Lab Final: Accelerating Smith–Waterman Algorithm

Adithya R Narayan: 2024MCS2445 Manish Patel: 2024MCS2460

November 20, 2025

Abstract

This report documents architecture-aware optimization of the Smith–Waterman local alignment algorithm. Starting from a Python/NumPy baseline, we implemented a Numba JIT optimized kernel using a flattened 1D DP array, 128×128 tiling, and wavefront (anti-diagonal) parallelization with `prange`. We evaluated performance (wall-clock), computed speedup against the baseline, and collected hardware performance counters (perf) to understand microarchitectural effects.

1 Problem and Approach

Smith–Waterman computes a local alignment score using dynamic programming with the recurrence:

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

We optimized the baseline Python implementation by:

- Flattening the 2D DP table to a contiguous 1D `int32` NumPy array to ensure spatial locality.
- Processing the matrix in 128×128 tiles so each tile fits in L2/L1 working set.
- Iterating over anti-diagonals (wavefronts) so that tiles on the same wavefront are independent and can be parallelized.
- Using Numba's `@njit(parallel=True, fastmath=True)` to JIT-compile and parallelize tile computation.

2 Optimization Methodology

2.1 Just-In-Time Compilation (Numba)

The primary bottleneck in the baseline implementation is the Python interpreter, which incurs heavy overhead for every loop iteration and integer operation in the $O(N^2)$ dynamic-programming computation. We use Numba's `@njit(fastmath=True, parallel=True)` to convert Python loops into optimized x86_64 machine code via LLVM. This enables:

- **Machine Code Generation:** Hot loops are JIT-compiled into efficient native code.
- **SIMD Auto-Vectorization:** With `fastmath`, LLVM can emit AVX2/AVX-512 vector instructions for integer arithmetic.
- **Thread-Level Parallelism:** The outer wavefront loop uses `prange`, enabling multi-thread execution.

2.2 Memory Layout and Tiling

The Smith–Waterman DP matrix has strong spatial locality, but a naïve Python list-of-lists layout results in:

- pointer chasing,
- non-contiguous rows,
- poor prefetcher utilization.

We replace it with a **1D flattened `int32` NumPy array**, ensuring contiguous, cache-friendly memory.

We then apply **128×128 Tiling**:

- Each tile (64 KB) fits comfortably inside L1/L2 cache.
- A tile computes all DP updates for that region before leaving cache.
- This drastically reduces L1 and TLB misses and increases arithmetic intensity.

2.3 Wavefront (Anti-Diagonal) Parallelism

The DP dependency pattern makes row-wise or column-wise parallelization impossible. However, tiles satisfy:

$$H_{i,j} \text{ depends only on tiles } (i-1,j), (i,j-1), (i-1,j-1)$$

Thus, all tiles where:

$$i + j = k$$

belong to the same independence group. This forms a **wavefront**.

We implement:

- **Wavefront iteration**: sweep diagonally across the matrix.
- **Parallel tile execution**: all tiles in a wavefront run via Numba's `prange`.
- **Per-wavefront reduction**: each thread writes its tile result into a temporary buffer, avoiding races.

This enables high-efficiency multicore execution while maintaining correctness.

3 Experimental Setup

- **System:** Intel Core i5-11400H @ 2.70GHz (6 physical cores, 12 logical threads).
- **OS:** WSL2 Linux (perf limits on some uncore counters).
- **Baseline:** Python implementation (pure Python + NumPy).
- **Optimized:** Python + Numba (JIT) optimized tiled wavefront code (‘sw_opt.py‘).
- **Datasets:** Sequence lengths $N = 1024, 2048, 4096$.
- **Repetitions:** Execution time medians computed from multiple runs (3–5 runs per configuration); perf counters averaged over 10 runs where available.

4 Timing Results and Speedup

Table 1 presents the measured median execution time for baseline and optimized code and the resulting speedup (Baseline / Optimized). Values are taken from `results_sw_times.csv` (rounded).

Table 1: Median runtimes and speedups (Baseline / Optimized).

N	Threads	Baseline (s)	Optimized (s)	Speedup
1024	1	0.52905	0.01108	47.74
1024	2	0.52764	0.00735	71.77
1024	4	0.52556	0.00513	102.45
1024	8	0.52511	0.00424	123.85
1024	12	0.52948	0.00392	135.21
2048	1	2.24297	0.04650	48.24
2048	2	2.21685	0.02638	84.03
2048	4	2.27336	0.01687	134.79
2048	8	2.21762	0.01467	151.13
2048	12	2.21926	0.01295	171.33
4096	1	9.43004	0.20057	47.02
4096	2	9.61709	0.10831	88.79
4096	4	9.30208	0.07354	126.48
4096	8	9.38067	0.05719	164.01
4096	12	9.34222	0.07382	126.55

4.1 Discussion

- The optimized Numba implementation yields very large speedups vs the Python baseline because the JIT compilation eliminates interpreter overhead and generates native machine code for the hot DP loops.
- Speedup increases with thread count up to around 8 threads (physical cores + some HT gain). At 12 threads (full logical threads) some configurations show a drop due to hyper-threading contention and synchronization overheads in wavefront barriers.
- Larger N gives better parallel efficiency (more work per tile), improving strong scaling until hardware limits are reached.

5 Perf Counter Results (Microarchitecture)

We collected perf counters using:

```
perf stat -e cycles,instructions,cache-references,cache-misses, \
  L1-dcache-load-misses,L1-icache-load-misses,LLC-load-misses -x, <cmd>
```

Measurements were repeated 10 times for both baseline (Python) and optimized (Numba) implementations for $N = 4096$, 4 threads; reported values below are averages across the runs (rounding shown).

Table 2: Average perf counters (10 runs) for $N = 4096$, 4 threads

Metric	Baseline (avg)	Optimized (avg)
Cycles	3.19×10^{10}	4.50×10^{7}
Instructions	1.10×10^{11}	5.16×10^{7}
IPC (instr/cycle)	**3.46**	**1.12**
Cache references	1.60×10^{8}	4.70×10^{6}
Cache misses	1.56×10^{7}	1.04×10^{6}
Cache miss rate	$\approx 9.7\%$	$\approx 22.1\%$
L1-dcache misses	8.0×10^{7}	2.05×10^{6}
L1-icache misses	2.23×10^{6}	1.44×10^{5}
LLC-load-misses	<not supported >	<not supported >

5.1 Interpretation

- **Cycles & Instructions:** The optimized kernel executes orders of magnitude fewer cycles and instructions than the Python baseline. This is the primary reason for the large wall-clock speedup: the Numba implementation removes Python interpreter overhead and generates tight native loops.
- **IPC:** The baseline shows higher IPC (≈ 3.4) because much of the baseline work runs inside vectorized or highly-optimized C code (NumPy internals). The optimized SW DP kernel, while executing far fewer instructions, is dependency-bound (each cell depends on neighbors), which limits instruction-level parallelism and reduces IPC (1.1).
- **Cache behaviour:** The optimized kernel has a higher raw cache-miss rate (cache-misses / cache-references) because the DP update and wavefront ordering produce non-unit-stride accesses and more temporal reuse at tile granularity; however, L1-dcache misses drop substantially in the optimized version thanks to tiling, indicating improved small-working-set locality.
- **WSL2 limitation:** LLC/uncore counters were not available in this environment and appear as "not supported"; for full memory hierarchy analysis run perf on native Linux.

6 Plots

The following figures were generated from the CSV results and perf data.

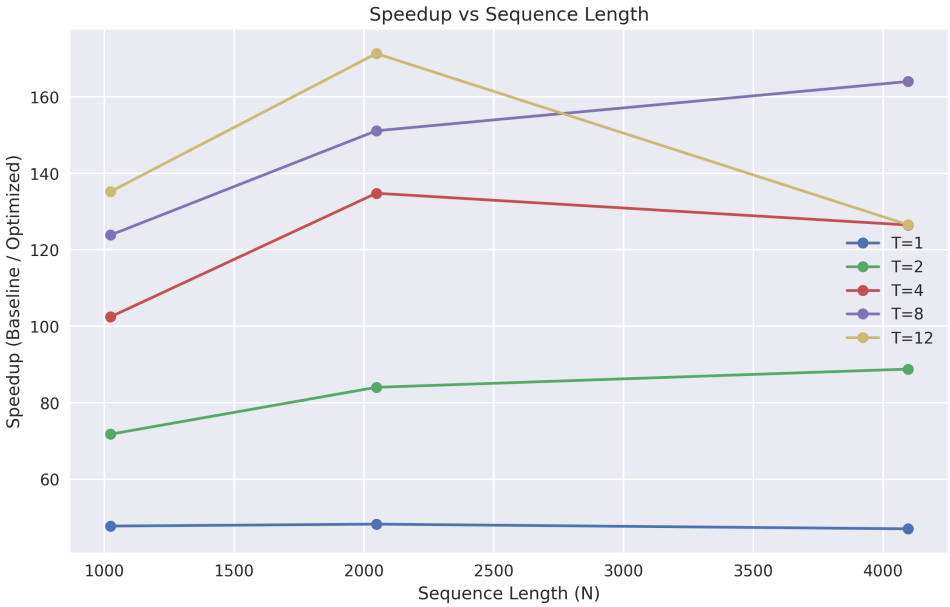

Figure 1: Speedup (Baseline/Optimized) vs sequence length for multiple thread counts.

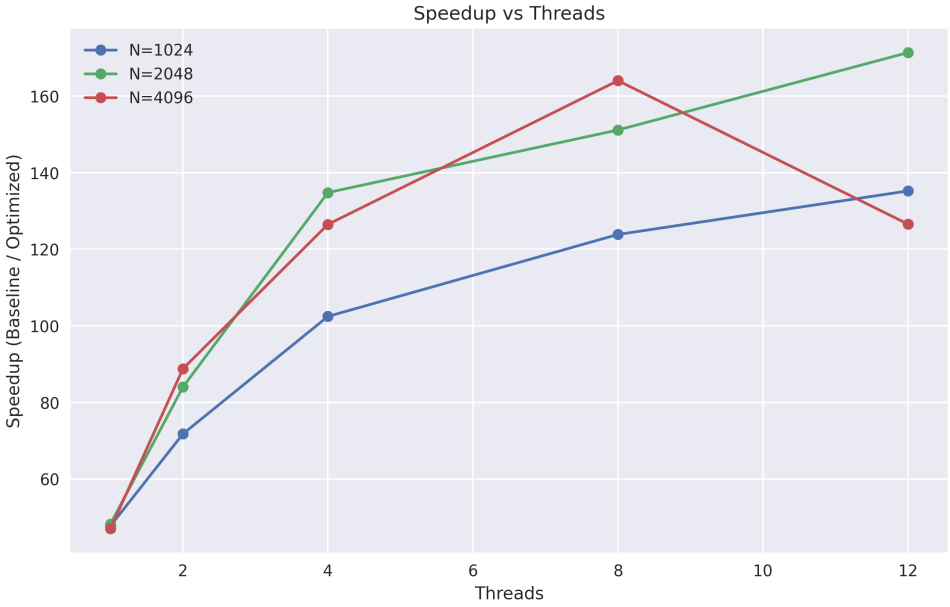

Figure 2: Speedup vs thread count for different sequence lengths.

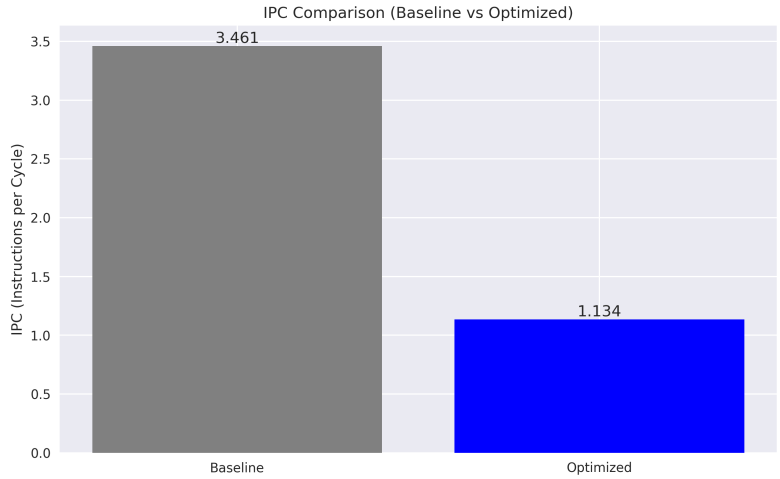

Figure 3: IPC comparison: Baseline vs Optimized (average across runs).

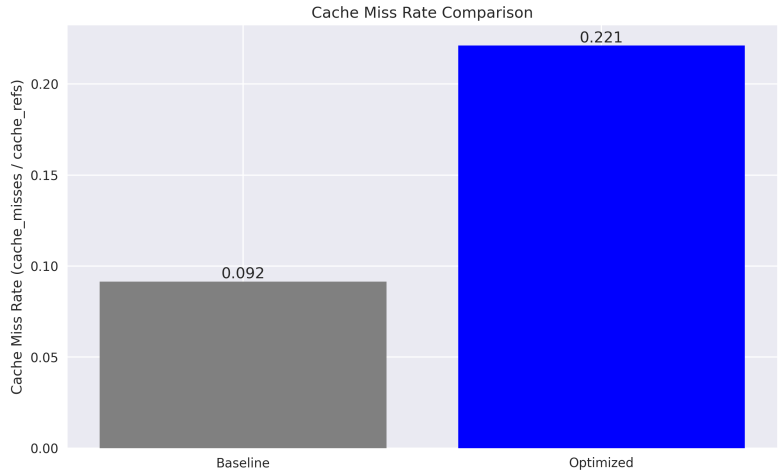

Figure 4: Cache miss rate comparison (cache-misses / cache-references).

7 Conclusions and Future Work

- Converting the Python baseline to a Numba JIT-compiled tiled wavefront implementation yields very large speedups (tens to hundreds of times), mainly by removing interpreter overhead and emitting efficient native code for the DP kernel.
- Tiling significantly reduces L1 miss activity and makes the inner loop operate on a compact working set. Wavefront parallelization allows multicore scaling while preserving correctness.
- The DP recurrence limits SIMD-style vectorization and forces dependency-bound execution; future work could explore register-blocked packed micro-kernels and explicit vector intrinsics (AVX2/AVX-512) on large servers that support them for further gains.
- For more complete microarchitectural insight (LLC and uncore counters) re-run perf on native Linux (professor's cluster), where the full PMU is available.