50,500 cells, making most of the designs unviable.

Besides the device occupancy, an important synthesis result is the maximum clock frequency achieved by the kernel as it influences the maximum throughput of the filter. figure 15 shows a boxplot of the clock frequency of all the designs by platform. The differences between the achieved maximum frequencies are mainly caused by the differences between FPGA families, but most accelerators target the 300 MHz range.

The Cyclone-based FPGA accelerator shows the slowest clock frequency. The Arria family systems provide an important improvement, which is still slightly lower to the achieved by Stratrix families. The Stratix 10 device show a 10% improvement with respect to Stratix V for our kernels. These results are basically in line with our expectations, since Cyclone family is addressing low cost, Stratix and Virtex are addressing High Performance applications, and Arria is targets a trade-off between the two aspects. Xilinx Alveo boards provide a clock frequencies above 300 Mhz, except the Xilinx system used in Amazon F1 instances, which is above 250 Mhz.

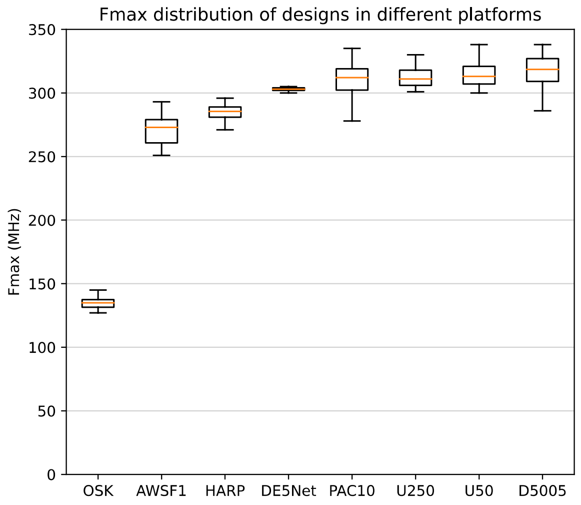


FIGURE 15. Kernel clock frequency variability for different designs in various FPGA accelerators.

TABLE 4. Synthesis results from the HDL designs in the literature compared with our OpenCL results for 100 bp.

			HDL on Xilinx XC7VX690T (original)			OpenCL on Intel 5SGXEA7 (ours)		
Design	Length	Th	LEs	FFs	Mem.	LEs	FFs	Mem.
SHD [17]	100	2	221 k	17 k	1 Mbit	46 k	70 k	2.5 Mbit
	100	5	311 k	17 k	1 Mbit	47 k	70 k	2.5 Mbit
Shouji [20]	100	2	255 k	17 k	1 Mbit	47 k	75 k	2.5 Mbit
	100	5	423 k	17 k	1 Mbit	48 k	78 k	2.5 Mbit
Sneaky Snake [22]	100	2	253 k	38 k	1 Mbit	49 k	74 k	2.5 Mbit
	100	5	390 k	63 k	1 Mbi1	52 k	76 k	2.5 Mbit

TABLE 5. Synthesis results of our exact filters for 100 bp.

			OpenCL on Intel 5SGXEA7 (ours)		
Design	Length	Th	LEs	FFs	Mem.
Banded-Lev (ours)	100	2	64 k	95 k	2.5 Mbit
	100	5	85 k	122 k	2.5 Mbit
Banded-Myers (ours)	100	2	47 k	73 k	2.5 Mbit
	100	5	52 k	83 k	2.5 Mbit

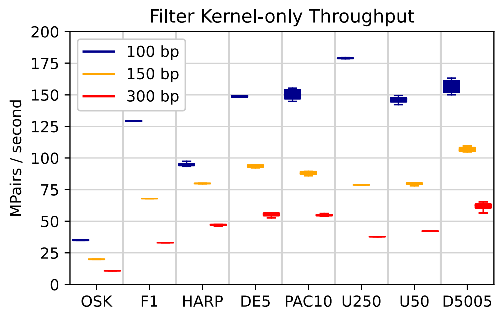


FIGURE 16. Throughput of kernel (only) execution for different FPGA accelerators. Different filters do not differ significantly as performance is more determined by the common adapter part rather than the implementation of each filter. Longer sequences require more memory bandwidth than shorter ones, which translates on lower performance.

C. RESOURCE OVERHEAD OF HLS DESIGNS

We analyze the resources of our HLS based implementations in comparison with HDL based ones reported in other works. In this case we do not report occupancy percentage, but the absolute number of resources. This can be used to give an idea of the overhead induced by HLS frameworks and the OpenCL runtime. An interesting point is that the original inexact filters (GateKeeper, Shouji, SneakySnake) use combinational logic designs and duplicate computing units to achieve better speedups. As their units are replicated, the f_{max} decreases. The introduction of pipelining is proposed by them with a grain of salt due to the concerns on an excessive cost in registers.

Table 4 shows a comparison of synthesis results of previous works and our proposals. Although the FPGA devices used in our work are different from those used in others' works, we can still get some useful information from the comparison. The replication of 16 filtering units done in previous proposals increase the Logic Elements (LE) count creating an imbalance between LEs and register use. In modern FPGAs combinational blocks and registers are combined inside the logic blocks of the devices, so having balanced LE/register designs can share the same logic block resources without increasing resource usage. Our OpenCL designs have a balanced number of LEs and registers. In our case, OpenCL introduces pipelining automatically when needed, so that the register count is higher but the LE count is lower. On the other hand, our designs already have a high occupancy of the memory bandwidth of the accelerator with a single filtering unit, so there are no incentives to replicate filtering units unless higher memory bandwidth is available.

As shown in table 5, our exact filter designs also have a balanced mix of LEs and registers. The resource count of the Banded-Lev design is higher than the Banded-Myers design, so, for the exact same accuracy Banded-Myers should be preferably used.

D. PERFORMANCE RESULTS

We analyze the performance of the filters by executing synthetic benchmarks having 10 million sequence pairs. The results are shown in table 6. When we focus our analysis to the execution time of the kernel (see figure 16), the boards with device memory show the highest performance. For Intel devices, D5005 platform has a slightly higher kernel performance than the PAC10 platform due to both, a higher bandwidth when fetching data from the device memory, and a higher clock frequency. Having a similar Arria10 FPGA fabric, the HARPv2 system runs the kernel slower than the PAC10 device due to the higher distance to the memory.

For Xilinx devices, the U250 platform achieves the highest performance thanks to the high bandwidth of their DDR4 memories. Notice that the performance is reduced as read length is increased due to the increase of the initiation interval mentioned earlier.

Memory read and write operations must be interlaced as the distance for every pair is computed and stored. Thus, as seen in figure 17, the available memory bandwidth is often underutilized due to an incapacity of maintaining a unidirectional transfers during long period of time. We only approach the maximum bandwidth when using long reads and the factor between the data read and the data written is increased. When including the memory transfer time between the host and the accelerator in the analysis, the shared memory approach of the HARPv2 platform shows a better overall performance thanks to a higher bandwidth

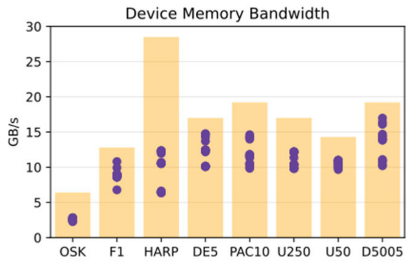


FIGURE 17. Memory bandwidth utilization during kernel execution. The columns illustrate the maximum available memory bandwidth for a device. A dot illustrates the consumed bandwidth for a particular design (targeting a certain combination of read-length, error threshold, and filter type).

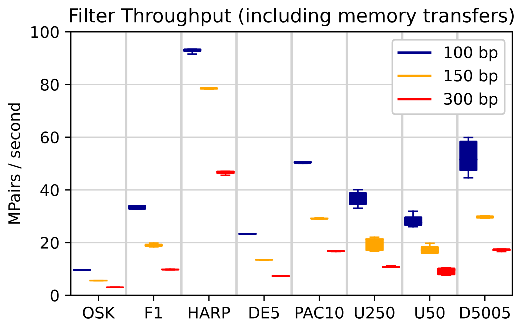


FIGURE 18. Throughput of filter execution (including memory transfers with the host) for different FPGA accelerators. Notice that PCIe based accelerators show a significant performance drop as designs are not able to reach the theoretical maximum bus bandwidth.

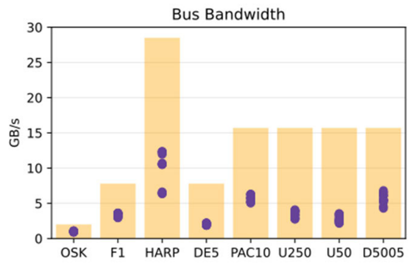


FIGURE 19. Bus bandwidth utilization before and after kernel execution. The columns illustrate the maximum available bus bandwidth for a device. A dot illustrates the consumed bandwidth for a particular design.

with the host memory (see figure 18). Regarding the OSK platform, the lower performance of the kernel is mainly explained by the lower clock frequency (around a third of the other platforms) and a lower bandwidth with the external memory. The lower bus memory bandwidth of its PCIe connection contributes to and even lower total throughput.

The invocation of the kernel requires to transfer the input data, execute the kernel, and collect the results. We do not overlap computation and communication, which is left to future optimizations. As shown in figure 19, this results on a low bus bandwidth utilization. However, we try to optimize the transfers with the host by using the SVM OpenCL primitives in HARP devices and clEnqueueMigrateMemObjects in Xilinx devices.

TABLE 6. Filtering Speed in million pairs per second (M Pairs/s) measured when executing our OpenCL filters over 10 million pairs.

		100-100 bases		150-150 bases		300-300 bases	
	Platform	kernel	+trans	kernel	+trans	kernel	+trans
OpenCL SHD	OSK	35.5	9.8	20.0	5.6	10.9	3.0
	DE5Net	149.2	23.3	94.4	13.5	56.8	7.4
	HARP	97.4	**97.4**	81.1	**79.7**	47.5	**47.0**
	PAC10	155.2	51.4	89.4	29.4	55.1	16.8
	D5005	161.1	51.5	**104.8**	29.7	**61.9**	18.4
	AWSF1	129.4	32.8	67.9	18.6	33.0	9.1
	U50	146.8	29.6	80.28	15.96	42.3	10.3
	U250	**179.2**	38.6	78.7	16.7	37.8	10.9
OpenCL Shouji	OSK	35.3	9.6	20.0	5.6	10.8	3.0
	DE5Net	149.2	23.3	94.4	13.5	52.7	7.3
	HARP	95.4	**93.4**	80.1	**78.7**	46.0	**45.5**
	PAC10	144.7	50.2	86.0	28.7	53.8	16.6
	D5005	158.4	59.9	**109.6**	30.0	**65.3**	17.2
	AWSF1	129.4	34.0	67.9	18.9	33.1	9.7
	U50	142.2	26.6	78.1	15.9	42.1	7.6
	U250	**178.6**	38.8	78.8	21.3	37.9	9.4
OpenCL Sneaky	OSK	35.3	9.6	20.0	5.6	10.8	3.0
	DE5Net	149.2	23.3	94.4	13.5	56.4	7.3
	HARP	94.0	**92.6**	79.8	**78.4**	46.7	**46.2**
	PAC10	146.8	50.5	87.1	29.2	56.2	16.9
	D5005	150.1	58.4	**108.3**	30.2	**63.1**	17.2
	AWSF1	99.8	30.5	67.9	18.4	33.1	9.8
	U50	144.4	26.0	79.1	17.2	42.1	7.9
	U250	**178.7**	40.1	78,7	17.0	37.9	11.1
OpenCL Banded-Lev	OSK	n/a	n/a	n/a	n/a	n/a	n/a
	DE5Net	148.2	23.2	92.3	13.45	n/a	n/a
	HARP	95.1	**93.2**	79.6	**78.2**	**47.3**	**46.9**
	PAC10	151.2	50.6	87.8	29.0	n/a	n/a
	D5005	152.0	44.6	**105.2**	29.3	n/a	n/a
	AWSF1	158.8	34.0	75.2	19.8	34.8	9.9
	U50	147.7	27.6	80.4	19.7	42.2	9.7
	U250	**179.5**	34.6	86.1	22.0	39.5	10.8
OpenCL Banded-Myers	OSK	33.3	9.6	19.0	5.5	n/a	n/a
	DE5Net	148.2	23.2	92.8	13.4	55.0	7.3
	HARP	93.3	**91.5**	79.7	**78.4**	47.5	**47.0**
	PAC10	154.2	50.0	89.5	29.2	54.6	16.7
	D5005	163.1	47.4	**106.1**	29.4	**56.5**	16.6
	AWSF1	129.1	33.7	67.8	19.3	33.1	9.8
	U50	149.5	31.9	79.8	18.4	42.0	10.2
	U250	**178.8**	33.0	78.7	21.3	37.7	10.6

E. RELATED WORK

The execution time of inexact filters compared with previous work is shown in table 7, which shows the million pairs per second (MPPS) achieved by pre-alignment filters when

TABLE 7. **Achieved filtering speed (MPairs/s) of 100bp and th=2 filters from previous work and our work.**

	HDL on Xilinx XC7VX690T		OpenCL on D5005 Stratix 10 (ours)			OpenCL on HARPv2 Arria 10 (ours)		
Design	MPPS	SF_{pre}	MPPS	SF_{pre}	SF_{HDL}	MPPS	SF_{pre}	SF_{HDL}
SHD [17]	41.52	133×	51.5	166×	1.2×	**97.4**	**314×**	**2.3×**
Shouji [20]	41.52	133×	59.9	193×	1.4×	**93.4**	**301×**	**2.2×**
Sneaky Snake [22]	41.47	133×	58.4	188×	1.4×	**92.6**	**298×**	**2.2×**

TABLE 8. **Achieved filtering speed (MPairs/s) of exact filters.**

			OpenCL on HARPv2 Arria 10		Kernel only OpenCL on D5005 Stratix10	
Design	Len.	Th	MPPS	SF_{pre}	MPPS	SF_{pre}
Banded-Lev	100	2	95.1	306×	152.0	490×
Banded-Myers	100	2	93.3	300×	163.1	526×

working on 10 million pairs. We report the throughput of FPGA HDL-based implementation in original papers and the throughput of our best performing OpenCL implementation on D5005 and the HARPv2 system. The speedup factor SF_{pre} is computed with respect to the execution time of the Levenshtein distance using Edlib with a single thread on a Xeon Gold 6230 (0.314 MPairs/s computing edit distance using 100 bp and 0.094 MPairs/s obtaining the backtrace). The results show that our OpenCL implementations on the external memory based accelerator D5005 is between 20% and 40% faster than the HDL coded solutions in the literature, and more than 120% faster on the HARPv2 platform, which is based on a shared memory architecture.

As shown in table 8, the exact filters (Banded-Lev and Banded-Myers) have a similar performance since they are also designed aiming a single clock throughput. The performance of the accelerators could still be improved by overlapping communication and computation if using multiple memory buffers and interleaving the transfer of input data. However, it is clear that the analyzed pre-alignment filter algorithms are memory bound and its execution time is totally determined by the available bandwidth to fetch the input data.

In addition to the total best throughput achieved in HARPv2 system, in table 7 we include the throughput achieved by the kernel execution in the second best performing system (the D5005 platform, using the Stratix 10 device). Since we work at the cycle throughput the theoretical peak performance of the hardware unit would be the clock frequency, which is around 330 MHz for the Stratix 10 device. This peak performance would require a sustained memory bandwidth of 21 GB/s, which is higher to the reported top memory bandwidth of D5005 (19 GB/s). The actual achieved top bus bandwidth (6.8 GB/s) is less than half of the theoretical peak bandwidth (15.7 GB/s). Since the same amount of information must be streamed through the bus and the device memory, the bus bandwidth underutilization is the clear limiting factor of the global performance.

As examined in section 2.4, the speedup provided by the accelerator into the extension phase of the read mapping process is a combination of the accuracy of the used filter and its speedup factor compared with the performance of the CPU-only execution of the extension phase. If we take, for instance, the Sneaky Snake filter in 100 bp sequences, we could assume to have $SF_{pre} = 298\times$ (as reported in table 7) and an average $FP_{rate} = 1\%$. This false positive rate is selected empirically after analyzing the filter response on some non-synthetic datasets. Using these values on equation 6 we would still need to determine N_{rate} to get an estimate of the whole acceleration. We can consider two scenarios to get an idea of the potentially achieved acceleration. With a pessimistic $N_{rate} = 50\%$ we would obtain an overall speedup factor of $SF_{align} = 1.58\times$. With a more optimistic $N_{rate} = 100\%$ we would obtain an overall speedup factor of $SF_{align} = 24.7\times$.

Using the Banded-Myers filter in 100 bp sequences (with $SF_{pre} = 314\times$, $FP_{rate} = 0$) the acceleration in the same previous scenarios would range from $SF_{align} = 1.6\times$ to $SF_{align} = 314\times$. These results are assuming that the Edlib alignment is still executed in the host CPU for the positives pairs to obtain the alignment path, and their Compact Idiosyncratic Gapped Alignment Report (CIGAR). This effect of how requiring the alignment path affects the overall alignment speed is illustrated in figure 20. The alignment can be divided in to processes: filtering (computing edit distance) and verifying (obtaining the alignment path). Amdahl law explains this behavior as with this approach we effectively parallelize the filtering process but not the verification one.

If the extend phase does not require the CIGAR, the achieved acceleration of Banded-Myers would be $SF_{align} = 314\times$ in all cases. As shown in figure 21, the execution time of an Edlib solution would be constant with N_{rate} as it would always retrieve the edit distance and never the alignment path. However, inexact filters would still require to execute Edlib to compute the edit distance to verify the positives.

To the best of our knowledge, the achieved performance of our implementation of Banded-Myers pre-alignment filter is the highest reported in the literature. In table 9 we compare the processed MPairs/s of various dynamic programming algorithm accelerators that can be used in the extend phase of read mappers, including Myers, NW, and SW. Although the quality of the alignment given by gap-affine SW is considered better than Levenshtein distance in the bioinformatics community, we provide a significant lower execution time, almost doubling the closest result in

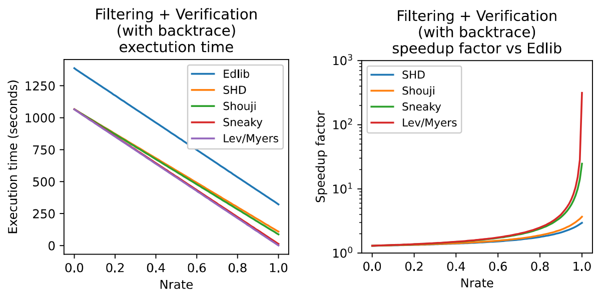


FIGURE 20. Execution time (in seconds) and speedup factor (with respect to single threaded Edlib) of 100 million reads of 100bp. In all cases first, the edit distance is computed (or estimated) and only if the result is lower than the threshold the CIGAR is retrieved.

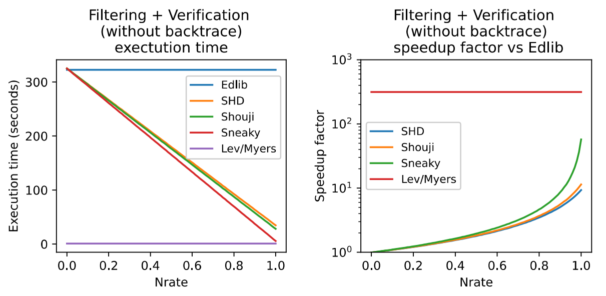


FIGURE 21. Execution time (in seconds) and speedup factor (with respect to single threaded Edlib) of 100 million reads of 100bp. The CIGAR is never retrieved, but inexact filters need to verify the positives. Thus, a significant speedup in inexact filters is only achieved if N_{rate} is big enough.

TABLE 9. Performance in MPairs/s for 300 × 300 sequences.

Design	Year	Alg.	Platform	Pr. Lang.	MPairs/s
[39] Lloyd	2009	NW	FPGA	VHDL	0.27*
[42] Settle	2013	SW	FPGA	OpenCL	0.26*
[43] Chacón	2014	Myers	GPU	CUDA	25.5*
[44] Sirasao	2015	SW	FPGA	OpenCL	0.85*
[45] Rucci	2015	SW	CPU	C/C++	3.90*
[46] Di Tucci	2017	SW	FPGA	OpenCL	0.46*
[25] Cai	2019	Myers	FPGA	OpenCL	9.60*
[30] Bautista	2020	Myers	FPGA	Verilog	0.18*
[47] Abdelhamid	2020	SW	FPGA	OpenCL	4.76*
[48] Chen	2021	SW	FPGA	n/a	0.56*
Banded-Myers on PAC10 (ours)		Myers	FPGA	OpenCL	16.90
Banded-Myers on HARPv2 (ours)		Myers	FPGA	OpenCL	**47.00**

* MPairs/s estimated from reported GCUPS

TABLE 10. Energy efficiency for 300 × 300 sequences.

Design	Device	Fndr. node (nm)	f_{clk} (MHz)	Perf. (MPairs/s)	Power (Watts)	Energy Efficiency (kPairs/J)
[42] Settle	Stratix V	28	193	0.26	25†	10.4
[43] Chacón	GTX Titan (GK110)	28	993	25.55	250*	102.2
[44] Sirasao	Virtex 7	28	n/a	0.85	28†	30.35
[45] Rucci	Xeon E5-2695	14	2100	3.93	120*	32.75
[46] Di Tucci	Kintex Ultrascale	20	n/a	0.46	25†	18.4
[47] Abdelhamid	Virtex Ultrascale+	16	n/a	4.76	56†	85
Banded-Myers PAC10 (ours)	Arria10	20	315	16.90	66*	256.06
Banded-Myers HARPv2 (ours)	Arria10	20	283	47.00	22	**2136.36**

* Power not measured, using TDP from specifications † No details on how power was estimated

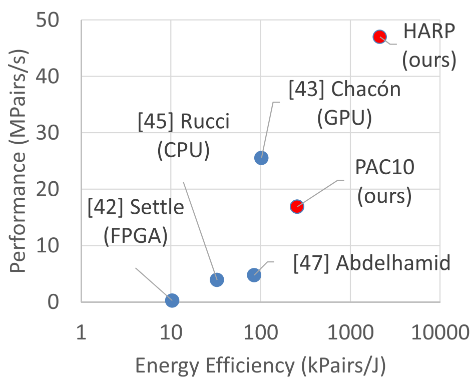


FIGURE 22. Energy efficiency of different alignment designs from the literature compared with our best implementations. We should note that [42], [45], and [47] implement SW instead of Myers, which is the base of [43] and one of our filters.

GPU [43]. An advantage of GPU implementations is that they are more flexible as they are not limited to a specific read length and supports longer reads. To address the first issue, the FPGA OpenCL approach can take profit of the dynamic reconfiguration ability to reprogram the FPGA fabric with a specific accelerator addressing the required read length and error threshold.

Besides performance, the expected major benefit from FPGA acceleration is energy efficiency. As detailed in [49], energy efficiency depends (among other factors) on the process node and the number of transistors of the computing platform. In table 10 we can see the energy efficiency achieved by different dynamic programming alignment accelerators. Since FPGAs, GPUs, and CPUs are all early adopters of new silicon foundry nodes, there is no huge difference between the node factor of latest works from the literature. We achieve the best energy efficiency value by more than one order of magnitude margin to other works (see figure 22). The key factor explaining this high number is the number of operations computed per cycle of our FPGA design using local (on-chip) fast memory accesses. In comparison, GPU implementations execute a large number of operations per clock cycle as well, but at the expense of a much higher memory bandwidth requirements to fetch instructions, private registers, and data, which increases their energy consumption.

VI. CONCLUSION

We have reviewed and re-implemented the most-relevant FPGA pre-alignment filters of the literature using HLS toolchains that can be executed through the OpenCL runtime. The designs can be integrated easily in genomic datacenters with FPGA accelerators providing important performance and energy efficiency improvements of several orders of magnitude with respect to CPU-based Levenshtein distance computation.

The algorithmic analysis to translate the existing proposals from HDL to C/C++ based HLS that we performed has unveiled many method simplifications and optimization opportunities. In turn, these results allow shorter and more understandable algorithms showing higher performance and lower resource usages. The maximum achieved throughput of 95.1 M Pairs/s including memory transferences of the Banded-Myers filter is the highest reported to date for an exact filter. Without considering memory transferences, the achieved throughput is even significantly higher: 178.8 MPairs/s in Xilinx U250. This result motivates our future research in addressing the system bottlenecks and the integration in read mappers with an expected double digit speedup factor.

Although some of the tested platforms are not addressing the data center environment (like the OSK) we have included them to demonstrate the degree of portability achieved with OpenCL and also make the bottlenecks of the system more evident to the reader. As an additional result from this work we have verified the productivity of C/C++ based HLS in implementing genomic accelerators. The lack of a full feature support for arbitrary precision integer library has been an important road-block. The metaprogramming approach has proven to be useful to work around it by sacrificing the readability of the code, which is a well-known factor to increase BUG probability and maintenance cost. Ignoring this issue, coding is very fast, and the generation of multiple variants is easy to do, facilitating the design space exploration. The HLS framework obviously hides the implementation details of many parts of the design. This has some drawbacks, like trying to identify exactly how a computing unit has been implemented and what are the performance bottlenecks. But also has some benefits, like avoiding implementing tedious blocks like memory load and store units, which are also key to high performance.

ACKNOWLEDGMENT

This work was supported in part by Xilinx under the Xilinx Adaptive Compute Cluster (XACC) Program. The authors would like to thank Intel for granting us access to the DevCloud System and let us join the HARP Research Program. The presented HARP-2 results were obtained on resources hosted at the Paderborn Center for Parallel Computing (PC2) in the Intel Hardware Accelerator Research Program (HARP2). They would also like to thank Cathal McCabe and Mario Ruiz for their valuable support.

REFERENCES

[1] E. L. van Dijk, H. Auger, Y. Jaszczyszyn, and C. Thermes, "Ten years of next-generation sequencing technology," *Trends Genet.*, vol. 30, no. 9, pp. 418–426, Sep. 2014.

[2] S. Goodwin, J. D. McPherson, and W. R. McCombie, "Coming of age: Ten years of next-generation sequencing technologies," *Nature Rev. Genet.*, vol. 17, no. 6, p. 333, 2016.

[3] Human Genome Sequencing Consortium, "Finishing the euchromatic sequence of the human genome," *Nature*, vol. 431, no. 7011, p. 931, 2004.

[4] J. Kim, M. Ji, and G. Yi, "A review on sequence alignment algorithms for short reads based on next-generation sequencing," *IEEE Access*, vol. 8, pp. 189811–189822, 2020.

[5] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Sov. Phys.-Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[6] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Mol. Biol.*, vol. 147, no. 1, pp. 195–197, 1981.

[7] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, Mar. 1970.

[8] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped BLAST and PSI-BLAST: A new generation of protein database search programs," *Nucleic Acids Res.*, vol. 25, no. 17, pp. 3389–3402, 1997.

[9] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa, "Fast gap-affine pairwise alignment using the wavefront algorithm," *Bioinformatics*, vol. 37, no. 4, pp. 456–463, Sep. 2020.

[10] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, no. 4, p. 357, 2012.

[11] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," 2013, *arXiv:1303.3997*.

[12] H. Li, "Minimap2: Pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[13] S. Marco-Sola, M. Sammeth, R. Guigó, and P. Ribeca, "The GEM mapper: Fast, accurate and versatile alignment by filtration," *Nature Methods*, vol. 9, no. 12, pp. 1185–1188, Dec. 2012.

[14] M. Alser, Z. Bingol, D. S. Cali, J. Kim, S. Ghose, C. Alkan, and O. Mutlu, "Accelerating genome analysis: A primer on an ongoing journey," *IEEE Micro*, vol. 40, no. 5, pp. 65–75, Sep. 2020.

[15] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating read mapping with FastHASH," *BMC Genomics*, vol. 14, no. S1, pp. 1–13, Jan. 2013.

[16] H. Xin, J. Greth, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, "Shifted Hamming distance: A fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping," *Bioinformatics*, vol. 31, no. 10, pp. 1553–1560, May 2015.

[17] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "Gate-Keeper: A new hardware architecture for accelerating pre-alignment in DNA short read mapping," *Bioinformatics*, vol. 33, no. 21, pp. 3355–3363, 2017.

[18] M. Alser, O. Mutlu, and C. Alkan, "MAGNET: Understanding and improving the accuracy of genome pre-alignment filtering," 2017, *arXiv:1707.01631*.

[19] J. S. Kim, D. S. Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies," *BMC Genomics*, vol. 19, no. S2, pp. 23–40, May 2018.

[20] M. Alser, H. Hassan, A. Kumar, O. Mutlu, and C. Alkan, "Shouji: A fast and efficient pre-alignment filter for sequence alignment," *Bioinformatics*, vol. 35, no. 21, pp. 4255–4263, Nov. 2019.

[21] J. C. G. Maghirang, R. L. Uy, K. V. A. Borja, and J. L. Pernez, "QCKer-FPGA: An FPGA implementation of Q-gram counting filter for DNA sequence alignment," in *Proc. IEEE 11th Int. Conf. Humanoid, Nanotechnol., Inf. Technol., Commun. Control, Environ., Manage. (HNICEM)*, Nov. 2019, pp. 1–6.

[22] M. Alser, T. Shahroodi, J. Gómez-Luna, C. Alkan, and O. Mutlu, "SneakySnake: A fast and accurate universal genome pre-alignment filter for CPUs, GPUs and FPGAs," *Bioinformatics*, vol. 36, nos. 22–23, pp. 5282–5290, Apr. 2021.

[23] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, A. Nori, A. Scibisz, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "GenASM: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis," 2020, *arXiv:2009.07692*.

[24] N. Ahmed, K. Bertels, and Z. Al-Ars, "A comparison of seed-and-extend techniques in modern DNA read alignment algorithms," in *Proc. IEEE Int. Conf. Bioinf. Biomed. (BIBM)*, Dec. 2016, pp. 1421–1428.

[25] B. Berger, M. S. Waterman, and Y. W. Yu, "Levenshtein distance, sequence comparison and biological database search," *IEEE Trans. Inf. Theory*, vol. 67, no. 6, pp. 3287–3294, Jun. 2021.

[26] S. Burkhardt, "Filter algorithms for approximate string matching," Ph.D. dissertation, Univ. des Saarlandes, Saarbrücken, Germany, 2002. [Online]. Available: https://publikationen.sulb.uni-saarland.de/bitstream/20.500.11880/25741/1/StefanBurkhardt_ProfDrHansPeterLehnhof.pdf
[27] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. Spring Joint Comput. Conf.-AFIPS (Spring)*, 1967, pp. 483–485.
[28] S. Krishnaprasad, "Uses and abuses of Amdahl's law," *J. Comput. Sci. Colleges*, vol. 17, no. 2, pp. 288–293, Dec. 2001.
[29] L. Cai, Q. Wu, T. Tang, Z. Zhou, and Y. Xu, "A design of FPGA acceleration system for myers bit-vector based on OpenCL," in *Proc. Int. Conf. Intell. Informat. Biomed. Sci. (ICIIBMS)*, Nov. 2019, pp. 305–312.
[30] D. P. Bautista, R. C. Aguilera, F. A. Acevedo, and I. A. Badillo, "Bit-vector-based hardware accelerator for DNA alignment tools," *J. Circuits, Syst. Comput.*, vol. 30, no. 5, Apr. 2021, Art. no. 2150087.
[31] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *J. ACM*, vol. 46, no. 3, pp. 395–415, May 1999.
[32] D. Castells-Rufas, S. Marco-Sola, Q. Aguado-Puig, A. Espinosa-Morales, J. C. Moure, L. Alvarez, and M. Moreto, "OpenCL-based FPGA accelerator for semi-global approximate string matching using diagonal bit-vectors," in *Proc. 31st Int. Conf. Field-Program. Log. Appl. (FPL)*, Aug. 2021, pp. 174–178.
[33] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "RIFFA 2.1: A reusable integration framework for FPGA accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 4, pp. 1–23, Sep. 2015.
[34] S. Pilz, F. Porrmann, M. Kaiser, J. Hagemeyer, J. M. Hogan, and U. Rückert, "Accelerating binary string comparisons with a scalable, streaming-based system architecture based on FPGAs," *Algorithms*, vol. 13, no. 2, p. 47, Feb. 2020.
[35] L. Forget, Y. Uguen, F. de Dinechin, and D. Thomas, "A type-safe arbitrary precision arithmetic portability layer for HLS tools," in *Proc. 10th Int. Symp. Highly-Efficient Accel. Reconfigurable Technol. (HEART)*, 2019, pp. 1–6.
[36] A. Ortiz, "An introduction to metaprogramming," *Linux J.*, vol. 2007, no. 158, p. 6, 2007.
[37] J. Hunt, "Java server pages," in *Java Object Orientation: An Introduction*. London, U.K.: Springer, 2002, pp. 361–370.
[38] C. Wolf, Ed., *RISC-V Bitmanip Extension. Document Version 0.93*, Symbiotic GmbH, 2021. [Online]. Available: https://github.com/riscv/riscv-bitmanip/releases/download/v0.93/bitmanip-0.93.pdf
[39] S. Lloyd and Q. O. Snell, "Hardware accelerated sequence alignment with traceback," *Int. J. Reconfigurable Comput.*, vol. 2009, pp. 1–10, Jan. 2009.
[40] M. Šošić and M. Šikić, "Edlib: A C/C++ library for fast, exact sequence alignment using edit distance," *Bioinformatics*, vol. 33, no. 9, pp. 1394–1395, May 2017.
[41] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, S. C. Sahinalp, R. A. Gibbs, and E. E. Eichler, "Personalized copy number and segmental duplication maps using next-generation sequencing," *Nature Genet.*, vol. 41, no. 10, p. 1061, 2009.
[42] S. O. Settle and others, "High-performance dynamic programming on FPGAs with OpenCL," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2013, pp. 1–6.
[43] A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, "Thread-cooperative, bit-parallel computation of levenshtein distance on GPU," in *Proc. 28th ACM Int. Conf. Supercomput. (ICS)*, 2014, pp. 103–112.
[44] A. Sirasao, E. Delaye, R. Sunkavalli, and S. Neuendorffer, "FPGA based OpenCL acceleration of genome sequencing software," *System*, vol. 128, no. 8.7, p. 11, 2015.
[45] E. Rucci, C. García, G. Botella, A. De Giusti, M. Naiouf, and M. Prieto-Matías, "An energy-aware performance analysis of SWIMM: Smith–Waterman implementation on Intel's multicore and manycore architectures," *Concurrency Comput., Pract. Exper.*, vol. 27, no. 18, pp. 5517–5537, Dec. 2015.
[46] L. Di Tucci, K. O'Brien, M. Blott, and M. D. Santambrogio, "Architectural optimizations for high performance and energy efficient Smith–Waterman implementation on FPGAs using OpenCL," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 716–721.
[47] R. B. Abdelhamid and Y. Yamaguchi, "A block-based systolic array on an HBM2 FPGA for DNA sequence alignment," in *Proc. Int. Symp. Appl. Reconfigurable Comput.*, 2020, pp. 298–313.
[48] Y.-L. Chen, B.-Y. Chang, C.-H. Yang, and T.-D. Chiueh, "A high-throughput FPGA accelerator for short-read mapping of the whole human genome," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 6, pp. 1465–1478, Jun. 2021.
[49] D. Castells-Rufas, A. Saa-Garriga, and J. Carrabina, "Energy efficiency of many-soft-core processors," in *Proc. HIP3ES Workshop*, 2016. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1601/1601.07133.pdf

DAVID CASTELLS-RUFAS received the B.S., M.S., and Ph.D. degrees in computer science from the Universitat Autònoma de Barcelona (UAB), Spain, in 1994, 2009, and 2016, respectively. From 2003 to 2018, he was a Research Assistant and a Postdoctoral Researcher with the CEPHIS/CAIAC Research Center, UAB, where he was also an Adjunct Professor. Since 2020, he has been with the Computer Architecture and Operating Systems Department, UAB. His research interests include high performance computing, reconfigurable systems, and embedded systems.

SANTIAGO MARCO-SOLA received the M.Sc. and Ph.D. degrees in computer science from the Universitat Politècnica de Catalunya (UPC), in 2017. During his Ph.D., he worked at the Algorithm Development and Bioinformatics Group, Spanish National Centre for Genome Analysis (CNAG). He is currently a Postdoctoral Researcher at the Barcelona Supercomputing Center (BSC) and a Lecturer at the Universitat Autònoma de Barcelona (UAB). He participates in the project "Designing RISC-V-based Accelerators for next-generation Computers (DRAC)." His research interests include high-performance computing, heterogeneous architectures, genomic data analysis, and machine learning algorithms in the context of bioinformatics and computational biology.

JUAN CARLOS MOURE is currently an Associate Professor with the Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona (UAB), where he teaches computer architecture, performance engineering, and parallel programming. He has participated in several European and Spanish projects related to high-performance computing. He is the author of more than 50 papers. His current research interests include massive parallel architectures, programming, and algorithms, mainly focused on computer vision, signal processing, and bioinformatics applications.

QUIM AGUADO received the B.Sc. degree in computer science from the Universitat Autònoma de Barcelona (UAB), in 2019, and the M.Sc. degree in innovation and research in informatics programme from the Universitat Politècnica de Catalunya (UPC). He works at UAB as a Research Engineer in the Project "Designing RISC-V-based Accelerators for next-generation Computers (DRAC)". His research interests include high-performance computing, massively parallel architectures, and GPU programming; with applications to genomics, computational biology, and sequence alignment.

ANTONIO ESPINOSA received the B.Sc. and Ph.D. degrees in computer science, in 1994 and 2000, respectively. He is currently an Associate Professor with the Department of Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona. During the last ten years, he has participated in several European and national projects related to bioinformatics and high-performance computing, in collaboration with a number of biotechnology companies and research institutions.