Improving the performance of Smith-Waterman Algorithm

Sai Kaushik S (2025CSZ8470)
Yosep Ro (2025ANZ8223)
Indian Institute of Technology Delhi

November 26, 2025

1 Introduction

The goal of this lab is to explore the performance impact of architecture-level optimizations for the Smith–Waterman local sequence alignment algorithm and to analyze their practical effects in terms of runtime, speedup, scalability, and memory usage. Smith–Waterman computes the best local alignment between two DNA-like sequences using a dynamic programming (DP) matrix and a simple scoring scheme. The provided baseline is a Smith-Waterman implementation written in Python, which we reimplemented and optimized in C++ to achieve high-performance execution.

To enhance overall system performance, we applied multiple key optimizations that target different aspects of the processor and memory hierarchy. The complete source code, simulation scripts, and data used for generating the plots in this report are publicly available on GitHub at `https://github.com/sai-kaushik-s/Enhanced-Smith-Waterman`.

2 Background and Assumption

2.1 Smith-Waterman Algorithm

The Smith–Waterman (SW) algorithm is a dynamic programming method for *local* sequence alignment. Given two sequences $A = a_1 \dots a_m$ and $B = b_1 \dots b_n$, it finds the highest scoring alignment between any substrings of A and B, in contrast to global alignment algorithms that align both sequences end to end.

Refer to the simple Python baseline implementation provided in the lab, which we use as the functional reference throughout this work, the algorithm constructs a $(m + 1) \times (n + 1)$ DP matrix as shown in the Fig. 1. Each cell $H[i, j]$ represents the best local alignment score that ends at positions (i, j) and is computed as Eqation 1, where $s(a_i, b_j)$ is the match/mismatch score and g is the gap penalty.

$$H[i,j] = \max\big(0,\ H[i-1,j-1] + s(a_i,b_j),\\ H[i-1,j] + g,\\ H[i,j-1] + g\big). \tag{1}$$

In this lab, we use a simple scoring scheme: match +2, mismatch −1, and gap −2, which directly instantiates the above recurrence.

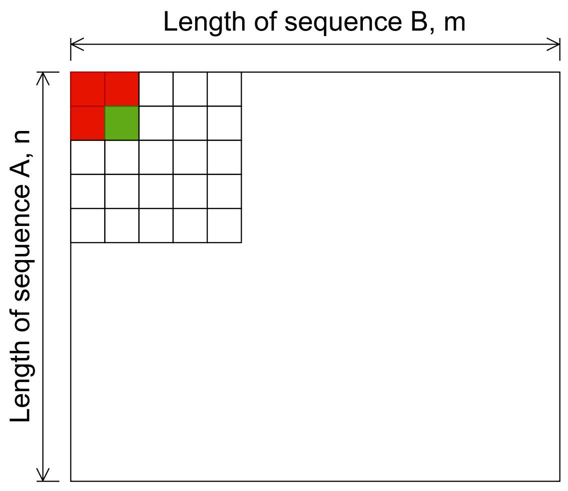

Figure 1: Baseline implementation

2.2 Assumption

Compared to the GEMM kernel in Assignment 1, the SW kernel in this lab exhibits much stronger *sequential* dependencies, as illustrated in Figure 1. Each DP cell $H[i, j]$ depends on its top, left, and top-left neighbours, so updates within a row or column cannot be reordered freely. This means that parallel implementations must carefully respect these row/column dependencies, and fine-grained parallelism is inherently more constrained than in GEMM.

A second difference from GEMM is the effective working set in memory. In matrix multiplication, the full input matrices are reused many times and the kernel is often strongly memory- and cache-bound. In SW, each DP cell is touched once in a regular streaming pattern, and only a small window of rows (or columns) is needed at a time. Under these conditions we assume that the kernel behaves closer to compute-bound than bandwidth-bound, and thus prioritize instruction-level parallelism and SIMD utilization over aggressive data prefetching.

3 Architectural Design

This section presents seven architecture-level optimization techniques applied to the baseline Smith-Waterman Algorithm implementation. Each subsection describes the technique, the architectural reasoning behind it.

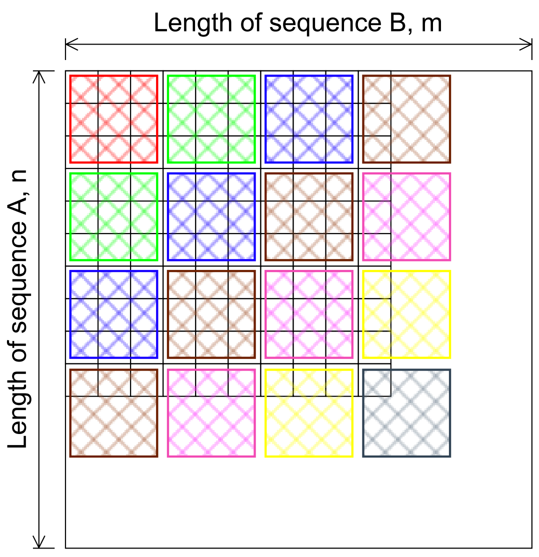

Figure 2: Tiling

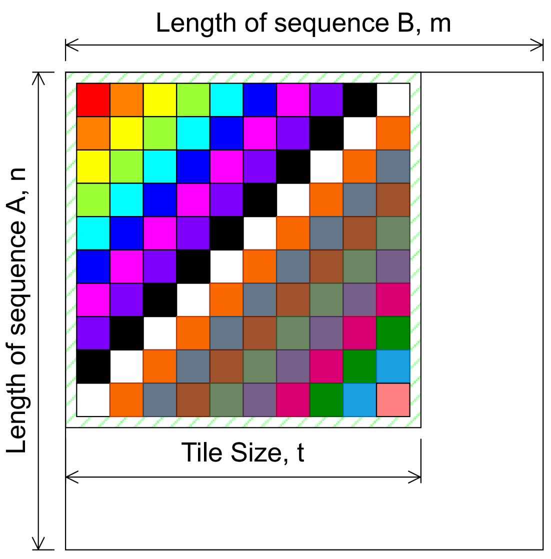

Figure 3: Diagonalization

One of the techinque used in the assignment is the wavefront parallism. Wavefront parallelism or anti-diagonal traversal is a strategy used in parallel computing where computations are done diagonal by diagonal instead of row-by-row or column-by-column. The cells whose dependencies are already resolved are processed. Each anti-diagonal is a wavefront that moves across the matrix.

3.1 Tiling (Cache Blocking)

We employed a **tiling (blocking)** technique that divides the DP Table into multiple smaller tiles. The tile size was carefully chosen to fit within the cache capacity, ensuring that there are no cache misses for the operation done within the same. The graphical illustration of tiling of the DP Table is in Figure 2.

The dependeny of the computation among the the tile is wavefront as colored in the Fig. 2. Each tile on the wavefront depends on the computed value of the previous wavefront. Also, there is no dependency among the tiles in same wavefront. Each tile on the wavefront can be computed at the same time by multiple threads, making the computation highly parallelizable.

We maximized the tile size to fit within the L1 cache ensuring that the working set of the computation always remains accessable without any misses. The tile size is determined according to the following constraint, where $\alpha = 0.5$ to account for code and metadata overhead for the further computation.

$$t^2(\text{Tile size of DP Table}) \leq \alpha \cdot \frac{\text{Cache size}}{\text{sizeof(int16\_t)}}$$

At the beginning of each iteration, one tile of the DP Table is created in memory and loaded to the L1 cache for further computation. Within that region, the algorithm follows the diagonalization ideology to make the firther computation independent of the previous iteration.

3.2 Wavefont parallelism

Within each tile, the Smith-Waterman scores are calculated in a similar diagonal fashion as shown in Fig. 3. From Eq. 1, we can deduce that for any given cell (`i, j`), the value depends on (`i-1, j`), (`i, j-1`) which are present in the previous wavefront and (`i-1,j-1`) which is present in the previous to previous wavefront and are not dependent on the elements on the same wavefront. Here, the upcoming Unrolling and AVX optimizations come into play.

The only optimiztion that cannot be applied to this is the cache prefetching as the data used is not contiguous and prefetching data would thrash the existing data increasing our misses.

3.3 Loop Unrolling

To push instruction-level parallelism (ILP) further and minimize loop-control overhead, we enable the compiler's loop-unrolling option (`-funroll-loops`) during compilation. The diagonal implementation allows effective unrolling of the inner loop leading to effective AVX optimization as mentioned.

3.4 AVX2 / AVX-512 Vectorization

To further increase the computational throughput of the inner kernel, we implicitly vectorize the innermost loop using AVX2 and AVX-512 intrinsics suing the OpenMP directive, `#pragma omp simd`, which auto-vectorizes the data based on the available instruction set and hardware. This is done at compile time when the binary is created based on the modules provided to it.

Theoretically, the loop would unroll 16 times for the AVX2 instruction and 32 tiles for the AVX512 instruction for the `int16_t` data type.

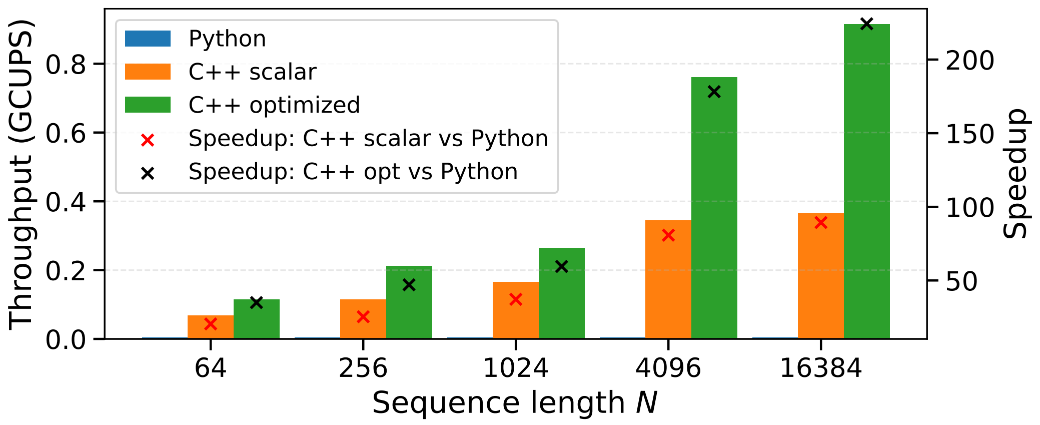

Figure 4: Single-core throughput and speedup over the Python baseline for the Python, C++ scalar, and C++ optimized implementations.

3.5 OpenMP Parallelization

As explained above with the tiling process, we use the multiple available threads to work on the wavefront tiles created based on the tile size. Similar to the diagonalization done within each tile, the tiles themselves follow the same principle and can be computed in parallel as soon as it previous diagonal tiles are done executing. Each thread pauses its next instruction until all the pre-requisites are done with their computations using the `_mm_pause` function.

3.6 NUMA-aware Memory Placement

On a multi-socket NUMA system, remote memory accesses incur significantly higher latency. To minimize cross-socket traffic, we ensure that thread placement and memory allocation occur on the same NUMA node. Specifically, the program allocates its working buffers after the OpenMP threads are pinned, enabling a first-touch allocation policy that aligns memory pages with the thread's NUMA domain.

4 Result

4.1 Methodology and Metrics

We evaluate the Smith–Waterman implementations on the host system described in Table 1. For each experimental setting, we run the benchmark at least three times and report the arithmetic mean of the runtimes to reduce random noise.

Table 1: System configurations.

Component	Specification
CPU	13th Gen Intel Core i7-13700 (16C/24T)
RAM	16 GB DDR5-4400 (Single Channel)
ISA	x86 64-bit
OS	Ubuntu 22.04.5 LTS

We measured two primary metrics: *speedup* and *throughput*. Speedup is computed with respect to the given Python baseline implementation. We measure the total execution time of each program and compute the speedup based on this runtime. As a throughput metric, we use GCUPS (Giga Cell Updates Per Second), which is standard for dynamic-programming sequence alignment kernels and plays a similar role to GFLOPs in the GEMM assignment. This counts how many DP cells are updated per second and allows a direct comparison of the computational throughput across different implementations and thread counts.

4.2 Single-Core Performance

We first compare different implementation stages in a single-core setting. To isolate the effect of algorithmic and microarchitectural optimizations, we fix the number of threads to one and use equal-length input sequences with $N = (64, 256, 1024, 4096, 16384)$. We evaluate the following four variants:

- **Python baseline**: pure Python implementation.
- **C++ scalar**: a direct C++ translation using nested loops and a full DP matrix in memory, without explicit blocking or SIMD.
- **C++ optimized**: our final implementation that combines row blocking, diagonal traversal, and AVX2 vectorization of the inner DP loop.

Figure 4 summarizes the single-core behaviour of these three implementations as we vary the sequence length $N =\in \{64, 256, 1024, 4096, 16384\}$ with one thread. The bars show throughput in GCUPS, while the point markers on the secondary axis show speedup relative to the Python baseline.

Even without any architecture-specific optimizations, the C++ scalar version already delivers more than an order-of-magnitude higher GCUPS than Python across all problem sizes, primarily by eliminating interpreter overhead and dynamic dispatch. The C++ optimized implementation further improves GCUPS by a factor of roughly 1.5–2.5× over the scalar version. This additional gain is due to the combined effect of row blocking, diagonal traversal, and AVX2 vectorization, which together increase data locality

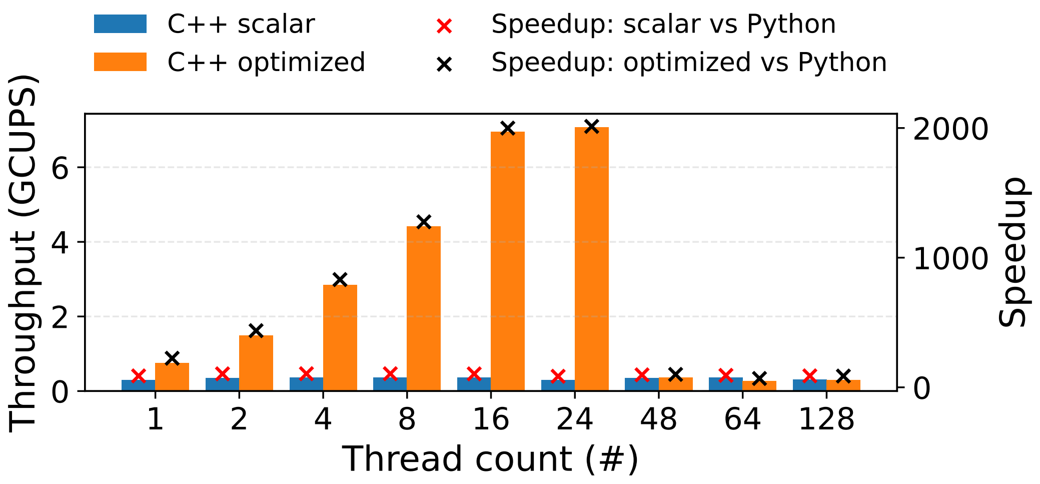

Figure 5: Multicore scaling throughput and speedup over the Python baseline for the Python, C++ scalar, and C++ optimized implementations.

and expose more instruction-level and data-level parallelism within the DP kernel.

The speedup markers highlight the same trend. Depending on N, the C++ scalar code achieves between approximately 20× and 90× speedup over the Python baseline, while the C++ optimized version reaches more than two orders of magnitude (over 200× at $N = 16384$). These results show that even on a single core, architecture-aware design choices have a dominant impact on the effective throughput of the Smith–Waterman kernel.

4.3 Multicore Scalability

We next evaluate how our design scales with the number of cores. For this experiment, we fix the sequence length to $N = 16384$ and vary the thread count over $T \in \{1, 2, 4, 8, 16, 24, 48, 64, 128\}$. For each configuration, we measure throughput in GCUPS and compute the speedup with respect to the Python baseline implementation. We compare our optimized C++ design against a pure C++ baseline (C++ scalar) that uses a straightforward nested-loop translation without architecture-specific optimizations.

The C++ scalar kernel shows almost flat throughput across thread counts. It achieves roughly 0.3–0.36 GCUPS from 1 to 24 threads, and neither runtime nor GCUPS improve significantly beyond the single-thread case. This indicates that the scalar implementation is effectively unable to exploit additional cores in our current parallelization strategy, and behaves mostly like a single-core, cache-resident DP kernel. The Python baseline exhibited a similar trend: its runtime only changes marginally as the thread parameter is increased, confirming that the pure Python implementation also does not meaningfully benefit from extra cores. Nevertheless, C++ scalar already delivers about two orders of magnitude speedup over the Python baseline (around 80×–100×), primarily by eliminating the interpreter overhead and using a more cache-friendly layout.

In contrast, the C++ optimized kernel scales much more effectively. Starting from approximately 0.76 GCUPS at 1 thread, throughput increases to around 7 GCUPS at 16–24 threads, corresponding to a parallel speedup of about 9× over the single-threaded optimized version and more than 2000× over the Python baseline at its peak. The scaling is close to linear up to the number of physical cores (16 on our system), after which it starts to saturate. Beyond 24 threads, the performance drops sharply. At 48, 64, and 128 threads the GCUPS and speedup both degrade, even lower than C++ scalar, reflecting oversubscription overheads such as context switching and synchronization costs.

Overall, these results highlight that merely translating the Python baseline to C++ is not sufficient to exploit the underlying hardware. The C++ scalar kernel already delivers around two orders of magnitude speedup over Python, but its GCUPS remains almost flat with increasing thread count and it fails to benefit from additional cores. In contrast, the C++ optimized kernel, which adds cache-conscious blocking, diagonal traversal, and vectorization, achieves both substantially higher single-core throughput and meaningful multicore scaling: at its peak, it is roughly an order of magnitude faster than the C++ scalar version and about 9× faster. The gap between the scalar and optimized C++ curves between various number of cores shows that careful architectural design is at least as important as adding more cores.

5 Discussion

Compared to the GEMM kernel in Assignment 1, the Smith–Waterman kernel exhibits a very different performance profile. Because only a small window of DP rows needs to be resident in fast memory at any given time, the working set fits comfortably in cache and the kernel behaves closer to compute-bound than memory-bound. As a result, the critical factor is not aggressive data reuse across large tiles, but how much computational parallelism we can safely extract from the DP recurrence.

The first key step was therefore to expose parallelism in an algorithm that is, by construction, highly sequential along rows and columns. The baseline Python and C++ scalar implementations update $H[i,j]$ row by row, so each cell depends on freshly written neighbours; empirically, both show almost no benefit from increasing the thread parameter. For $N = 16384$ the Python runtime stays around 78–79 s and the C++ scalar runtime around 0.74–0.90 s from 1 to 24 threads, with GCUPS nearly flat in both cases. By restructuring the computation to update cells along diagonals and combining this with row blocking, the optimized kernel can safely process independent wavefronts in parallel.

Once this parallelism was enabled, however, we observed new bottlenecks in the memory system. With multiple cores concurrently updating different wavefronts of the DP matrix, cache traffic increased and the measured high L2 miss rates rose compared to the baseline case. This meant that the optimized kernel, which initially behaved as compute-bound, began to encounter memory-related bound in system at higher parallelism. To address this, we introduced a tiling scheme that keeps only a carefully sized window of DP rows and columns cache-resident per core. This reconstructed design supported by memory optimization allows the OpenMP to scale: at $N = 16384$ the optimized kernel improves from about 0.76 GCUPS at 1 thread to roughly 7 GCUPS at 16–24 threads, corresponding to a $\approx 9\times$ speedup over its own single-threaded version and more than $2000\times$ speedup over the Python baseline at its peak. These results show that, for this DP workload, carefully exploiting parallelism at both the algorithmic level (diagonal wavefront updates) and the microarchitectural level (AVX2 SIMD) is significantly effective, while supported by memory-level optimizations such as tiling at the same time.