2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 979-8-3503-6460-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPSW63119.2024.00113

Further Optimizations and Analysis of Smith-Waterman with Vector Extensions

Reza Sajjadinasab*‡, Hamed Rastaghi*, Hafsah Shahzad*, Sanjay Arora†, Ulrich Drepper†, and Martin Herbordt*‡
*ECE Department, Boston University †Red Hat Inc. ‡{sajjadi, herbordt}@bu.edu

***Abstract*—Sequence alignment based on dynamic programming, e.g., Smith-Waterman (SW), remains central to bioinformatics both in various standalone scenarios and as a function in other critical bioinformatics tasks. Much work has been done in optimizing SW for various accelerators; CPU-based platforms, however, are possibly still the most accessible computational resource for bioinformatics and our focus here. CPU acceleration necessarily involves applying vector extensions; a recent CPU/SW study resulted in the creation of an extensive autotuning framework. We add to this corpus a number of contributions: (i) a new optimization based on a novel mapping of the scoring matrix together with interleaving data coming from the substitution matrix; (ii) an implementation that combines disjoint optimizations from previous studies; (iii) a variable (8/16) bit width implementation; (iv) a finding of the benefits of learned compiler hyperparameters; (v) performance comparisons with respect to different usage scenarios; and (vi) a results showing that, despite appearing memory bound, multicore SW remains CPU bound. Among the most significant findings are that for protein alignments these approaches result in performance superior to that of the Parasail library's approaches, with the added benefit of determinism and robustness. Versus the deterministic Parasail+diag the improvement is 3.9×; versus the non-deterministic Parasail+scan and Parasail+striped, it is 1.9× and 1.5×, respectively.**

***Index Terms*—Smith-Waterman, Sequence Alignment, Intel Intrinsics, Vector Extensions**

I. Introduction

Bioinformatics relies on efficient and accurate sequence alignment to extract valuable insights from the ever-growing volumes of biological sequence data [1]. Smith-Waterman algorithm (SW) offers a provably optimal method of providing high sensitivity and precision with a computational complexity of $O(nm)$ where n and m are the database and query size, respectively. However, aligning large-scale datasets poses computational challenges that necessitate the use of high performance methods [2]. This is especially true in many application, such as multiple sequence alignment and genome sequencing, where SW is invoked repeatedly. Research has been rich with respect to the use of GPUs (e.g., [3]–[5]) and FPGAs (e.g., [6]–[9]).

The day-to-day workhorses, however, for thousands of bioinformatics practitioners remain workstation CPUs and CPU clusters [10], [11]. Continuing recent advances in sequence alignment using vector intrinsics (e.g., [12]), we explore further enhancements in SW/CPU for protein sequence alignment in the evolving computational landscape. In addition, a further question is investigated: with the increase in CPU compute capabilities, has SW transitioned from being compute-bound to memory-bound (as in [13])?

This work has the following contributions and innovations.
Enhanced Alignment Kernel: At the core of this paper is the refinement of the alignment kernel, which is crucial for the performance improvement obtained. One aspect is the redesign of the data memory layout so that it is optimized to align effectively with current CPU memory hierarchies, an approach previously applied for GPUs [14]. A further aspect is that the kernel mixes interleaving and wavefront approaches.
Compiler Hyperparameter Optimization: Complementing the kernel enhancement, we delve into compiler hyperparameter optimization. This approach exploits compiler settings to enhance kernel performance, demonstrating significant gains in computational efficiency.
Memory and Microarchitecture Analysis: Extending previous research, we include an analysis of recent advances in memory technologies and their impact on sequence alignment. This analysis offers strategies for overcoming memory-related challenges through hardware-software co-design.
Client-Specific SW Algorithm Variations: Addressing diverse client needs, we introduce algorithmic variations optimized for different operational contexts, from high-throughput to memory-limited environments.
Comprehensive Portability Analysis: A detailed analysis of the methodologies' adaptability across various platforms is presented, showcasing the practical applicability of the proposed improvements.

Among the most significant findings is that these approaches result in performance comparable, or superior to, performance of the Parasail library's approaches, but with the benefit of determinism and robustness. Versus the deterministic Parasail+diag the improvement is 3.9×; versus the non-deterministic Parasail+scan and Parasail+striped, it is 1.9× and 1.5×, respectively.

II. BACKGROUND

A. DNA vs Protein alignment

In bioinformatics, understanding the key differences between protein and DNA alignments is crucial. Protein alignments involve sequences of 20 different amino acids. The lengths are highly variable, extending from a few dozen to thousands of amino acids, necessitating flexible alignment strategies. For protein alignments, scoring matrices like BLOSUM or PAM are typically used [15]. On the other hand, DNA alignments, though dealing with just four nucleotides (adenine,

979-8-3503-6460-6/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPSW63119.2024.00113

thymine, cytosine, and guanine), confront the challenge of managing extremely long sequences, particularly in eukaryotes where genomes can span billions of base pairs. DNA alignments employ simpler scoring matrices. The choice between affine and linear gap penalties depends on the alignment task: protein alignments commonly use affine gap penalties to better reflect the significant impact of gaps in protein structures, while DNA alignments use either affine or linear gap models, tailored to the sequence characteristics [16].

B. Smith-Waterman

In this study, we implement the affine gap penalty model using a modified Smith-Waterman algorithm [12]. The model employs two auxiliary arrays, F and E, to effectively manage gap penalties, with F being of the same size as the database sequence for storing horizontal gap penalties, and E corresponding to the query sequence for vertical gap penalties. This approach is specifically designed to handle the complexity of affine gap penalties, where the cost of opening a gap is different from extending it. Equation (1) shows the implementation.

$$\begin{aligned} H(i,j) &= \max\{0, H(i-1,j-1) + S[q(i-1), r(j-1)]\} \\ H(i,j) &= \max\{H(i,j), F[j], E[i]\} \\ F[j] &= \max\{F[j] + e, H(i,j) + o\} \\ E[i] &= \max\{E[i] + e, H(i,j) + o\} \end{aligned} \tag{1}$$

After completing the construction of the matrix H, which captures the optimal scoring of subsequence alignments, the traceback process determines the best-aligned subsequences. This phase traverses the matrix H from the highest-scoring cell (local alignment), or from the bottom-right cell (global alignments). This step is computationally demanding, as it requires both additional memory to store the traceback matrix and additional processing to interpret the traceback path [17].

C. Smith-Waterman Usage Scenarios

SW is adaptable across various usage scenarios, with different data set assumptions. In particular, different studies have made different usage assumptions, which sometimes leads to difficulty in making direct comparisons.

Scenario 1: Single Query versus Database. For protein queries it is likely that the query resides in the highest level of storage; the database is streamed with little reuse.

Scenario 2: Batch of Queries versus Database. Multiple query sequences are aligned against a database, either from a single client or multiple clients batched in a shared resource. This scenario becomes a many-to-many problem with much potential data reuse.

Scenario 3: SW as a Subroutine. Here SW is often applied to aligning small query sequences against a small database, as demonstrated by the SSW Library for optimal protein or genome sequence alignment [10]. There is substantial reuse and the working set often fits in the highest level memory.

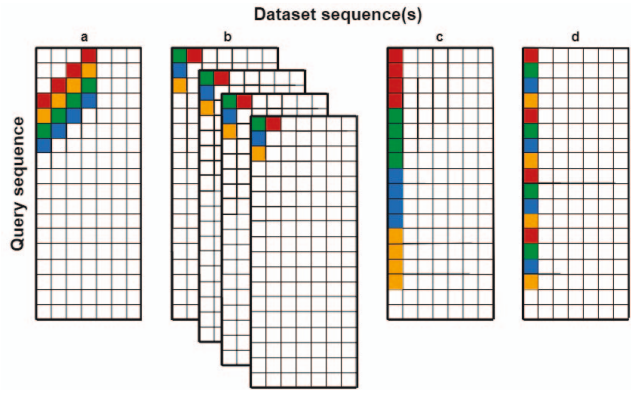


Fig. 1: Different vectorization approaches.

D. Data parallelism and SIMD instructions

A key step in SW is constructing and traversing a similarity matrix. This matrix, typically denoted as H, is filled following the dependencies between its elements. Each cell $H_{i,j}$ depends on its neighboring cells: $H_{i-1,j}$, $H_{i,j-1}$, and $H_{i-1,j-1}$.

Researchers have developed several ways to fill in the similarity matrix [18]. Fig. 1 illustrates four such methods. In method (a), the matrix is filled diagonally, with each lane of the vector working on one element of the diagonal at a time [19]. This method is a classic way of resolving dependencies. In method (b), every lane of the vector is responsible for building a separate similarity matrix for different a pair of sequences, which also avoids dependency problems altogether [20]. The other two methods shown, (c) and (d), do face dependency challenges, but researchers have come up with clever solutions for these with some correction loops. In method (c), each vector is in charge of a portion of the query sequence [21], while in method (d), each lane of a vector processes a different part of the query sequence [22].

We chose method (a) for our project because it doesn't have the issue of dependencies, and it helps build a robust and deterministic implementation of SW. Also, Using this approach improves the locality, which is important for larger sequences, since the current vector will be reused as the diagonal neighbor of the previous vector. In the next section, we'll describe how we implemented this method effectively.

E. Compiler Hyperparameters

Compiler hyperparameters, set before compilation, direct the compiler in code optimization. They include optimization levels, which determine the degree of code optimization [23]; memory management parameters, which guide how the compiler allocates and utilizes memory; and instruction scheduling, which involves the ordering of machine code instructions. Properly configured hyperparameters can lead to significant enhancements in execution speed, memory efficiency, and overall computational resource management [24].

III. IMPROVED ALIGNMENT KERNEL

This section presents the development and optimization of the kernel. Sequence alignment demands both precision and

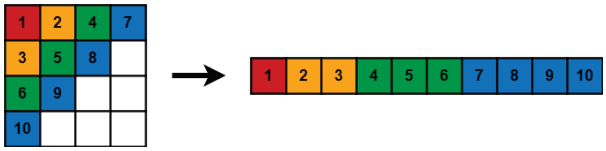


Fig. 2: Diagonal-based linearization of the memory improves access and results in better spatial locality.

efficiency. We introduce several heuristic techniques specifically designed to address common challenges.

A. Diagonal-Based Memory Indexing

We depart from the traditional row-based traversal of the alignment matrix, opting instead for a diagonal-based method. This shift in traversal strategy, as illustrated in Fig. 2, yields significant benefits. By traversing the matrix diagonally, we align the memory access pattern with the diagonal filling order. This means that, as the matrix is being filled, we access and fill elements that are consecutive in memory along each diagonal. This arrangement optimizes spatial locality, as elements that are logically adjacent in the alignment are also physically adjacent in memory. Also, it improves temporal locality, since each row will be used for updating the next row. The advantage of this setup is twofold: it reduces the number of memory accesses and enhances processing speed. Additionally, diagonal-based traversal is more aligned with the inherent flow of sequence alignment calculations due to the natural data dependencies between different elements.

Fig. 2 shows the diagonal-based indexing with its advantageous arrangement of elements in memory.

B. Handling Variable-Length Diagonal Segments

One of the challenges in implementing the diagonal approach for sequence alignment is dealing with segments of the matrix diagonal that are shorter than the number of lanes in the vector. Our solution involves using zero-padding for the unused lanes in these segments. For small segments, we revert to standard CPU instructions. A crucial point is that it enables the computation of as many cells as possible at a given time. While this means that the highest capacity of the CPU is not always fully harnessed, this is a limitation stemming from the nature of the problem itself and only affects a minor portion of the computations, roughly around 15%. This balance is essential for optimizing performance within the constraints of the alignment problem. Fig. 3 shows how some segments are smaller than the size of the array and pinpoints where in the process the zero padding occurs.

C. Enhanced Substitution Matrix Utilization

In this work, the substitution matrix is reorganized by reordering its columns and rows and adding extra ones for characters that don't represent an amino acid. This reorganization aids in efficient data retrieval using the *gather* vector instructions of AVX2. Fig. 4 demonstrates this process of transforming characters into indices for accessing the substitution matrix.

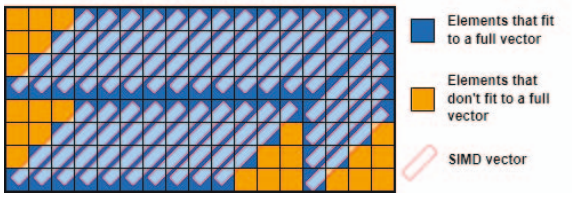


Fig. 3: Vector assignment of different segments of the similarity matrix. Yellow sections need padding with zeros.

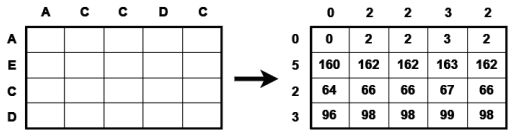


Fig. 4: Index calculation of the substitution matrix for gather instruction. Each character is an index to access the matrix.

Although this approach can be useful for 32-bit and 16-bit implementations, the performance degrades with 8-bits. To address this issue, a query profile is made before computing the alignment. The query profile contains the possible score for each query residue. For the sake of a good alignment in memory (as described above) the substitution matrix has 32 residues in each row, which fits exactly into 256 bits and can be read with a single AVX instruction.

In addition to the query profile, which is made at runtime, the database can be organized for more efficient access. This is done once, offline. The database sequences are stored in batches containing 32 transposed sequences, i.e., 32 for the number of lanes in AVX2 when using 8-bit integers. This enables the immediate use of AVX shuffling instructions. As shown in Figure 5, each adjacent transposed residue represents a residue from a different sequence. Then for every batch we compute the score once and store it in a scratch buffer. At this point, there are numerous ways to tune the kernel depending on cache parameters.

In computing the maximum score, the local maximum for each vector lane is stored. A reduction using shift operations on the matrix is then employed to find the overall maximum score. This approach not only simplifies the calculation, but also improves the use of vector processing capabilities.

D. Maximum Score Calculation

During sequence alignment, determining the maximum alignment score is a crucial yet computationally expensive task. Calculating the maximum score after computing the scores along each diagonal would require a reduction operation on a vector. Such operations, especially when performed frequently, can be costly: while AVX512 does offer instructions for this purpose, it is limited in the size of integers it can handle and is not fast. To optimize this process and minimize computational overhead, we have adopted a strategy of storing the local maxima of each lane in a separate vector. This approach allows us to defer the computationally intensive task

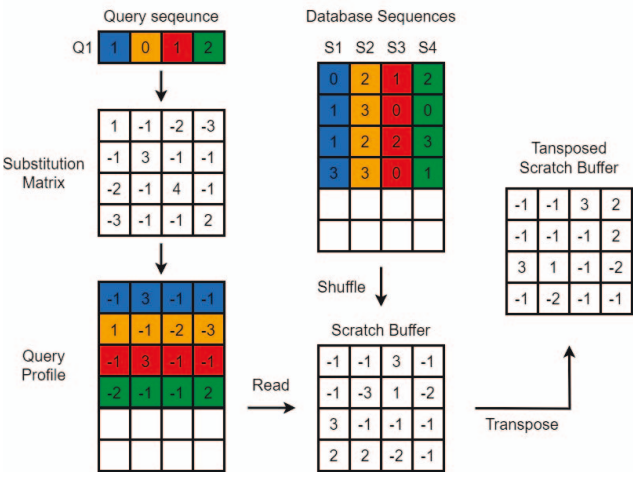


Fig. 5: Extracting scores from the substitution matrix based on query and batch of database sequences

of finding the global maximum score until the end of the computation.

E. Hyperparameter Tuning in Compiler Optimization

Besides code optimization, we also investigated improving SW performance through tuning of compiler hyperparameters using an evolutionary algorithm. This method diverges from traditional software tuning techniques, leveraging the principles of natural selection to evolve a population of potential solutions over time. Such an approach is particularly suited for dynamic and adaptive optimization, where traditional deterministic methods might fall short.

IV. RESULTS AND DISCUSSION

This section presents findings and analyses. It encompasses a detailed examination of SIMD length choices, various design implementations, compiler hyperparameter optimizations, and assessments of memory and microarchitecture. The results are further augmented by investigating three specific usage scenarios of the Smith-Waterman algorithm and a comparative analysis with the Parasail library.

A. Methodology

Processors: Our study utilized four Intel architectures: Intel Haswell E5-2660 (8 core), Intel Broadwell E5-2680 (14 core) for baseline performance, and Intel Skylake Gold 6132 (16 core), Intel Cascadelake Gold 6242 (16 core) for advanced processing capabilities. For memory analysis, we used the Intel Alderlake i9-12900HK (10 core).

Dataset and Queries: We used the protein UniProtKB/Swiss-Prot [25] dataset. For queries, we randomly selected 10 proteins from this dataset with a range of lengths. The reason for this limited number (used, e.g., [22]) is that the execution is deterministic with respect to query size and only behaviors related to size need to be measured.

Compilation: The code, designed with macro definitions for flexibility, was compiled using GCC version 11.2.0 with -O3 optimization.

B. AVX512 versus AVX2

We focused on two of the newer (but not the latest) Intel architectures to compare the performance implications of using AVX 512, a deprecated version of AVX. As indicated in Fig. 6, use of AVX512 did not result in a significantly enhanced performance as initially anticipated. Despite expectations of double the performance of AVX2, the results did not align with these projections. This discrepancy led us to continue our analyses primarily with AVX2, which not only is the current version of AVX in newer Intel CPUs, but also offers compatibility with older architectures. This choice was guided by the relative performance benefits and the broader applicability of AVX2 across various hardware platforms, ensuring a more generalizable and practical approach.

C. Design Choices Based on Client Needs

Effect of Affine Gap Penalty: Incorporating affine gap penalties did not result in a noticeable performance drop as shown in Fig. 7. This finding is significant as it suggests that the added complexity of an affine gap model, which is generally expected to be computationally more intensive, does not necessarily compromise performance.

Traceback Functionality: The integration of traceback functionality, while necessitating the storage of extensive information in memory for backtracking, surprisingly did not degrade performance (as shown in Fig. 8). This includes recording from which cell (up, left, or diagonal) a particular cell was updated. The ability to maintain performance despite the additional memory requirements suggests an effective utilization of memory resources. This finding is particularly relevant for applications where detailed alignment histories are crucial, as it demonstrates that incorporating traceback does not come at the cost of reduced processing speed.

Impact of Substitution Matrix: The implementation of a substitution matrix significantly impacted performance compared to the case with fixed alignment scores, primarily leading to a core-bound scenario due to extensive shuffling and register reads as depicted in Fig. 9. The utilization of Intel's gather function, although not exceptionally fast, played a key role in this aspect. Despite these challenges, the performance was reasonably good, especially for smaller-sized queries, which are more common in protein datasets. This highlights a crucial trade-off between the complexity of matrix operations and the accuracy benefits it brings, particularly relevant for applications where query size varies.

Previously the performance of the 8-bit version was degraded (versus the 16-bit version) because there is no 8-bit gather in Intel. The new approach, presented in the previous section and used here, alleviates this problem and the performance is now comparable.

D. Compiler Hyperparameter Optimization and Its Impact

Inspired by the genetic algorithm, we employed a random initialization to grow a population that evolves randomly into a new one. Within each population, we select the best possible solution to maximize the real-time performance of the SW

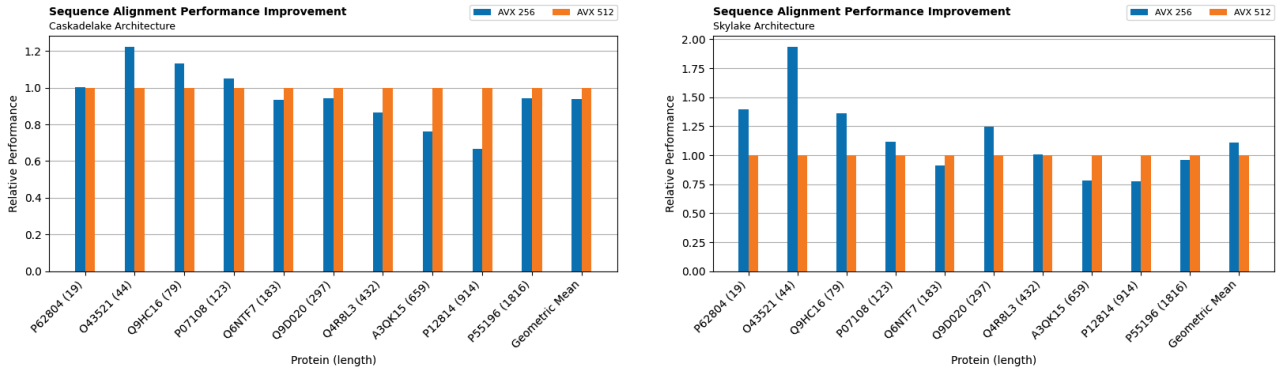


Fig. 6: Performance evaluation for AVX 256 vs AVX 512 on different architectures for 10 different protein queries

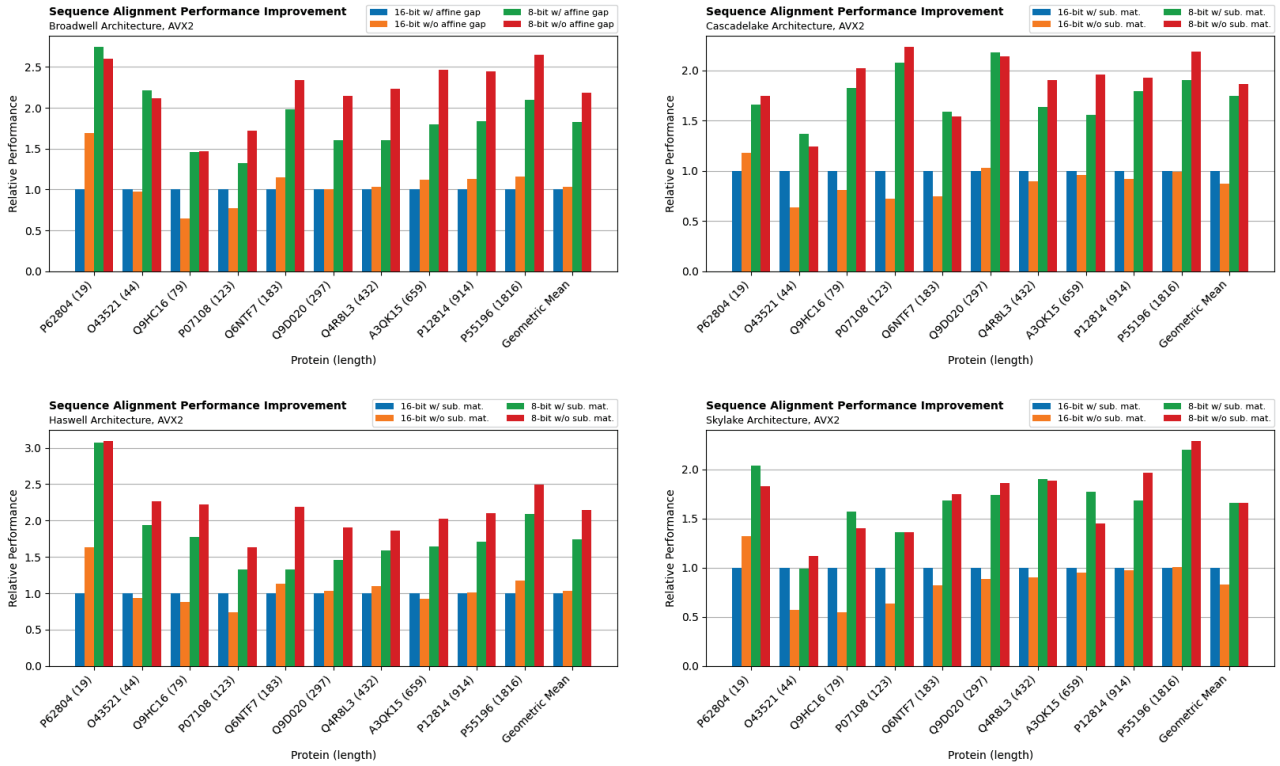


Fig. 7: Performance for designs with and without affine gap penalty on different architectures for 10 different proteins

implementation. This method allows us to fine-tune the GCC parameters over several iterations, further optimizing our SW implementation. We start with a random initialized population of hyperparameter values (all those provided by GCC). Each hyperparameter then evolves within its particular allowable set of values. Each new population is evaluated and the best is selected for the final selection. Since this approach is not guaranteed to find the best solution the results and the fine-tuned versions of the program might vary. This variability is influenced by several factors, including the datasets used and the specific tuning materials applied. As illustrated in Fig. 10, this approach led to improvements in performance across almost all architectures. However, it's noteworthy that some architectures exhibited significantly better enhancements compared to others.

Moreover, the size of the query emerged as a crucial factor affecting performance. Our analysis revealed that tuning could substantially improve performance for certain query sizes. This indicates that the effectiveness of hyperparameter optimization is not uniform across different scenarios; rather, it is highly dependent on specific conditions such as architecture and query size.

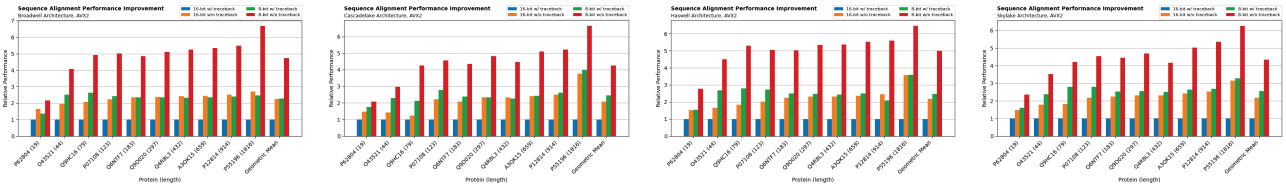


Fig. 8: Performance for designs with and without traceback on different architectures for 10 different proteins

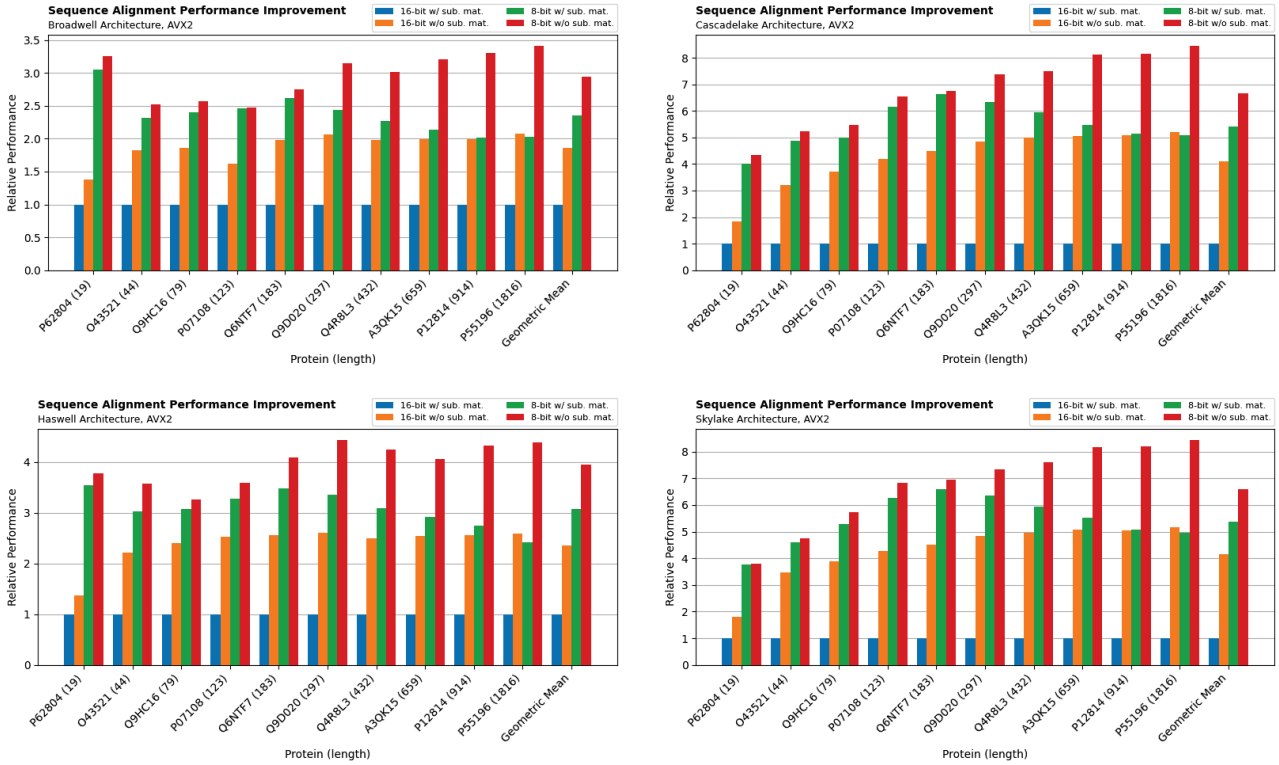


Fig. 9: Performance for designs with and without substitution matrix on different architectures for 10 different proteins

This variability underscores the complexity of achieving optimal performance and highlights the importance of context-specific tuning in sequence alignment applications. It also suggests that while general improvements are attainable, maximizing efficiency requires careful consideration of the unique characteristics of each use case.

E. Multi-Threading and Scalability Analysis

Given that SW is both a batch application and that performance per core degrades with the number of cores, our hypothesis was that this was due to memory contention. We found rather that it was due to variations in operating frequency. A microbenchmark specifically designed for analyzing CPU frequency revealed that the frequency is not consistently stable in multi-core mode. This variability led us to recalibrate our performance metrics, particularly for the single-threaded version, to take into account the frequency drop during multi-core operations.

As demonstrated in Fig. 11, this recalibration was crucial, especially in the case of hyperthreading and full-core utilization. For larger queries, which rely heavily on AVX2 computations, the performance closely matched our projections. This was notably apparent in scenarios with hyperthreading across

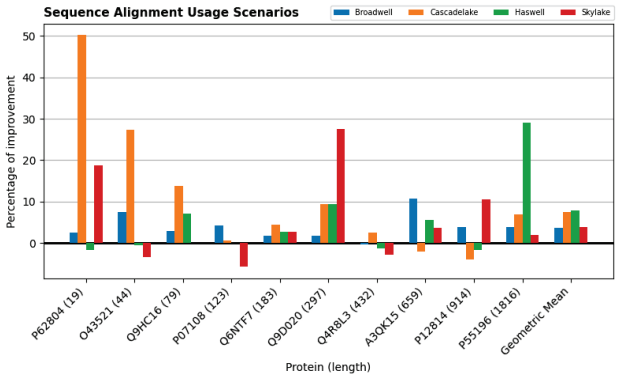


Fig. 10: Performance improvement after hyperparameter tuning for different architectures

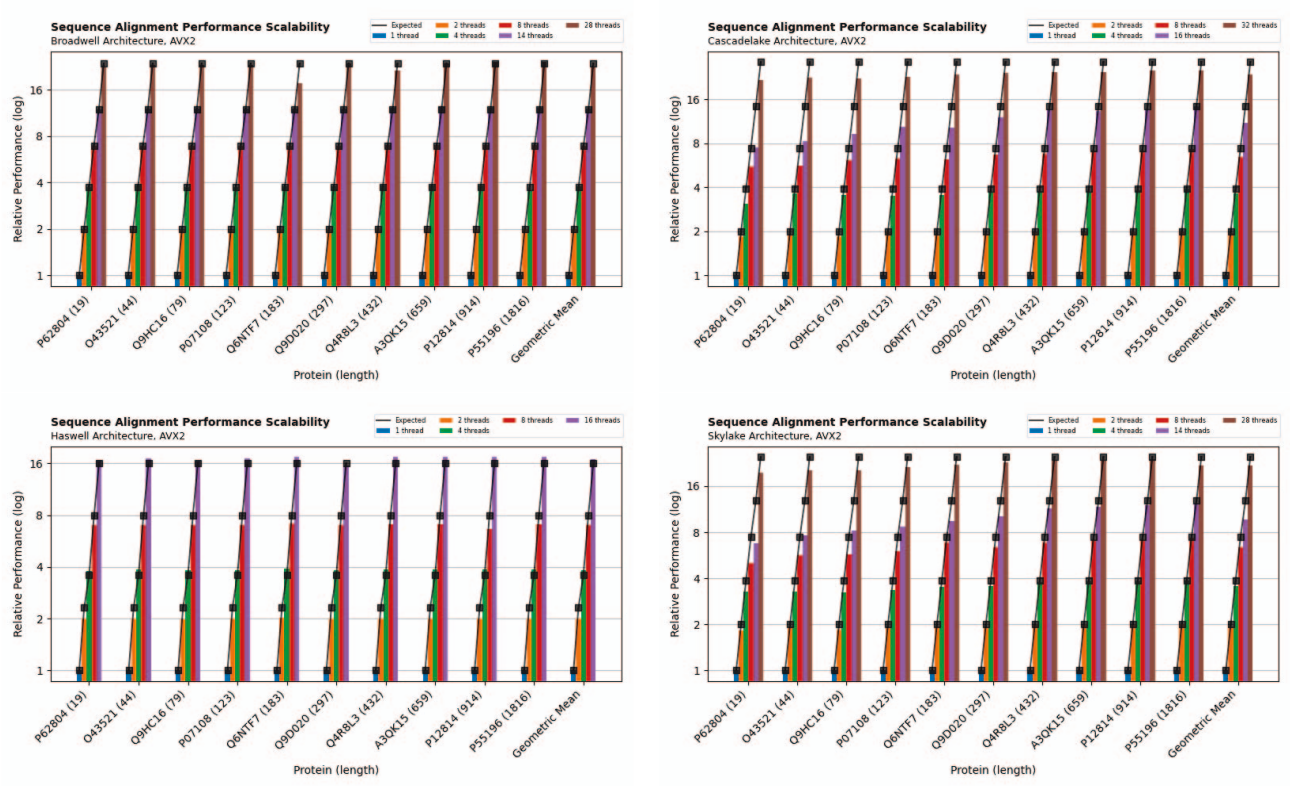


Fig. 11: Performance scaling using more threads for different architectures according to the number of cores

all cores, where we consistently observed high efficiency. Furthermore, the effectiveness of hyperthreading in improving performance provides additional evidence supporting the notion that our computation is not significantly limited by memory. The ability of hyperthreading to efficiently manage CPU resources and enhance throughput, especially in larger query scenarios, suggests that the primary constraints are related to CPU performance rather than memory capability.

This finding has important implications for our understanding of SW's performance limitations. With the CPU frequency showing considerable variability and hyperthreading significantly boosting performance, it becomes clear that the implementation is not currently constrained by memory. Instead, the scalability and efficiency appear to be more influenced by the interplay between CPU multi-threading capabilities, frequency stability, and the strategic use of hyperthreading.

F. Microarchitecture Analysis Using Vtune Profiler

A microarchitectural investigation using the Vtune profiler was used to reveal detailed behaviors within different scenarios, particularly those involving large queries. Although smaller protein results, as shown in Fig. 12, were found to be less reliable, the analysis with larger queries provided robust insights.

Given its significant impact on performance, we focused on scenarios with and without a substitution matrix. Notably, in scenarios with a substitution matrix, the execution was predominantly CPU bound. This was largely due to the core limitations while executing gather instructions.

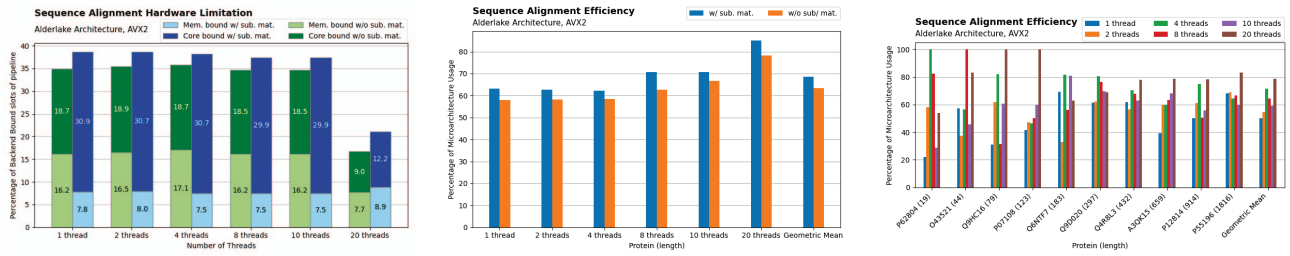
Moreover, our findings were further reinforced when examining scenarios where hyperthreading was employed, using two threads per core. Here, we observed an increase in efficiency, aligning with our earlier observations in the thread scalability analysis. This efficiency boost is likely due to the decreased backend-bound limitations, as the second thread could progress while one was busy, thus reducing idle time and enhancing overall throughput.

In both scenarios with and without the substitution matrix, at least 8 percent of the slots were memory-bound, and up to 18 percent in cases without the substitution matrix. However, the introduction of hyperthreading and the resultant efficient use of CPU pipeline slots suggest that while memory constraints are a factor, the optimization of CPU resources through hyperthreading plays a crucial role in overcoming backend-bound limitations, particularly in CPU-intensive scenarios.

G. Smith-Waterman Usage Scenarios Analysis

We structured our analysis around three distinct scenarios to understand the Smith-Waterman algorithm's performance under varying conditions:

Single Query Scenario: Here, we computed the average performance across different proteins, as there was noticeable variability. Interestingly, larger query sizes yielded better performance. This could be attributed to the efficient utilization



(a) Analysing the backend bound limitation of our work. Backend bound can be either due to being memory bound or core bound.

(b) Efficiency of using the pipeline slots for different numbers of threads for large number of query sequences

(c) Efficiency of using the pipeline slots for different numbers of threads and different query protein

Fig. 12: Analysis using Intel Vtune profiler

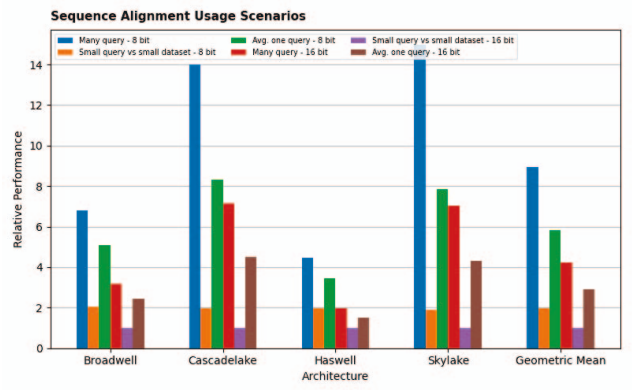


Fig. 13: Performance for different SW usage scenarios

of the dataset by multiple threads, where each thread handles a different segment of the database. Larger queries allow for more optimal usage of these segments.

Multiple Query Scenario: When handling a large number of query sequences, we observed an enhanced performance efficiency. This improvement likely results from the increased parallel processing capability, where multiple queries effectively utilize the database's different sections. In practical applications, this suggests that in environments with a centralized server handling multiple queries, it may be more efficient to accumulate several queries before beginning the computation. This approach could lead to a shorter overall processing time compared to processing each query individually.

Small Sets of Queries and References: In this scenario, we compared a small number of queries with a small database. The performance in this case provided insights into the algorithm's efficiency when dealing with limited data, crucial for smaller-scale or more targeted sequence alignment tasks.

These varied scenarios underscore the adaptability of the Smith-Waterman algorithm to different data scales and query sizes as illustrated in Fig. 13. The findings highlight that the algorithm's performance is not just a function of its computational design but is also heavily influenced by how it is deployed in different operational contexts.

H. Comparison with Parasail Library

In the final phase of this study, we conducted a comparative analysis between our implementation and the established benchmarks of the Parasail library [26]. Parasail was selected because it has a collection of many different implementations for SW, including, wavefront [19], scan [18], and striped [22]. These are well maintained, documented, and easy to use. Moreover, the various implementations are all modular functions within a unified framework giving some degree of fairness when their performance is compared. Some other SW implementations were not used here because they are limited in the queries or data sets supported [27], or because an open source version could not be found [28].

This comparison revealed some insights into the benefits of our approach. Notably, our method consistently outperformed the scan, diag, and striped approaches of Parasail across all cases. Of note is that our approach is diag-based, whereas the best performing Parasail version is striped. Also significant is that the striped and scan versions in Parasail do not always yield stable results: speculative computations are used, which necessitates subsequent correction loops to rectify potential miscalculations [29].

I. Discussion

This study opens several avenues for further research. Further investigation is needed of the centralized server scenario. Our findings suggest that processing a larger number of queries concurrently can significantly boost performance. Another area for future investigation is to make an autotuner to decide what should be the block size in the case of using a substitution matrix which we are currently hand-tuning.

Exploring compiler optimization tuning, including optimization phase ordering and selection, is especially promising. Such tuning has demonstrated improvements in various applications and could be particularly impactful when coupled with advanced hyperparameter tuning strategies [30].

Finally, it could be possible to adapt the proposed methods to FPGAs and GPUs.

V. Related Work

GPU-Based Sequence Alignment: Several studies (e.g., CUDASW++ 3.0 [31] and [32]), have focused on using GPUs

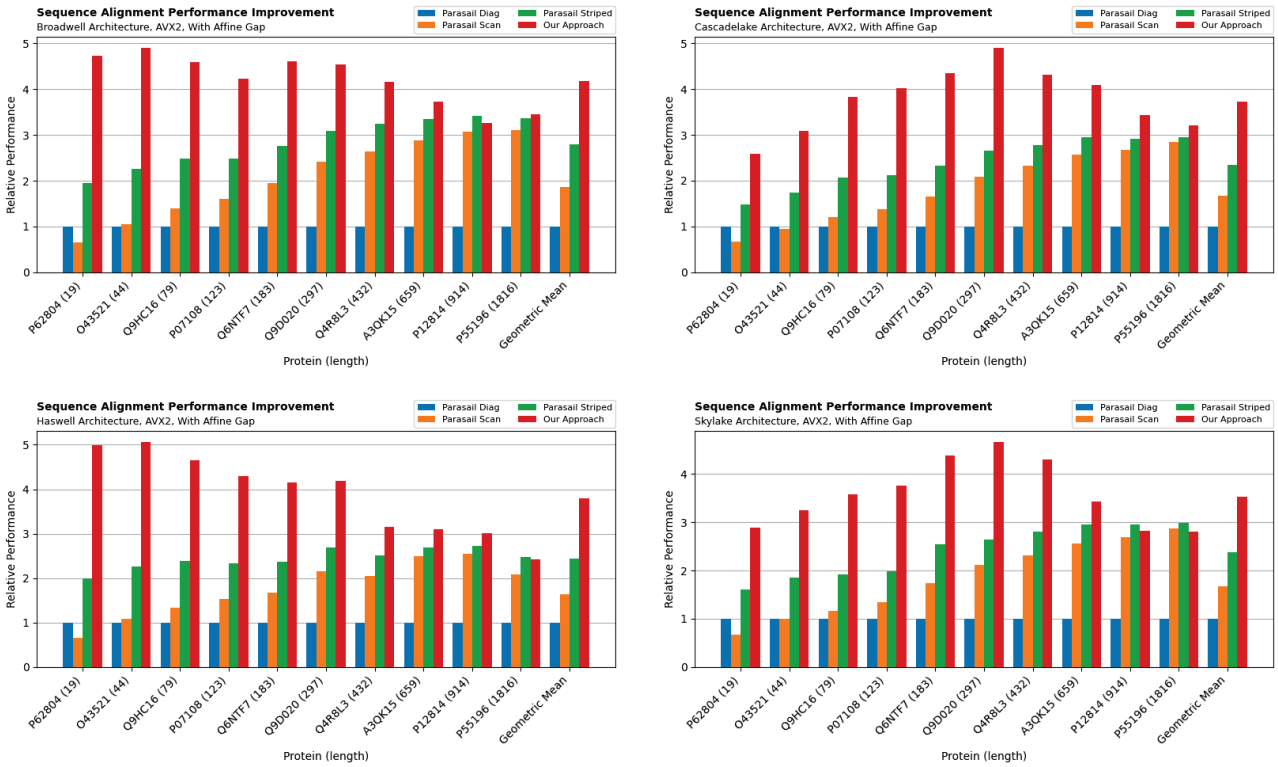


Fig. 14: Performance evaluation for Parasail library implementation of SW versus our work on different architectures for 10 different proteins as the query

to accelerate sequence alignment. The review article [33] highlights the significant performance benefits. [3] investigated the theoretical hardware capability and compared it with their approach.

FPGA-Based Sequence Alignment: There has been much work in FPGA-based SW (e.g., [34]). [35] proposes fine-grained parallelized SW algorithms including backtracking. [36] focuses on implementing multiple sequence alignment. [37] used OpenCL to implement design.

CPU Vectorization-Based Sequence Alignment: Many studies have explored CPU vectorization techniques, e.g. [38] for AVX2 and [39] for ARM's Scalable Vector Extension (SVE). The Parasail library includes [26] three different implementations of SW (as describe above). Works such as [12] and [27] compare their models with some of these functions. Each work demonstrates a different approach in mapping, computation, or algorithm. Other work studies the implementation of the substitution matrix: [21] and [18] precompute the database. [10] introduces a new way of computing the maximum, computing it in an array for all the vector lanes.

We applied a disjoint set of these implementations and show the benefits of this approach. Note that combining multiple methods often involves substantial recoding, e.g., to deal with dependencies and SIMD instruction limitations.

Hybrid (CPU-GPU or Heterogeneous) Approaches: [40] presents CloudSW, a hybrid computing framework that leverages Apache Spark and SIMD instructions for large-scale biological sequence database search on heterogeneous computing environments. [41] introduces OSWALD, a hybrid approach combining OpenMP multithreading and SIMD computing on the host with OpenCL on FPGAs to optimize sequence alignment efficiency. [42] discusses experiences in porting the SWdb sequence alignment tool from CUDA to DPC++ for utilization on heterogeneous architectures.

VI. Conclusion

This study provides an exploration of sequence alignment using the Smith-Waterman algorithm, emphasizing its application across a variety of Intel architectures and computational scenarios. Our analysis covers critical aspects such as SIMD length choices, the role of substitution matrices, affine gap penalties, and the importance of traceback functionality.

One of the most significant advances comes from compiler hyperparameter tuning, which resulted in an average performance improvement of 10% and up to 50% in certain cases. These improvements, however, were found to be query size-dependent, underlining the importance of targeted optimization strategies. Additionally, the implementation of a centralized server for batch processing of sequence alignment queries proved to be highly effective, enhancing computational efficiency by a factor of two in some cases.

Our approach demonstrated robust performance, in particular, for all query sizes, and exhibited a deterministic nature,

ensuring consistent results. This was further validated in our comparative analysis with the Parasail library, where our methods also showed significantly improved performance.

ACKNOWLEDGMENTS

This work was supported, in part, by the NSF through awards CCF-1919130 and CCF 2151021; the NIH through award R44GM128533; and by Red Hat through awards 2023-01-RH13 and 2024-01-RH01.

REFERENCES

[1] M. Orobitg, F. Guirado, F. Cores, J. Llados, and C. Notredame, "High performance computing improvements on bioinformatics consistency-based multiple sequence alignment tools," *Parallel Computing*, vol. 42, pp. 18–34, 2015.

[2] M. R. Aniba, O. Poch, and J. D. Thompson, "Issues in bioinformatics benchmarking: the case study of multiple sequence alignment," *Nucleic Acids Research*, vol. 38, no. 21, pp. 7353–7363, 2010.

[3] L. Ligowski and W. Rudnicki, "An efficient implementation of Smith Waterman algorithm on GPU using CUDA for massively parallel scanning of sequence databases," in *IEEE International Symposium on Parallel & Distributed Processing*, 2009.

[4] E. F. d. O. Sandes and A. C. M. de Melo, "Smith-Waterman alignment of huge sequences with GPU in linear space," in *IEEE International Parallel & Distributed Processing Symposium*, 2011.

[5] Y. Liu, W. Huang, J. Johnson, and S. Vaidya, "Gpu accelerated smith-waterman," in *6th Int. Conf. on Computational Science*. Springer, 2006.

[6] T. Van Court and M. C. Herbordt, "Families of fpga-based accelerators for approximate string matching," *Microprocessors and Microsystems*, vol. 31, no. 2, pp. 135–145, 2007.

[7] Z. Nawaz, M. Nadeem, H. Van Someren, and K. Bertels, "A parallel FPGA design of the Smith-Waterman traceback," in *Int. Conf. on Field-Programmable Technology*, 2010.

[8] Y. Yamaguchi, H. K. Tsoi, and W. Luk, "FPGA-based Smith-Waterman algorithm: Analysis and novel design," in *7th Int. Symp. on Reconfigurable Computing: Architectures, Tools and Applications*, 2011.

[9] B. Harris, A. C. Jacob, J. M. Lancaster, J. Buhler, and R. D. Chamberlain, "A banded Smith-Waterman FPGA accelerator for Mercury BLASTP," in *Int. Conf. Field Prog. Logic and Applications*, 2007.

[10] M. Zhao, W.-P. Lee, E. P. Garrison, and G. T. Marth, "SSW library: an SIMD Smith-Waterman C/C++ library for use in genomic applications," *PloS one*, vol. 8, no. 12, 2013.

[11] D. Zou, Y. Dou, and F. Xia, "Optimization schemes and performance evaluation of Smith–Waterman algorithm on CPU, GPU and FPGA," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 14, pp. 1625–1644, 2012.

[12] D. T. Popovici, M. G. Awan, G. Guidi, R. Egan, S. Hofmeyr, L. Oliker, and K. Yelick, "Designing Efficient SIMD Kernels for High Performance Sequence Alignment," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2023, pp. 167–176.

[13] A. Mahram and M. Herbordt, "NCBI BLASTP on High Performance Reconfigurable Computing Systems," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, pp. 6.1–6.20, 2015.

[14] L. Lindsey, M. Haseeb, H. Sundar, and M. Awan, "TANGO: A GPU optimized traceback approach for sequence alignment algorithms," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 760–765.

[15] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks," *Proceedings of the National Academy of Sciences*, vol. 89, no. 22, pp. 10 915–10 919, 1992.

[16] M. Vingron and M. S. Waterman, "Sequence alignment and penalty choice: Review of concepts, case studies and implications," *Journal of Molecular Biology*, vol. 235, no. 1, pp. 1–12, 1994.

[17] S. Lloyd and Q. O. Snell, "Hardware accelerated sequence alignment with traceback," *Int. J. of Reconfigurable Computing*, pp. 1–10, 2009.

[18] T. Rognes, "Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation," *BMC Bioinformatics*, vol. 12, no. 1, pp. 1–11, 2011.

[19] A. Wozniak, "Using video-oriented instructions to speed up sequence comparison," *Bioinformatics*, vol. 13, no. 2, pp. 145–150, 1997.

[20] B. Alpern, L. Carter, and K. Su Gatlin, "Microparallelism and high-performance protein matching," in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, 1995.

[21] T. Rognes and E. Seeberg, "Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors," *Bioinformatics*, vol. 16, no. 8, pp. 699–706, 2000.

[22] M. Farrar, "Striped Smith–Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[23] W. Von Hagen, *The definitive guide to GCC*. Apress, 2011.

[24] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys*, vol. 51, no. 5, pp. 1–42, 2018.

[25] E. Boutet, D. Lieberherr, M. Tognolli, M. Schneider, and A. Bairoch, "UniProtKB/Swiss-Prot: the manually annotated section of the UniProt KnowledgeBase," in *Plant Bioinformatics: Methods and Protocols*. Springer, 2007, pp. 89–112.

[26] J. Daily, "Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments," *BMC Bioinformatics*, vol. 17, no. 1, pp. 1–11, 2016.

[27] E. Rucci, C. Garcia Sanchez, G. Botella Juan, A. D. Giusti, M. Naiouf, and M. Prieto-Matias, "SWIMM 2.0: enhanced smith–waterman on intel's multicore and manycore architectures based on AVX-512 vector extensions," *Int. J. of Parallel Programming*, vol. 47, pp. 296–316, 2019.

[28] K. Hou, H. Wang, and W.-c. Feng, "AAlign: A SIMD framework for pairwise sequence alignment on x86-based multi- and many-core processors," in *IEEE International Parallel and Distributed Processing Symposium*, 2016, pp. 780–789.

[29] R. Snytsar, "De (con) struction of the lazy-f loop: improving performance of Smith Waterman alignment," in *IEEE 19th International Conference on Bioinformatics and Bioengineering*, 2019, pp. 7–10.

[30] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2012, pp. 147–162.

[31] 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, pp. 1–10, 2013.

[32] A. Khajeh-Saeed, S. Poole, and J. B. Perot, "Acceleration of the Smith–Waterman algorithm using single and multiple graphics processors," *J. Computational Physics*, vol. 229, no. 11, pp. 4247–4258, 2010.

[33] Z. Xia, Y. Cui, A. Zhang, T. Tang, L. Peng, C. Huang, C. Yang, and X. Liao, "A review of parallel implementations for the Smith–Waterman algorithm," *Computational Life Sciences*, pp. 1–14, 2021.

[34] 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, pp. 1–7, 2007.

[35] X. Fei, Z. Dan, L. Lina, M. Xin, and Z. Chunlei, "FPGASW: accelerating large-scale Smith–Waterman sequence alignment application with backtracking on FPGA linear systolic array," *Computational Life Sciences*, vol. 10, pp. 176–188, 2018.

[36] A. Mahram and M. C. Herbordt, "FMSA: FPGA-accelerated ClustalW-based multiple sequence alignment through pipelined prefiltering," in *Int. Symp. on Field-Programmable Custom Computing Machines*, 2012.

[37] E. Rucci, C. Garcia, G. Botella, A. De Giusti, M. Naiouf, and M. Prieto-Matias, "SWIFOLD: Smith-Waterman implementation on FPGA with OpenCL for long DNA sequences," *BMC Systems Biology*, vol. 12, no. 5, pp. 43–53, 2018.

[38] M. Malik, S. Malhotra, and N. Prasanth, "Time Improvement of Smith-Waterman Algorithm Using OpenMP and SIMD," in *Futuristic Trends in Networks and Computing Technologies*. Springer, 2020, pp. 686–697.

[39] D.-H. Park, J. Beaumont, and T. Mudge, "Accelerating smith-waterman alignment workload with scalable vector computing," in *2017 IEEE International Conference on Cluster Computing*, 2017, pp. 661–668.

[40] B. Xu, C. Li, H. Zhuang, J. Wang, Q. Wang, and X. Zhou, "Efficient distributed Smith-Waterman algorithm based on Apache Spark," in *10th International Conference on Cloud Computing*, 2017, pp. 608–615.

[41] E. Rucci, C. Garcia, G. Botella, A. E. De Giusti, M. Naiouf, and M. Prieto-Matias, "OSWALD: OpenCL Smith–Waterman on Altera's FPGA for Large Protein Databases," *Int. J. High Performance Computing Applications*, vol. 32, no. 3, pp. 337–350, 2018.

[42] M. Costanzo, E. Rucci, C. García-Sánchez, M. Naiouf, and M. Prieto-Matías, "Migrating CUDA to oneAPI: A Smith-Waterman case study," in *Bioinformatics and Biomedical Engineering*, 2022, pp. 103–116.