Lab Report: Accelerating the Smith-Waterman Algorithm

Student Name 1 (EntryNo1) Student Name 2 (EntryNo2)

November 15, 2025

Abstract

This report details the iterative optimization of the Smith-Waterman algorithm for local sequence alignment. Starting from a baseline Python implementation, we describe a series of architecture-aware optimizations, including C++ reimplementation, SIMD vectorization, and multicore parallelism. We present a detailed performance analysis, including speedup, scalability, and profiling data, to quantify the impact of each optimization. We conclude with a reflection on the most significant architectural factors that dominated performance.

Contents

1 Introduction **2**
1.1 System Environment 2

2 The Smith-Waterman Algorithm **2**

3 Baseline Implementation and Performance **3**

4 Optimization Strategies and Reasoning **3**
4.1 C++ Re-implementation (Serial) 3
4.2 Compiler Optimizations 3
4.3 Cache-Friendly Blocking (Tiling) 3
4.4 SIMD Vectorization (e.g., AVX2/AVX-512) 4
4.5 Multicore Parallelism (Multi-threading) 4

5 Performance Analysis and Results **5**
5.1 Overall Performance Improvement 5
5.2 Scalability Analysis 5
5.3 Profiling Evidence (Cache/IPC) 5

6 Reflection and Conclusion **5**
6.1 Reflection: Dominant Architectural Factor 7
6.2 Conclusion 7

1 Introduction

The Smith-Waterman (S-W) algorithm is a fundamental algorithm in bioinformatics used for local sequence alignment. It finds the most similar regions between two biological sequences (e.g., DNA or protein) using a dynamic programming approach.

The objective of this assignment is to accelerate a provided Python baseline implementation of the S-W algorithm. We explore a series of optimizations targeting modern CPU architectures, moving from a high-level language to architecture-aware C++ code.

This report is structured as follows:

- **Section 2** briefly describes the S-W algorithm and its recurrence relation.
- **Section 3** analyzes the baseline implementation and its performance.
- **Section 4** details each optimization strategy applied, from compiler flags to vectorization and multi-threading.
- **Section 5** presents the performance results, including speedup, scalability, and profiling evidence.
- **Section 6** provides a final reflection on the key performance bottlenecks and takeaways.

1.1 System Environment

All benchmarks were conducted on the following system. This is critical for reproducibility.

- **CPU:** [e.g., Intel Core i9-XXXX or AMD Ryzen 9 XXXX]
- **Cores/Threads:** [e.g., 16 Cores / 32 Threads]
- **L1/L2/L3 Cache:** [e.g., 32KB/256KB/16MB]
- **SIMD Support:** [e.g., AVX2, AVX-512]
- **OS:** [e.g., Ubuntu 22.04 LTS]
- **Compiler:** [e.g., g++ 11.4.0]
- **Compiler Flags (Baseline):** [e.g., -O2]

2 The Smith-Waterman Algorithm

The core of the S-W algorithm is the computation of a scoring matrix H where $H(i, j)$ is the maximum alignment score between a suffix of the first sequence (ending at i) and a suffix of the second sequence (ending at j).

The recurrence relation is defined as:

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

where $s(a_i, b_j)$ is the score for a match or mismatch, and gap is the penalty for an insertion or deletion. For this assignment, the scoring scheme is:

- **Match:** +2
- **Mismatch:** -1
- **Gap:** -2

3 Baseline Implementation and Performance

The baseline implementation is a pure Python script [Name of baseline script, e.g., `sw_baseline.py`]. It implements the recurrence relation directly using nested loops to fill the DP matrix.

[Describe the input sequences used for benchmarking, e.g., synthetic DNA sequences of length N and M].

The wall-clock time for the baseline implementation on the test input was:

- **Baseline Runtime:** [e.g., 120.5 seconds]

This runtime serves as the basis for all speedup calculations.

4 Optimization Strategies and Reasoning

We applied a series of optimizations iteratively, measuring the performance at each step.

4.1 C++ Re-implementation (Serial)

The first step was to port the algorithm from Python to C++. This eliminates the overhead of the Python interpreter and allows for more direct control over memory and execution.

- **Reasoning:** Python's dynamic typing and interpreted nature introduce significant overhead for tight numerical loops. C++ is a compiled language that can generate much more efficient machine code.
- **Performance:** [Runtime and Speedup]

4.2 Compiler Optimizations

We compiled the C++ code with aggressive optimization flags.

- **Flags Used:** `-O3 -march=native -funroll-loops`
- **Reasoning:** `-O3` enables a wide range of optimizations. `-march=native` allows the compiler to generate instructions specific to our CPU, including auto-vectorization. `-funroll-loops` can reduce branch overhead in loops.
- **Performance:** [Runtime and Speedup]

4.3 Cache-Friendly Blocking (Tiling)

The S-W algorithm has data dependencies that make simple blocking difficult. However, we can use techniques like wavefront parallelism or tiled computation.

- **Reasoning:** [Explain your blocking strategy. The standard DP matrix computation is cache-unfriendly as it sweeps through large amounts of memory. Blocking aims to keep the "hot" working set in the cache.]
- **Diagram:** Below is a placeholder for a diagram illustrating our data blocking/tiling strategy.

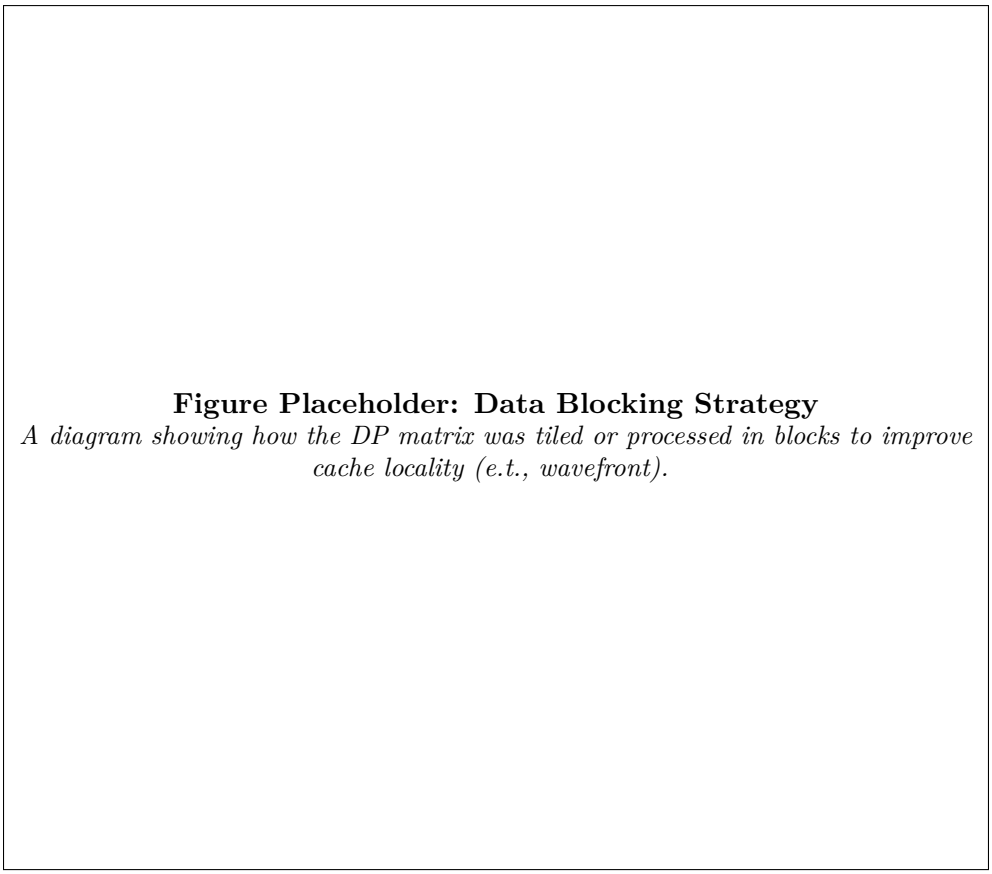

Figure 1: Illustration of the data blocking strategy.

4.4 SIMD Vectorization (e.g., AVX2/AVX-512)

We manually vectorized the innermost loop using AVX intrinsics.

- **Reasoning:** The S-W recurrence involves multiple independent max operations, which are well-suited for SIMD. A single AVX-512 instruction can perform operations on 16 integers (32-bit) in parallel.
- **Implementation:** [Describe which intrinsics were key, e.g., `_mm512_max_epi32`, `_mm512_add_epi32`, `_mm512_loadu_si512`, etc.]
- **Performance:** [Runtime and Speedup]

4.5 Multicore Parallelism (Multi-threading)

Finally, we parallelized the computation across multiple CPU cores using [e.g., OpenMP, std::thread, C++ Pthreads].

- **Reasoning:** [Describe your parallelization strategy. For example, did you parallelize an outer loop? Did you use task-based parallelism for the wavefront blocks?]
- **Synchronization:** [Describe any synchronization needed, e.g., barriers, locks, or if it was embarrassingly parallel.]
- **Performance:** [Runtime and Speedup on N cores]

5 Performance Analysis and Results

This section quantifies the performance of each optimization.

5.1 Overall Performance Improvement

The table below summarizes the runtime and speedup at each stage of optimization.

Table 1: Performance at each optimization step.

Optimization Step	**Runtime (sec)**	**Speedup (over baseline)**
1. Python Baseline	[e.g., 120.50]	1.00x
2. C++ Serial (O2)	[e.g., 10.20]	[e.g., 11.8x]
3. C++ Serial (O3, native)	[e.g., 8.50]	[e.g., 14.2x]
4. + Blocking	[e.g., 7.10]	[e.g., 17.0x]
5. + AVX-512	[e.g., 2.30]	[e.g., 52.4x]
6. + Multicore (N cores)	[e.g., 0.45]	[e.g., 267.8x]

5.2 Scalability Analysis

The plot below shows the speedup of the final parallel implementation as the number of threads increases from 1 to [Max Cores].

5.3 Profiling Evidence (Cache/IPC)

To validate our reasoning, we used `perf` to profile the code.

- **Cache Misses:** The cache-blocking optimization reduced L3 cache misses from [X%] to [Y%], demonstrating improved data locality.
- **IPC:** Manual vectorization with AVX-512 significantly improved the Instructions Per Cycle (IPC). The scalar C++ version had an IPC of [e.g., 0.8], while the vectorized version achieved an IPC of [e.g., 2.5].

Example `perf stat` output for the final optimized version:

Listing 1: Perf stat output for optimized run

```
Performance counter stats for './sw_optimized':

  1,393.94 msec    task-clock   # 1.887 CPUs utilized
  59             context-switches
  1,95,451       page-faults
  4,25,73,83,614 cycles       # 3.054 GHz
  4,48,34,89,980 instructions # 1.05 insn per cycle
  51,04,34,834   branches
  1,47,317       branch-misses  # 0.03% of all branches

  0.738770142 seconds time elapsed
```

6 Reflection and Conclusion

This lab demonstrated a systematic approach to code optimization. We achieved a total speedup of [e.g., 267.8x] over the Python baseline.

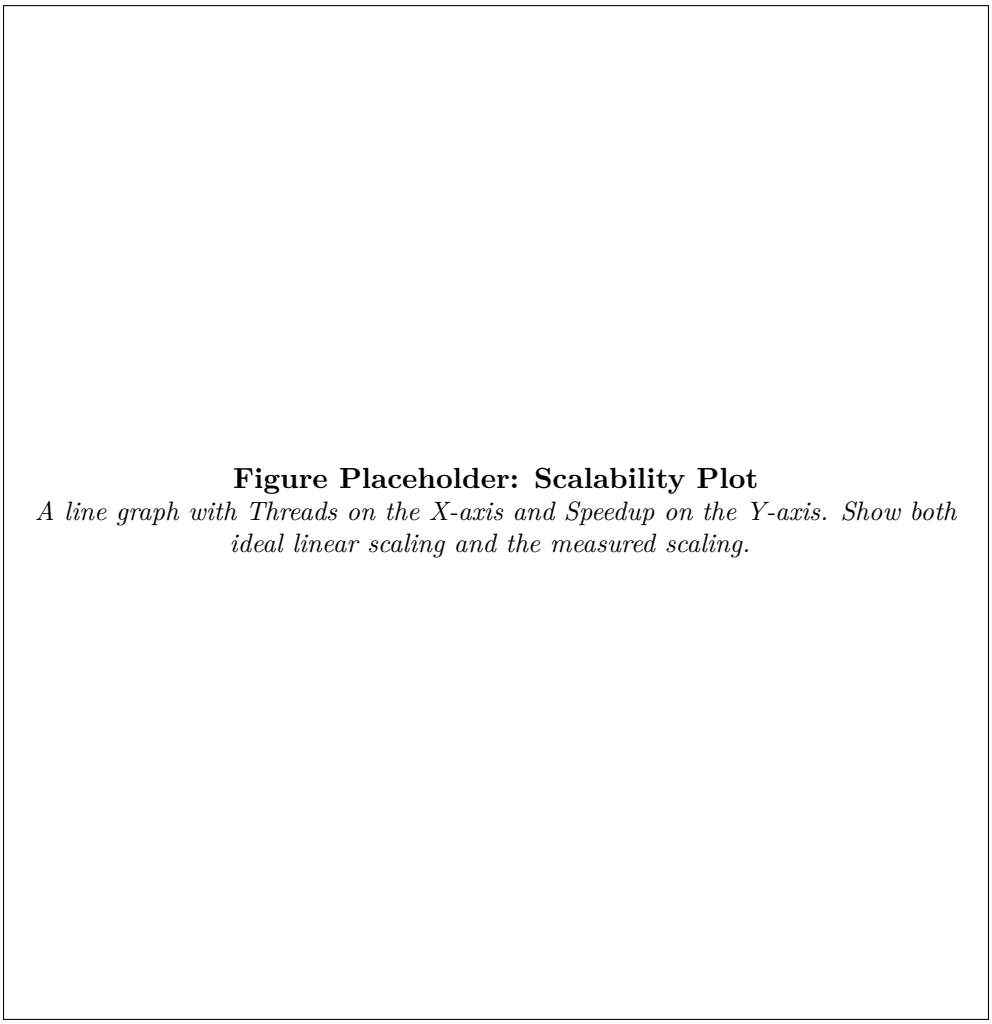

Figure 2: Scalability: Speedup vs. Number of Threads.

6.1 Reflection: Dominant Architectural Factor

[This is the most important part of the conclusion.]

The architectural factor that dominated performance was [Choose one and explain]:

- **Data-Level Parallelism (SIMD):** The transition from scalar to AVX-512 operations provided the single largest jump in performance (a [Y]x speedup). This shows that for algorithms with regular, independent operations like S-W, exploiting SIMD is critical.
- **Memory-Hierarchy (Cache):** While SIMD was important, the cache-blocking strategy was essential to feed the vector units. Without it, the vectorized code was stalled on memory, and the speedup was only [Z]x. Therefore, managing the memory hierarchy was the true key.
- **Thread-Level Parallelism (Cores):** The problem was highly parallelizable, and the near-linear scaling up to [N] cores shows that the dominant factor was simply the amount of parallel hardware available.

[Elaborate on your choice]. For example: While multi-threading gave the final best result, the most *architecturally interesting* optimization was the AVX-512 vectorization. It forced us to rethink the data dependencies and provided a [Y]x speedup on a single core, which is a testament to the power of data-level parallelism.

6.2 Conclusion

Through this lab, we successfully applied optimization principles to accelerate the Smith-Waterman algorithm significantly. The process highlighted the trade-offs at each step and the importance of a profiling-guided approach.

References

[1] Wikipedia contributors. (2025, October). *Smith–Waterman algorithm*. Wikipedia. `https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm`

[2] GeeksforGeeks. (n.d.). *Sequence Alignment Problem*. `https://www.geeksforgeeks.org/dsa/sequence-alignment-problem/`

[3] [Your CPU Manufacturer, e.g., Intel]. (2024). *Intel 64 and IA-32 Architectures Optimization Reference Manual*. [Add URL if you used it]