Wavefront Alignment Algorithm

Noah Edmiston, Logan Miller, Ishmeal Lee

(Dated: December 7, 2024)

Innovations in sequencing technology have led to a need for alignment algorithms that can handle ever-increasing read lengths. The Wavefront Alignment (WFA) algorithm is a modern approach to sequence alignment that offers superior time complexity [1]. In contrast to traditional quadratic models, WFA scales for a sequence of read length n, and alignment score s as $O(ns)$. We re-implement the algorithm in C++ in two separate ways. First, we implement the algorithm as described in the paper, relying on the compiler to vectorize the algorithm. Then, we manually vectorize the algorithm using the Kokkos SIMD library. We compare both these implementation against the naive approach to sequence alignment, and to the WFA2-lib implementation provided by the original paper. We find manual vectorization to have poor results, though we are able to achieve similar performance to WFA2-lib in low error rates.

I. INTRODUCTION

The advent of high-throughput sequencing technologies has revolutionized bioinformatics, enabling the generation of massive amounts of sequence data at unprecedented speeds. A cornerstone task in analyzing this data is pairwise sequence alignment this involves identifying regions of similarity between two sequences to infer functional, structural, or evolutionary relationships. Pairwise alignment is indispensable in applications such as read mapping, variant detection, and genome assembly, often demanding flexible scoring systems that can accommodate diverse biological contexts [1, 2].

Traditional alignment algorithms, such as those implementing dynamic programming for global or local alignment, have well-established theoretical underpinnings but are often hindered by computational inefficiencies. For example, the Smith-Waterman-Gotoh (SWG) algorithm, widely recognized for its robustness, incurs a quadratic time complexity that becomes prohibitive for longer sequences or high-throughput settings [2].

The Wavefront Alignment (WFA) algorithm represents a modern, efficient solution to these challenges, achieving a time complexity of $O(ns)$ for sequences of length s [1]. WFA leverages dynamic wavefront propagation to significantly reduce computational overhead while maintaining alignment accuracy. We explore the practical implementation and optimization of the WFA algorithm in C++, comparing its performance to both the naive dynamic programming approach and the state-of-the-art WFA2-lib library provided by the algorithm's authors.

Through detailed analysis, we evaluate the performance trade-offs associated with compiler-assisted vectorization and manual SIMD optimization via the Kokkos SIMD library [3]. This SIMD library is the example implementation for the `std::simd`, set to arrive in C++26. This is an opportunity to test the process of a more ergonomic approach to manual vectorization. Our findings underscore the promise of WFA as a scalable alignment tool, albeit with challenges in certain settings that merit further exploration, particularly with the use of SIMD abstractions.

II. PRIOR WORK

Pairwise alignment algorithms have been implemented in numerous different ways with different distance or cost models. The simplest model is the edit distance model which has been well optimized in tools like the C/C++ Edlib [4] and DALIGN [5], both capable of handling noisy reads. However, the distance model fails to capture the biological complexities of gaps.

A better model is the gap-linear penalties. More general is the gap-affine penalties as described in the Smith–Waterman–Gotoh (SWG) algorithm [2]. The alignment is computed on a dynamic programming matrix, with each dimension representing the progress through one of the two sequences being aligned. Thus, the time and memory requirements are quadratic in the length of the reads.

Much work has been put into accelerating the SWG algorithm. Many techniques have focused on accelerating and vectorizing certain operations along the diagonals of the matrix. The C Gaba library introduced the method of storing the deltas of the scores in the matrix, allowing values to fit within 8 bit integers. This results in a speedup with better packing into SIMD vectors [6]. This method is implemented in MiniMap2.

Many more implementations can be found which implement interesting vectorization or parallelization strategies. This algorithm has found enough prevalence for previous work to attempt different hardware, including Graphic Processing Units, Field-Programmable Gate Arrays, and further custom hardware.

The Wavefront Alignment (WFA) algorithm, introduced by Marco-Sola et al,, represents a significant leap in pairwise sequence alignmentby addressing the computational and memory challenges of traditional approaches [1]. WFA employs a wavefront representation to focus calculations only on active regions of the alignment matrix, reducing time complexity to $O(ns)$, where n is the read length and s is the alignment score, and memory complexity to $O(s^2)$. This efficiency is further enhanced by the algorithm's intrinsic adaptability, allowing it to handle diverse alignment problems such as short-read mapping, long-read alignment, and even highly divergent sequences.

The Bidirectional Wavefront Alignment (BiWFA) algorithm is an evolution of WFA, addressing its memory limitations while retaining computational efficiancy. BiWFA achieves optimal gap-affine alignments with $O(ns)$ time complexity and $O(s)$ memory complexity, significantly reducing the $O(s^2)$ memory requirements of the original WFA [7]. By aligning sequences in forward and reverse directions and identifying a midpoint break point, BiWFA enables genome-scale alignments that were previously impractical. On datasets like noisy Oxford Nanopore reads, BiWFA demonstrates substantial memory reductions - up to 438-fold - while achieving up to a 26.5 fold speedup compared to WFA2-lib in challenging scenarios. These advancements make BiWFA highly effective for large-scale genomic applications, including variant calling, pangenome assembly, and structural variant analysis, while maintaining compatibility with diverse scoring models.

Efforts to parallelize WFA across different hardware platforms, including GPUs and FPGAs, have also demonstrated significant speedups while maintaining alignment accuracy. Notable, Marco-Sola's work on GPU accelerated WFA illustrates the algorithm's potential for scaling bioinformatics workflows in high-throughput environments [8]. The integration of these hardware accelerations with the WFA framework ensures that it remains a leading choice for bioinformatics tools requiring rapid and accurate alignments. We took heavy inspiration from these papers, as the process of moving the algorithm to dedicated compute units resulted in similar styles of parallelism to the SIMD approach we took.

In summary, the WFA and its subsequent adaptions exemplify the evolution of sequence alignment algorithms toward efficiency and scalability. By addressing the challenges of diverse sequence datasets, long reads, and hardware integration, WFA continues to set a benchmark for modern alignment tools. Its ability to handle large-scale genomic analyses efficiently makes it an indispensable asset in bioinformatics Research.

III. METHODS

A. WFA Implementation Details

To begin the discussion of the implementation details, we will start with the descriptions of the wavefront data itself in memory. Naively, there each wavefront is composed of three vectors, one for the insertion offsets, the deletion offsets, and the matching offsets. The size of each of these vectors is dependent on the size of the wavefront, which grows with increasing score. For SIMD reasons discussed later, we would like each column to be contiguous. Then, as we proceed through the algorithm, we add wavefronts until we reach the final offset.

One of the primary approaches to achieving high-performance was to maximize data locality, and minimize memory allocations. To do this, the underlying data for the wavefronts is stored completely contiguously in a reusable memory area type `wavefront_arena_t`. The type will grow as needed, can be reused for subsequent runs of the algorithm for other pairs of strings. From a memory allocation perspective, instead of reallocating for every `next` step in WFA, reuse of the arena results in near-zero allocations beyond the first several alignments.

The data type for an individual wavefront is `wavefront_entry_t`. Primarily, this type contains a `std::span` over a portion of the arena, providing array style access to the underlying data without duplication. Beyond this, the type primarily stores metadata about the wavefront, such as the low and high diagonals covered by the wavefront, and the number of diagonals per column, where the insertion, deletion, and match data are the three columns of the wavefront data. The usage of the span and arena allows the type to remain reasonably small while still including meta-data. There are a variety of utility methods for performing bounded and unbounded lookups in the data, which aid in the conversion of column and diagonal indecision into pointer offsets.

The metadata for a full run of the alignment process is stored in a `wavefront_t`, and this type is passed around into each step of the WFA implementation. First, the type stores a reference to the underlying wavefront arena, and a vector of wavefront entries corresponding to the various wavefronts. Finally, it also additionally stores a mapping vector. As wavefronts can only spawn from other wavefronts, there are scores that are completely impossible to achieve. However, much of the indexing throughout the algorithm is performing lookups by score. As such, the mapping vector is indexed by all scores, and the indexed element is a -1 if the wavefront does not exist, and an index to the wavefront with the corresponding score in the wavefront entries vector otherwise. We found that this combination of two memory access was generally faster than the overhead of a `unordered_map<int32_t, int32_t>`. Beyond this, the type contains utility functions for lookups, along with the logic for inserting a new wavefront.

For both the SIMD and non-SIMD implementations of the algorithm, they are structured as closely as possible to the original algorithms on the paper. Each implementation is broken up into an overall `wavefront` function, with two supporting functions, `next` and `extend`. While there is little to discuss on the non-SIMD implementation, which very closely mirrors the algorithm described in [1], the general approach was to perform as minimal bounds checking as possible and minimize duplicate computation.

Implementing the algorithm in the SIMD approach was a complex endeavor for both the `next` and `extend` phases of the algorithm. In both cases, the implementation attempts to use the SIMD implementation as much as possible, and then switches to a tradition implementation to clean up the remainder.

The implementation of the SIMD version of `extend` focuses on vectorizing the character comparisons along the diagonal. This allows us to move very quickly in cases of very close strings, as each loop jumps the offset much farther forward. Due to the limitations of the Kokkos SIMD implementation, the smallest datatype is a `int32_t`. As such, we pack four `chars` into each `int32_t` element of the vector register, and the perform the comparison. The first non-matching mask element is identified, and then the four `char` values corresponding to that mask entry are iterated through to identify the correct index.

The SIMD implementation of the `next` step of WFA was far more complicated. Though the original paper [1] is correct in its depiction of the data dependencies of this stage as simple, it rather undercuts the more complicated cases of bounds checking. The `next` calculations for solving a new wavefront entry with a given score are as follows:

$$\widetilde{I}_{s,k} = \max\left(\widetilde{M}_{s-o-e,k-1}, \widetilde{I}_{s-e,k-1}\right) + 1 \quad (1)$$

$$\widetilde{D}_{s,k} = \max\left(\widetilde{M}_{s-o-e,k+1}, \widetilde{D}_{s-e,k+1}\right) \quad (2)$$

$$\widetilde{M}_{s,k} = \max\left(\widetilde{M}_{s-x,k} + 1, \widetilde{D}_{s,k}, \widetilde{I}_{s,k}\right) \quad (3)$$

$$\widetilde{M}_{0,0} = 0 \quad (4)$$

where k denotes the diagonal currently being computed. From inspection, it can be seen that each entry into $\widetilde{I}_{s,k}$, $\widetilde{D}_{s,k}$, or $\widetilde{M}_{s,k}$ depends only on previously computed wavefronts and the current values of k and s. As such, we attempt to vectorize these calculations over k. As our columns are laid out contiguously in memory, the entries, for instance, of $\widetilde{M}_{s-o-e,k-1}$, for a range of k values can be loaded directly into a vector register. We can then apply the vector max operations, before once again taking use of the contiguous columns to write the calculated vector register $\widetilde{M}_{s,k}$ directly to the wavefront data.

However, this ideal pitch of load, compute max, and store turns out to be far more complicated in practice. It is extremely common for lookups into previous wavefronts to hit wavefronts that don't exist, or for wavefronts that do exist to hit diagonals that do not exist. In these cases, a value of -1 is returned to ensure the algorithm completes successfully. As such, even though the columns for wavefronts are contiguous in memory, there is a large amount of conditional logic needed before the column can be loaded into the register. The most costly of this logic occurs in cases where the wavefront we intend to load from exists, except the range of diagonals we need to load from is partially out of bounds. To handle these cases, the `simd_lookup` function calculates a mask of in bounds diagonals, and then uses this mask to perform a conditional load into memory. We then have to invert and re-use this mask to then write the -1 values needed.

B. Artificial Data Generation and Testing

In order to test our implementation, we required a source of artificial data with controllable error rates. This is implemented by generating random sequences of 'A', 'T', 'G', and 'C' to the length and number desired. Then, these sequences are iterated through to generate a companion sequence, where each character has a user-provided random chance of being redrawn from the set of nucleobases. When running benchmarks, it is the original string, and the manually error string that are aligned.

To test the correctness of the program, a naive implementation of the SWG dynamic programming approach was implemented. Similarly, we also retrieved the original WFA2-lib, as implemented by the paper's authors, for purposes of comparison. We used the C++ bindings in the comparison process.

C. Performance Profiling

Performance Profiling was performed using Intel VTune, set to hardware events-based sampling at a resolution of 1 ms. The primary style of analysis performed was hotspot CPU profiling, though memory profiling was also used to study cache misses and memory overhead. More simple forms of benchmarking were also performed by running large sets of sequences through the various WFA implementations at varying error levels, corpus sizes, and read-lengths. For the sake of consistency, all of the numbers presented in the paper were measured on a Intel(R) i7-14700K, inside of a Ubuntu WSL2 instance running on Windows 11.

IV. RESULTS

As expected, in comparison to the naive dynamic programming implementation, we can see nearly two orders of magnitude performance improvement against the naive implementation at low error rates in Figure 1. The $O(ns)$ scaling is clearly visible, as the score and required wavefronts increase, the performance decreases as more wave fronts are needed.

In comparison with WFA2-lib, as seen in Figure **??**, we see slightly better performance at low error rates in both the SIMD and non-SIMD algorithms than WFA2-lib, though our scaling with error rate is noticably worse. This issue is most clearly visible in the joint sensitivity data in Figure 2. We are slightly faster in the upper left of low error rates, but as the need for more wavefronts increases, we rapidly lose our edge to WFA2-lib. Even for incredibly short sequence lengths, high error rates quickly cause us to fall behind.

This scaling issue is supported by the data from the hardware sampling. In both the SIMD and non-SIMD implementations, the `next` stage of the algorithm dominates time spent. In both cases, the source of this is due to the overhead in the `lookup` functions pulling data from previous wavefronts. The bounds checking for identifying scores with no corresponding wavefronts, or checking for out of bounds diagonals dominates the time spent, causing lookups to take up nearly half of the time spent within the `next` function. Future iterations would likely require a adjustment of how bounds checking is handled overall in order to improve the performance to match WFA2-lib in the case of high wavefronts.

It is also of note, as visible in all of the data, that our SIMD implementation rarely performs significantly differently than the normal implementation. While the synthetic data identifies the issue, the detail is once again found within the hardware level benchmarks. We can see that the vectorization is in fact occurring, but the constant marshaling of data in and out of the vector registers heavily cuts down on the performance improvements. It also makes visible how Kokkos's limitations hamper the implementation. For both `extend` and `next`, we see very similar performance between the manually vectorized and the non-vectorized components.

In the case of `extend`, the 32bit minimum vector element size, and the subsequent repeated remainder calculations take longer than all of the vector operations to compute the mask itself. Based on the sampling data, we spend as long extracting the characters from the 32bit mask as we do in the entire rest of computation.

This is also the case in `next`. Though vector registers are being used, such using `VPMASKMOV` for the conditional loads, the surrounding code generation is almost entirely non-vector instructions marshaling data to setup the operation. It is highly like that superior vector performance could be gained by manually implementing operations. Most egregiously, the SIMD max operations appear to have defaulted to a non-SIMD implementation under the hood, as opposed to have using standard SIMD max operations. From both the data and the assembly, it is apparent that one is much better off taking the plunge into writing intrinsics, as opposed to hoping that Kokkos can figure it out.

V. CONCLUSIONS

While we did successfully implement the algorithm and its SIMD variant, we still under-perform the WFA2-lib. An area of particular issues is those with large wavefronts, where our implementation seems to have a worse scaling factor, causing performance degradation in the case of high error rates or extremely long sequences. Similarly, while the pitch of "easy platform abstracted intrinsics" is very appealing from and implementation perspective, the ease of use of the SIMD tools inversely corresponded to how much of a performance benefit they gave.

For future work, optimizing the `next` functions of our implementation would bring us in line with WFA2-lib. Given that we are outperforming WFA2-lib at low error rates, better scaling could possibly allow for us to consistently outperform WFA2-lib, even with large wavefronts.

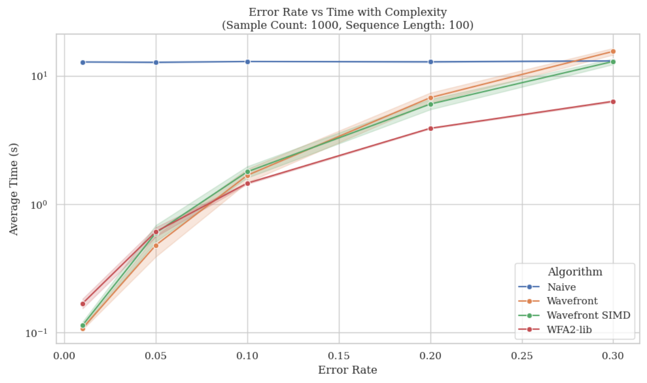

FIG. 1. Impact of error rate on alignment performance. At lower error rates, algorithms (excluding naive) maintain high efficiency, but performance scales differently as error rates increase

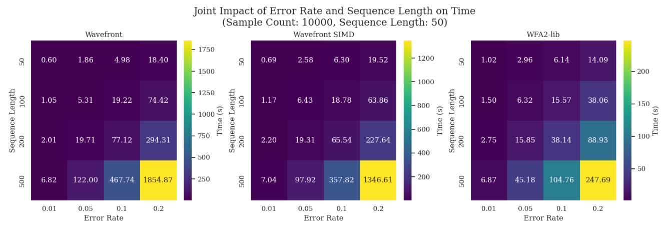

FIG. 2. Joint analysis of error rate and sequence length on alignment time.

VI. AUTHOR CONTRIBUTIONS

Edmiston: Implemented the SIMD versions of the WFA library. Performed hardware benchmarking and performance iteration. Contributed to SIMD details in the paper.

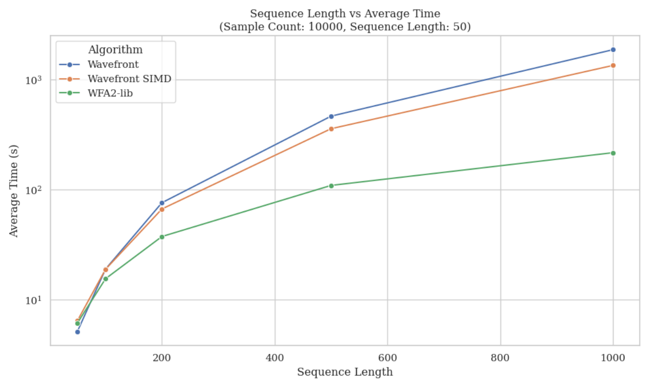

FIG. 3. Relationship between sequence length and alignment time for different algorithms.

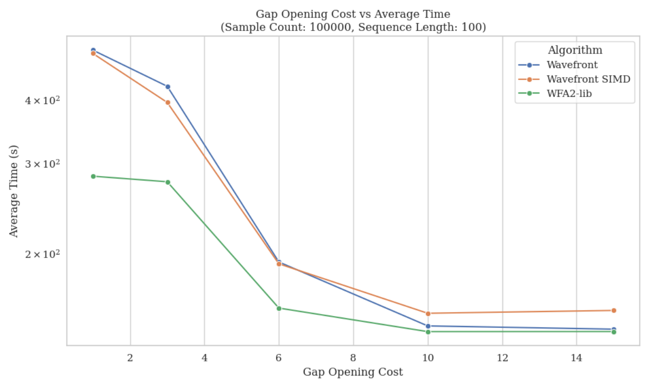

FIG. 4. Effect of gap-opening penalties on alignment time, Algorithms demonstrate varying sensitivities to increasing penalties, reflecting differences in computational strategies for handling gaps.

Lee: Implemented the first version of naive and WFA algorithms and contributed to the paper.

Miller: Contributed towards the implementation and testing of the naive and DP-WFA algorithms. Significantly contributed to benchmarking the implementations against the WFA2-lib library and other approaches. Implemented the production of visual aids. Contributed to the introduction, prior works, and figures of the report.

[1] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa, Bioinformatics **37**, 456 (2020), https://academic.oup.com/bioinformatics/article-pdf/37/4/456/50359789/btaa777.pdf.

[2] O. Gotoh, Journal of Molecular Biology **162**, 705 (1982).

[3] kokkos, *Kokkos Core Wiki*.

[4] M. Šošić and M. Šikić, Bioinformatics **33**, 1394 (2017), https://academic.oup.com/bioinformatics/article-pdf/33/9/1394/49038512/bioinformatics_33_9_1394.pdf.

[5] G. Myers, in *Algorithms in Bioinformatics*, edited by D. Brown and B. Morgenstern (Springer Berlin Heidelberg, Berlin, Heidelberg, 2014) pp. 52–67.

[6] H. Suzuki and M. Kasahara, BMC Bioinformatics **19** (2018).

[7] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa, Bioinformatics **38**, 3938 (2022), https://academic.oup.com/bioinformatics/article-pdf/38/17/3938/50789588/btad074.pdf.

[8] Q. Aguado-Puig, S. Marco-Sola, J. C. Moure, D. Castells-Rufas, L. Alvarez, A. Espinosa, and M. Moreto, IEEE Access **10**, 63782 (2022).