SMITH WATERMAN ALGORITHM

Cache Efficient Vectorization & Parallelization for Protein Alignment

Muhammad Aseef Imran

Table of Contents

Introduction & Scope

For this project, we shall implement the Smith-Waterman dynamic programming algorithm. The algorithm is foundational in computational biology and offers a unique chance to try to solve a computational problem from a biological perspective. While very biologically relevant, as we shall see, the algorithm is very memory intensive. Nevertheless, many parts of this algorithm are computed in local proximity to another in memory while other parts can be computed independent from another making it a compelling candidate for further study and optimization.

The Smith-Waterman Algorithm: Standard Algorithm

The standard Smith-Waterman (SW) algorithm is a dynamic programming method for local sequence alignment, designed to identify similar regions between two arbitrary sequences. In biology, this is typically DNA, RNA, or protein sequences. It constructs a scoring matrix by comparing all possible character alignments between two sequences and assigns scores based on match, mismatch, and gap penalties. The key idea is to fill the matrix such that each cell stores the score of the best local alignment ending at that position, and then trace back from the cell with the maximum score to find the optimal local alignment[1].

Formally, for two sequences a and b, the scoring matrix H is initialized to zero, and then updated using the following recurrence:

$$H(i,j) = \max \begin{cases} 0, \\ H(i-1,j-1) + s(a_i,b_j), \\ H(i-1,j) - d, \\ H(i,j-1) - d \end{cases}$$

Here, $s(a_i,b_j)$ is the substitution score for aligning (or misaligning) characters a_i and b_j and d is a fixed gap penalty[2]. In the standard SW algorithm, $s(a_i,b_j)$ assigns fixed scores for alignment or misalignment. Extensions to this algorithm allow $s(a_i,b_j)$ to assign each sequence a specific score as described later.

[1] Korf
[2] "Smith–Waterman Algorithm."

Time Complexity

The time complexity of the standard SW algorithm is O(mn), where m and n are the lengths of the two input sequences. This is because the dynamic programming table has m×n entries, and each entry is computed in constant time.

Space Complexity

The space complexity is also O(mn), due to the need to store the entire dynamic programming matrix.

Arithmetic Intensity (AI)

Most implementations of SW deal exclusively with integers. Since Arithmetic Intensity is traditionally defined as the number of floating-point operations per memory access, quantifying the AI is meaningless in the traditional sense.

However, if we relax this definition to include other operations, then we can find the AI of the SW algorithm as follows per iteration:

- Integer Operations:
    - 3 integer max operations over four integers (zero, top, left, top-left)
    - 3 integer additions (for adding the penalties)
    - 1 character comparison
- Memory Reads:
    - 2 for the current sequence character in sequence a and b,
    - 3 cells from the DP matrix (top, left, top-left)
    - 1 write for the current i,j cell

Putting this all together, we get an AI of 7/6 = 1.1667 for the base SW algorithm (where 7 is the number of computational operations and 6 is the number of memory operations).

SW-Extensions: Towards Biological Realism

While the standard Smith-Waterman algorithm provides exact alignments, it simplifies the biological reality of mutation and evolutionary pressures. To improve this, we employ two major extensions: affine gap penalties and substitution matrices[3]. These changes better model actual biological processes by assigning more nuanced costs to insertions/deletions and by adapting to different types of similarity (e.g., evolutionary vs. functional)[4] [5].

[3] "Smith–Waterman Algorithm."
[4] "Point Accepted Mutation."
[5] "BLOSUM."

Affine Gap Extensions

In biological sequences, long gaps are generally more likely to be the result of a single insertion/deletion event rather than many small ones. To account for this, the affine gap model introduces two parameters: a gap opening penalty g and a gap extension penalty e, where g > e. This penalizes starting a new gap more than extending an existing one, which better reflects real-world sequence variations.

The affine gap model requires tracking additional state, so instead of using a single dynamic programming matrix, it uses three:

- One for match/mismatch scores,
- One for tracking vertical gaps (insertions),
- One for tracking horizontal gaps (deletions).

This increases the algorithmic complexity by three fold for both time and space requirements[6]. However, the overall asymptotic complexity remains O(mn) in both time and space, since constants are ignored in big-O notation.

Substitution Matrix

The substitution matrix is crucial in scoring alignments. It quantifies the cost of substituting one amino acid or nucleotide for another, based on empirical data or evolutionary models. Different matrices serve different purposes. For instance, PAM (Point Accepted Mutation) matrices emphasize evolutionary relationships[7], while BLOSUM (BLOcks SUbstitution Matrix) matrices are tailored for functional comparisons[8].

The inclusion of this substitution matrix requires maintaining a 2D table containing scores for the comparison of every possible character in the “alphabet” (i.e. all possible nucleotides or DNA/RNA bases). In addition, one must perform an additional memory operation per iteration. This increases both time and space complexity of the algorithm by a constant factor.

While in this project, we use the “PAM250” matrix for all experiments, we could have just as well used any other substitution matrix without any notable impact to performance.

[6] R. Sajjadinasab
[7] "Point Accepted Mutation."
[8] "BLOSUM."

Protein Databases

In practice, sequence alignment isn't done one-on-one. Instead, real use cases often involve querying one sequence against a large database of sequences (one-to-many), or even many query sequences against a large database (many-to-many)[9]. While for this project, we focused on the "one-to-many" alignment case, the methods and optimizations explored here, however, extend naturally to many-to-many scenarios.

Our Scope and Assumptions

For this project, we will be implementing the full-fledged SW algorithm with all its bells and whistles: the sequence alignment, the substitution matrix, and the affine-gap penalties. Formally, this is defined as follows:

$$E[i][j] = \max \begin{cases} H[i][j-1] - g, \\ E[i][j-1] - e \end{cases}$$

$$F[i][j] = \max \begin{cases} H[i-1][j] - g, \\ F[i-1][j] - e \end{cases}$$

$$H[i][j] = \max \begin{cases} 0, \\ H[i-1][j-1] + s(a_i, b_j), \\ E[i][j], \\ F[i][j] \end{cases}$$

Here E and F are the horizontal and vertical gap matrices respectively, while H and $s(a_i,b_j)$ are the same as defined in the original formulation. Then, g and e are the gap opening and gap extension penalties as previously mentioned[10].

To keep the scope of this project reasonable, we will not be implementing the traceback logic. This is because the objective of this project is optimizations. Because the bulk of the algorithm's time and space complexity stems from the step of the algorithm where we fill the DP matrix, our time is best spent optimizing that. Therefore, for each alignment we only keep track of the highest scoring cell in the matrix and return that.

This is a reasonable design choice, as in many practical applications involving large-scale database searches biologists are primarily interested in the similarity scores between sequences, rather than the exact alignment of each character. The full alignment is only

[9] R. Sajjadinasab
[10] R. Sajjadinasab

examined for top matches in downstream analysis (which is much more computationally feasible).

We also assumed this implementation would only be used for protein alignment using one of the popular PAM or BLOSUM matrices. Later this assumption helps me limit the width, height, and bytes per element of the substitution matrix.

Arithmetic Intensity of our Implementation

The modifications we described to SW changes its arithmetic intensity (even if its asymptotic complexity remains the same).
Integer Operations:

    - 9 operations from the 3 integer max operations over four integers (zero, top, left, top-left) over 3 matrices (3*3=9),
    - 1 max operation to get the max score in the DP matrix so far,
    - 9 operations from the 3 integer additions (for adding the penalties) over 3 matrices.

- Memory Reads:
    - 2 reads for the current sequence characters in sequence a and b,
    - 1 read for the substitution matrix
    - 9 total reads for 3 cells from the 3 DP matrix (top, left, top-left) (3*3=9),
    - 3 writes for the current i,j cell in 3 matrices.

Putting this all together, we get an AI of 19/15 = 1.2667 for our implementation.

Code

The code for this project can be found at: https://github.com/Aseeef/seq-align-omp.

Asserting Correctness & Benchmarking

Correctness Validation

To validate correctness, we used an existing working implementation of Smith-Waterman[11]. This also served as the base for all of our code, as discussed later.

The original tool only supported one-to-one comparisons. To test against a full database, we wrapped the executable with a Python script that extracted sequences from a FASTA file and ran one-by-one comparisons against the query. We then ran our modified code and

[11] Turner

compared outputs. Any discrepancies were flagged immediately. This allowed us to confidently verify correctness throughout development.

Benchmark Methodology

The full program includes setup work like reading the database, loading substitution matrices, parsing arguments, loading batches, etc. These were excluded from benchmarking and mostly ignored in terms of optimization.

Benchmarks report only the time spent inside alignment_fill_matrices: the core DP matrix computation. This was a reasonable choice because, for example, profiling the entire program would allow things like kernel page caching to interfere with results. It would also make the impact of changes to the hot loop harder to isolate due to added noise from unrelated parts of the program.

Once again, our program is specialized for protein databases. For all tests and benchmarks, we use the UniProt Swiss-Prot database—one of the most widely used and biologically relevant protein datasets available.

Each benchmark was run 6 times, and both the mean and standard deviation were recorded. This is how we determined whether a parameter made a statistically significant impact on performance.

Finally, all reported benchmarks were compiled using GCC with the -O3 optimization flag unless noted otherwise.

Hardware Used

All tests were conducted on Intel(R) Xeon(R) Gold 6242 CPU @ 2.80GHz on BU's Shared Computing Cluster (SCC). Some important specifications of this CPU include:

- 16 Physical Cores, 32 Virtual Cores
- L1 Instruction Cache: 16 x 32 KB
- L1 Data Cache: 16 x 32 KB
- L2 Cache: 16 x 1024 KB
- L3 Cache: 22 MB

Serial Baseline

Repository Origin

Our code was built on top of https://github.com/noporpoise/seq-align, an 8-year-old general purpose implementation of Smith-Waterman with affine gap penalties and support for substitution matrices. It also includes Needleman-Wunsch for global alignment and can process FASTA files.

However, it does not support querying against a user-defined database. Instead, it either performs one-to-one comparisons or runs a batch similarity search that returns pairwise scores for all internal sequence pairs: neither of which suited our use case. Though it was possible to use this original version for correctness validation (via a Python wrapper) as mentioned earlier, for benchmarking, this would not be fair: the Python wrapper introduces significant overhead after all.

Moreover, while this foundational code base was solid, the inner loop that fills the DP matrix does extra work like wildcard substitution checks, per-character lowercasing, and other flexible but costly features that further add unnecessary overhead and adding them limits optimizations.

All in all, comparing this repository as-is would exaggerate our speedups given the specialized assumptions we are operating under. Therefore, the serial baseline had to be refactored.

Refactoring to Our Use-Case

Thus, to adapt to our use case, we refactored the code to accept a query file and a database file, enabling one-to-many comparisons (i.e., comparing a single query sequence against all sequences in a FASTA-formatted database). Modified usages are documented in the repositories README (link provided below).

We also removed everything not relevant to our use case:

- Needleman-Wunsch logic
- Backtracking code
- Configurable substitution scores with wildcard support
- Extra runtime arguments and general-purpose flexibility

The result was a cleaner, more focused baseline implementation tailored to our needs. This more focused implementation will also open room for further optimizations as will be discussed later.

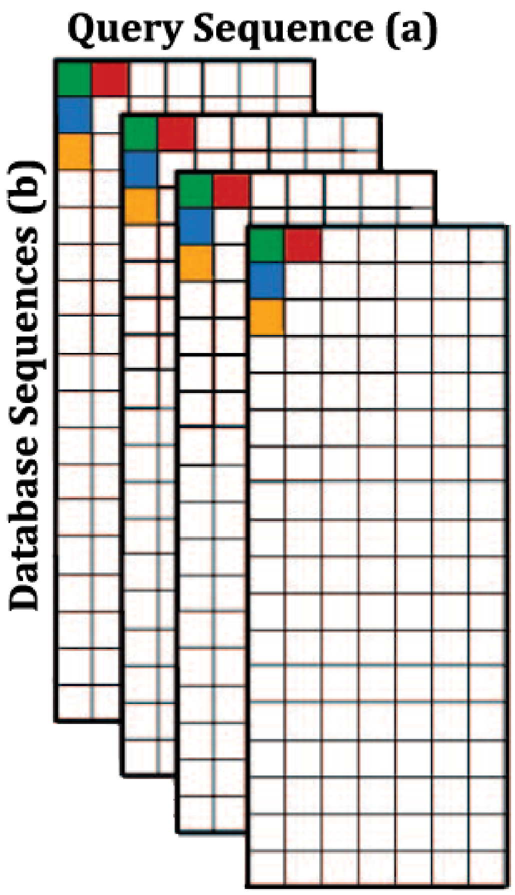
Benchmarks

In our benchmarks, this initial serial version took **27832ms** to run, with a standard deviation of 89ms across 6 runs.

This refactored, serial baseline lives on the serial branch of our repository: https://github.com/Aseeef/seq-align-omp/tree/serial.

AVX Vectorization

General Vectorization Strategy



The figures above illustrate our general vectorization strategy. The figure was adapted from R. Sajjadinasab's paper titled "Further Optimizations and Analysis of Smith-Waterman with

Vector Extensions"[12]. Each stacked matrix represents the dynamic programming (DP) matrix for aligning the same query sequence against different sequences from the database. This structure aligns with how AVX2 vectorization is applied: we vectorize across a batch of independent query-database comparisons.

Originally, the memory layout for each alignment was `dp_matrix[h][w]`, created fresh for each call to alignment_fill_matrices. The most natural extension would be to store multiple matrices as `dp_matrix[b][h][w]`, where b is the batch size. However, this layout is inefficient for vectorization—it would require gathering elements from memory by jumping h*w elements per lane, which is cache-unfriendly.

A more efficient layout is `dp_matrix[h][w][b]`, so that elements from the same position in each batch matrix are stored contiguously in memory. This allows for efficient loading of vector lanes with a single aligned memory read.

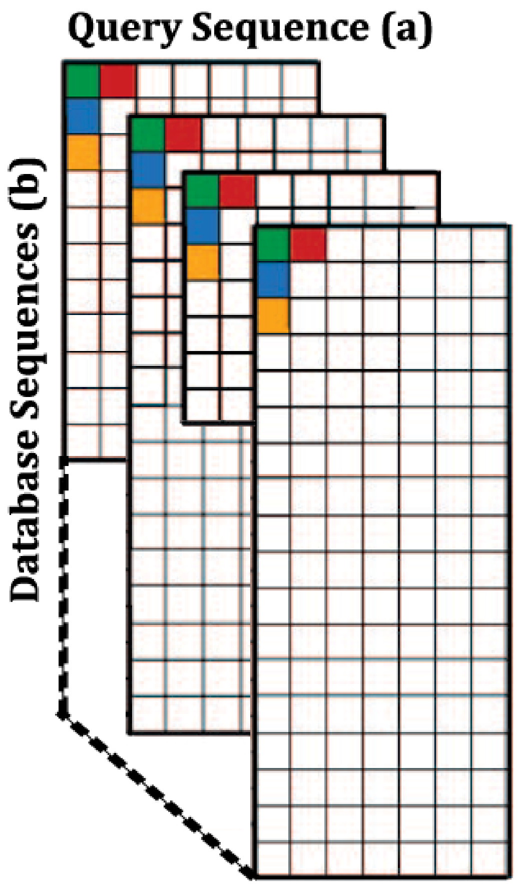
Challenges

Of course, reality is a bit more complicated.

- With this layout, values for the same query-database pair are now spaced b-elements apart, which makes pointer math harder to reason about and debug (since now if you wish to print out the sequences of matrices from the same sequence for debugging, you must jump across the batches). And indeed, this was a great challenge.
- Instead of 2D, we now perform 3D pointer arithmetic inside a performance-critical kernel—something that is notoriously easy to make mistakes in.
- Another challenge here is that sequences in the database are not all the same length. This means DP matrices will be of different sizes. Let us go into more detail here.

To make vectorization over a batch possible, each sequence in the batch must be padded so that all DP matrices in the batch are the same size (i.e. equal to the **largest** sequence). This is wasteful in both memory and computation. Shorter sequences must wait for the longest one to finish in that vector batch, and all padding must be carried through the loop logic. This conundrum is depicted in the figure below.

[12] R. Sajjadinasab



A simple and effective workaround is to sort the database by sequence length in descending order offline. Then, when forming vector batches, each batch will contain sequences of similar size, minimizing both padding overhead and wasted computation.

32-bit Integer Vectorization

AVX2 provides strong support for 32-bit integer operations, with well-optimized and fully documented instruction sets. Many of the operations we care about such as memory loads, max, and integer addition have near one-to-one mappings from serial code to AVX2 intrinsics. For example:

- `_mm256_load_si256` is used for aligned vector loads
- `_mm256_max_epi32` for computing max
- `_mm256_add_epi32` for computing penalties

The Challenge with the Substitution Matrix

Most of the scalar logic translated naturally to vector code with minimal effort. However, one major challenge was vectorizing the substitution matrix lookup, which in the scalar version looks like:

```
*score = scoring->swap_scores[(size_t) a][(size_t) b];
```

Here, `swap_scores` is defined as an `int32_t[255][255]` lookup table indexed by ASCII values of characters from sequences `a` and `b`. Each row `i` of `swap_scores` contains the scores of matching character a_i from the query and which row `j` the scores for batching character `b_j` from the database sequence. According to `perf` profiles, this was a hotspot, and vectorizing it properly was important for performance.

The issue is that each vector lane contains a different `b_index` (i.e. the so we can't just load a full row. We need to gather values from potentially different rows and columns. Here's how I handled it:

```
__m256i base = _mm256_set1_epi32(a_index * 32);
__m256i idx  = _mm256_add_epi32(base, _mm256_load_si256((__m256i
*) b_indexes));

int32_t * swap_scores = (int32_t *) scoring->swap_scores;
__m256i scores = _mm256_i32gather_epi32(swap_scores, idx, 4);
```

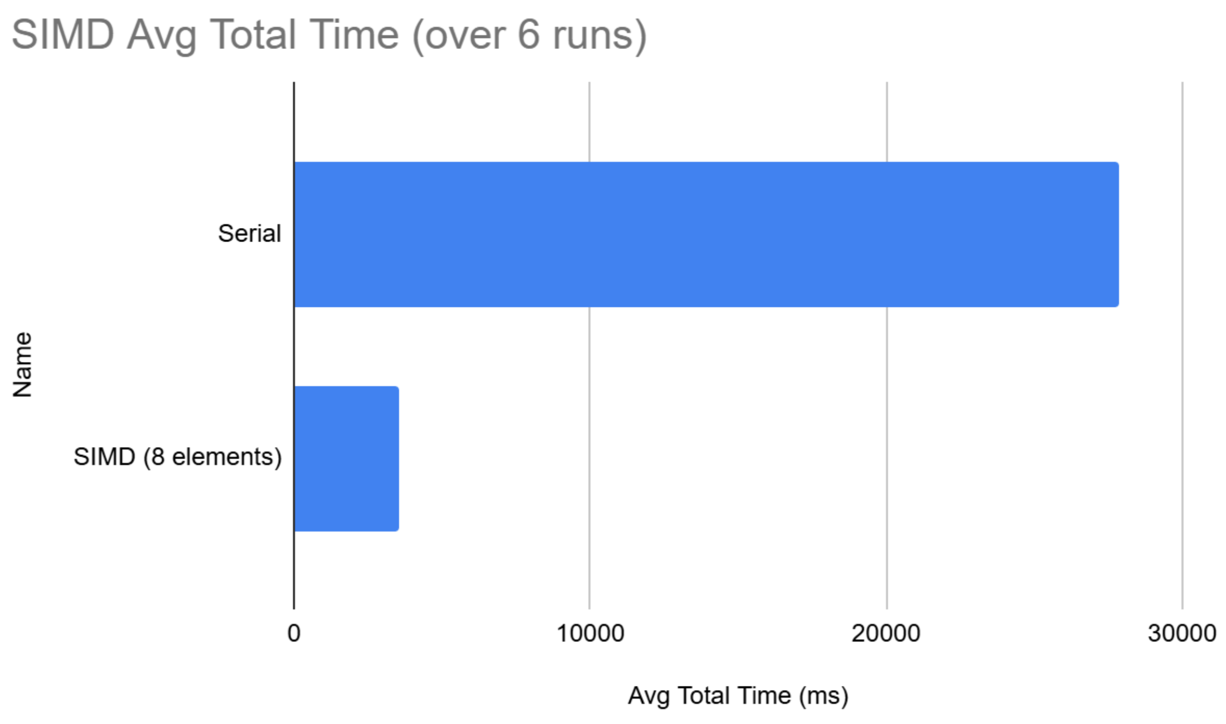
This works by:

- Computing the base offset for the current `a_index` row (`a_index * 32`, because the substitution matrix width is padded/aligned to 32),
- Loading the `b_indexes` into a vector,
- Adding the base offset to each lane,
- Using `_mm256_i32gather_epi32` to fetch the corresponding substitution penalties.

This allowed efficient vectorized substitution scoring with full use of AVX2 gather instructions, removing a major bottleneck.

Benchmarks

The benchmarks for the 32-bit AVX2 implementation were promising. Here, we see a leap from an average runtime of 27832ms (std=89ms) down to 3545ms (std=184ms). A **7.85x** speed up.



16-bit Integer Vectorization

Similar to the 32-bit implementation, many of the AVX2 instructions used earlier translate directly to 16-bit vectorization:

- `_mm256_load_si256` remains the same
- `_mm256_add_epi32` becomes `_mm256_add_epi16`
- `_mm256_max_epi32` becomes `_mm256_max_epi16`

These substitutions allowed most of the inner loop logic to be adapted with minimal effort.

The Challenge with the Substitution Matrix (Part 2)

The main complication again came from the substitution scoring lookup. AVX2 does not support 16-bit gathers. In the 32-bit version, this was handled using `_mm256_i32gather_epi32`, but no such equivalent exists for 16-bit integers.

One idea was to use two 32-bit gathers followed by a packing step:

```
const int32_t *swap_scores = scoring->swap_scores[a_index];
__m256i idx_low  = _mm256_loadu_si256((__m256i *)b_indexes);
__m256i idx_high = _mm256_loadu_si256((__m256i *)(b_indexes +
8));
__m256i scores_low  = _mm256_i32gather_epi32(swap_scores,
idx_low, 4);
__m256i scores_high = _mm256_i32gather_epi32(swap_scores,
idx_high, 4);
__m256i packed_interleaved = _mm256_packs_epi32(scores_low,
scores_high);
__m256i final_packed       =
_mm256_permute4x64_epi64(packed_interleaved, 0xD8);
```

The key issue here was that `_mm256_packs_epi32` doesn't merge the two input vectors sequentially, which initially led to incorrect results. A Stack Overflow post[13] helped clarify this behavior. Ultimately, Gemini 2.5 Pro (prompt citation included in citations) suggested applying `_mm256_permute4x64_epi64` after packing to reorder the values correctly. This resolved the issue giving me a working 16-bit SIMD implementation.

Caching Improvements and De-vectorizing

To further assist the gather operation, I replaced the original `int32_t[255][255]` scoring table with `int32_t[32][32]`, since there are only 20 amino acids, plus 4 special characters (e.g., X, *) for unknowns or padding. This significantly improved cache locality. (Note: For fairness, the same lookup optimized table was used in the 32-bit version as well for benchmarking comparison.)

Nevertheless, despite the improvements, `perf` still showed the scoring gather logic as a major bottleneck. The overhead from using two gathers plus packing and shuffling outweighed the benefits as we observed very little uplift over the 32-bit AVX2 implementation.

Consequently, we attempt one more strategy which ends up as our final solution: to scrap the AVX2 implementation of the substitution matrix all together in the 16-bit implementation. Previously, to support AVX2's gather instructions, the substitution matrix had to be defined as `int32_t[32][32]`, since `_mm256_i32gather_epi32` only

[13] https://stackoverflow.com/questions/67979078/mm256-packs-epi32-except-pack-sequentially

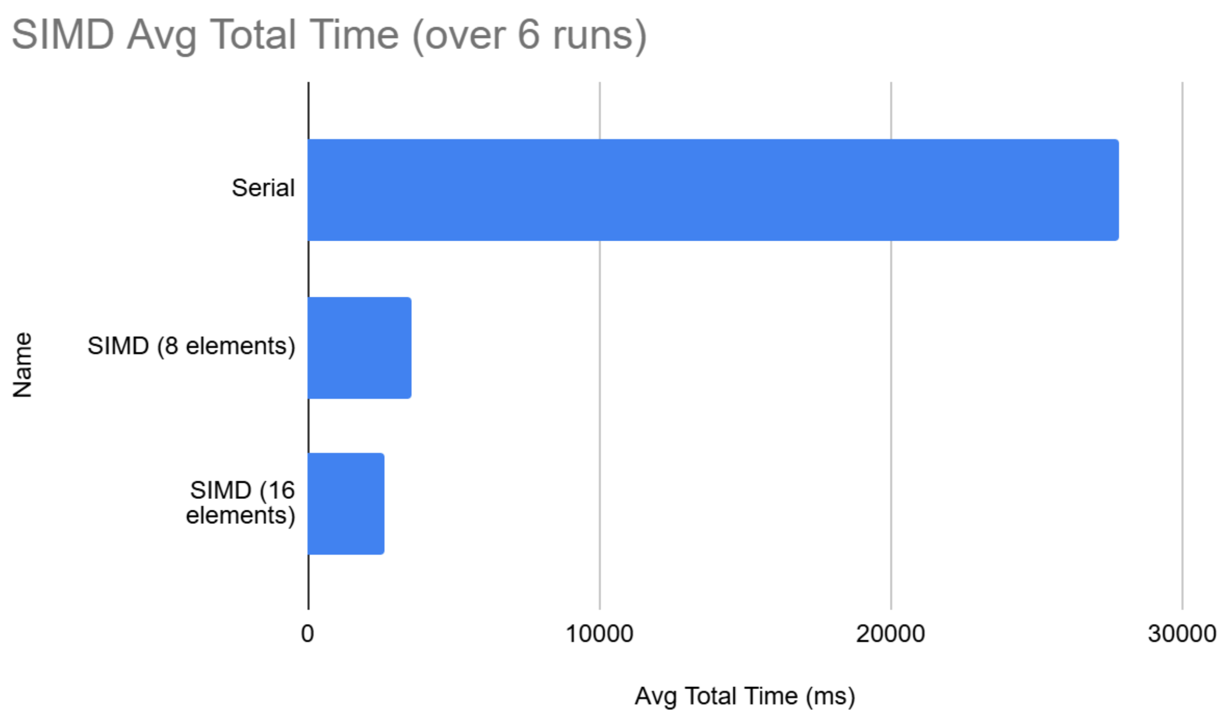
operates on 32-bit integers. But once I dropped the AVX gather approach, I was free to use `int8_t`. This was a reasonable choice since common substitution matrices like PAM and BLOSUM never contain values exceeding 127. While “de-vectorizing” sounds like a step backward, here it proved to be faster than the vectorized gather for the 16-bit AVX implementation as previously shown. Since each row was now only 32 bytes, we got excellent memory performance (in terms of cache and memory bandwidth saturation). After all, `int8_t[32][32]` fits well inside the L1-cache (with a memory requirement of just 1KB). Perhaps surprisingly (or not), it seems the cache benefits of the serial version with its lower instructions (even if they weren’t vectorized) proved to have superior real-world performance. The final code for the scoring lookup is shown below:

```
const int8_t *swap_scores = scoring->swap_scores[a_index];
alignas(32) int16_t indexes[16];
for (int32_t i = 0; i < 16; i++) {
    indexes[i] = (int16_t) swap_scores[b_indexes[i]];
}
return _mm256_load_si256((__m256i *) indexes);
```

Benchmarks

While benchmarks showed notable improvements over the 32-bit AVX2 implementation, the benefits of 32-bit AVX -> 16-bit AVX weren’t as drastic as the serial -> 32-bit AVX.

Particularly, I only got a **10.44x** speedup from the serial (as opposed to the x16 theoretically possible). While the 32-bit AVX version as previously mentioned took 3545ms (std=184ms) over 6 runs, this new 16-bit ran in 2666ms (std=25ms).



Memory Optimizations

SW with Linear-Space Scoring

By default, Smith-Waterman requires a full 2D matrix per sequence comparison to compute alignment scores. In our 16-bit AVX2 implementation, each vector batch processes 16 database sequences at once.

Protein sequence lengths can be large. The largest known protein (titin) exceeds 35,000 residues. Even setting aside such outliers, sequences with lengths around 1,000 are not uncommon[14]. Since then it makes sense for our program must support large inputs, we consider the following realistic worst case:

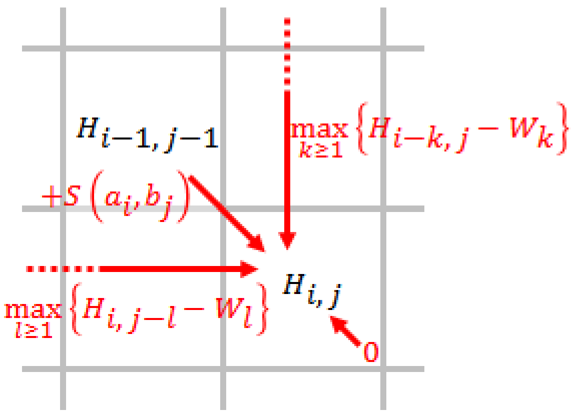
```
1000 (query length) × 1000 (db sequence length) × 16 (batch
size) × 2 bytes (element size) = 32 MB per batch
```

Later in the report, we'll see that parallelization benefits from launching batches of batches. Suppose we use ~100 batches per thread (which by the way is still too little to fully

[14] Zhang

amortise the cost of multithreading) and 32 threads, this scales to over **3 GB** of memory just for the scoring matrices. This is clearly unsustainable.

Double Row Implementation


$H_{i-1,j-1}$
$\max_{k\geq 1}\{H_{i-k,j}-W_k\}$
$+s(a_i,b_j)$
$H_{i,j}$
$\max_{l\geq 1}\{H_{i,j-l}-W_l\}$
0

The figure above shows how the score for each DP cell is computed (borrowed from Wikipedia; notation differs slightly, but the logic is equivalent to the SW algorithm previously described). Notice that the value at `H[i][j]` only depends on the top, left, and top-left neighbors.

We thus observe that each cell depends only on:

- the cell directly above,
- the cell to the left,
- and the top-left diagonal cell.

This implies that we don't need the full matrix in memory at once—we only need two rows: the current row being computed and the previous row. We process left-to-right in the inner loop, and top-to-bottom in the outer loop. At the end of each row, we simply swap the pointers: the current row becomes the previous row, and vice versa. This reduces memory usage by a factor of `h`, the number of rows, and provides significant performance gains as shown in the benchmarks section below. This is the approach mentioned in Chapter 3 of *Blast an Essential Guide to the Basic Local Alignment Search Tool* by Ian Korf[15].

Single-Row Implementation

However, this idea can in fact be taken further. Instead of two rows, we can use a single row. The trick is to treat the old value at each cell (before overwriting it) as the value from the

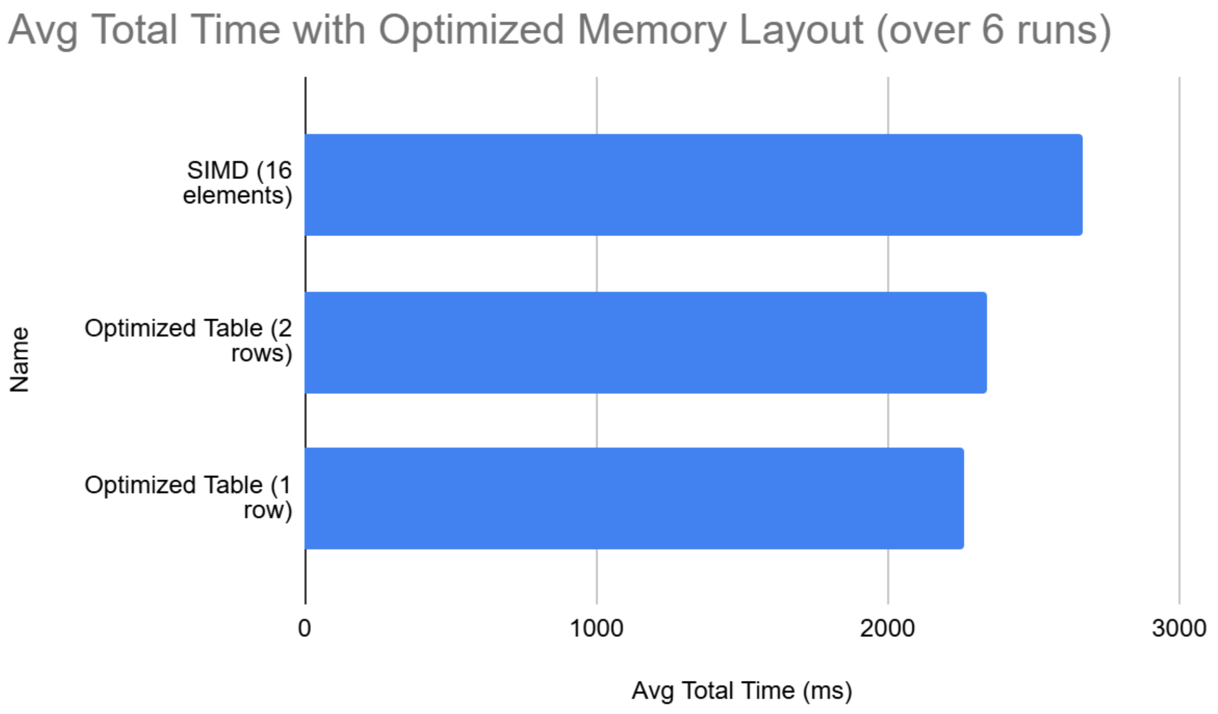
[15] Korf

"row above." This can be done by holding it in a temporary register before writing the new value. This method increases register pressure, but in practice we observed a noticeable performance gain, likely due to reduced memory bandwidth pressure and better temporal locality.

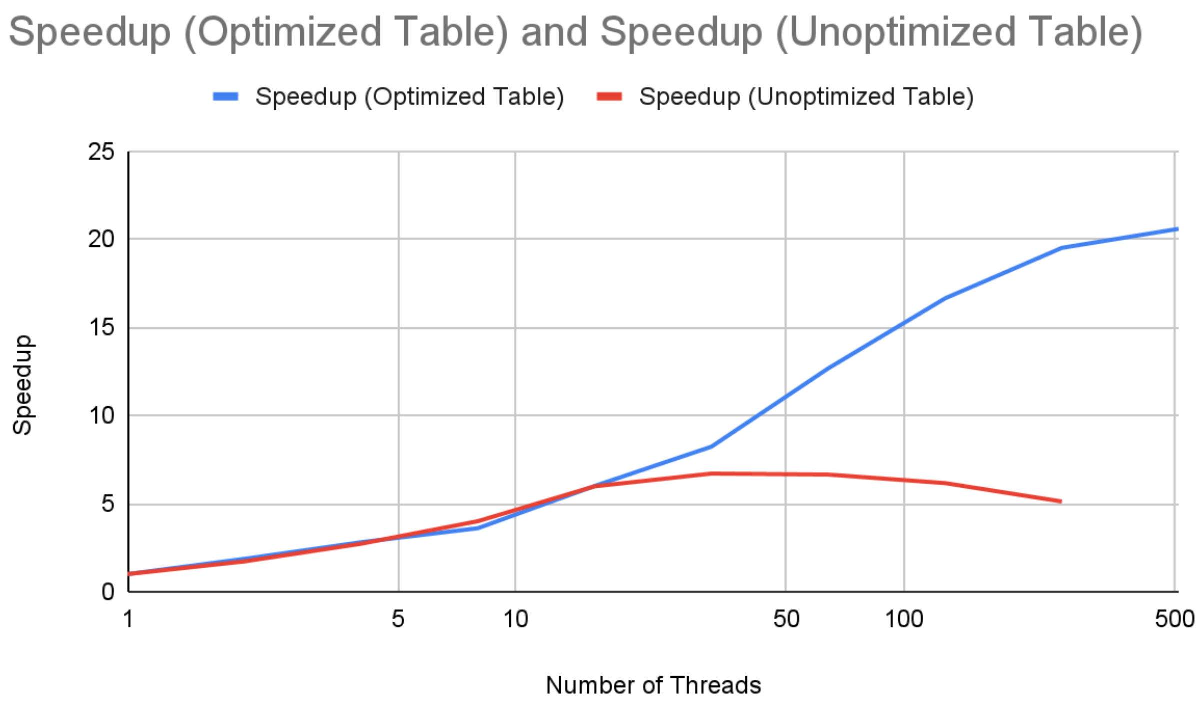
Benchmarks

The benchmarks below show the improvements over the 16-bit AVX implementation. These benchmarks highlight that not only do the memory optimization make our problem more tractable in space complexity, but also significantly improve performance.

The 16-bit AVX version achieved a mean runtime of 2666ms (std = 25ms), the 2-row version improved this to 2343ms (std = 10ms), and the 1-row version further reduced it to 2263ms (std = 31ms). These correspond to speedups of **10.44×, 11.88×, and 12.30× over the serial baseline**, respectively. Compared to the 16-bit AVX version, the 1-row version corresponds to a **1.18x speedup**.



Another notable advantage of the Optimized 1-row table, and the Unoptimized 2D table is scaling. Notice, how when we add more threads, compared to the original 2D table version, the optimized 1-row version scales more linearly (and in fact, in the later section on Parallelization we get it to scale perfectly linearly by tuning the batch size). The exact parallelization strategy used is discussed in a later section.



Prefetching

There's one optimization we have been carrying throughout the code that hasn't been discussed yet: prefetching.

During early development, we observed that manually prefetching memory significantly improved performance. So it was added and has been kept in since relatively early. However, upon revisiting the impact of prefetching under proper benchmarking conditions, we found that any prefetching actually degraded performance.

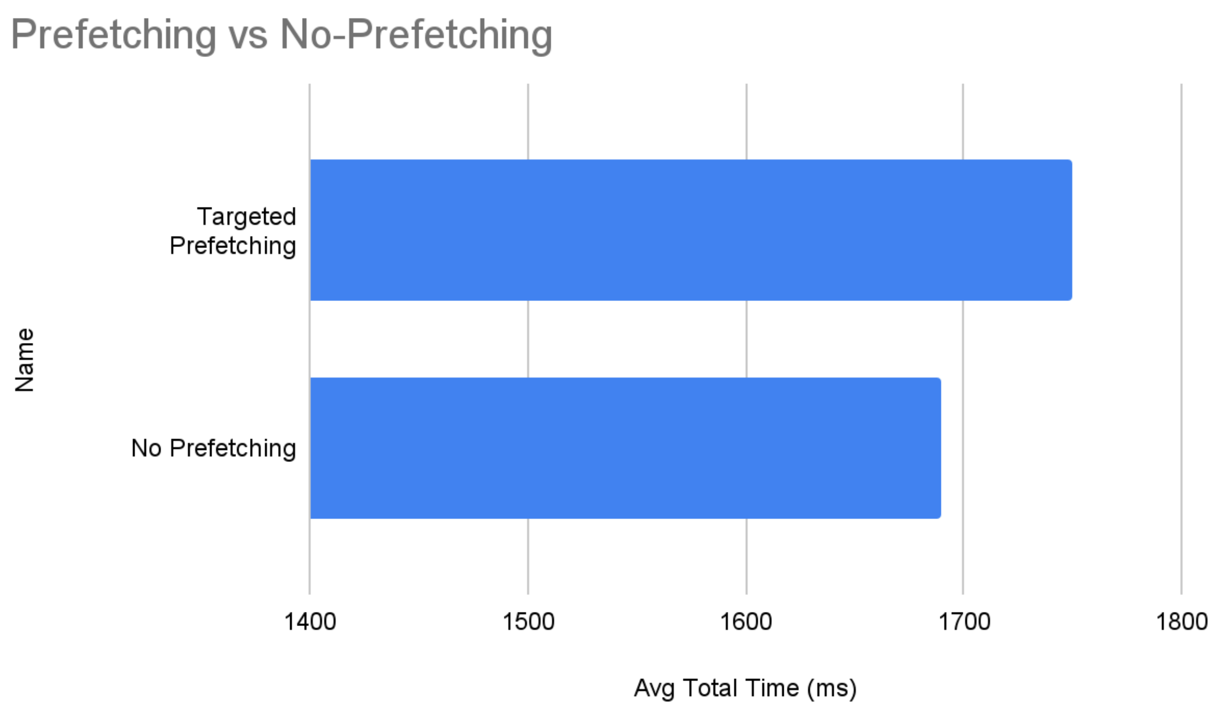
Looking back, we believe the initial perceived benefit came from testing under `-O0` compiler optimization flags. In that context, prefetching may have helped compensate for otherwise unoptimized memory access. But once the compiler was allowed to optimize (e.g., `-O3`), its scheduling and instruction reordering became far more effective than anything we were doing manually.

Interestingly, after inspecting the assembly output, we saw no explicit prefetch instructions inserted by the compiler. In any case, by the final implementation, all manual prefetching was removed, and performance was better for it.

Benchmarks

The benchmarks below show the impact of prefetching. With manual, targeted prefetching, the runtime was 1750 (std = 5ms), while no prefetching at all resulted in a faster runtime of 1690ms (std = 21ms). This means the version without any prefetching is approximately 1.057x faster than with prefetching. This is despite the fact that our prefetching logic was targeted based on memory bottlenecks reported by `perf`. We also attempted to vary the number of iterations ahead to prefetch but even that did not help.

The conclusion is clear: any manual prefetching degrades performance in this context.



Note: Compared to the runtimes in the previous section, the absolute numbers here are lower. This is because additional optimizations (not yet discussed) were implemented between the previous benchmark and this one.

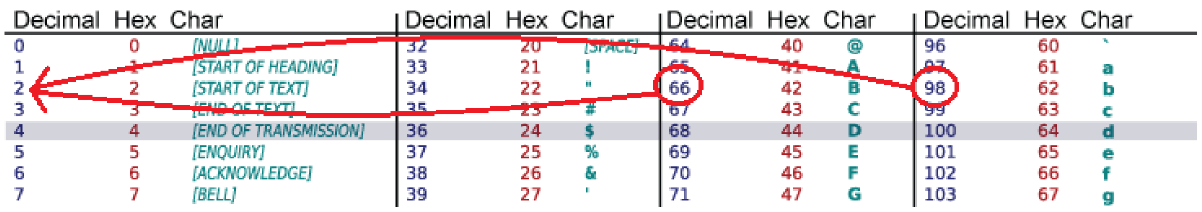
Substitution Matrix Layout Improvement

We also experimented with reordering the indices used in the substitution matrix to explore potential gains in cache locality.

Currently, character indices are translated to fall within a compact 0–31 range using simple arithmetic on ASCII values:

```
int8_t letters_to_index(char c) {
    if (c >= 97 && c < 123) {
        return c - 96;
    } else if (c >= 65 && c < 91) {
        return c - 64;
    } else if (c == 42) {
        return 31;
    } else {
        printf("Error: %c is not a legal character for the
substitution matrix!\n", c);
        exit(1);
    }
}
```

This keeps the substitution matrix small and fits all relevant amino acid symbols into a 32×32 table. The mapping is simple and easy to work with as shown in the figure below. But the question we wanted to explore was: what if this layout isn't optimal for cache performance?



Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g

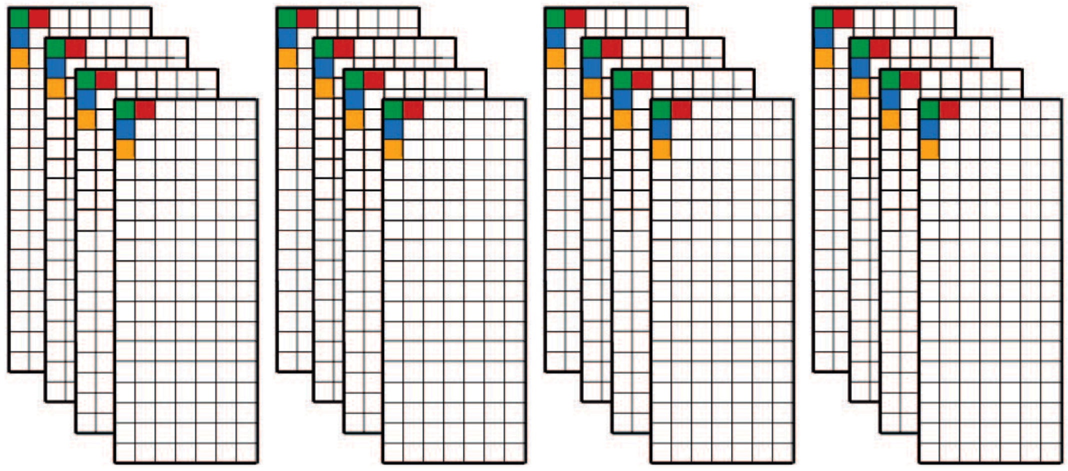
To test this, we tried reordering the character indices based on amino acid abundance. For example, Leucine (L) is the most frequent amino acid (~9.68%), followed by Alanine (A), Serine (S), etc. The idea was that if frequently occurring amino acids are placed next to each other in the substitution matrix, then the probability of cache hits would increase since a single cache line can load two adjacent rows from the matrix. For example, if we try to access row L (which we frequently would since it's the most common amino acid), I would also load row A (the second most common amino acid). Therefore, there is a good chance that A would be retrieved soon.

However, in practice, this made no measurable difference. Even though the reasoning was sound, it's likely that the substitution matrix is already small enough to stay resident in cache (as we pointed out previously).

So while conceptually interesting, this optimization was ultimately not beneficial for this implementation.

Parallelization with OpenMP

Basic Strategy



Up to this point, we've focused on vectorizing the alignment process by processing multiple database sequences in parallel using AVX across a single vector batch.

Parallelization with OpenMP is a natural extension of this idea: we now parallelize across multiple batches of vectorized work: essentially a "batch of a batch".

The figure above illustrates this approach. Each thread is assigned a separate batch of batches, allowing independent processing with minimal synchronization overhead. Since all batches are independent, as we shall see this strategy scales well with thread count.

In our implementation, we synchronize at the end of each batch and load the new batches. As previously mentioned, we do not include the time spent loading these batches in our benchmarks. Optimizing this batch loading was considered outside the scope of this project however, this batch loading processes could in theory also be done in parallel.

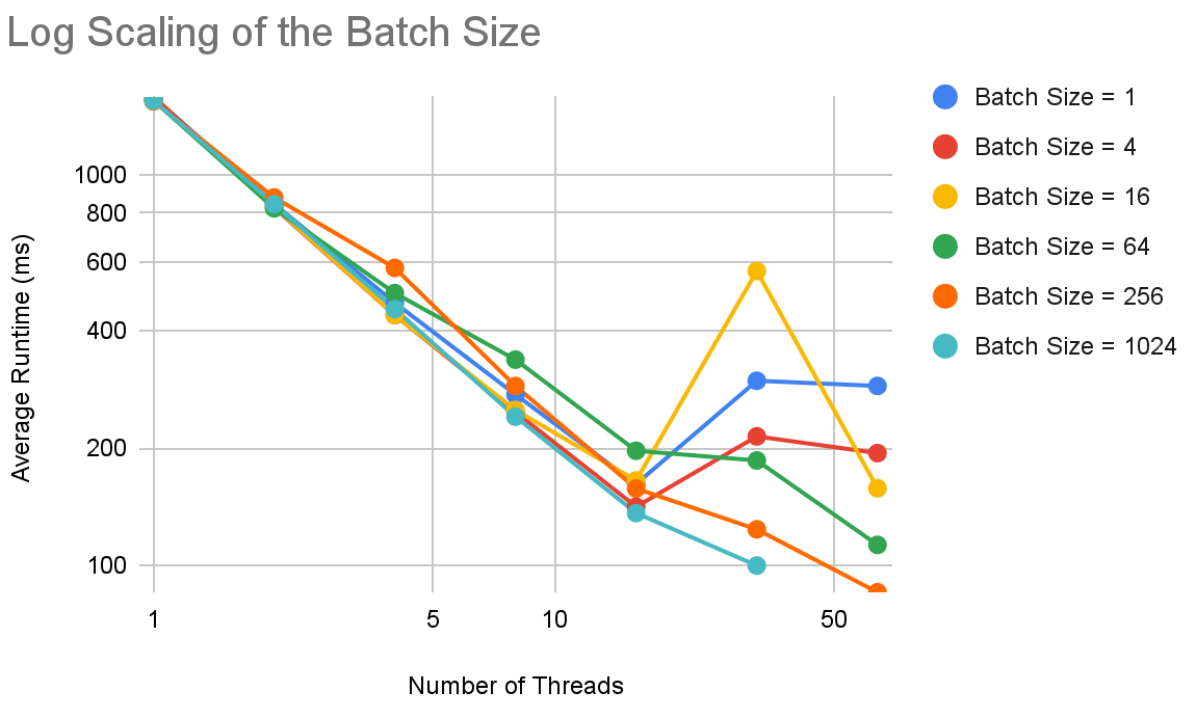
Batch Sizes

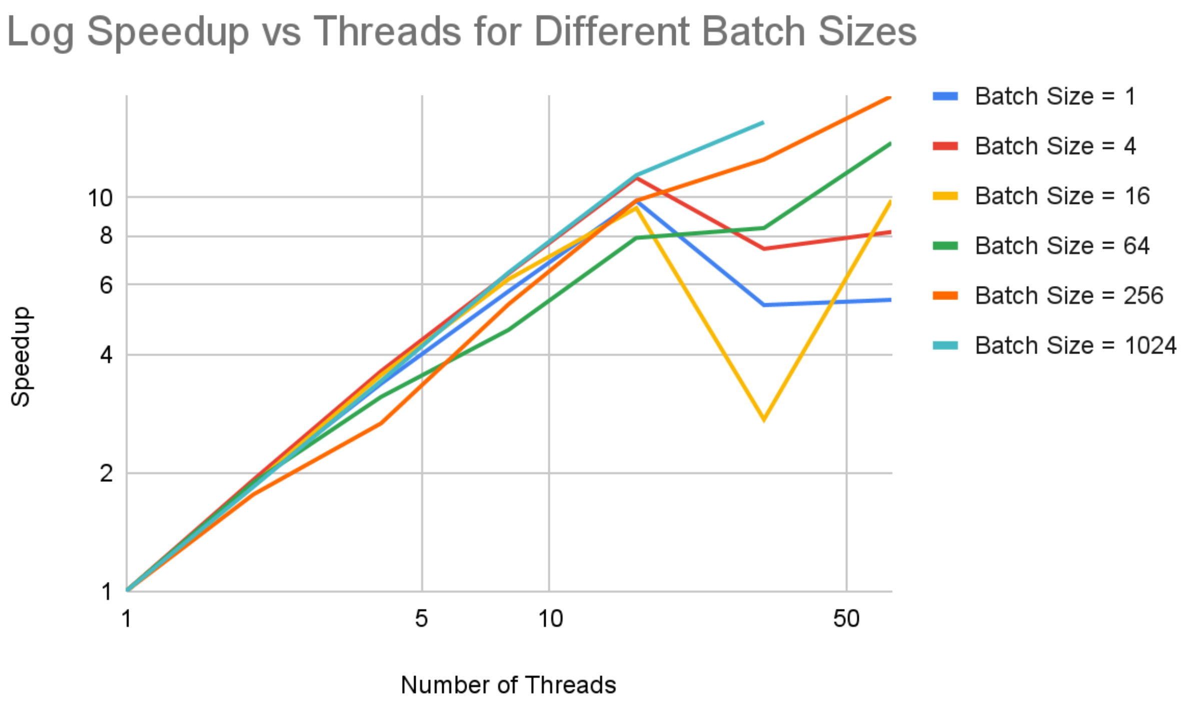
Batch size ends up being an important parameter. It refers to how many batches of vectorized work each thread processes. Larger batch sizes help amortize the overhead of OpenMP. So in general, the larger the batch, the better the performance.

However, larger batches also increase memory usage. In our experiments, we kept increasing the batch size and observed consistent performance improvements (more or less) from larger batch sizes.

Benchmarks

The below benchmarks show the impact of batch size on the scalability of multithreading with OpenMP as discussed. Notice, in some of these benchmarks, we continue to observe a speedup even after surpassing the 32 virtual threads provided by the hardware. This should not happen if each thread is being utilized fully. Perhaps this is why I observed that with a batch size of 1024, the program scales the best (most linearly) as threads are added since we are able to better utilize each thread due to the larger batch. Unfortunately after the batch size exceeds 1024, we begin having memory limitations which is why our best configuration ends up being 64 threads with a batch size of 256. Once again, all data points here were averaged over 6 runs.





Note: While the best result in this benchmark ends up being 64 threads with a batch size of 256, in a later test, we found that a batch size of 512 with 64 threads performs even better.

Dynamic or Static?

When parallelizing with OpenMP, we experimented with two scheduling strategies:

```
#pragma omp parallel for schedule(dynamic, 1)
for (i = 0; i < batch_cnt; i++) {
    alignment_fill_matrices(aligners[i]);
}
```

vs

```
#pragma omp parallel for schedule(static, 1)
for (i = 0; i < batch_cnt; i++) {
    alignment_fill_matrices(aligners[i]);
}
```

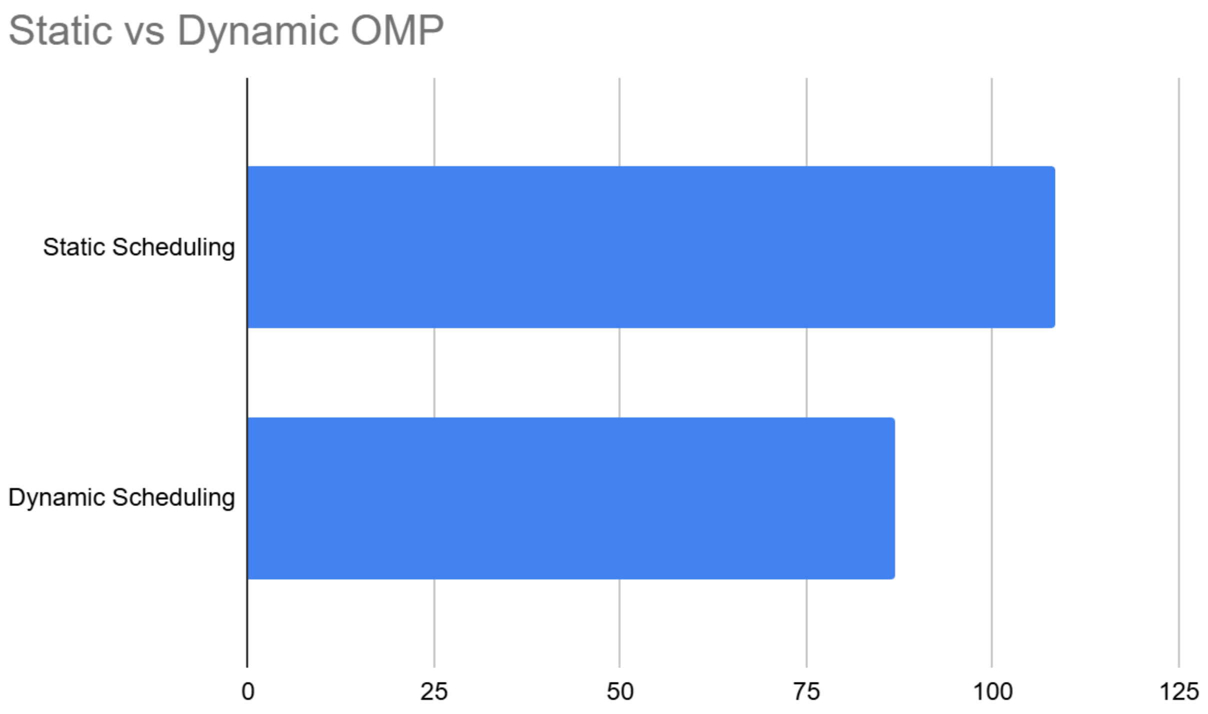
With `static`, work is pre-assigned to threads in a round-robin fashion. It has less overhead but assumes the workload is evenly distributed. `dynamic`, on the other hand, assigns work to whichever thread is available, at the cost of slightly more scheduling overhead[16].

At first, I expected `static` to perform better since the database was sorted by sequence length, so I expected work across the batch to be relatively balanced since each sequence is close to the same length. However, in practice, I observed that `dynamic` scheduling performed slightly better. This suggests that the actual work per batch is more imbalanced than I originally thought.

Benchmarks

The test below was run with 64 threads using a batch size of 256.

With static scheduling, the average runtime was 108ms (std = 4ms), while dynamic scheduling achieved 87ms (std = 5ms). This means dynamic scheduling was 1.24× faster than static in this setup!



[16] Liu

Other Optimizations

Loop Unrolling

We attempted manual loop unrolling on this critical part of the substitution matrix lookup logic:

```
for (int32_t i = 0; i < 16; i++) {
    indexes[i] = (int16_t) swap_scores[b_indexes[i]];
}
```

This is the same loop discussed earlier in the Caching Improvements and De-vectorizing section. Unrolling this loop had no impact on performance. This is likely because the compiler already unrolls such simple, fixed-size loops automatically under optimization flags like `-O3`.

Code Motion

In my transition from the serial to AVX implementation, I performed a number of code motion optimizations by moving pointer dereferences out of the tight loop in `alignment_fill_matrices`. I didn't profile each one individually at the time.

Much later into development, I realized I had missed one important case—because I assumed the compiler would handle it:

```
__m256i substitution_penalty = scoring_lookup(scoring,
aligner->seq_a_indexes[seq_i], aligner->seq_b_batch_indexes +
(seq_j * FULL_VECTOR_SIZE));
```

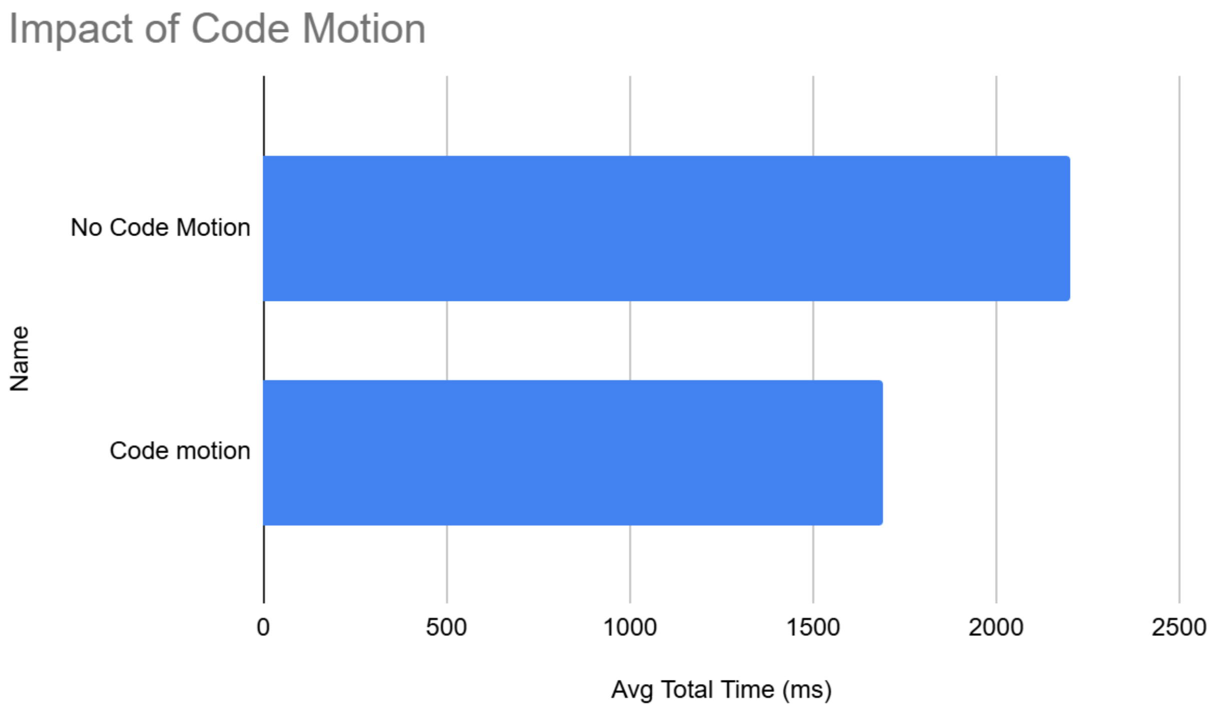
I moved both `aligner->seq_a_indexes` and `aligner->seq_b_batch_indexes` into local variables before the loop, and to my surprise, this resulted in significant performance gains.

This was unexpected because struct fields are stored contiguously in memory, so I didn't think dereferencing them would be expensive. Even more surprising: just two instances of code motion made that much of a difference.

I had seen this line show up in `perf` profiles for cache misses before, but I had mistakenly assumed the cost was from `aligner->seq_a_indexes[seq_i]` alone. In reality, it was the repeated dereferencing of both fields. This showed that even small, predictable pointer dereferences can have a non-trivial performance impact, and that manual code motion in hot loops is still worth doing.

Benchmarks

The benchmark below shows a whopping **1.303x speedup** from this simple change using one thread. We saw the average run time drop from 2203ms (std = 11ms) without code motion to 1690ms (std = 21ms) with code motion. This benchmark was conducted using a single OpenMP thread.



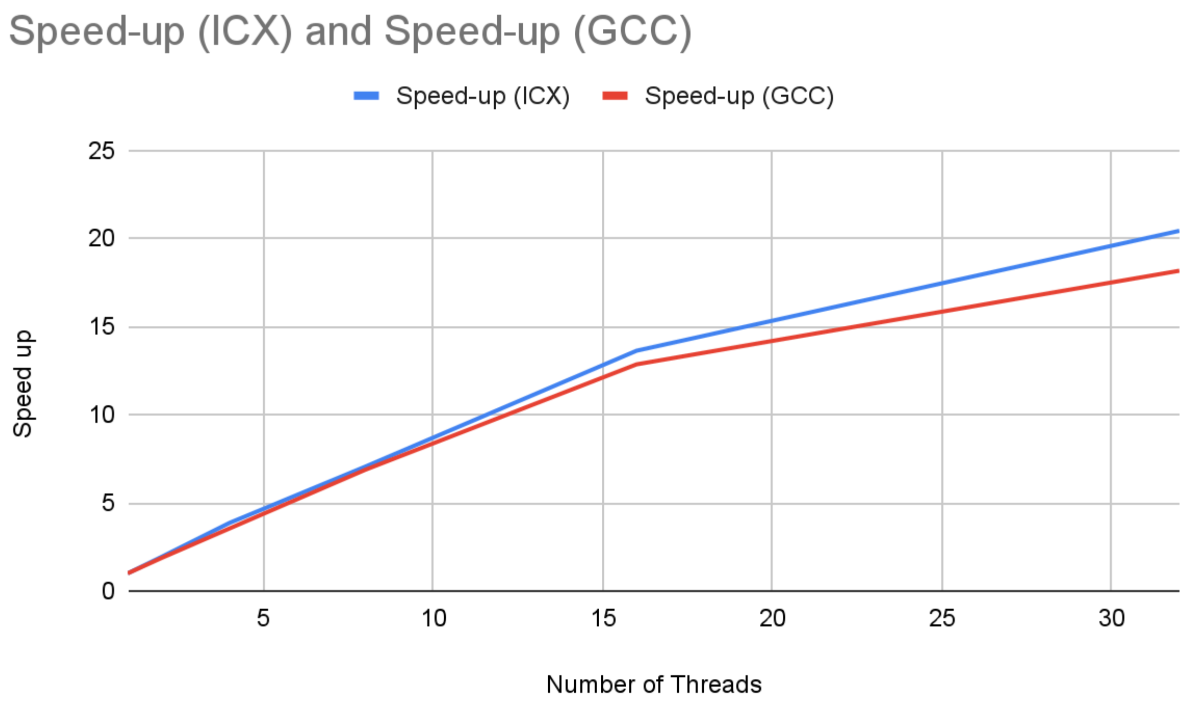
GCC vs ICX

We also experimented with using Intel's ICX (Intel's C compiler) to see if we would get better results. Intel's C compiler is generally "supposed to" perform better on Intel machines by up to 18% [17]. Since the SCC computer used for all benchmarks runs an Intel CPU as previously described, this was worth the time however the results were mixed.

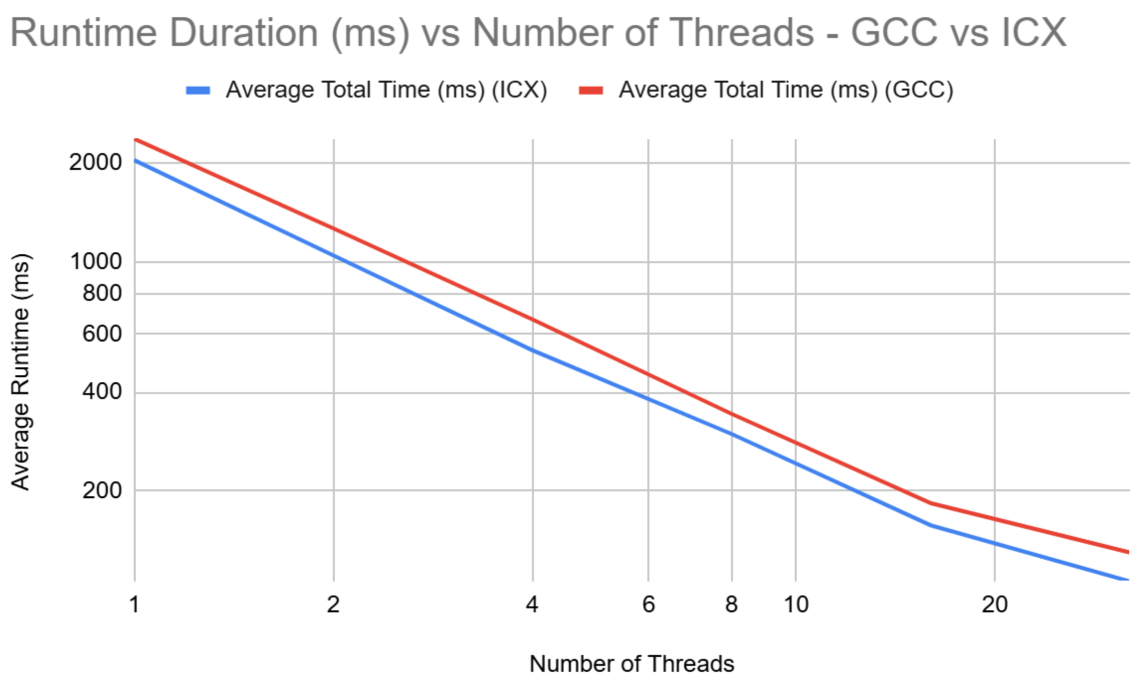
[17] Reinders

Benchmarks

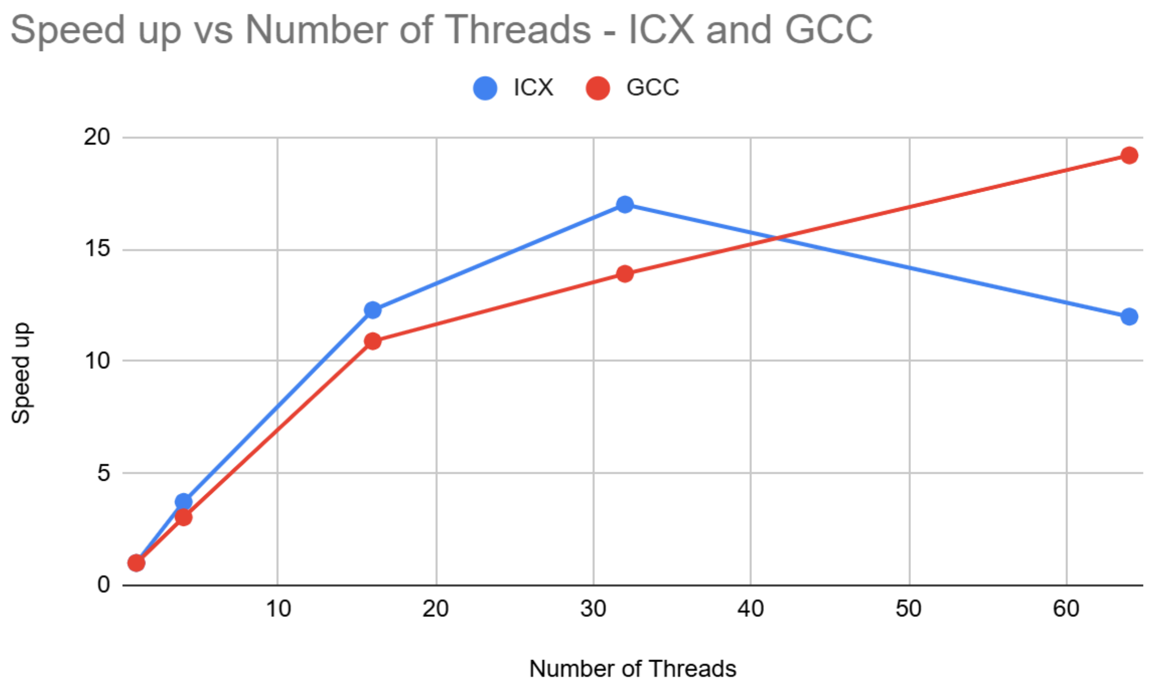
While earlier, before we had implemented all optimizations described above, comparing ICX and GCC delivered promising results. The below benchmarks were recorded with a batch size of 1024 showing a clear lead for ICX.



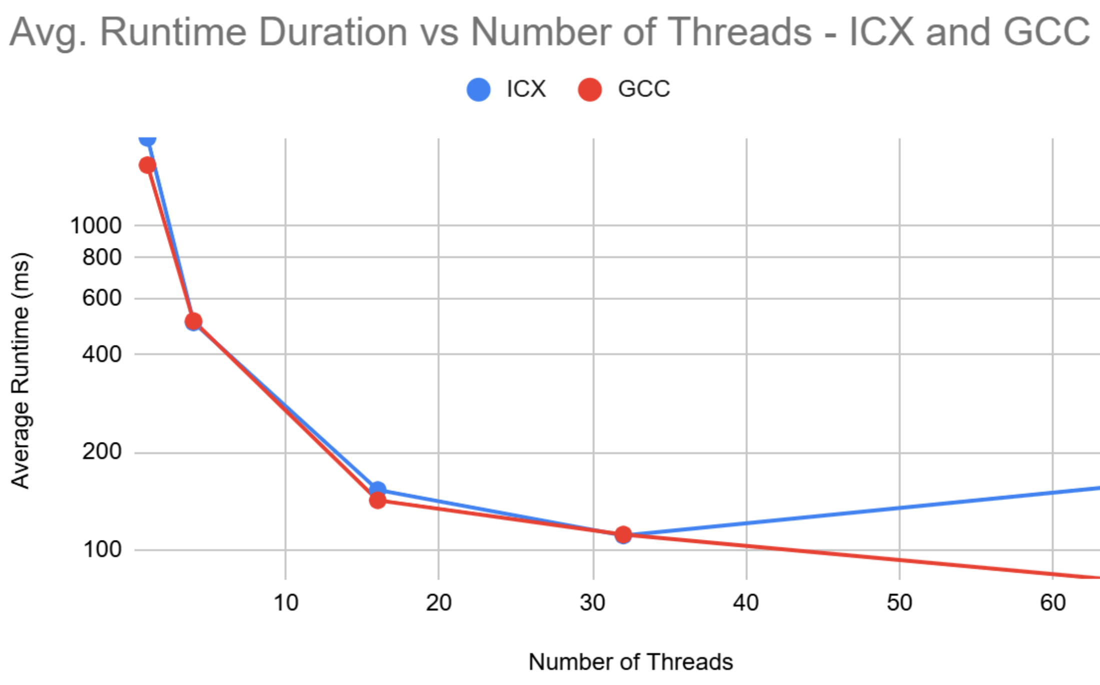
Note: The speed-up reported here is relative to itself and not the serial baseline (yet).



Interestingly however, by the time we finalized our repository with every single optimization we could think of, GCC performed consistently better regardless of the batch size used. The particular benchmark reported below used a batch size of 512, however, other batch sizes showed similar results.



Note: The speed-up reported here is relative to itself and not the serial baseline (yet).



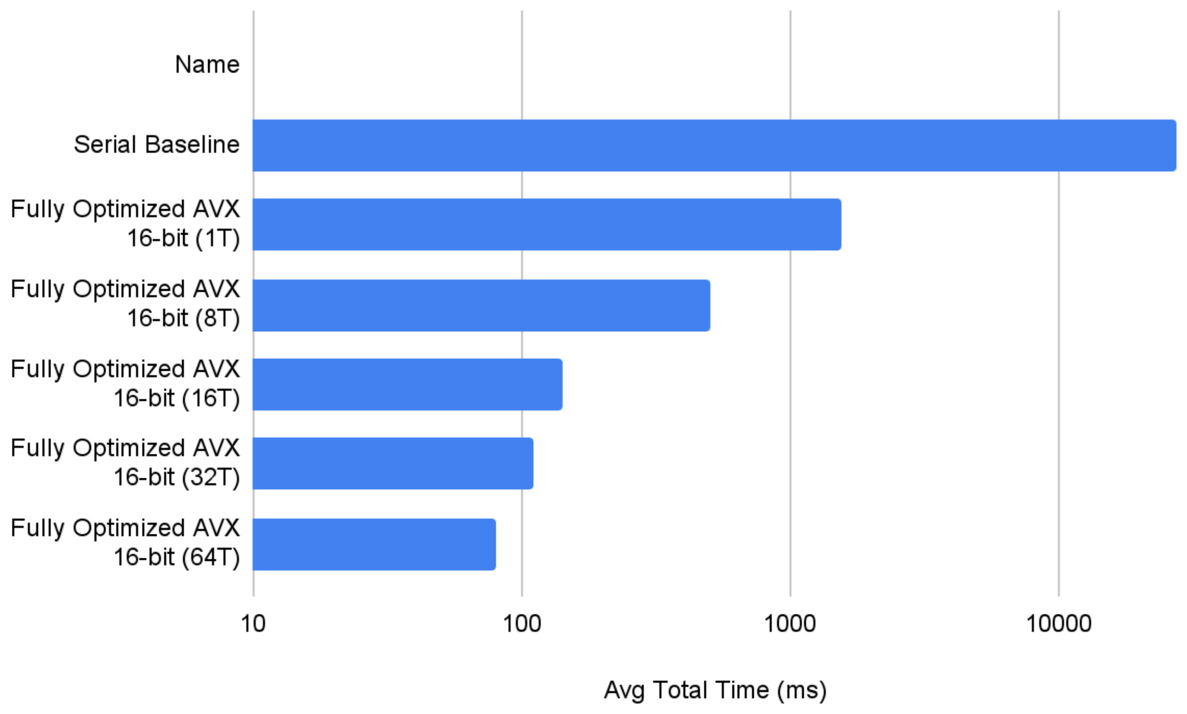
We also consistently saw that for code compiled by ICX, performance always dropped off as we went above 32 threads (the SCC computer has 32 virtual threads). We also tried compiling the unoptimized serial baseline to see if it would perform better when compiled by ICX and here as well we observed a performance degradation. We conclude that at least in our case, whether a different compiler will improve or degrade performance depends largely on the program itself and it is worth trying different compilers for each project individually.

Conclusion

We believe this project was a resounding success.

After weeks of profiling, debugging, and squeezing out every drop of performance, we reached a point where we can't think of a single meaningful optimization left undone. Despite the hurdles—vectorization edge cases, cache pitfalls, compiler quirks—we got Smith-Waterman, a traditionally memory-bound algorithm, to **scale linearly** across threads and run orders of magnitude faster than the baseline.

In case the reader got lost among all the transformations, here's the final performance tally:



Name	**Runtime (ms)**	**Std (ms)**	**Speedup Over Serial**	**Elements per Second**
Serial Baseline	27832	89	1.00×	20309
Fully Optimized AVX 16-bit (1T)	1547	1	17.99×	365,383
Fully Optimized AVX 16-bit (8T)	509	25	54.68×	1,110,505
Fully Optimized AVX 16-bit (16T)	142	3	196.00×	3,980,613
Fully Optimized AVX 16-bit (32T)	111	2	250.73×	5,092,315
Fully Optimized AVX 16-bit (64T)	81	4	343.60×	6,978,358

The “fully optimized” label reflects the best configuration we found, incorporating AVX vectorization, memory layout tuning, loop code motion, OpenMP tuning, and cache-aware scoring.

Notice, even on a single thread, we are beating the serial baseline by a whopping 18x! The 64-thread version represents the absolute maximum performance we were able to squeeze out of the Intel(R) Xeon(R) Gold 6242 CPU these benchmarks were running on. And at 81ms, it’s **over 343× faster** than where we began!

For our benchmarks, we were using the Uniprot Swis-Prot as previously reported. This database had 565,247 records at the time it was downloaded. So using this information, to put this all into perspective, we report how many database records our algorithm can process in a second to highlight its potential usefulness for large-scale database queries.

All in all: we took a classic algorithm, pushed it to its limits, and made it fly!

Citations

"AI Studio Prompt." *Google AI Studio*, https://aistudio.google.com/app/prompts?state=%7B%22ids%22:%5B%221-Vbs7o7xRHvi1D2gWGdpiWL0vUtobbdp%22%5D,%22action%22:%22open%22,%22userId%22:%22112153521177605316268%22,%22resourceKeys%22:%7B%7D%7D&usp=sharing. Accessed 26 April 2025.

"Amino Acid." *Wikipedia: The Free Encyclopedia*, https://en.wikipedia.org/wiki/Amino_acid. Accessed 4 May 2025.

"BLOSUM." *Wikipedia: The Free Encyclopedia*, https://en.wikipedia.org/wiki/BLOSUM. Accessed 20 April 2025.

bumpbump. "_mm256_packs_epi32, Except Pack Sequentially." *Stack Overflow*, 15 June 2021, https://stackoverflow.com/questions/67979078/mm256-packs-epi32-except-pack-sequentially. Accessed 26 April 2025.

Korf, Ian, et al. *Blast an Essential Guide to the Basic Local Alignment Search Tool Ian Korf, Mark Yandell, and Joseph Bedell. Ed.: Lorrie Lejeune*. O’Reilly, 2003.

Liu, Yiling. "OpenMP – Scheduling (Static, Dynamic, Guided, Runtime, Auto)." Yiling's Tech Zone, 15 July 2020, https://610yilingliu.github.io/2020/07/15/ScheduleinOpenMP/. Accessed 30 April 2025.

"Point Accepted Mutation." *Wikipedia: The Free Encyclopedia*, https://en.wikipedia.org/wiki/Point_accepted_mutation. Accessed 20 April 2025.

R. Sajjadinasab, H. Rastaghi, H. Shahzad, S. Arora, U. Drepper and M. Herbordt, "Further Optimizations and Analysis of Smith-Waterman with Vector Extensions," 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), San Francisco, CA, USA, 2024, pp. 561-570, doi: 10.1109/IPDPSW63119.2024.00113.

Reinders, James. "Intel® C/C++ Compilers Complete Adoption of LLVM." *Intel*, 9 Aug. 2021, https://www.intel.com/content/www/us/en/developer/articles/technical/adoption-of-llvm-complete-icx.html. Accessed 5 May 2025.

"Smith–Waterman Algorithm." *Wikipedia: The Free Encyclopedia*, https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm. Accessed 20 April 2025.

Turner, Isaac. seq-align: Fast, Portable C Implementations of Needleman-Wunsch and Smith-Waterman Sequence Alignment. GitHub, 18 Aug. 2015, https://github.com/noporpoise/seq-align. Accessed 20 April 2025.

Zhang, Jianzhi. "Protein-Length Distributions for the Three Domains of Life." *Trends in Genetics*, vol. 16, no. 3, Mar. 2000, pp. 107–109. Elsevier Science Ltd. https://websites.umich.edu/~zhanglab/publications/1996-2001/Zhang_2000_TIG_16_107.pdf.