Abstract

In bioinformatics, effective sequence alignment is essential, and the Smith-Waterman technique is one of the industry standards for local alignment. Our project's main goals were to benchmark and optimize several Smith-Waterman implementations and investigate alternate alignment techniques such as Wavefront Alignment (WFA). Using SIMD vectorization, we improved a non-SIMD baseline implementation of Smith-Waterman, obtaining 1.5x to 2x speedups, especially for DNA alignments. We additionally parallelized and aligned memory to provide incremental improvements in an existing SIMD-accelerated implementation. We went beyond Smith-Waterman and implemented WaveFront Alignment to solve the edit distance problem, which outperformed conventional edit-distance techniques by up to 22%, demonstrating excellent scalability. These findings demonstrated how important it is to improve sequence alignment performance using both adaptive algorithmic techniques and hardware-level enhancements.

Introduction

Our goal was to optimize and benchmark various Smith-Waterman implementations. We wanted to see how much we can improve the performance of a basic implementation of Smith-Waterman by using SIMD operations and how would it compare against a state-of-the-art SIMD-accelerated implementation. We also hoped to improve the existing SIMD-accelerated implementation. Furthermore, we explored alternative alignment strategies, such as wavefront alignment, and measured its performance against traditional edit distance computations.

By implementing SIMD-accelerated Smith-Waterman from scratch and comparing it to our non-SIMD benchmark, we hoped to quantify the direct impact of vectorization on both DNA and protein sequence alignments. Furthermore, by improving an already optimized SIMD implementation, we made incremental performance improvements when starting from an already highly efficient codebase. Ultimately, we also wanted to measure how WaveFront Alignment (WFA) compares to a simpler approach to the edit-distance problem, with expectations that WFA would deliver superior performance as sequences grow.

In conclusion, our work revealed that even a previously optimized, SIMD-accelerated Smith-Waterman implementation can benefit, even just marginally, from careful memory alignment, parallelization, and vectorization improvements. Our custom SIMD version showed substantial speedups over the non-SIMD benchmark, especially for DNA sequence alignment, and SIMD itself offered massive performance benefits for both DNA and protein workloads. Regarding WFA, our experiments yielded data indicating WFA was approximately 22% faster than the simple edit-distance algorithm.

Prior Work

The work by Rognes and Seeberg (2000) showed the first major speedup of Smith-Waterman database searches by using parallel processing on standard microprocessors, achieving a six-fold performance increase. Building on this, Farrar (2007) introduced a"striped" approach to Smith-Waterman implementation using SIMD (Single Instruction Multiple Data) that also achieved a six-fold speedup over other SIMD implementations at the time. The striped approach in Farrar's implementation divides sequence data into vertical stripes that align with SIMD vector widths, which allows for better parallelization.

Rognes (2011) advanced the field again by developing inter-sequence SIMD parallelization techniques for Smith-Waterman searches, achieving even better performance. Zhao et al. (2013) created the SSW library, which provided an implementation of these SIMD optimizations in multiple programming languages (C, C++, Java, Python, R).

While these SIMD-based advancements have been important for speeding up exact pairwise alignment, they are still based on the traditional dynamic programming (DP) approach. While these techniques made scoring each cell in the Smith-Waterman matrix much faster, the techniques still had the inherent complexity of filling the DP table.

To solve these issues, Marco-Sola and colleagues (2021) proposed a different perspective with the Wavefront Alignment (WFA) algorithm. The WFA algorithm employs a fundamentally different view of the alignment space: instead of expanding along the full dynamic programming matrix, it uses a "wavefront" that progresses through the matrix in a diagonal manner. Instead of computing every cell, the WFA algorithm identifies and follows promising alignment paths, which effectively narrows the search space and achieves substantial reductions in both computation time and memory usage. This allowed WFA to handle large alignments with affine gap models at near-optimal complexity, setting a new performance benchmark for exact pairwise alignment.

Methods and software

Implemented Smith-Waterman

We wrote our own non-SIMD Smith-Waterman implementation. Our implementation uses a 2D dynamic programming matrix, as discussed in class, written in C++.

The code can be found in *smith-waterman/smith-waterman-1-non-simd/ f*older. There are two versions: *smith-waterman-dna-simd.cpp* for dna and *smith-waterman-protein-simd.cpp* for protein. The instructions to run can be found in the directory's README.

Implemented Smith-Waterman with SIMD

We enhanced the performance of our basic Smith-Waterman implementation using SIMD (Single Instruction, Multiple Data) operations. Our original version computed the scoring matrix iteratively which is computationally intensive, especially for large sequences. To optimize performance, the program was modified to leverage SIMD capabilities using ARM Neon intrinsics.

In the updated version, we optimized the scoring operations by processing four cells simultaneously instead of one. Using SIMD registers (int32x4_t), we vectorized all operations like computing match/mismatch scores, gap penalties, and finding the maximum score among possible alignments (match, insertion, deletion, or zero). For sequence lengths not divisible by four, we reverted to scalar operations to handle the remaining cells, ensuring accuracy.

The code can be found in *smith-waterman/smith-waterman-2-simd/* folder. There are two versions: *smith-waterman-dna-simd.cpp* for dna and *smith-waterman-protein-simd.cpp* for protein. The instructions to run can be found in the directory's README.

Optimizing Striped Smith-Waterman

We found an existing SIMD-accelerated Smith-Waterman implementation on Github and attempted to optimize it. The Github repository implemented Smith-Waterman in C, C++, Python, Java, and R, but we modified the C implementation since it is the most performant. This repository was also one of the few which let us run SIMD Smith-Waterman seamlessly on our devices (ARM-based MacBooks) without any environment issues. We describe the optimizations we made to the library below.

Query Profile Optimization (qP_byte)

We made three key optimizations to the qP_byte function in ssw.c, which generates a query profile for optimized sequence alignment (see diff here):

- Aligned Memory Allocation:
    - The library initially allocated memory using `malloc`, which does not ensure alignment. We changed it so memory is allocated using `_mm_malloc`, which ensures the memory is 16-byte aligned, allowing for more efficient SIMD loads and stores. We thought proper alignment would reduce the chance of costly unaligned memory accesses.
- OpenMP Parallelization:
    - Previously, the loop over `nt` was serialized and running on a single thread. After our changes, the `#pragma omp parallel for` directive parallelizes the outer loop over `nt`. This distributes the workload across multiple threads.
- More Efficient Vectorization:
    - The original code wrote one byte at a time directly to the memory location pointed by `t`. We modified it so each 128-bit vector gets constructed in a register first by directly assigning each of the 16 bytes. After populating the 128-bit vector, we use `_mm_store_si128()` to store it into the output array in a single operation.

Backward Trace Optimization

For this optimization, we focused on the backward trace section. The backward trace is in charge of finding the alignment ending position. We went about it by applying a SIMD-based optimization, improving runtime efficiency and scalability, specifically for larger datasets, by replacing scalar operations with vectorized processing (see diff here):

- Replacing Scalar Comparisons with SIMD Vectorization:
    - Before: In the original code, they used a scalar loop to check each element in pvHmax with the maximum value. One element was processed in each loop iteration, and if a match was discovered, the alignment ending position was updated. Large data sets experienced increased latency as a result of the original code sequentially processing the items. Not only that but there was possible reliance on branching for the comparisons, which could lead to inaccurate predictions.

    - After: We replaced the scalar loop with SIMD instructions, allowing 16 elements to be processed in parallel. The optimized code used `_mm_cmpeq_epi8` to compare 16 elements at a time. It also generated a bitmask with `_mm_movemask_epi8` to identify matches in constant time. We also employed `_builtin_ctz` to locate the least significant set bit in the mask to enable efficient updates to the alignment position. In the end, for any remaining non-multiple-of-16 cases, we employed a fallback scalar loop to catch these edge cases.

- Mask-Based Filtering for Branch Reduction:
    - Before: In the original code of the backward trace, there were conditional branches (if statements) that were in the loop to check whether each element in the array matched the max value. This branching made room for pipeline stalls and potential performance degradation when conditions weren't consistently met.
    - After: To combat this, we introduced mask-based filtering to eliminate the branching. This leveraged SIMD comparison and bitmask operations to avoid branching and process multiple matches in the same iteration, in an attempt to be more efficient.

WaveFront Alignment (WFA)

In this part of the project, we implemented WaveFront Alignment (WFA) in C++ from scratch. Once implemented, we could benchmark this against the Levenshtein distance calculation algorithm we covered in class, to ascertain how much of a performance gain there is from using the WFA technique. We chose WFA as the technique to implement because it should, in theory, scale somewhat better than quadratic as input sequences get longer. WFA optimizes computation by focusing only on some parts of the alignment matrix; this is unlike Smith-Waterman, which will always generate values for the entire matrix. Further, in situations where two sequences have long stretches of homology, WaveFront should also perform better than the benchmark algorithm. Implementing WaveFront was also an opportunity for us to explore some more advanced algorithmic techniques beyond what we covered in class.

To facilitate this, we wrote two implementations that solve the edit distance problem. One used a WaveFront alignment technique, and the other took the basic Levenshtein DP approach; both of these C++ files are in the *wavefront-alignment* directory. See *wavefront-alignment/README.md* for directions to run this code. Our experiment design was to take each of our 3 FASTA DNA files (*wavefront-alignment/dna-input*), and mutate each one 3 times, at a rate of 0%, 25%, and 50%. Mutate here means a number of positions (based on the rate, 25% means 25% of the bases) were selected and randomly mutated. Now we had 3 input files, and each has 3 files with the same sequence, but mutated to some degree. This is 9 different tests, where we try to align (and find the edit distance between the two files); each of the 9 tests was run using both the edit_distance.cpp algorithm and wavefront.cpp algorithm (3 times each to average results), and the results were compared in the table below. In addition to the code for the two algorithms, we also provide the program used to generate the mutated sequences, given an original sequence and a mutation rate; the mutated files we used for our experiment are in the *wavefront-alignment/dna-input-mutated* directory.

Specifically, our goal here was to measure not only how the WFA algorithm scales compared to the regular Smith-Waterman approach, but also how the number of mutations between the two sequences affect performance for the two algorithms, knowing that WFA shines when the two sequences have long stretches of the same bases.

Results

Benchmarking Smith-Waterman

This part of the project's goal was to benchmark the performance gains of a SIMD-accelerated implementation of Smith-Waterman, versus a non-SIMD implementation. The motivation for conducting this benchmarking experiment is to ascertain just how much of a performance gain SIMD provides, especially on long sequences of data. We also wanted to compare the performance differences between the alignment of DNA and protein sequences.

For the DNA sequence alignment benchmark, we tested our implementation against target sequences of varying lengths, from about 1,000 bases to over 50 million bases, while holding the query sequence constant. Each dataset was processed three times and averaged. We compared the performance of four implementations: a non-SIMD baseline, our SIMD-accelerated code, the SIMD implementation from GitHub, and an optimized version of it. Our SIMD implementation demonstrated consistent improvement over the non-SIMD reference, generally offering about 1.5x to 2x speedups. The GitHub SIMD implementation performed orders of magnitude faster than both our SIMD and the non-SIMD versions. Our optimized version of the implementation showed a modest improvement over the original.

A similar pattern emerged with the protein sequence alignment benchmarks. We evaluated the alignment of three sets of protein queries (100, 1,000, and 10,000 queries) against fixed reference sequences. Once again, our SIMD implementation outpaced the non-SIMD version, though only by a modest factor compared to the gains observed in some of the DNA tests. Similar to DNA, the GitHub SIMD implementations were far more efficient than our SIMD implementation. However, our optimization version of the Github library did not perform better than the library itself. We are not sure why, but perhaps it is because the protein alphabet is larger than the DNA alphabet (20 amino acids vs 4 bases). This might mean that our test device was not powerful enough to utilize parallelism or SIMD-optimization for protein sequences, but was good enough for DNA sequences.

We conducted our experiments on an Apple M2 MacBook Air with a 8-core CPU and 16 GB of RAM. The results of our experiments are shown in the table below.

DNA Sequence Alignment

	1k.fa (length 1,001 bases)	10k.fa (length 10,001 bases)	100k.fa (length 100,001 bases)	1M.fa (length 1,000,001 bases)	10M.fa (length 10,000,001 bases)	Homo_sapiens.GRCh38.dna_rm.chromosome.22.fa (length 50,818,468 bases)
non-SIMD	0.01644 s	0.052414 s	0.451222 s	4.138367 s	41.117629 s	254.634861 s
SIMD (ours)	0.005189 s	0.036570 s	0.237849 s	2.184179 s	23.101347 s	123.166507 s
SIMD (github)	0.000271 s	0.001416 s	0.011568 s	0.080957 s	0.551085 s	2.225050 s

SIMD (github optimized)	0.000191 s	0.001259 s	0.005509 s	0.057751 s	0.517614 s	2.182498 s

Protein Sequence Alignment

	protein-reads100.fa 136 queries against 36 target sequences Avg read length: 211 Avg ref length: 280	protein-reads1000.fa 1036 queries against 36 target sequences Avg read length: 222 Avg ref length: 280	protein-reads10000.fa 10036 queries against 36 target sequences Avg read length: 224 Avg ref length: 280
non-SIMD	15.375676 s	123.454744 s	1189.254530 s
SIMD (ours)	11.163812 s	92.117955 s	941.656703 s
SIMD (github)	0.285043 s	7.187794 s	11.231611 s
SIMD (github optimized)	0.279794 s	7.484043 s	11.252006 s

DNA Alignment Improvement (library vs optimized library)

Average CPU time speed-up over 20 trials, each having 100 runs (%)	4.8
% of runs where optimized version ran faster (19/20)	95

For these results, we wrote a script called benchmark.sh which runs the original program and our optimized program 100 times on the same (DNA sequence, query) pair and reports the total CPU time taken. We ran benchmark.sh 20 times and computed the metrics above. While the degree of speed-up varied quite a lot from run to run, with some being negligible, we noticed that a speed-up occurred consistently almost every time we ran the script. We added the output of 20 runs here.

Benchmarking WaveFront Alignment

The goal here was to measure how well WaveFront alignment would perform as sequences got longer, and more mutations were introduced. See **Methods and Software** for a description of the experiment we run. We conducted our experiments on an Apple M2 MacBook Air with a 8-core CPU and 16 GB of RAM. The results of our experiments are shown in the table below. The 1k.fa, 10k.fa, and 100k.fa sequences are 1,000, 10,000, and 100,000 bases long, respectively. ED = Edit Distance; WF: WaveFront Alignment

	1k.fa		10k.fa		100k.fa	
	ED	WF	ED	WF	ED	WF
0% Mutated	0.038021	0.034236s	3.528410s	2.793481s	370.48s	251.415s

25% Mutated	0.051920s	0.041609s	3.753007s	2.990397s	374.313s	278.117s
50% Mutated	0.054503s	0.046666s	3.753007s	3.318075s	372.184s	325.171s

There are a few trends to unpack here. First is focusing on just the regular dynamic programming edit distance matrix calculation. We expect as we 10x the input sequence lengths, a 100x increase in the runtime (10 x 10 = 100). This is indeed what we see from 10k to 100k, but the 1k to 10k doesn't exactly follow this. Our hypothesis as to why this happens is simply that with the smaller data input, the fixed costs of the program dominate, and therefore the 1k.fa test doesn't follow the trend of the other two sequence lengths. As the sequences grow, the CPU-intensive matrix filling dominates the runtime, and we see the trend we expect.

Looking at WaveFront, we also expect to see generally quadratic growth, as the computational problem is the same, but for lower mutation rates, there should be some performance gain, as WF can take advantage of homologous sequences to compute fewer dynamic programming cells. Similar to ED, the 1k.fa input doesn't follow the 100x trend, but the 10k.fa to 100k.fa increase does. However, in all three mutation rates, we can see a slight improvement over strict quadratic growth (less than 100x). Specifically, for the 0% mutation rate, from 2.793s to 251.415s is a 90x change, reflecting the WF performance gain because the sequences are identical. Similar but slightly less drastic is the change from 2.990s to 278.117s, a 93x jump. Finally, at the much higher 50% mutation rate, the jump is 98x, and this improvement compared to 100x could perhaps even just be chalked up to noise. All this said, this was an interesting finding, as it meant that highly homologous sequences, as 0% and 25% mutated sequences are, allow WaveFront's optimizations to apply, and create a performance gain. Finally, strictly comparing WF and ED, WaveFront also performs better. WF was around 17% faster on the 10k-length sequences, and 24% better for the 100k-length sequences. Again, based on the WF optimizations discussed, this disparity makes sense and is exactly what we were looking for.

Conclusion

Our work set out to explore and optimize various Smith-Waterman sequence alignment implementations and then compare them against newer alignment strategies. By starting with a basic non-SIMD Smith-Waterman implementation and then introducing SIMD vectorization, we were able to measure tangible performance improvements (1.5x to 2x speedup for DNA sequence alignments). We also experimented with optimizing a highly efficient, state-of-the-art SIMD-accelerated Smith-Waterman implementation, managing to extract incremental gains through careful attention to memory alignment and parallelization techniques. While our SIMD improvements were more pronounced for DNA sequences than for protein sequences, likely due to the smaller DNA alphabet and better fit for vectorization, SIMD optimizations nevertheless outperformed their non-SIMD counterparts across the board. Beyond Smith-Waterman, we looked into wavefront alignment and found it could run approximately 22% faster than a straightforward edit-distance-based method. In conclusion, our findings showed that SIMD can significantly enhance classical DP-based alignment algorithms, while more adaptive methods like WaveFront Alignment hold promise for even greater efficiency gains.

Literature

1. Torbjørn Rognes and Erling Seeberg. Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors. Bioinformatics, 16(8):699–706, 2000.

2. Michael Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. Bioinformatics, 23(2):156–161, 2007.

3. Torbjørn Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. BMC bioinformatics, 12(1):221, 2011.

4. Hajime Suzuki and Masahiro Kasahara. Acceleration of nucleotide semi-global alignment with adaptive banded dynamic programming. bioRxiv, page 130633, 2017.

5. Zhao M, Lee W-P, Garrison EP, Marth GT (2013) SSW Library: An SIMD Smith-Waterman C/C++ Library for Use in Genomic Applications. PLoS ONE 8(12): e82138. doi:10.1371/journal.pone.0082138

6. D. -H. Park, J. Beaumont and T. Mudge, "Accelerating Smith-Waterman Alignment Workload with Scalable Vector Computing," 2017 IEEE International Conference on Cluster Computing (CLUSTER), Honolulu, HI, USA, 2017, pp. 661-668, doi: 10.1109/CLUSTER.2017.91. keywords: {Sequential analysis;Bioinformatics;Genomics;Acceleration;Computer architecture;DNA;Organisms;Accelerator architectures;Genomics;Dynamic programming},

7. Santiago Marco-Sola, Juan Carlos Moure, Miquel Moreto, and Antonio Espinosa. Fast gapaffine pairwise alignment using the wavefront algorithm. Bioinformatics, 37(4):456–463, 2021

8. Santiago Marco-Sola, Jordan M Eizenga, Andrea Guarracino, Benedict Paten, Erik Garrison, and Miquel Moreto. Optimal gap-affine alignment in o (s) space. Bioinformatics, 39(2):btad074, 2023.