PALADIN

FPGA Long Read X-Drop Aligner

Alberto Zeni, Guido Walter Di Donato, Francesco Peverelli,
Politecnico di Milano, Milano, Italy,
{*alberto.zeni,guidowalter.didonato, francesco.peverelli*}@mail.polimi.it

June 2020

XILINX
| UNIVERSITY PROGRAM

Xilinx Open Hardware 2020
Team: xohw20_405
Advisor: Marco D. Santambrogio
YouTube video: youtu.be/7Bn-XBZIsmA
Github repository: github.com/albertozeni/XDropXOHW-Public

POLITECNICO MILANO 1863 NECST laboratory POLITECNICO MILANO 1863

1 Introduction

Pairwise alignment is one of the most commonly used workhorses of sequence analysis. It is used to correct raw sequencer reads, assemble them into more complete genomes, search databases for similar sequences, and many other problems. The optimal solutions for this problem require quadratic time (i.e. they take $O(mn)$ time for aligning a sequence A of length m and a sequence B of length n). Namely, Needleman–Wunsch (NW) [1] is used to find the best global alignment by forcing the alignment to extend to the endpoints of both sequences. Alternatively, Smith–Waterman (SW) [2] computes the best local alignment by finding the highest scoring alignment between continuous subsequences of the input sequences.

The popular X-drop [3] algorithm avoids the full quadratic cost by searching only for high-quality alignments, and can be viewed as an approach to accelerate both NW and SW. Most applications of alignment will throw out low quality alignments, which arise when the two strings are not similar. Instead of exploring the whole $m \times n$ space, the X-drop algorithm searches only for alignments that results in a limited number of edits between the two sequences. X-drop keeps a running maximum score and does not explore cell neighborhoods whose score decreases by a user-specified parameter X. It gets its performance benefits from searching a limited space of solutions and stopping early when a good alignment is not possible.

Zhang et al. [3] proved that, for certain scoring matrices, the X-drop algorithm is guaranteed to find the optimal alignment between relatively similar sequences. In practice, the algorithm eliminates searches between sequences that are clearly diverging. This feature is especially effective for many-to-many alignment problems when there is an attempt to align many sequences to many other possibly matching sequences, i.e., the cost is high as is the possibility that some pairs will not align. With X-drop, any spurious candidate pair is readily eliminated because the optimal score quickly drops. Consequently, X-drop and its variants are the algorithm of choice in some of the most popular sequence mapping software including BLAST [4], LAST [5], BLASTZ [6] with Y-drop, and minimap2 [7] with Z-drop.

Although X-drop is a heuristic for cutting the cost of alignment, it also produces good quality results, which are sometimes better than a more complete search. Frith et al. [8] show that a large X does not necessarily produce better alignments. Without the X-drop feature, the alignment algorithm can incorrectly glue two independent local alignments into a large one. For example, consider two sequences, one of the form S = A-B-C and other of the form R = A-D-C. Since the regions A and C produce high-scoring alignments, likely a high X would incorrectly determine that $\mathit{score}(\text{S,R}) > \mathit{max}(\mathit{score}(\text{A,A}), \mathit{score}(\text{C,C}))$ provided that B and D regions are short enough.

Although there are numerous FPGA implementations of the full $O(mn)$

SW and NW algorithms that often achieve impressive computational rates (measured in CUPS or cell updates per second), they are rarely incorporated into high-impact genomics pipelines due to their quadratic complexity. By contrast, a FPGA implementation of X-drop is notably missing from the literature despite its benefits and popularity. This is likely due to the increased complexity of implementing X-drop efficiently on a FPGA, compared with NW and SW methods, because of the dynamic nature of the computation, its adaptive band, and the need to check for completion.

Our main contributions are:

- We present the first FPGA implementation of the X-drop algorithm, named PALADIN, which achieves significant speed-ups over leading versions on state-of-the-art processors.
- We present an architecture with multiple cores already designed to scale up for any database size.

2 Related Works

The majority of hardware acceleration efforts for pairwise alignment have focused on the Smith-Waterman (SW) and Needleman–Wunsch (NW) algorithms. These find exact alignments and have quadratic complexity in the lengths of the reads. Along with some of the most successful NW and SW acceleration efforts, we review the few efforts to accelerate heuristics more similar to our own. Though they are more generally applicable, exploiting FPGA parallelism in these heuristics is more challenging due to their adaptive nature. As a common success metric, we report Giga Cell Update per Second (GCUPS), as reported by the original work, throughout this section. This measures the amount of cell updates has been made in the amount of time. It is important to keep in mind that the GCUPS rates presented in this section were collected by the respective authors using different architectures than examined in our study.

A recent work by Turakhia et al. [9], Darwin, exploits FPGAs to speed up the alignment process achieving up to 45 GCUPS. Darwin uses a seed-and-extend heuristic (GATC) that performs the extension stage in the seed-and-extend paradigm in which Dynamic Programming (DP) is used around the seed hit to obtain local alignments similar to SW.

Feng et al. [10] recently presented accelerator-based optimizations for minimap2 [11]. Leveraging the GPU architecture, they accelerate minimap2's *seed-chain-extend* pairwise alignment algorithm, which is quadratic in the length of the reads when computing traceback and linear otherwise. They reported performance of 96.5 GCUPS (a 7.1× speed-up over the minimap2's SIMD software implementation). Our X-drop alignment algorithm in this study computes a similar heuristic to minimap2's pairwise alignment. Despite a large number of sophisticated implementations, only Zeni et al. [12] presented a valid implementation of the algorithm on GPU, as the overwhelming majority of the proposed hardware accelerations studies implement the exact SW or NW algorithms. Although the latter X-drop implementation is very efficient, we believe that it can be improved if properly implemented as FPGAs have already proven to be more effective to solve this particular type of problems [13].

3 Background

This section provides an overview of the X-drop implementation proposed by Zhang et al. [3] and implemented in the SeqAn library [14], a C++ library for sequence analysis. First, we review the formal definition of alignment. A *pairwise alignment* of sequences s and t over an alphabet Σ is defined as the pair (s', t') such that $s', t' \in \Sigma \cup \{-\}$ and the following properties hold:

1. $|s'| = |t'| = l$
2. $\forall_{i=1}^{l} \;\; s'_i \neq - \;\; OR \;\; t'_i \neq -$
3. Deleting all "$-$" from s' yields s, and deleting all "$-$" from t' yields t.

A *scoring scheme* is used to distinguish high-quality alignments from the many (valid) alignments of a given pair of sequences. Scoring schemes generally reward matches and penalize mismatches, insertions, and deletions.

3.1 The X-drop Algorithm

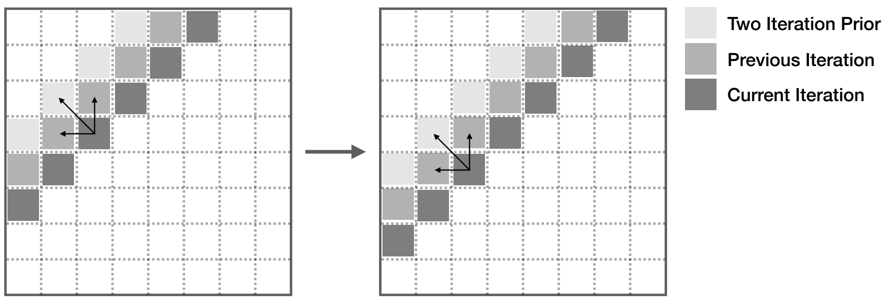

Figure 1: Each cell at the current iteration has two dependencies on cells from the previous iteration and one dependency on a cell at two iteration prior.

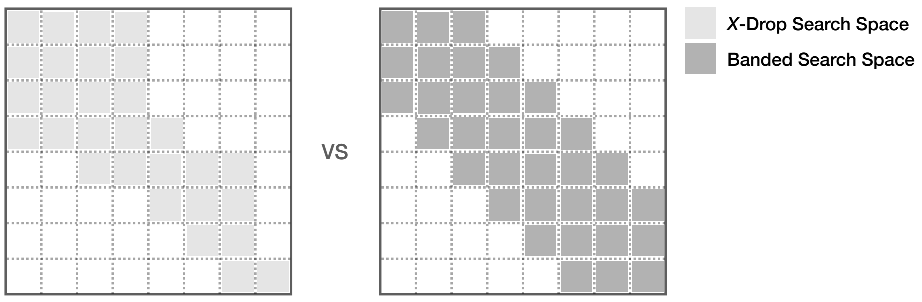

Figure 2: Comparison between the search space of an X-drop alignment algorithm versus the search space of a banded-alignment algorithm.

Given two DNA sequences $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$ of length m and n, the goal of the X-drop algorithm is to find the highest-scoring *semi-global* alignment between A and B of the forms $a_1a_2 \ldots a_i$ and $b_1, b_2 \ldots b_j$, for some $i \leq m$ and $j \leq n$ that are chosen to maximize the score.

For a given i and j, we define S as the alignment matrix and $S(i,j)$ as the alignment score between A and B. A positive *match* score is added to $S(i,j)$ for each pair of identical nucleotides. If nucleotides do not match, the algorithm can either subtract a *mismatch* score to $S(i,j)$ and move diagonally or subtract a *gap* score and move horizontally (gap into the vertical sequence) or vertically (gap into the horizontal sequence) in the dynamic programming grid. More formally, each cell of the alignment matrix S is computed as follows:

$$S(i,j) = \begin{cases} S(i-1,j-1) + match & \text{if } i > 0, j > 0 \text{ and } a_i = b_j \\ S(i-1,j-1) + mismatch & \text{if } i > 0, j > 0 \text{ and } a_i \neq b_j \\ S(i,j-1) + gap & \text{if } j > 0 \\ S(i-1,j) + gap & \text{if } i > 0 \end{cases}$$

Figure 1 shows the three dependencies of a cell during the computation: two dependencies on cells from the previous iteration and one dependency on a cell at two iterations prior. Note that SW, NW, as well as the majority of their heuristic implementations show these dependencies. SW and NW compute the entire S matrix to find the optimal alignment. This quadratic algorithm is extremely inefficient in the case of either misalignment or when aligning almost identical sequences. A misalignment could happen, for example, when two sequences have a tiny region in common due to a genomic repetition. The SW algorithm would spend significant computational resources calculating the entire dynamic programming (DP) matrix and report a very poor alignment score between the two sequences. On the other hand, SW or NW on two almost identical sequences would compute the whole DP matrix with no additional benefit, since the optimal alignment score would

Algorithm 1 Pairwise alignment of S_q and S_t with X-drop

```
procedure PairwiseAlignment($S_q$, $S_t$, $X$)
    $A1, A2, A3$                                  ▷ Create anti-diagonal
    $best \leftarrow 0$                           ▷ Initialize best score to 0
    while $A1.size() \neq 0$ do                   ▷ DP matrix
        $A1 \leftarrow A3$.                       ▷ Anti-diagonal swap
        $A2 \leftarrow A1$.
        $A3 \leftarrow A2$.
        ComputeAntiDiag($A1$, $A2$, $A3$)
        $best \leftarrow A1.max()$
        for $k \leftarrow 0$ to $A1.size()$ do
            if $A1[k] = -\infty$ then
                ReduceAntiDiagFromStart($A1$)
        for $k \leftarrow A1.size()$ to 0 do
            if $A1[k] = -\infty$ then
                ReduceAntiDiagFromEnd($A1$)
    return($best$)                                ▷ $X$-drop termination
```

remain close to the diagonal of the DP matrix.

The concept of the X-drop termination consists of halting the computation if the alignment score drops more than X below the best alignment score σ seen so far for that pair of sequences. σ is potentially updated at each anti-diagonal iteration. If $S(i,j) < \sigma - X$, we set the cell $S(i,j)$ equal to $-\infty$ and no longer consider that cell for the subsequent iterations of S. The cells set to $-\infty$ are used to compute the lower and upper bound for the next anti-diagonal iteration. This approach limits the anti-diagonal width, reducing the search space of the algorithm, and automatically provides a termination condition. The X-drop algorithm is particularly efficient when two sequences do not align. A pseudo-code of the X-drop algorithm is shown in Algorithm 1.

Note that X-drop should not be confused with the popular banded-SW method. This approach constrains the search space to a fixed band along the diagonal, regardless of the drop in the score. The areas of the $m \times n$ dynamic programming grid explored by these algorithms are characteristically different from X-drop's search space, which is reminiscent of a rugged band with changes in the length of each anti-diagonal, as shown in Figure 2. To better understand the difference in practice, consider two sequences that have very high (over 50%) differences in terms of substitutions but have no indel (insertion or deletion) differences. The optimal path would be along the diagonal because both a mismatch and match move the cursor in both sequences. X-drop will correctly terminate the search early due to a significant drop in the score whereas banded-SW would explore the entire band.

4 Implementation

The key aspect of our implementation is exploiting two different levels of parallelism as possible on the Field Programmable Gate Array (FPGA). Intra-level parallelism is achieved by scheduling multiple Compute Units (CUs) based on the value of X and through the use of in-warp parallelization to find the maximum of the anti-diagonals. To achieve inter-level parallelism, we implemented the parallel execution of multiple alignments by building two cores and scheduling one core to execute the seed extension on the right and the other one on the left.

In this section, we describe the design of our implementation.

4.1 FPGA Implementation

We first describe how we implemented a single core of our architecture. Each core exploits intra-sequence parallelism, which is the parallelization of a single pairwise alignment and its anti-diagonals computation. Given that the X-drop algorithm we decided to port to FPGA does not perform alignment trace-back, we do not store the entire alignment matrix on the device for each alignment. Therefore, we can reduce the memory footprint of our kernel on the FPGA by storing only three anti-diagonals per alignment: current, previous, and two iterations prior, as highlighted in Figure 1. We store all of the antidiagonals and the sequences on the High Bandwidth Memory (HBM) of the FPGA. In this way we are able to store a lot of data on the FPGA, while still mantaining a fast data access. Similarly to SW and NW algorithms, we compute the cell updates of each anti-diagonal of the alignment matrix in parallel. Each anti-diagonal cell has three dependencies on cells from anti-diagonals at previous iterations. Nevertheless, we can leverage the independence between cells belonging to the same anti-diagonal. To compute an anti-diagonal update in parallel, we assign each cell to a FPGA CU and compute them independently, where each of our cores has 64 CUs. To overcome this limitation and ensure the computation of any anti-diagonal length, we split each anti-diagonal into *segments*, as shown in Figure 3. The anti-diagonal is split into segments whose width is equal to the number of threads within a block. Once a segment of the anti-diagonal is completed, the kernel initiates the computation of the subsequent segment. This process is repeated until the entire anti-diagonal is computed.

For each cell, the corresponding CU sets the score to $-\infty$ if the cell score drops X below the global maximum of the alignment matrix, that is *best* in Algorithm 1. The overall maximum score is a shared variable within the considered block. The global maximum of the scoring matrix is updated after any anti-diagonal has been completely calculated since it needs to consider the newly computed scores. Computing the global maximum naively would significantly slow down the execution since it would require

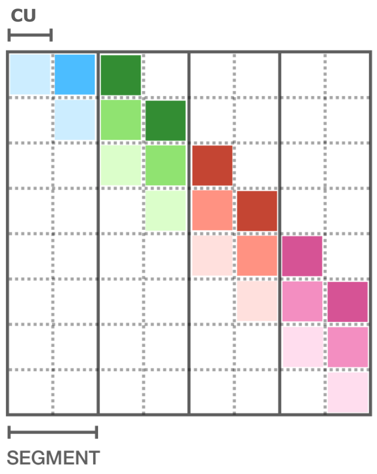

Figure 3: Each anti-diagonal is divided in segments, whose width is equal to the number of CU scheduled in a block

serial comparison of each cell with the all others in a given anti-diagonal. Thus we speed up this process by computing the maximum anti-diagonal score via a parallel reduction.

Algorithm 2 Computation of the anti-diagonal in parallel

```
1: procedure AntiDiag($A1, A2, A3, S_q, S_v, X, best$)
2:     $CU \leftarrow ComputeUnit$
3:     while $CU < A1.size()$ do
4:         if $S_q[CU] == S_v[CU]$ then
5:             $A1[CU] \leftarrow A3[CU-1] + \textbf{match}$
6:         else
7:             $A1[CU] \leftarrow A3[CU-1] + \textbf{mismatch}$
8:         $tmp \leftarrow \textbf{max}(A2[CU] + \textbf{gap},\ A2[CU-1) + \textbf{gap})$
9:         $A1[CU] \leftarrow \textbf{max}(A1[CU],\ tmp)$
10:        if $A1[CU] < best - X$ then
11:            $A1[CU] \leftarrow -\infty$
12:        $CU \leftarrow CU + numOfCUs$
```

Algorithm 2 illustrates the parallel computation of the anti-diagonal. Finally, the size of the next anti-diagonal to be computed is updated by checking if there are cells marked with $-\infty$ at the end or the start of the current anti-diagonal. PALADIN continues computing the alignment matrix until either it reaches the end of the shortest read or the size of the current anti-diagonal is set to zero, meaning that it has satisfied the condition.

Our design also permits for easy replication. Since the X-Drop alignment is composed by a left and right extension of a given seed, we implemented two

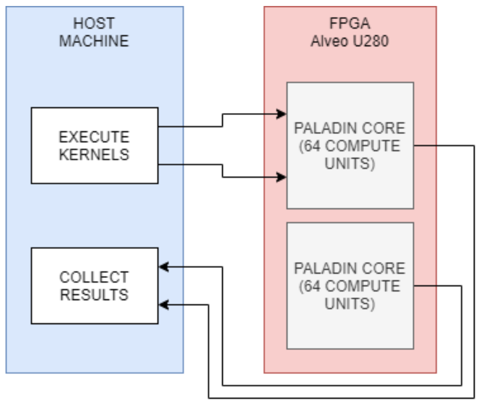
HOST MACHINE
EXECUTE KERNELS
COLLECT RESULTS
FPGA Alveo U280
PALADIN CORE (64 COMPUTE UNITS)
PALADIN CORE (64 COMPUTE UNITS)

Figure 4: Each core receives input independently and then computes everything in parallel. When a core has completed execution, it proceeds to send data to the host machine independently to the other core.

cores in our architecture. Each core is independent to the other and interacts independently with the host (Figure 4). The cores operate in parallel, when the alignment is complete the result of the right or left extension is sent from the core to the host. The host then combines the two outputs together and produces the final result.

5 Experimental Results

5.1 Experimental Settings

The design of the architecture has been described using C++ and Xilinx Vitis HLS. Xilinx Vitis has been used to perform the bitstream generation step. We benchmarked the architecure using the OpenCL events that provide an easy to use API to profile the code that runs on the FPGA device. We compare PALADIN against the CPU-based X-drop algorithm as implemented in SeqAn [14]. To compare PALADIN against SeqAn's, we generate a set of 4.5K read pairs with read length between 2,500 and 7,500 characters and an error rate of ≈15% between two reads of a given pair. All the execution times were measured taking into account sequence preprocessing and PCIe communication, so that our execution times comprehend the same operations that SeqAn performs from start to finish. The results were collected on a server with an 8-core/16-thread Intel Xeon Processor E5-2640 and an Alveo U280, with 128 GB DDR4 of RAM on the NIMBIX Platform.

5.2 Performance Analysis

Figure 5 shows PALADIN's speed-up using the Xilinx Alveo U280 compared against SeqAn's implementation using 16 threads on the Xeon processors.

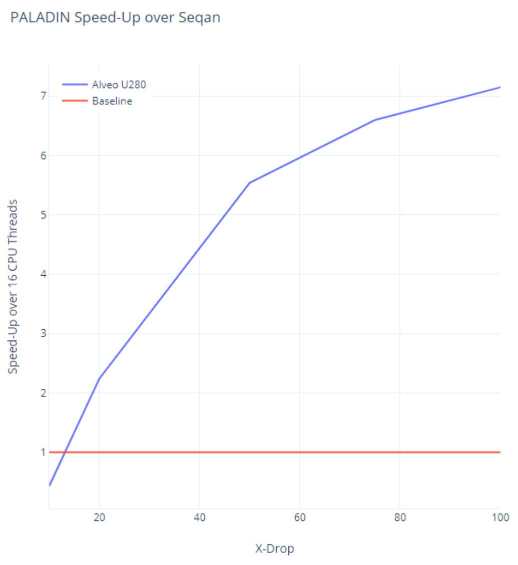

Figure 5: PALADIN's speed-up over SeqAn for 4.5K alignments. Intel Xeon Platform with Xilinx Alveo U280.

Table 1: **PALADIN** and **SeqAn** execution times in seconds for 4.5K alignments (Intel Xeon platform with the Alveo U280).

***X*-Drop**	**SeqAn 16 CPU Threads**	**PALADIN FPGA**
10	4.6	10.6
20	25.6	11.4
50	73.8	13.3
75	102.2	15.4
100	129.4	18.1

Details of the execution time are shown in Table 1. We can see that PALADIN is able to achieve a 7.1× improvement in performance. Note that PALADIN's execution times remain roughly constant for large values of X. In these scenarios, we can exploit the full parallelism of the FPGA architecture, resulting in much faster execution times. As expected, PALADIN achieves higher speed-ups as the value of X increases, since the alignment runs for a longer time.

6 Conclusions and future works

Our work presents PALADIN, the first FPGA implementation of the X-drop alignment algorithm. X-drop is employed in several important genomics applications, however it is particularly challenging for FPGA parallelization due to its adaptive banding and continual termination checks. Detailed results and analyses show significant performance acceleration using our novel optimization approach. PALADIN demonstrated runtime improvements of up to 7.1× using the Alveo U280, compared with the original CPU algorithm. Future work will focus on implementing more cores inside of PALADIN's architecture. As each alignment is independent to the others, we are planning to introduce even more cores inside of PALADIN. Another aspect that we'll be investigating is data compression. By compressing data on the host, we can exploit even better parallelism on FPGA, since we will be able to schedule more CUs and also more cores.

References

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

[2] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[3] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller, "A greedy algorithm for aligning DNA sequences," *Journal of Computational biology*, vol. 7, no. 1-2, pp. 203–214, 2000.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[5] S. M. Kiełbasa, R. Wan, K. Sato, P. Horton, and M. C. Frith, "Adaptive seeds tame genomic sequence comparison," *Genome research*, vol. 21, no. 3, pp. 487–493, 2011.

[6] S. Schwartz, W. J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller, "Human–mouse alignments with BLASTZ," *Genome research*, vol. 13, no. 1, pp. 103–107, 2003.

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

[8] M. C. Frith, M. Hamada, and P. Horton, "Parameters for accurate genome alignment," *BMC bioinformatics*, vol. 11, no. 1, p. 80, 2010.

[9] Y. Turakhia, G. Bejerano, and W. J. Dally, “Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly,” in *ACM SIGPLAN Notices*, vol. 53. ACM, 2018, pp. 199–213.

[10] Z. Feng, S. Qiu, L. Wang, and Q. Luo, “Accelerating long read alignment on three processors,” in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019. New York, NY, USA: ACM, 2019, pp. 71:1–71:10. [Online]. Available: http://doi.acm.org/10.1145/3337821.3337918

[11] H. Li, “Minimap2: pairwise alignment for nucleotide sequences,” *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 05 2018. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty191

[12] A. Zeni, G. Guidi, M. Ellis, N. Ding, M. D. Santambrogio, S. Hofmeyr, A. Buluç, L. Oliker, and K. Yelick, “Logan: High-performance gpu-based x-drop long-read alignment,” *arXiv preprint arXiv:2002.05200*, 2020.

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

[14] A. Döring, D. Weese, T. Rausch, and K. Reinert, “SeqAn an efficient, generic C++ library for sequence analysis,” *BMC bioinformatics*, vol. 9, no. 1, p. 11, 2008.