# LOGAN: High-Performance GPU-Based $X$-Drop Long-Read Alignment

Alberto Zeni*, Giulia Guidi†‡, Marquita Ellis†‡, Nan Ding‡, Marco D. Santambrogio*,
Steven Hofmeyr‡, Aydın Buluç†‡, Leonid Oliker‡, Katherine Yelick†‡

*Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy
†Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA, USA
‡Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA
**Availability**: https://github.com/albertozeni/LOGAN
**Contact**: alberto.zeni@mail.polimi.it, gguidi@lbl.gov

***Abstract*—Pairwise sequence alignment is one of the most computationally intensive kernels in genomic data analysis, accounting for more than 90% of the runtime for key bioinformatics applications. This method is particularly expensive for *third-generation* sequences due to the high computational cost of analyzing sequences of length between 1Kb and 1Mb. Given the quadratic overhead of exact pairwise algorithms for long alignments, the community primarily relies on approximate algorithms that search only for high-quality alignments and stop early when one is not found. In this work, we present the first GPU optimization of the popular $X$-drop alignment algorithm, that we named LOGAN. Results show that our high-performance multi-GPU implementation achieves up to 181.6 GCUPS and speed-ups up to 6.6× and 30.7× using 1 and 6 NVIDIA Tesla V100, respectively, over the state-of-the-art software running on two IBM Power9 processors using 168 CPU threads, with equivalent accuracy. We also demonstrate a 2.3× LOGAN speed-up versus ksw2, a state-of-art vectorized algorithm for sequence alignment implemented in minimap2, a long-read mapping software. To highlight the impact of our work on a real-world application, we couple LOGAN with a many-to-many long-read alignment software called BELLA, and demonstrate that our implementation improves the overall BELLA runtime by up to 10.6×. Finally, we adapt the Roofline model for LOGAN and demonstrate that our implementation is near optimal on the NVIDIA Tesla V100s.**

## I. 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 $\mathit{score}(\mathrm{S},\mathrm{R}) > \mathit{max}(\mathit{score}(\mathrm{A},\mathrm{A}), \mathit{score}(\mathrm{C},\mathrm{C}))$ provided that B and D regions are short enough.

Although there are numerous GPU 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 GPU 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 GPU, 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 high-performance, multi-GPU implementation of the $X$-drop algorithm, named LOGAN, which achieves significant speed-ups over leading versions on state-of-the-art processors.
- We integrate LOGAN within BELLA, a long-read many-to-many overlapping and alignment software and demonstrate performance improvements up to $10\times$.
- We adapt the Roofline Model to LOGAN implementation and underlying hardware, and demonstrate that performance is near optimal on NVIDIA Tesla V100s.

A key aspect of our implementation is combining different levels of parallelism. Specifically, we implement the *intra-sequence* parallelism via dynamic thread scheduling and in-warp parallelism, while accomplishing *inter-sequence* parallelism by assigning each GPU block to a single alignment. Finally, we carry out parallelism across multiple GPUs through a load balancer.

The remainder of the paper is organized as follows. Section II provides an overview of the related work, while Section III describes the original software algorithm we port on GPU. Section IV describes our implementation and optimizations. Section V presents LOGAN integration within BELLA [9], a long-read overlapping and alignment software. Section VI illustrates our experimental results, while Section VII describes the Roolfline model used to analyze our implementation. Finally, Section VIII summarizes our contributions and outlines future work.

## II. Related Work

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 GPU parallelism in these heuristics is more challenging due to their adaptive nature. As a common success metric, we report Giga Cell Updates Per Second (GCUPS), as reported by the original work, throughout this section. 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. In Section VI, we collect comparative performance data with the ksw2 algorithm on equivalent platforms.

The implementation of Michael Farrar [10], which is adopted in Bowtie2 [11], stands out amongst software implementations of the SW algorithm. It leverages SIMD instructions and reaches performance of more than 20 GCUPS on an Intel Xeon Gold with 40 CPU threads. The same software implementation has been optimized for the PlayStation 3 processor and the IBM QS20 architecture [12], achieving performance of $15.5$ and $11.6$ GCUPS, respectively.

CUDASW++3 [13] accelerates the SW algorithm combining SIMD instructions and GPU parallelism. The implementation achieves up to $185.6$ GCUPS when aligning reads with length less than $400$ characters. However, the performance drops significantly when the sequence length exceeds $400$ characters. Additionally, when running only using the GPU, their maximum attained performance is $68$ GCUPS (roughly $1/3$ of their peak performance).

Muhammadzadeh presented MR-CUDASW++ [14], which was inspired by CUDASW++3 but optimized for "medium length" reads. Muhammadzadeh compares MR-CUDASW++ to other tools, with CUDASW++3 as its closest contender, across sequences lengths of 1K, 10K, and 100K. MR-CUDASW++ achieved speedups of $1-2\times$ over CUDASW++3. The results were below 85 GCUPS using an NVIDIA Tesla V100, which is the same GPU used for our benchmarks. Li et al. [15] accelerate the SW algorithm achieving a speed-up of over $160\times$ compared to software implementation using an Altera Nios II Field Programmable Gate Array (FPGA). Nevertheless, the performance of their proposed solution is comparable to existing optimized software implementations. The SW implementation of Di Tucci et al. [16], running on a Xilinx Virtex 7 and a Kintex Ultrascale platforms, achieves up to $42.5$ GCUPS and a speed-up of $1.7\times$ over the state-of-the-art FPGA implementation. However, this work is limited to aligning short sequences that have a number of characters not exceeding the number of processing elements in the architecture.

A recent work by Turakhia et al. [17], 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. [18] recently presented accelerator-based optimizations for minimap2 [19]. 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 $7.1\times$ 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. In Section VI, we compare our work with ksw2 [20] (i.e., minimap2's alignment kernel), showing that LOGAN achieves higher performance in terms of GCUPS than both ksw2 and the performance reported by Feng et al. for their GPU-accelerated implementation.

Despite a large number of sophisticated implementations, the overwhelming majority of the proposed hardware accelerations studies implement the exact SW or NW algorithms.

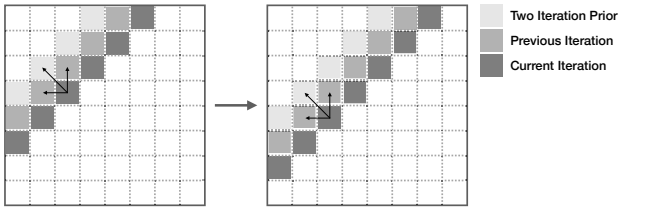


Fig. 1. Each cell at the current iteration has two dependencies on cells from the previous iteration and one dependency on a cell at two iteration prior.

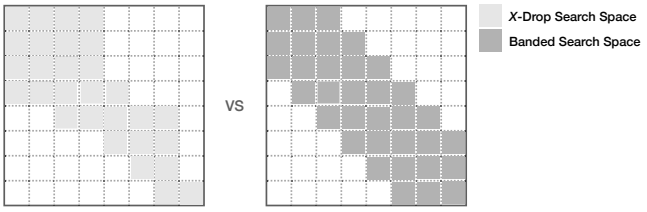


Fig. 2. Comparison between the search space of an $X$-drop alignment algorithm versus the search space of a banded-alignment algorithm.

We believe the although $X$-drop algorithm is the most practical choice for targeting large-scale alignments, it requires more challenging parallelization than the original SW or NW methods. Though relatively unexplored due to this challenge, we demonstrate that GPU optimization results in significant acceleration.

## III. Background

This section provides an overview of the $X$-drop implementation proposed by Zhang et al. [3] and implemented in the SeqAn library [21], 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 $\Sigma$ 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.

### A. The $X$-drop Algorithm

Given two DNA sequences $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$ 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 $a_1 a_2 \dots a_i$ and $b_1, b_2 \dots b_j$, 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 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 $\sigma$ seen so far for that pair of sequences. $\sigma$ 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.

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

```
1:  procedure PairwiseAlignment(S_q, S_t, X)
2:      A1, A2, A3                          ▷ Create anti-diagonal
3:      best ← 0                            ▷ Initialize best score to 0
4:      while A1.size() ≠ 0 do              ▷ DP matrix
5:          A1 ← A3.                        ▷ Anti-diagonal swap
6:          A2 ← A1.
7:          A3 ← A2.
8:          ComputeAntiDiag(A1, A2, A3)
9:          best ← A1.max()
10:         for k ← 0 to A1.size() do
11:             if A1[k] = −∞ then
12:                 ReduceAntiDiagFromStart(A1)
13:         for k ← A1.size() to 0 do
14:             if A1[k] = −∞ then
15:                 ReduceAntiDiagFromEnd(A1)
16:     return(best)                        ▷ X-drop termination
```

## IV. Implementation

The key aspect of our implementation is exploiting as many levels of parallelism as possible on the Graphical Processing Unit (GPU). Intra-level parallelism is achieved by dynamically scheduling the threads 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 implement the parallel execution of multiple alignments by assigning each GPU block to a single alignment. Multi-GPU parallelism is obtained by implementing a GPU load balancer that adapts the execution of LOGAN to leverage multiple GPUs.

In this section, we describe the design of our implementation. Section V then presents our optimized kernel integration within BELLA, a real-world application. Note that we refer to GPU threads and GPU blocks simply as *threads* and *blocks*.

### A. Intra-Sequence Parallelism

We first consider the intra-sequences 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 GPU 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 GPU by storing only three anti-diagonals per alignment: current, previous, and two iterations prior, as highlighted in Figure 1.

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 GPU thread and compute them independently, where a GPU block can schedule up to 1024 threads. 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.

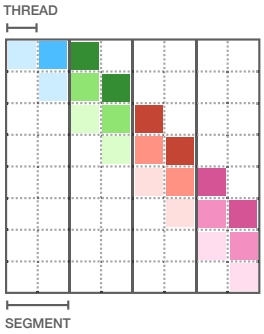


Fig. 3. Each anti-diagonal is divided in segments, whose width is equal to the number of *threads* scheduled in a block

**Algorithm 2** Computation of the anti-diagonal in parallel

```
1:  procedure AntiDiag(A1, A2, A3, S_q, S_v, X, best)
2:      tid ← threadID
3:      while tid < A1.size() do
4:          if S_q[tid] == S_v[tid] then
5:              A1[tid] ← A3[tid − 1] + match
6:          else
7:              A1[tid] ← A3[tid − 1] + mismatch
8:          tmp ← max(A2[tid] + gap, A2[tid − 1) + gap)
9:          A1[tid] ← max(A1[tid], tmp)
10:         if A1[tid] < best − X then
11:             A1[tid] ← −∞
12:         tid ← tid + numScheduledThreads
```

For each cell, the corresponding thread 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 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.

Once each thread is assigned to a cell of the anti-diagonal, our algorithm leverages in-warp thread communication to compare the values inside cells and perform the reduction, where a warp is a set of 32 threads executing the same code and sharing the same data. All threads within a warp communicate using registers, which maximizes communication speed. Algorithm 2 illustrates the parallel computation of the anti-diagonal. Finally, the size of the next anti-diagonal to be

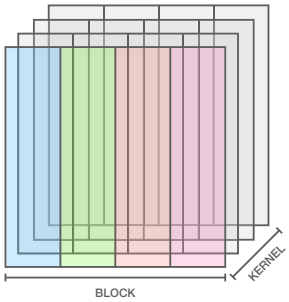


Fig. 4. Each alignment is assigned to one block. The kernel executes all the block in parallel, in order to leverage inter-alignment parallelization.

computed is updated by checking if there are cells marked with $-\infty$ at the end or the start of the current anti-diagonal. LOGAN 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.

### B. Inter-Sequence Parallelism

Intra-sequence parallelism optimizes the alignment computation for a single pair of sequences, however, it does not effectively leverage the large volume of GPU computational resources. We therefore exploit the GPU computational potential by designing LOGAN to align multiple pairs of sequences in parallel by assigning each alignment to a GPU block — thus taking advantage of inter-sequences parallelism. LOGAN schedules the number of GPU blocks based on the number of alignments needed to be performed (Figure 4). Each NVIDIA V100s GPU block can store up to 64KB in shared-memory and performs an independent alignment. Since only three anti-diagonals need to be stored, we could ostensibly store them into GPU shared-memory (the fastest memory available after the registers). However, despite the potential of reserving 64KB of memory per block, this cannot be implemented in practice as the device has only 96KB of memory per streaming multiprocessor (SM). Given that each SM on the device can execute up to 32 blocks in parallel, if a single block has reserved too much memory, a SM is forced to exchange data with the DRAM of the GPU every time it computes a block. Furthermore, given that only a single block could fit on a single SM, the execution would be limited to a single block per SM. Given our goal of achieving the best possible board-level utilization, we need to compute multiple blocks per SM in parallel. Consequently, to overcome these limitations and avoid shared memory contention, LOGAN stores the three anti-diagonals of each alignment on the High Bandwidth Memory (HBM) of the GPU. Doing so, removes the limitation on the number of blocks per SM, and achieves significantly more effective parallelism.

To further improve our resource utilization, LOGAN schedules a number of threads per block lower than the maximum of 1024. In fact, if the number of threads exceeds the anti-diagonal length, many of the threads will stall, decreasing

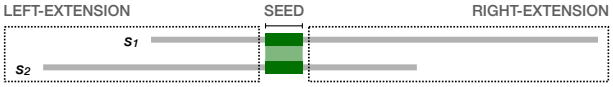


Fig. 5. In the seed-and-extend alignment paradigm, the seed location determines where the read pair is split into two different alignments: left- and right-extension.

TABLE I
$X$-DROP EXECUTION TIMES ON GPU USING $X = 100$ AND EXPLOITING DIFFERENT LEVELS OF PARALLELISM.

| Parallelism | Pairs | Threads | Blocks | Time | Speed-Up |
|---|---|---|---|---|---|
| None | 1 | 1 | 1 | 1.50s | - |
| Intra-sequence | 1 | 128 | 1 | 0.16s | 9.3× |
| Intra-sequence | 100K | 128 | 1 | 45h | - |
| Intra- and inter-sequence | 100K | 128 | 100K | 7.35s | 22000.0× |

overall performance. Additionally, we need to store the intermediate results of the parallel reduction in shared memory to enable in-warp thread communication when computing the anti-diagonal maximum score. Since the number of intermediate results is equal to the number of scheduled threads, reducing the number of scheduled threads per block also reduces the risk of shared memory contention. Given that the length of each anti-diagonal is proportional to the value of $X$, our implementation schedules a number of threads proportional to $X$, significantly reducing the number of stalled threads. This scheduling increases our performance and improves our resource utilization. Table I shows the impact of the various degrees of parallelization implemented in the LOGAN kernel. The first two rows of the table show the impact of intra-level parallelism over a single-thread execution, while the second two show the impact over inter-level parallelism for 100K alignments of read pairs. Note that intra-level parallelism improves the performance by a factor of about $9\times$, while the introduction of inter-level parallelism improves performance by an impressive factor of $22,000\times$ with respect to intra-parallelism alone. The intra-sequence parallelism has insufficient work to consume available GPU resources, hence its impact on performance is limited compared to inter-sequence parallelism. Notably, inter- and intra-sequence parallelisms are complementary to each other. LOGAN therefore implements both to better exploit the resources of the GPU and maximize our kernel performance.

To make the LOGAN processing more efficient on the GPU, we additionally introduce CPU host optimization. First, LOGAN loads the length of the sequences and the seed locations for each pair of sequences and stores them into two buffers. Each pair of sequences is split in two based on the seed's location, resulting in a *left-extension* pair and a *right-extension* pair, as shown in Figure 5. Left-extension and right-extension pairs are stored into two different vectors, and these alignments are computed independently by scheduling two different streams on the GPU. Traditionally, when aligning two sequences, one of them is accessed backward, resulting

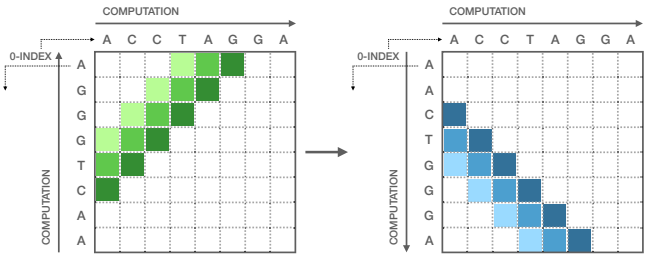


Fig. 6. The query (vertical) sequence is reversed to exploit coalesced memory access on the GPU.

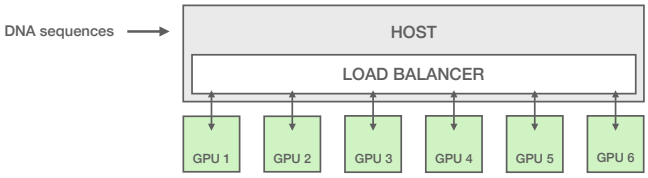


Fig. 7. Load balancing scheme of our multi-GPU implementation to distribute alignments across GPUs.

in memory performance degradation since characters are read in the opposite direction of the memory. To ensure coalesced data access on the GPU and exploit memory burst, one of the two sequences (for each given pair) is reversed, as shown in Figure 6. This optimization linearizes the GPU memory data access, thus increasing performance while preserving the correct solution. Finally, note that kernel execution is scheduled asynchronously from the host, enabling the retrieval of alignment results as soon as they are available, instead of waiting for all the alignments to complete.

### C. Implementation with Multiple GPU Devices

To effectively exploit the available multiple GPU resources, LOGAN leverages a load balancer as shown in Figure 7. This optimization allows LOGAN to run on varying GPU configurations, since it can adapt the load to the specific number of GPUs present within a given system.

The host application balances the computation by scheduling the number of alignments for each GPU. The host switches context multiple times and then replicates the operations for each GPU to simplify its task. The pre-processing of the sequences occurs as in the single device implementation and the load balancer divides the sequences into different groups that are then assigned to the GPUs. The HBM memory of the GPU represents a limiting resource for LOGAN, since in the single GPU implementation it is fully utilized. To ensure balance, we schedule the number of alignments per GPU considering both the number of available GPUs and the length of the sequences. Once the division is completed, the host allocates the necessary memory on the different GPUs, enabling each GPU to execute its set of alignments independently. The host then schedules each GPU kernel to be executed in parallel and collects alignment results asynchronously. Once the GPU devices completed their execution, the load balancer collects and organizes the results.

## V. BELLA Integration

To demonstrate the impact of the work, we integrate LOGAN into a long-read analysis software, called BELLA [9]. BELLA is a recently released, publicly-available software for long-read many-to-many overlap detection and alignment. Detecting overlaps is a crucial and computationally intense step in many long-read applications, such as *de novo* genome assembly and error correction. BELLA uses a seed-based approach for overlap detection implemented as an efficient sparse matrix-matrix multiplication (SpGEMM) kernel. Before performing overlapping, BELLA provides a new algorithm for pruning the $k$-mers, substrings of fixed length $k$ used as seeds. The $k$-mers are pruned because unlikely to be useful in overlap detection and their retention would cause unnecessary computational overhead and potential errors. Once overlaps are identified, a seed-end-extend pairwise alignment step is performed to filter out spurious overlaps. BELLA chooses the optimal $k$-mer to begin alignment extension, as illustrated in Figure 5, through a *binning mechanism*, where $k$-mer locations are used to estimate the overlap length and to "bin" $k$-mers to form a consensus. BELLA additionally uses the novel approach of separating true alignments from false positives using an *adaptive threshold* based on a combination of alignment techniques and probabilistic modeling.

BELLA relies on SeqAn's $X$-drop implementation [21] for pairwise alignment, which constitutes about $90\%$ of the overall runtime when using real data sets. Once overlaps are computed via SpGEMM, BELLA performs the pairwise alignment and determines if the aligned pair should be kept. The current implementation is efficient for SeqAn as the processor computes independent pairwise alignments in parallel using OpenMP [22]. However, this approach is inefficient for the GPU architecture, since it limits the amount of parallelism between alignments. To better exploit inter-alignment parallelism, we modify BELLA to batch the entire set of alignments together and send them to the GPU devices. The host CPU then retrieves and post-processes the alignment results. Our optimized BELLA version with LOGAN integration produces equivalent results as the original version.

## VI. Discussion

In this section, we describe the experimental settings used to evaluate the LOGAN methodology and present our performance results.

### A. Experimental Setting

We first compare LOGAN against the CPU-based $X$-drop algorithm as implemented in SeqAn [21]. Next, we evaluate LOGAN against two GPU-based algorithms: the current state-of-the-art implementation of full Smith-Waterman (SW), CUDASW3++ [13], and the closest heuristics to ours proposed by Feng et al., manymap [18]. Finally, we integrate LOGAN into

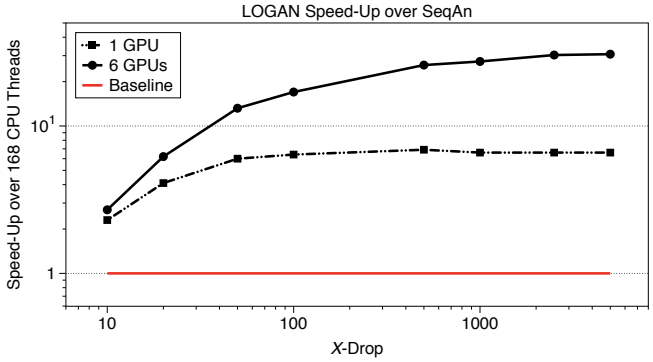


Fig. 8. LOGAN's speed-up over SeqAn for 100K alignments (log-log scale). POWER9 Platform with 6 NVIDIA Tesla V100s.

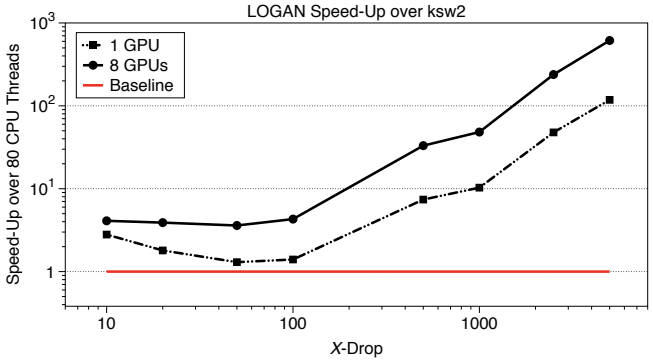


Fig. 9. LOGAN's speed-up over ksw2 for 100K alignments (log-log scale). "Skylake" Platform with 8 NVIDIA Tesla V100s.

TABLE II
**LOGAN** AND **SEQAN** EXECUTION TIMES IN SECONDS FOR 100K ALIGNMENTS (POWER9 PLATFORM WITH 6 NVIDIA TESLA V100S).

| $X$-Drop | SeqAn 168 CPU Threads | LOGAN 1 GPU | LOGAN 6 GPU |
|---|---|---|---|
| 10 | 5.1 | 2.2 | 1.9 |
| 20 | 12.7 | 3.1 | 2.1 |
| 50 | 29.6 | 5.0 | 2.2 |
| 100 | 45.7 | 7.2 | 2.7 |
| 500 | 102.6 | 14.9 | 4.0 |
| 1000 | 133.3 | 20.2 | 4.9 |
| 2500 | 168.0 | 25.3 | 5.6 |
| 5000 | 176.6 | 26.7 | 5.8 |

TABLE III
**LOGAN** AND **KSW2** EXECUTION TIMES IN SECONDS FOR 100K ALIGNMENTS ("SKYLAKE" PLATFORM WITH 8 NVIDIA TESLA V100S).

| $X$-Drop | ksw2 80 CPU Threads | LOGAN 1 GPU | LOGAN 8 GPU |
|---|---|---|---|
| 10 | 6.9 | 2.5 | 1.7 |
| 20 | 7.0 | 3.8 | 1.8 |
| 50 | 7.7 | 5.8 | 2.1 |
| 100 | 10.4 | 7.3 | 2.4 |
| 500 | 113.0 | 15.2 | 3.4 |
| 1,000 | 209.5 | 20.4 | 4.3 |
| 2,500 | 1235.8 | 25.9 | 5.2 |
| 5,000 | 3213.1 | 27.2 | 5.2 |

the BELLA long-read application to demonstrate its benefit in a real-world computation.

To compare LOGAN against SeqAn's, we generate a set of 100K read pairs with read length between 2,500 and 7,500 characters and an error rate of $\approx$15% between two reads of a given pair. The results were collected on a dual-socket server with two 22-core IBM POWER9 processors and 6 NVIDIA Tesla V100s (16 GB HBM2) with 512 GB DDR4 of RAM. Each processor has 21 compute cores with 4 threads per core. Also, we compare LOGAN to minimap2's [19] vectorized $Z$-drop alignment algorithm, called ksw2 [20], using the same data set of 100K pairs described above. Note that we conducted these comparisons on a different hardware platform: a dual-socket computer with two 20-core Intel Xeon Gold 6148 CPU processors, each running at 2.40 GHz with 384 GB DDR4 2400 MHz memory and 8 NVIDIA Tesla V100s (16 GB HBM2) GPUs. A different platform was required since the POWER9 processors are not compatible with ksw2's SSE2 SIMD instructions. On the Intel Xeon Gold platform with 8 NVIDIA Tesla V100s, we also perform the comparison between LOGAN, CUDASW++, and manymap using the same 100K pairs as above.

Additionally, we integrate LOGAN into the BELLA long-read application [9], described in Section V, by evaluating the performance difference of replacing SeqAn with LOGAN. For this comparison, we used a real *E. coli* and a synthetic *C. elegans* data sets, requiring 1.8M and 235M alignments, respectively. We analyzed these experiments on the same hardware platform as SeqAn evaluation.

### B. Results

Figure 8 shows LOGAN's speed-up using both one GPU and the entire set of six GPUs compared against SeqAn's implementation using 168 threads on two POWER9 processors. Details of the execution time are shown in Table II. Note that LOGAN's execution times remain roughly constant for large values of $X$. In these scenarios, we can exploit the full parallelism of the GPU architecture, resulting in similar execution times. Observe that LOGAN attains speed-ups ranging from 2.3$\times$ to 6.6$\times$ for a single GPU and from 2.7$\times$ to 30.7$\times$ using all six GPUs. As expected, LOGAN achieves higher speed-ups as the value of $X$ increases, since the alignment runs for a longer duration. We also note that LOGAN multiple GPU implementation scales better for longer execution runs. This is due to amortizing the load balancing overhead when dividing the sequences into different groups.

Figure 9 presents LOGAN's performance using both 1 GPU and the entire set of 8 GPUs when compared against ksw2's

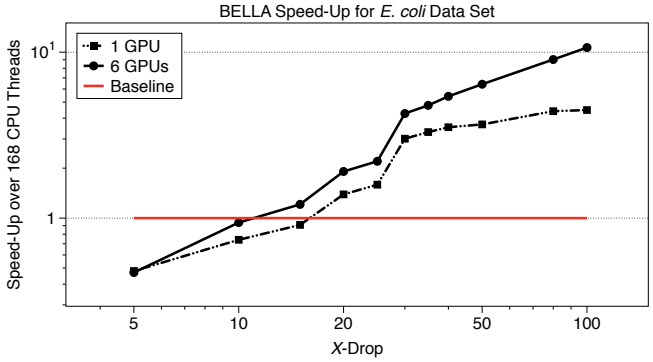


Fig. 10. BELLA's speed-up replacing its pairwise alignment kernel (SeqAn) with LOGAN for the *E. coli* data set for 1.8M alignments (log-log scale). POWER9 Platform with 6 NVIDIA Tesla V100s.

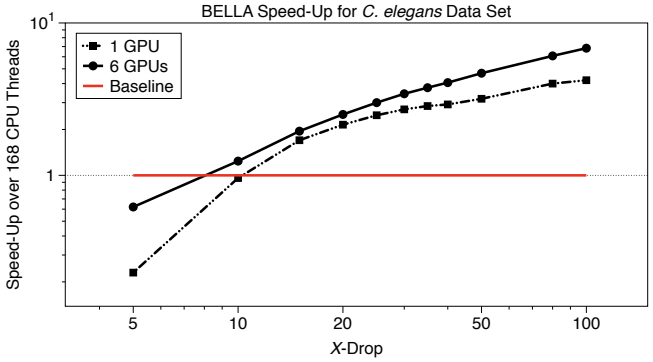


Fig. 11. BELLA's speed-up replacing its pairwise alignment kernel (SeqAn) with LOGAN for the *C. elegans* data set for 235M alignments (log-log scale). POWER9 Platform with 6 NVIDIA Tesla V100s.

TABLE IV
EXECUTION TIMES ON POWER9 PLATFORM WITH 6 NVIDIA TESLA V100S IN SECONDS FOR 1.82M ALIGNMENTS (*E. coli*).

| $X$-Drop | BELLA 168 CPU Threads | LOGAN 1 GPU | LOGAN 6 GPU |
|---|---|---|---|
| 5 | 53.2 | 110.4 | 114.3 |
| 10 | 108.6 | 146.4 | 115.3 |
| 15 | 139.0 | 152.9 | 114.8 |
| 20 | 226.7 | 162.7 | 118.4 |
| 25 | 275.3 | 173.5 | 125.3 |
| 30 | 558.0 | 185.3 | 130.6 |
| 35 | 654.1 | 198.4 | 136.8 |
| 40 | 750.1 | 212.7 | 138.4 |
| 50 | 913.1 | 248.5 | 141.4 |
| 80 | 1303.7 | 295.8 | 142.4 |
| 100 | 1507.1 | 336.3 | 144.5 |

TABLE V
EXECUTION TIMES ON POWER9 PLATFORM WITH 6 NVIDIA TESLA V100S IN SECONDS FOR 235M ALIGNMENTS (*C. elegans*).

| $X$-Drop | BELLA 168 CPU Threads | LOGAN 1 GPU | LOGAN 6 GPU |
|---|---|---|---|
| 5 | 131.7 | 577.1 | 213.1 |
| 10 | 723.3 | 750.2 | 579.7 |
| 15 | 1467.7 | 865.6 | 749.8 |
| 20 | 1954.8 | 908.9 | 777.0 |
| 25 | 2518.8 | 1015.5 | 838.9 |
| 30 | 3047.1 | 1125.0 | 888.0 |
| 35 | 3492.5 | 1226.5 | 927.0 |
| 40 | 3887.0 | 1329.0 | 955.9 |
| 50 | 4607.7 | 1449.0 | 983.7 |
| 80 | 6367.7 | 1593.9 | 1046.1 |
| 100 | 7385.3 | 1753.3 | 1080.9 |

CPU vectorized implementation on the Skylake processor. Both algorithms are benchmarked using the same set of 100K alignments used to compare LOGAN and SeqAn. Results show that LOGAN attains significant speed-ups ranging from $3.1\times$ to $120.4\times$ with a single GPU and from $3.7\times$ to $558.5\times$ using eight GPUs. Additionally, we can observe that ksw2 performs better when aligning the sequences using a small value of $X$ and its performance degrades drastically when increasing the $X$-drop value, as shown in Table III. Given LOGAN and ksw2 implement two slightly different heuristics, we also report a comparison based on the GCUPS metric. LOGAN achieves up to $181.4$ GCUPS with a single GPU for $X = 5000$, while ksw2 best performance is only $77.6$ GCUPS for $X = 100$. Importantly, LOGAN always outperforms ksw2 both in terms of runtime and GCUPS, independently from their respective peak performance at different values of $X$.

Figure 12 illustrates LOGAN's performance compared to two GPU-based algorithms, CUDASW++ and manymap. Notably, each of these three implementations performs a different amount of work, therefore we report the performance in terms of GCUPS. Furthermore, CUDASW++ uses hybrid GPU/SIMD computation by default. We report its performance with both hybrid and GPU-only computation. LOGAN consistently outperforms both CUDASW++ and manymap with performance up to $181$ GCUPS on a single GPU, while CUDASW++ and manymap achieve at most $70$ and $96$ GCUPS, respectively. Running with eight GPUs, LOGAN computes $3.2\times$ more GCUPS than GPU-only CUDASW++.

Finally, Figures 10 and 11 present BELLA's performance improvements when using LOGAN as pairwise alignment kernel. Our results show that BELLA attains significant speed-ups up to $7\times$ and $10\times$ on one GPU and six GPUs, respectively. Tables IV and III show the runtime of the original software in the column named "BELLA" and the runtime of BELLA using LOGAN as pairwise alignment kernel in the column "LOGAN". For large values of $X$, results show that LOGAN's runtime does not drastically degrade with increasing $X$. Notably, BELLA operates in a context where sequences have an error rate of about $10 - 15\%$. In this scenario, small values of $X$ can potentially lead to early drop-outs, even when

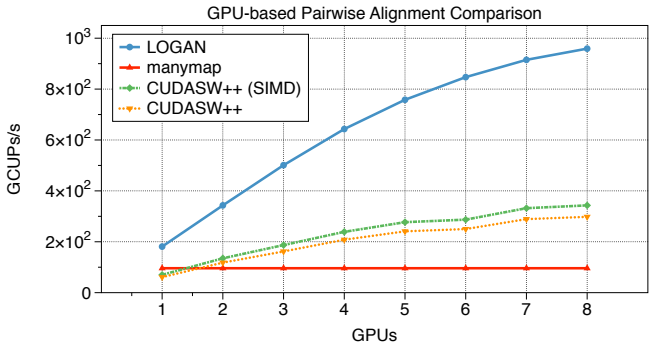


Fig. 12. Comparison amongst GPU-based pairwise alignment algorithms in terms of GCUPs per second ("Skylake" platform with 8 NVIDIA Tesla V100s). Higher is better. manymap is single GPU only, hence we report its performance as a flat line.

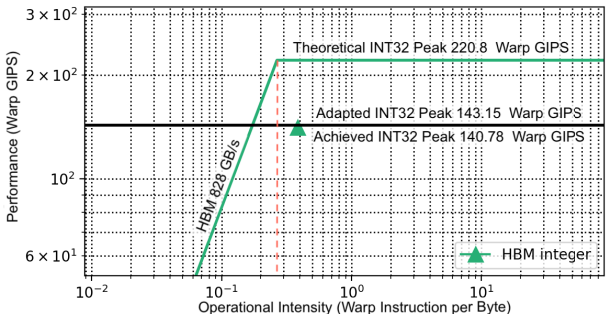


Fig. 13. Roofline analysis for our kernel on the NVIDIA Tesla V100 GPU performing 100K alignment and using $X = 100$.

sequences are supposed to align until the endpoints. Up to a certain point, increasing the value of $X$ increases the number of true alignments and makes it easier to differentiate true alignments from false positives. LOGAN's integration would allow BELLA to use larger $X$ values, resulting in higher accuracy without a notable increase in runtime.

Computing time scales linearly, however, the communication with multiple GPUs introduces an overhead that increases with the number of GPUs.

## VII. LOGAN Roofline Analysis

In this section, we provide a detailed analysis of the optimized LOGAN GPU performance by adapting the Roofline model [23, 24] to fit our specific computational characteristics. The Roofline model is a visually-intuitive method to understand the performance of a given kernel based on a *bound and bottleneck* analysis approach. The model outlines which factors affect the performance of computer systems, relating processor performance to off-chip memory traffic. The Roofline model characterizes a kernel's performance in billions of instructions (GIPS, y-axis) as a function of its operational intensity (OI, x-axis). We use *Operational Intensity* as the x-axis and, given that our kernel performs only integer operations, use billions of warp instructions per second (Warp GIPS) as the y-axis. *Operational Intensity* is defined as instructions per byte of DRAM traffic, which measures traffic between the caches and memory. Thus, our Roofline analysis combines integer performance, operational intensity, and memory performance into a 2D log-log scale graph, as shown in Figure 13.

On one NVIDIA Tesla V100 GPU, 80 Streaming Multiprocessor (SM)s are available, where each SM consists of four processing blocks, called *warp schedulers*. Each warp scheduler can dispatch only one instruction per cycle. As such, the theoretical maximum (warp-based) of instruction/s is $80\ \text{SM} \times (4 \times \text{warp scheduler}) \times (1 \times {}^{\text{instruction}}/_{\text{cycle}}) \times 1.53\ \text{GHz} = 489.6$ GIPS. Besides, each processing block contains 16 FP32 cores, 8 FP64 cores, and 16 INT32 cores. The maximum attainable integer performance is ${}^{16}/_{32} \times 489.6 = 220.8$ integer warp GIPS since 16 INT32 cores can only support 16 threads out of 32 threads in one warp. Peak Performance is upper bounded by both the theoretical INT32 peak rate and the peak memory bandwidth, which define the green line in the plot. The actual Warp Giga Instructions Per Second (GIPS) depends on the operational intensity and the ceiling line determines the limit of the actual performance. A kernel is memory-bound if the Warp GIPS are limited by the memory bandwidth (left of the red dotted line), and is compute-bound if limited by the hardware performance limit (right of the red dotted line). This maximum attainable performance represents a ceiling in the Roofline model plot for the considered GPU platform and is independent from the executed algorithm. To adapt this ceiling to the $X$-drop algorithm, we use the following formula:

$$Ceiling = \frac{1}{N} \sum_{i=1}^{N} \frac{f \times N_{op,i} \times B}{\lceil (T \times B)/MAX_R \rceil} \tag{1}$$

Equation (1) defines a new ceiling by averaging the number of cells that GPU can compute in parallel. $N$ indicates the total number of parallel iterations for a given algorithm, $f$ is the theoretical ceiling (220.8 warp GIPS), $B$ is the number of scheduled blocks, $N_{op,i}$ indicates the number of operations that need to be computed at each iteration, $T$ is the number of scheduled threads per block, and, finally, $MAX_R$ indicates the number of INT32 cores available. LOGAN's overall performance behavior is shown in Figure 13. Result show the operational intensity of our kernel on the HBM memory of the GPU, indicating that we are not memory bound and that we are bound by the adapted theoretical ceiling. In other words, the operational intensity of our kernel is high enough to be in the compute-bound area of the Roofline, thus it is not limited by the HBM memory. Note that the optimized performance of our algorithm is very close to the adapted theoretical ceiling. Considering that the adapted ceiling does not take into account memory latency, the results of our implementation are extremely close to the maximum achievable performance. Therefore, LOGAN represents a near-optimal implementation of the $X$-drop algorithm and it is only limited by the compute capability of the GPU.

## VIII. Conclusions

Our work presents LOGAN, the first high-performance multi-GPU implementation of the $X$-drop alignment algorithm. $X$-drop is employed in several important genomics applications, however it is particularly challenging for GPU parallelization due to its adaptive banding and continual termination checks.

Detailed results and analyses show significant performance acceleration using our novel optimization approach. LOGAN demonstrated runtime improvements of up to $30.7\times$ using six GPUs, compared with the original CPU algorithm. Additionally, results show speed-ups up to $614.4\times$ using six GPUs compared with the SIMD vectorized ksw2 algorithm, which implements a similar heuristics. Finally, LOGAN integration resulted in performance improvement up to $10.7\times$ on BELLA, a real-world many-to-many long-read overlapper and aligner.

Finally, our work provided an adaptation of the Roofline model that captures the unique aspects of our computation in the context of the underlying GPU hardware configuration. Roofline analysis demonstrates that our $X$-drop design methodology results in near-optimal performance. Our overall results show that our optimized kernel is flexible, efficient, and can be easily integrated into long-read application performing pairwise alignment.

Future work will focus on reducing LOGAN's load balancing overhead, to enable linear performance improvements with increasing GPU counts independent of the value of $X$. Given our implementation can be easily adapted to solve other similar problems, we also plan to extend LOGAN to support protein alignment and expect the $X$-drop algorithm to be effective in protein homology searches.

## Acknowledgments

We would like to thank Francesco Peverelli and Muaaz Awan for useful suggestions and valuable discussions.

This work is supported by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the DOE under contract number DE-AC02-05CH11231. This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

We used resources of the NERSC supported by the Office of Science of the DOE under Contract No. DEAC02-05CH11231. This research also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

## 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] G. Guidi, M. Ellis, D. Rokhsar, K. Yelick, and A. Buluç, "BELLA: Berkeley efficient long-read to long-read aligner and overlapper," *bioRxiv*, p. 464420, 2018.

[10] M. Farrar, "Striped Smith–Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2006.

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

[12] A. Szalkowski, C. Ledergerber, P. Krähenbühl, and C. Dessimoz, "SWPS3–fast multi-threaded vectorized Smith-Waterman for IBM Cell/BE and x86/SSE2," *BMC research notes*, vol. 1, no. 1, p. 107, 2008.

[13] Y. Liu, A. Wirawan, and B. Schmidt, "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions," *BMC bioinformatics*, vol. 14, no. 1, p. 117, 2013.

[14] A. Muhammadzadeh, "MR-CUDASW GPU accelerated Smith-Waterman algorithm for medium-length (meta)genomic data," Master's thesis, University of Saskatchewan, Saskatchewan, July 2014.

[15] I. T. Li, W. Shum, and K. Truong, "160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)," *BMC bioinformatics*, vol. 8, no. 1, p. 185, 2007.

[16] 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 *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 716–721.

[17] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics coprocessor provides up to 15,000 x acceleration on long read assembly," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 199–213.

[18] 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.

[19] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 05 2018.

[20] H. Suzuki and M. Kasahara, "Introducing difference recurrence relations for faster semi-global alignment of long sequences," *BMC bioinformatics*, vol. 19, no. 1, p. 45, 2018.

[21] 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.

[22] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *Computing in Science & Engineering*, no. 1, pp. 46–55, 1998.

[23] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, vol. 52, no. 4, 2009.

[24] N. Ding and S. Williams, "An instruction roofline model for gpus," *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019.