Concurrent Programming Project: DNA Sequence Alignment

Klaara Huima, Esma Masic, Giselle Feng

June 7, 2025

Abstract :

Biological sequence alignment is an important task in bioinformatics. The goal is to compare DNA, RNA or protein sequences to find areas of similarity. This constitutes a fundamental task in computational biology, aimed at identifying regions of similarity between two or more sequences, which may indicate structural, functional, or evolutionary relationships.

1 Introduction

Based on the research paper Parallel Optimal Pairwise Biological Sequence Comparison Sandes et al., 2016, we decided for our project to implement the Gotoh algorithm (1982) , which uses three dynamic programming matrices to track different alignment scenarios. This approach was implemented for both CPU and GPU architectures. Parallelization was used in order to accelerate the computation of alignment scores.

1.1 Contribution

Esma Masic

- CPU implementation of algorithm

Giselle Feng

- GPU implementation of algorithm
- Linear memory space algorithm

Klaara Huima

- GPU implementation of algorithm
- Database processing
- Performance analysis

1.2 Code Repository

The full project is found at: `https://github.com/gisellefg/Concurrent_alignemnt_project`
We would like to note that in commit c090690, Klaara committed approximately 1.6 million lines of data from the UniRef50 database.

2 Background

There are two main types of parwise sequence alignment: global alignment and local alignment. The fist attempts to align two sequences from beginning to end and is generally used when the two sequences in question are of the same length. The latter identifies the most similar subsequences within the two studied sequences. A gap in our context, refers to an insertion or an deletion in a biological sequence. When calculating the cost, in order to better reflect reality, most alignment algorithms apply a penalty. This can be a linear one where every gap costs the same or an affine one where there's a higher penalty for opening a new gap or a samller penalty for exentending existing one. The latter penalty is considered a more accurate alignment. Gotoh's algorithm integrates affine gap penalties. Our work build on his algorithm, by implementing it both for CPU and GPU, we explore the performance in parallel affine gap alignemnt.

2.1 Literature Review

There are already several foundational algorithms for biological sequence alignment. The Needleman-Wunsch algorithm (1970) for example is a dynamic programming model that uses the global alignment method with linear gap penalty. Another famous algorithm is the Smith-Waterman algorithm (1981) that provides a model for local alignment. A pseudocode using linear gap penalty is showcased in figure 1. Both of these algorithms have high computation costs. Gotoh's algorithm (1982) improves biological realism when it comes to gap modeling as it utilizes affine gap penalties, which distinguishes the cost between opening a new gap or extending an existing one. His algorithm uses three dynamic programming matrices to separately track alignments that end in a match, an insertion, or a deletion. Recent work has focused on improving the performance of exact alignment methods by using parallel computing. Special tools like CPU instructions (SSW) or very powerful GPUs (CUDASW++) considerably speed up the calculations. The research paper Sandes et al., 2016 offers an overview of the different approaches as well as the state of the art models.

ALGORITHM 1: Skeleton to Compute DP Matrices

```
1: procedure ComputeDPMatrix(S_0, S_1)
2:     InitializeFirstRow(H[0, 0..m])
3:     InitializeFirstColumn(H[0..n, 0])
         for i ← 1, m do
           for j ← 1, n do
4:           H[i, j] := RecurrenceRelation(i, j, S_0[i], S_1[j]);
           end
         end
5:     best:=RetrieveOptimalScore();
6:     return best
7: end procedure
```

Figure 1. Simplified pseudocode showing a dynamic programming structure commonly used in classical alignment algorithms such as Needleman–Wunsch and Smith–Waterman. These algorithms operate with a single scoring matrix and typically apply linear gap penalties. Gotoh's algorithm extends this structure by introducing three matrices and separate recurrence relations to support affine gap penalties.

3 Gotoh's algorithm

The algorithm we used came from Erhard, 2016. It uses three dynamic programming matrices:

- **M** – score ending with a match or mismatch,
- **I** – score ending with a gap in sequence B (insertion),
- **D** – score ending with a gap in sequence A (deletion).

$$M_{i,j} = \max \begin{cases} 0 \\ I_{i,j} \\ D_{i,j} \\ M_{i-1,j-1} + p(i,j) \end{cases} \qquad I_{i,j} = \max \begin{cases} I_{i,j-1} - extendGap \\ M_{i,j-1} - openGap \end{cases} \quad D_{i,j} = \max \begin{cases} D_{i-1,j} - extendGap \\ M_{i-1,j} - openGap \end{cases}$$

3.1 CPU Implementation and Parallelization Details

In our CPU implementation of Gotoh's affine-gap alignment algorithm we use dynamic programming and leverage parallelization to speed up the core computation. The algorithm maintains three score matrices of size $(m+1) \times (n+1)$: $M[i][j]$ holds the best score for aligning $A[i]$ with $B[j]$ (match or mismatch), $I[i][j]$ stores the best score for inserting a gap in sequence B, and $D[i][j]$ stores the best score for deleting from sequence A. The matrices are initialized such that $M[0][0] = 0$, $I[i][0] = -\texttt{openGap} - (i-1) \times \texttt{extendGap}^*$ for $i \in [1,m]^*$ and for gaps in B, and $D[0][j] = -\texttt{openGap} - (j-1) \times \texttt{extendGap}\ ^*$ for $j \in [1,n]^*$for gaps in A. Additionally, three trace matrices—`traceM`, `traceI`, and `traceD`—are used to record the backtracking path, using flags to denote the optimal predecessor state.

In order to efficiently parallelize the computation while also respecting data dependencies, we used an anti-diagonal sweep strategy. This consists of computing cells along diagonals where $i + j = \texttt{constant}$ at the same time. Each cell (i, j) only depends on the values $(i-1, j)$, $(i, j-1)$, and $(i-1, j-1)$ from the previous diagonal, which allows safe parallel updates in the same diagonal.

The core implementation loops over all diagonals from 1 to $m + n$ and, for each diagonal, it determines the valid range of indices i such that $i_{\min} = \max(1, \texttt{diag} - n)$, and $i_{\max} = \min(m, \texttt{diag})$. Then, the total number of computable cells for that diagonal is divided into chunks, whose amount is the number of hardware threads. We then dispatch these chunks to a thread pool for concurrent execution, and use a countdown latch to synchronize threads before proceeding to the next diagonal.

There are $N = \texttt{hardware\_concurrency()}$ worker threads in the thread pool which we initialize at program start. Each thread retrieves a chunk of cells to process from a dynamic task queue. Chunk sizes differ by at most one cell in order to ensure load balancing. For synchronization, a `CountdownLatch` ensures that all threads complete their tasks before proceeding.

During the matrix filling phase, we evaluate the recurrence relations in parallel for each cell (i, j) in the current diagonal. Then, we calculate the insertion score as $I[i][j] = \max(M[i-1][j] - (\texttt{openGap} + \texttt{extendGap}), I[i-1][j] - \texttt{extendGap})$, and the deletion score is given by $D[i][j] = \max(M[i][j-1] - (\texttt{openGap} + \texttt{extendGap}), D[i][j-1] - \texttt{extendGap})$. The match or mismatch score is computed as $M[i][j] = \max(M[i-1][j-1] + \texttt{submat}[A[i-1]][B[j-1]], I[i][j], D[i][j])$. During this step, we update the trace flags simultaniously so that we can record which transition led to the optimal score.

Backtracking, which is inherently sequential, starts from the matrix cell (m, n) which contains the highest final score among $M[m][n]$, $I[m][n]$, and $D[m][n]$. The algorithm then follows the trace flags to reconstruct the alignment in a way that the transitions through M emit aligned characters from both sequences, transitions through I emit a gap in B, and transitions through D emit a gap in A. This continues until the algorithm reaches cell $(0, 0)$.

We keep track of the execution timing using `std::chrono`,which reports the total runtime in milliseconds along with the final alignment score. The anti-diagonal approach offers significant advantages over row- or column-wise parallelization. It avoids conflicts during writes and removes the need for fine-grained locking or atomic

operations, since diagonal k only does the reading from diagonal $k-1$. Middle diagonals (where $i+j \approx \frac{m+n}{2}$) contain the most cells, maximizing parallel throughput. Since they are smaller, early and late diagonals complete quickly. This chunk-based workload distribution also reduces thread idle time. Latch synchronization adds minimal overhead (once per diagonal) since only $m+n$ synchronizations occur. The synchronization overhead per diagonal is minimal.

It is worth mentioning that some trade-offs remain. Backtracking is not parallelized due to the strict dependency chain. Moreover, the memory footprint is high: six matrices of $O(mn)$ are used, but this is essential for supporting affine-gap alignment with proper traceback functionality.

To support this design, we implemented two helper classes: `ThreadPool` and `CountdownLatch`. The `ThreadPool` manages a group of worker threads that wait on a condition variable for tasks. Each task is a lambda function capturing the range of cells to compute. Clean shutdown is done with a `shutDown` flag. We ensure the correct synchronization with `CountdownLatch` by decrementing an atomic counter for each completed task and blocking the main thread until the counter reaches zero.

In summary, our CPU implementation runs in $O(mn)$ time and scales well across multiple CPU cores. Our method of computing diagonals in parallel works especially well with Gotoh's algorithm, since it respects the order in which data needs to be calculated. It performs better than simpler approaches that try to parallelize entire rows or columns, and it avoids complex locking mechanisms that would slow things down.

3.2 Gotoh's algorithm on GPU

To accelerate the algorithm, we implemented a GPU-parallelized version of Gotoh's algorithm using CUDA. Since each cell $(i,j)(i,j)$ depends on the cells to its left, top, and top-left, we used anti-diagonal wavefront parallelism. In anti-diagonal parallelism, all cells along the same anti-diagonal are independent and can be computed simultaneously. We parallelize the computation by using CUDA kernels for each anti-diagonal, allowing all entries along that diagonal to be computed in parallel. We synchronize between successive anti-diagonals to ensure each kernel completes before the next is launched.

The backtracking (recovering the optimal alignment sequences) is still done on the host, since it is by nature a sequential process. Throughout the parallelized matrix computations, we store only the necessary scores on the GPU. After filling up the score matrices, we copied them back to the host and reconstructed the alignment path using the same logic as the CPU version.

4 Results

We tested the speed of our algorithms on protein sequences imported from the UniRef50 database (Suzek et al., 2007). The protein sequences were read from FASTA format files and cut to various lengths to study the performance of the implemented algorithms. All the programs were run on the school machines.

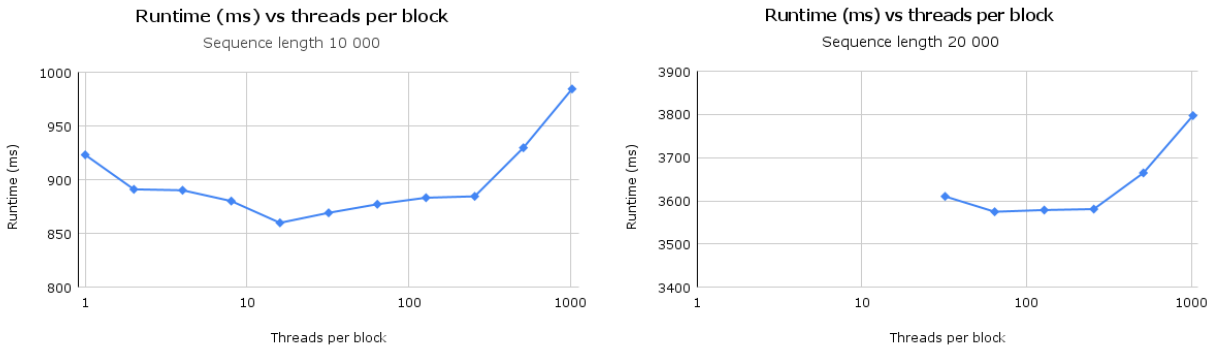
Below, we study the performance of 3 implementation of Gotoh's algorithm with an affine gap model: a sequential implementation, a concurrent implementation of Gotoh's algorithm on CPU, and a concurrent implementation on GPU using CUDA. Each algorithm takes as input 2 strings and returns an alignment score, defined by the penalties associated to different operations (inserting a gap, extending a gap, a matched character, and a mismatched character) and the optimal alignment pattern of the sequences. For every test case, all algorithms returned the same alignment score and aligned sequences.

4.1 Kernel parameters

The size of each kernel launch is defined by the number of threads per block. To ensure full coverage of the anti-diagonals, the algorithm computes the number of blocks required such that each element on an anti-diagonal is processed by some thread and no blocks are left empty. This is given by:

$$\text{blocks} = \lceil \frac{\text{total elements}}{\text{threads per block}} \rceil.$$

We experimented with thread counts that are powers of two. For each value of threads per block, we ran the sequence alignment algorithm on 50 different protein sequence pairs and computed the average runtime. This experiment was conducted for sequence lengths of 10 000 and 20 000. In the figure below, we plot the runtime as a function of threads per block on a log-linear scale, since the thread counts increase exponentially (as powers of two). CUDA supports up to 1024 threads per block.



(a) Sequence length 10 000 **(b)** Sequence length 20 000

Figure 2. Runtime with respect to number of threads per block

We notice in Fig. 2a that the runtime decreases when increasing thread count from 1 to 16, where it reaches its minimum. Between 32 and 256 threads, the runtime slightly slows down, and beyond 256, runtime increases more steeply. Since the GPU executes its threads in groups of 32 threads, using less than 32 threads per block results in idle execution units in the warp and thus bad occupancy. One would expect the best performance when the thread count per block is a multiple of 32, for maximal occupancy.

For sequences of length 20 000, in Fig. 2b, the runtime is equally optimal when using 64, 128, and 256 threads per block. The performance worsens when increasing thread count to 512 and 1024. This suggests that increasing the number of threads per block beyond a certain point may entail reduced parallelism due to limitations of shared memory. One may see that compared to the case of 10 000-length sequences, the 20 000-length sequences benefit from higher thread counts before the runtime increases.

4.2 CPU thread count

To determine how the number of threads used for the CPU implementation affected performance, we kept the input sequences constant and compared the runtimes with respect to the number of threads. We tested every thread count between 1 to 20 for three different lengths of sequences. For every length and every thread count, we ran the algorithm 50 times and took the average runtime. Running the algorithm on 1 thread corresponds to a sequential implementation. The data on average runtimes is in Table 1.

Table 1
Average runtime of CPU Gotoh's algorithm (in ms) for different thread counts and sequence lengths

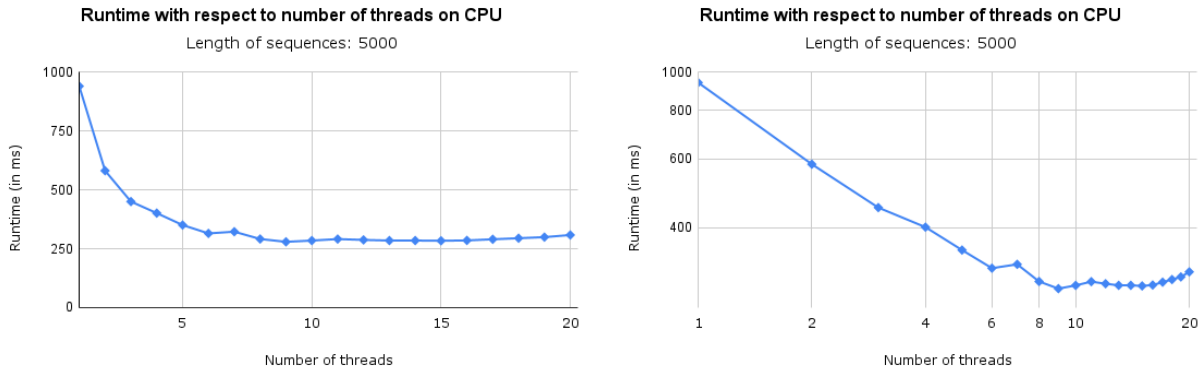
Thread Count	Seq. Length 5000	Seq. Length 10000	Seq. Length 20000
1	941	3870	24717
2	582	2268	13590
3	451	1754	9838
4	402	1474	8295
5	351	1024	7340
6	315	917	6647
7	323	865	6326
8	292	859	6242
9	280	875	6358
10	285	932	6824
11	291	1056	7362
12	288	1011	5423
13	285	979	5081
14	285	949	4761
15	284	951	4498
16	286	908	4266
17	290	904	4123
18	295	891	3986
19	300	896	3891
20	309	929	4034

4.2.1 Sequence length 5000 We've visualised the collected data in Fig. 3. The average runtimes of the algorithm for sequences of length 5000 are in between 250 and 1000 ms. The fastest runtime, 280 ms, was achieved when using 9 threads. The slowest runtime, 941 ms, was achieved when using 1 thread, as expected, as this corresponds to a sequential run.

We notice a deep decline in the runtime of the algorithm when increasing the thread count in the range 1 to 6. Plotting the data on a log – log scale (3b) reveals that the speedup of the algorithm in this range follows a nearly exponential curve. Ideally, each added thread would split the runtime in half, resulting in a perfect exponential relationship. In reality, creating, managing, and synchronizing threads, as well as memory access contention add overhead that prevents perfect halving of runtime.

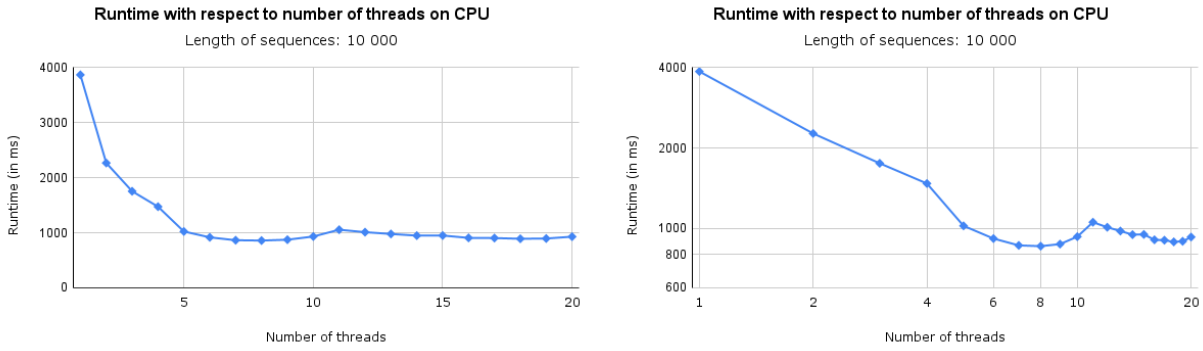
The runtime decreases rapidly initially because adding threads up to the number of physical cores improves CPU utilization and workload division. The runtime is slightly higher when using 7 threads, and remains mostly constant in the range 7 to 20. The plateau suggests that beyond 6 threads, the workload is sufficiently parallelized and that additional threads will not contribute in runtime reduction.

4.2.2 Sequence length 10000 As seen in Fig. 4, the average runtime for sequences with 10 000 elements are in between 850 and 4000 ms. The fastest runtime, 859 ms, was achieved when using 8 threads.
Similarly to when studying sequences with 5000 elements, we notice a deep decline in the runtime of the algorithm when increasing the thread count in the range 1 to 4. The speedup of the algorithm in this range also follows a similar exponential curve (visualized in Fig. 4b). The runtime dips to its minimum value at 8 threads, increases when the thread count is increased up until 11, after which it remains largely constant. This is likely due to the computer the code was run on having 8 physical cores. With the work split onto 8 threads, each thread can be



(a) Linear scale **(b)** Logarithmic scale

Figure 3. Runtime with respect to thread count, sequence length 5000



(a) Linear scale **(b)** Logarithmic scale

Figure 4. Runtime with respect to thread count, sequence length 10 000

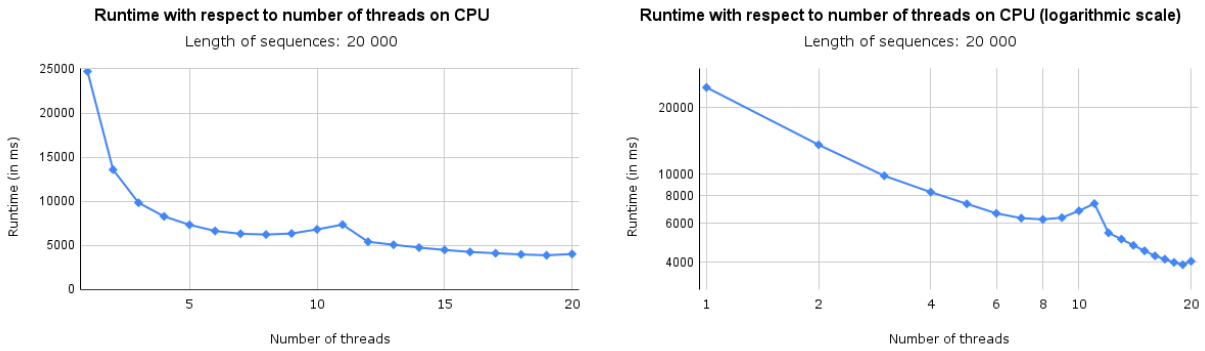
assigned to its own physical core without contention, leading to efficient parallel execution. Beyond 8 threads, the system relies on simultaneous multithreading. This can lead to relatively smaller speedups, and beyond this point, adding more threads does not significantly improve runtime.

4.2.3 Sequence length 20000 As seen in Fig. 5, the average runtimes for sequences with 20 000 elements are in between 3800 and 25 000 ms. The fastest runtime, 3891 ms, was achieved when using 19 threads. We again notice that the speedup from 1 to 6 threads follows an exponential curve (Fig. 6b). The runtime slows down when increasing the thread count up to 11, at which the runtime peaks (locally). When further increasing the number of threads, the algorithm continues to speed up.

4.3 Runtime of different algorithms

Based on the experiments above, we decided to compare the runtime of the CPU, GPU, and sequential implementation of Gotoh's algorithm with the following parameters.

- 20 threads on CPU
- 64 threads per block on the GPU
- 1 thread on the CPU for a sequential version



(a) Linear scale **(b)** Logarithmic scale

Figure 5. Runtime with respect to thread count, sequence length 20 000

4.3.1 Sequence length We ran the algorithm on 50 pairs of sequences for each of the sequence lengths in the figure below and used their average runtime for analysis. In the table below, the CPU Speedup and GPU Speedup columns are calculated by taking

$$\text{speedup} = \frac{\text{sequential runtime}}{\text{concurrent runtime}}$$

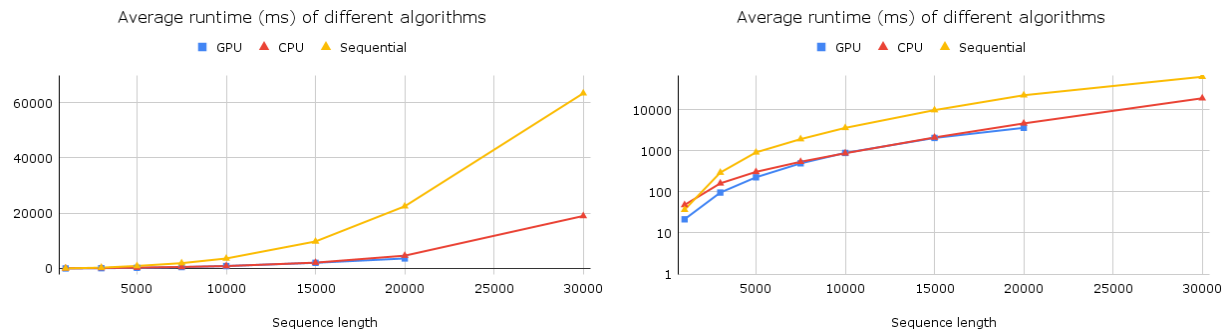
All runtimes are noted in milliseconds.

Table 2
Average runtime and speedup comparison for CPU and GPU implementations of Gotoh's algorithm

Seq. length	GPU runtime	CPU runtime	Sequential runtime	CPU speedup	GPU speedup
1000	21.5001	47.876	38.2142	0.7650	1.7069
3000	95.9847	160.992	294.816	1.8312	3.0715
5000	225.969	306.628	914.320	2.9819	4.0462
7500	493.792	539.684	1923.09	3.5634	3.8945
10000	888.558	871.578	3610.86	4.1429	4.0637
15000	2052.38	2110.32	9769.58	4.6294	4.7601
20000	3619.09	4634.71	22492.7	4.8531	6.2150
30000	–	18990.4	63376.5	3.3373	–

We have visualized the data in Fig. 6 on a linear and log-linear scale for clearer analysis of differences in results. The runtime results show that as sequence length increases, the computational cost grows quickly for all implementations. The sequential algorithm's runtime is by far the slowest, reaching over 63 seconds for sequences of length 30,000.

The CPU implementation's speedup with respect to the sequential version increases with sequence length, starting below 1 at length 1000 due to parallel overhead and reaching about 4.85x faster runtime at length 20 000. The GPU performs better, reaching an approximately 6x faster runtime at length 20 000. The GPU outperforms the CPU for shorter sequences as well, with a speedup of approximately 3.07 at length 3000 compared to the CPU's 1.83. For lengths 7500 - 15 000, the CPU and GPU algorithms have very similar average runtimes. For the largest sequence length with CPU data (30 000), the CPU speedup decreases to around 3.34, indicating the algorithm is not as efficient with larger entries, possibly caused by increased memory demands and overhead.



(a) Linear scale **(b)** Log-linear scale

Figure 6. Runtime comparison with respect to sequence length

We could not test our GPU implementation on sequences longer than 20 000 characters due to lack of memory. Our algorithm involves storing 3 $M \times N$ matrices, where M and N are the lengths of the sequences. The GPU memory is limited, and its maximum capacity is exhausted when the sequence lengths exceed 20 000. To address this issue, we considered implementing a linear space version of the algorithm derived from Myers et al., 1988. This is further discussed in section 4.4.

4.3.2 Input similarity We then compared the runtime with respect to input similarity on all 3 implementations. We measure the similarity of 2 inputs by the number of positions in which the 2 sequences do not match. We fix a sequence length (5000), draw a protein sequence from the database at random, shorten it to the chosen length, and then randomly change a fixed number of characters in it to some other (random) character. We again took the average runtime over 50 random pairs of input sequences. The collected data is in Table 3. From the data, we conclude that all the tested algorithms have a constant runtime with respect to input similarity. This is visualized in Fig 7.

Table 3
Runtime comparison of GPU, CPU, and sequential implementations at varying levels of sequence dissimilarity.

Proportion of different characters	GPU (ms)	CPU (ms)	Sequential (ms)
0	236.142	323.413	811.492
0.1	238.308	314.800	802.443
0.2	237.430	313.113	787.275
0.3	238.915	337.087	859.633
0.4	256.45	321.981	807.270
0.5	241.318	301.964	822.046
0.6	238.500	310.888	846.874
0.8	237.434	309.335	844.102
1	238.728	313.939	855.440

4.4 Linear space algorithm

The main motivation to use a linear space algorithm is to drastically reduce memory consumption. The previous implementations require storing matrices of size $O(MN)$. For long sequences, the available memory in RAM can be quickly exceeded. Linear space algorithms reduce the required memory to $O(N)$ (or $O(M)$), although at the cost of losing information (optimal alignment pattern). In the linear space approach, the alignment score is calculated by only storing the current and previous rows of the dynamic programming matrices (M, I, D). This is

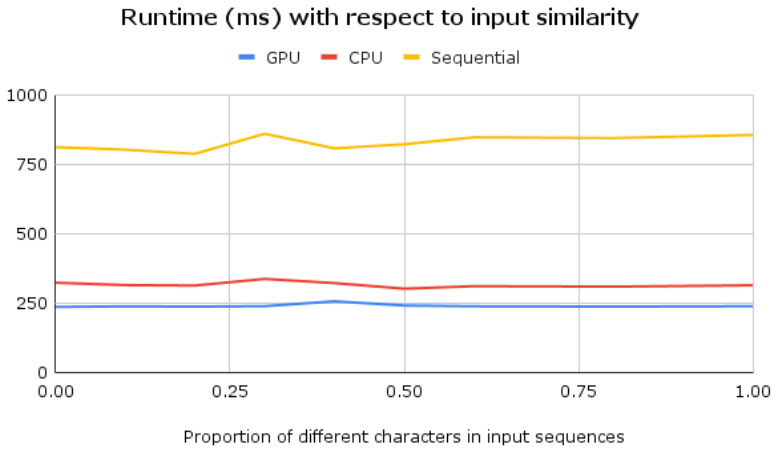


Figure 7. Average runtime of different algorithms with respect to input similarity

possible as to calculate any cell's score in the current row, one only needs the values from the cell directly above (previous row) and the cell to its left (current row). This method then iteratively calculates each row's scores, swapping the current and previous row buffers after each complete row. Unfortunately without storing the full matrices we are not able to derive the optimal alignment pattern between the sequences, and the algorithm only returns the alignment score.

4.4.1 Discrepancy in Alignment Scores We implemented a linear space Gotoh's algorithm for GPU so we can run the comparison on very long sequences (> 25 000 characters). However, this attempt provided incompatible alignment scores with the original GPU implementation. This discrepancy arose from a problem in the parallelization strategy. Indeed, the chosen row-wise parallelization assigned one thread per column, creating a race condition because the calculation for the deletion matrix at position $D(i, j)$ depended on the value of the adjacent cell $D(i, j-1)$. Since threads in the same kernel launch cannot guarantee the order of execution or access to results, the dependency and order was not respected. Hence, the required data flow was not respected which caused incompatible scores with the original implementations. This led us to believe that maybe if we implemented it for CPU, where each row is computed sequentially, then we might get the same scores. However, the CPU linear space implementation also produced inaccurate alignment scores. The issue was that we only stored the current and previous rows and dependencies that spanned more than one row (the ones in affine gap matrices like deletion matrix D) could then not be properly calculated. Indeed, it relies on the whole column. The problem arises because the optimal path for $D(i, j)$ for example might involve a long deletion that started many rows ago. This moreover made us realized that even if the race condition problem was solved for the linear space GPU version, it would still be impacted by values that are not readily available.

However, the latter algorithm did manage to run for very large sequences for GPU. We ran it with sequences of 50 000 characters. This was done by concatenating different index of sequences. The alignment score provided is only of interest if one compares it with the alignment score obtained with the same linear space algorithm.

5 Discussion

This project highlighted the trade-offs involved in implementing Gotoh's algorithm across different hardware and memory models. For the CPU model, we saw consistent speedups as the number of threads increased. One sees that the average runtime remains constant for shorter sequences if we keep increasing the number of threads beyond 7. For sequences of length 20 000 characters, the runtime performance kept increasing as we increased

the number of threads available to use for the program (up to 19 threads). The CPU speedup peaks for sequences with 20 000 characters, and decreases when the sequence length is increased to 30 000.

For the GPU model, one sees that increasing the number of threads per block improves performance up to a certain point, depending on the sequence length being processed. After that, the runtime performance worsens. To over come GPU memory limits, we implemented a linear space version of Gotoh's algorithm. This version significantly reduced memory usage and allowed us to align sequences of 50 000 characters and more. However, the alignment scores produced were incorrect due to limitations of data flow. This issue also appeared in the linear space GPU implementation, although the calculations were sequential there was a flaw in how the affine gap penalty logic was implemented. The linear space variant still proved useful when comparing sequences of very large length internally using the same algorithm.

In general, both parallelized versions outperform the sequential version. The best speedup was seen on GPU with sequences of length 20 000. The CPU and GPU implementations perform relatively similarly for short sequences (<15 000 characters).

5.1 Future improvements

Possible improvements to our project would be to include partial backtracking in the linear space model: this variant is explored in Myers et al., 1988 by Myers and Miller, and still maintains a complexity of $O(N)$. This could lead to more efficient memory management as well as achieve correctness in alignment scores.

6 Conclusion

We benchmarked three implementations of Gotoh's algorithm: a sequential one, a multithreaded CPU one and a CUDA-based GPU version. Our results showed consistent alignment scores and showcased significant performance improvement with parallelism. This is specifically relevant for sequences of longer lengths. We wanted to overcome memory limitations by experimenting with the linear space variant, but this introduced inaccuracies. This highlights the key challenge of overcoming memory constraints when it comes to parallelizing dynamic programming. It is not always straightforward trying to ensure correctness and balancing runtime performance with efficient memory usage.

References

Erhard, Florian (2016). *Gotoh2: Fixing Gotoh's Algorithm*. https://florianerhard.github.io/2016/gotoh2. Accessed: 2025-06-07.

Myers, Eugene W. and Webb Miller (1988). "Optimal alignments in linear space". In: *Bioinformatics* 4.1, pp. 11–17. DOI: 10.1093/bioinformatics/4.1.11.

Sandes, Edans Flavius de Oliveira, Azzedine Boukerche, and Alba Cristina Magalhaes Alves de Melo (2016). "Parallel Optimal Pairwise Biological Sequence Comparison: Algorithms, Platforms and Classification". In: *ACM Computing Surveys (CSUR)* 49.4, pp. 1–38. DOI: 10.1145/3003784.

Suzek, Baris E et al. (2007). "UniRef: comprehensive and non-redundant UniProt reference clusters". In: *Bioinformatics* 23.10, pp. 1282–1288. DOI: 10.1093/bioinformatics/btm098.