Memory-aware Optimization of Mass-univariate EEG Statistical Inference

Accelerating statistical inference in the Neurophysiological Biomarker Toolbox using data layout optimization, vectorization, and native execution

Pepijn van Egmond

Supervisor
Arthur-Ervin Avramiea

Responsible Professor
Ricardo Marroquim

1 Motivation

Electroencephalography (EEG) measures electrical brain activity and is widely used in neuroscience research and clinical studies of neurological disorders.

The Neurophysiological Biomarker Toolbox (NBT) supports EEG biomarker analysis, where many statistical tests are applied across biomarkers frequency bands, and source locations. Although a single statistical test is small, the total cost grows quickly as the analysis covers more dimensions.

This makes performance depend not only on arithmetic computation, but also on memory layout, data movement, SIMD efficiency, and parallel execution.

2 Background

Modern CPUs are fastest when data is accessed contiguously and reused from cache. If cohort values are separated by large strides, each statistical test may trigger more memory traffic and less efficient cache use.

Contiguous data also makes it easier for compiled kernels to use SIMD instructions, where one CPU instruction processes multiple values at once. Since many EEG statistical tests are independent, they can also be distributed across CPU cores for parallel execution.

The optimization therefore focuses on arranging the EEG data so cohort-based tests become cache-friendly, SIMD-friendly, and easier to parallelize.

3 Optimization Strategy

The baseline layout stores biomarker values contiguously, but this is poorly aligned with cohort-based testing. Reading all subject values for one test requires strided access across memory:

```
eeg_data[subject][freq][source][biomarker]
```

The optimized layout changes the dominant access pattern by storing subject values contiguously:

```
eeg_data[biomarker][source][freq][subject]
```

This improves spatial locality, reduces indirect indexing, and makes each test operate on compact subject buffers. As a result, the workload becomes more suitable for cache-efficient execution and compiled kernels.

Because the tests are independent, they can be distributed across CPU threads with limited synchronization. The contiguous buffers also expose regular loops that are easier to auto-vectorize and more likely to benefit from cache locality.

Vectorized Python
Groups many tests into larger **NumPy** operations, reducing Python-level loop and dispatch overhead while using **NumPy**'s compiled kernels.

Native Rust
Moves the performance-critical kernel into compiled Rust code using **PyO3** and **Maturin**, while **Rayon** provides explicit control over parallel execution.

Two optimized implementations were developed and compared against the original NBT baseline. The first remains within Python and improves how the workload is expressed using NumPy. The second moves the performance-critical computation into a native Rust backend to evaluate the benefit of compiled execution and explicit parallel control.

4 Benchmarking Setup

Each benchmark run uses a synthetic EEG dataset with configurable numbers of subjects, sources, frequency bands, and biomarkers. In the benchmark grid, the subject count ranges from **4** to **4096**, the number of sources is fixed at **100**, the number of frequency bands is fixed at **11**, and the biomarker count ranges from **1** to **64**. All configurations use an unpaired t-test and are executed with **2** warm-up runs followed by **5** measured runs. Runtime is measured directly around the statistical analysis call, while hardware counters are collected using `perf stat` in an isolated subprocess.The `perf` measurements include process-level activity such as Python startup,data loading, memory allocation, and result handling, so they are interpretedasd general indicators rather than exact staticial kernel only measurements.

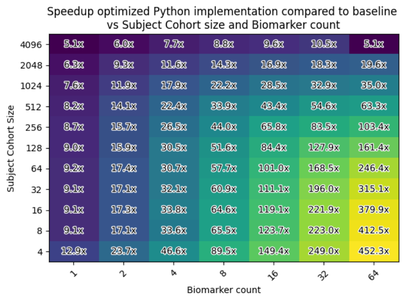

Figure 1: Speedup of the vectorized Python implementation relative to the baseline. The largest gains occur for small subject cohorts and many biomarkers.

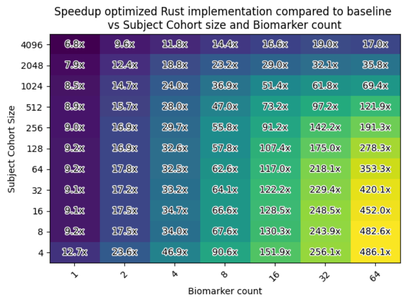

Figure 2: Speedup of the native Rust backend relative to the baseline. Rust achieves the highest overall speedup and remains strong at larger workloads.

5 Results

Baseline runtime scales mainly with biomarker count. Since each additional biomarker adds another set of independent statistical tests, runtime grows almost linearly in the baseline implementation.

After optimization, this growth becomes much weaker. Both vectorized Python and Rust execute many tests inside larger compiled kernels, so increasing the biomarker count adds far less Python-level overhead.

452.3x	486.1x	76.57s → 0.11s
Max speedup for vectorized Python	Max speedup for Rust kernel	Best absolute runtime improvement

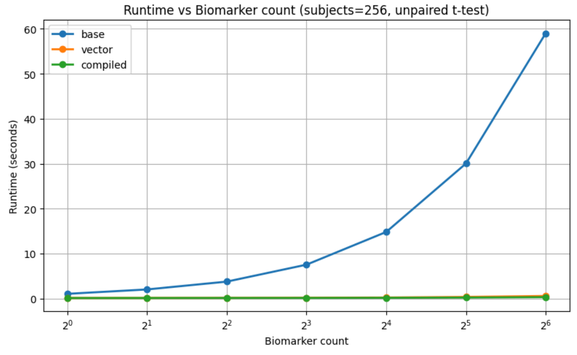

Figure 3: Runtime scaling with biomarker count. The optimized implementations show much weaker growth than the baseline.

6 Cache Behaviour

Cache metrics change only modestly, even though runtime improves by orders of magnitude, so the speedup is not explained by cache hit-rate improvements alone. The optimized implementations are much less sensitive to biomarker growth because many tests are processed inside larger compiled kernels. At higher biomarker counts, the Rust backend often benefits most from the improved layout, but the vectorized python implementation is not far behind.

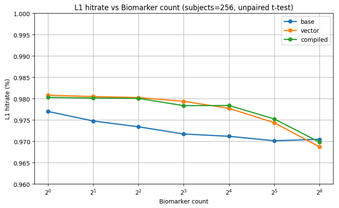

Figure 4: L1 cache behaviour: only small differences between implementations.

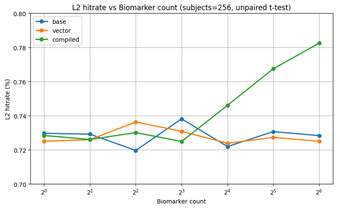

Figure 5: L2 cache behaviour: modest improvement for optimized kernels.

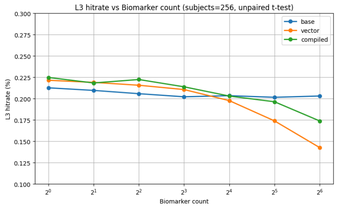

Figure 6: L3 cache behaviour: cache trends do not match the runtime gains.

7 SIMD and CPU

The optimized implementations expose more regular, contiguous work to compiled kernels. This increases SIMD density, because more operations can be executed as vectorized floating-point instructions rather than as many small scalar operations.

CPU utilization also improves, since independent biomarker-frequency-source tests can be distributed across multiple cores. The Rust backend benefits from explicit parallel execution, while the vectorized Python version gains from NumPy's native kernels.

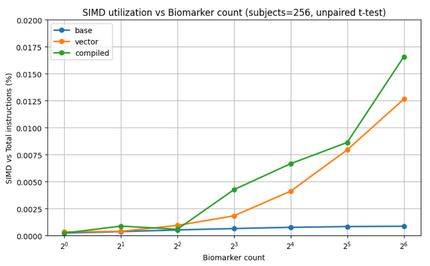

Figure 7: SIMD density across biomarker counts.
Native kernels expose more regular loops for vectorized floating-point execution.

8 Limitations

Hardware counters are collected with `perf` at process level, so they include setup costs such as Python startup, data loading, allocation, and result handling.

Measurements were also collected on a single hardware configuration, so cache behaviour and CPU utilization may differ on other processors.

Runtime is therefore the primary performance metric, while hardware counters are used as explanatory indicators.

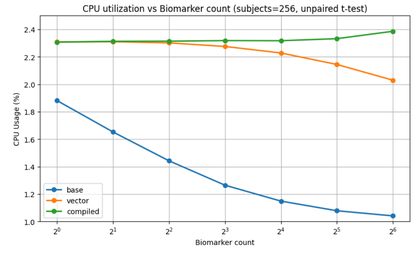

Figure 8: Effective CPU utilization across biomarker counts. Optimized implementations make better use of available cores. Unfortunately, process-level profiling overhead and serial setup work keep the measured overall CPU utilization modest.

9 Conclusion

Memory-aware restructuring substantially accelerates mass-univariate EEG statistical inference in NBT. **Largest gain:** replacing many small Python-level calls with larger batched NumPy kernels, which removes most baseline overhead and makes runtime much less sensitive to biomarker growth. Rust adds further speedup through native execution, explicit memory-layout control, and parallelization, but the main bottleneck is already reduced by vectorization and batching.

Future work: Finer profiling, handwritten SIMD, additional statistical tests, and GPU acceleration.