Memory-aware optimization of mass-univariate statistical inference on EEG datasets

Accelerating the statistical testing pipeline of the Neurophysiological Biomarker Toolbox using memory-aware data layouts, vectorization, and native execution

Pepijn van Egmond
Supervisor: Arthur-Ervin Avramiea
Responsible professor: Ricardo Marroquim
EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 20, 2026

Name of the student: Pepijn van Egmond
Final project course: CSE3000 Research Project
Thesis committee: Ricardo Marroquim, Arthur-Ervin Avramiea, Thomas Abeel

An electronic version of this thesis is available at http://repository.tudelft.nl/.

1 Abstract

This paper investigates memory-aware optimization of mass-univariate EEG statistical inference in the Neurophysiological Biomarker Toolbox. A vectorized Python implementation and a native Rust backend are evaluated as optimized alternatives to the existing NumPy/SciPy-based statistical testing pipeline. The optimized implementations reorganize EEG biomarker data for cohort-based access, improving support for cache locality, SIMD execution, and parallel processing. Synthetic benchmarks show speedups of up to 452.3x for the vectorized Python implementation and up to 486.1x for the Rust backend. The optimized implementations also substantially reduce sensitivity to increasing biomarker counts, resulting in much weaker runtime growth across the measured benchmark space. Profiling shows increased SIMD density and CPU utilization, while cache behaviour improves only modestly. These results suggest that the primary limitation is not the statistical operation itself, but the overhead introduced by how the workload is structured and executed. Much of the available speedup can therefore be achieved by expressing the computation as larger batched and vectorized operations.

2 Introduction

Electroencephalography (EEG) is a non-invasive technique for measuring electrical activity in the brain and is widely used in neuroscience research, brain–computer interfaces, and clinical diagnostics. It plays an important role in the study and monitoring of neurological and neurodevelopmental disorders such as Parkinson's disease[1] , Alzheimer's disease [2], attention-deficit/hyperactivity disorder (ADHD) [3], and autism spectrum disorder [4], as well as in evaluating treatment effects and medication responses[5].

EEG recordings are high-dimensional, noisy, and weakly structured. As a result, meaningful signal extraction typically relies on aggregating large numbers of derived biomarkers and applying statistical hypothesis testing across multiple. This leads to workloads that are computationally light per operation but large in aggregate. This suggests that EEG statistical analysis pipelines are not only shaped by arithmetic computation, but also by memory access patterns and data movement through the memory hierarchy.

The Vrije Universiteit Amsterdam is developing an EEG research platform based on the Neurophysiological Biomarker Toolbox (NBT) to streamline and standardize EEG analysis workflows. As the platform is intended to support large-scale biomarker analysis, the efficiency of its statistical analysis pipeline becomes increasingly important. This motivates the investigation of memory-aware optimization strategies that can improve runtime performance while preserving the existing workflow and numerical results.

2.1 Section overview

This paper is structured as follows. Section 3 introduces the relevant background on EEG statistical testing, memory layout, SIMD, and parallel execution. Section 4 describes the optimization strategy, focusing on data reshuffling and memory-aware computation. Section 5 presents the Python and Rust implementations. Section 6 explains the benchmark setup and measured performance metrics. Section 7 presents the results, including runtime, cache behaviour, SIMD usage, and CPU utilization. Section 8 discusses the result, and Sections 9 and 10 present the conclusions, future work, and responsible research considerations.

3 Background

3.1 Statistical Analysis of EEG Recordings

After preprocessing and artifact removal, EEG recordings are transformed into a collection of derived biomarkers that summarize relevant properties of the signal. Examples include spectral power within frequency bands, functional connectivity measures, and complexity metrics. Statistical analysis is then applied to determine whether these biomarkers differ significantly between subject groups.

Statistical testing is then used to determine whether observed differences in these biomarkers are likely to reflect meaningful effects rather than random variation. For each biomarker, a hypothesis test, such as a t-test, can compare measurements between groups or conditions. For example, such a test may evaluate whether alpha-band power differs between a control group and a patient group, or whether a connectivity measure changes between experimental states. Statistical analysis in EEG pipelines therefore consists of repeatedly applying hypothesis tests across multidimensional biomarker data. This process converts large collections of derived signal features into interpretable evidence about group differences, condition effects, or relationships between neurological measurements.

3.2 Memory Layout and Locality

Modern processors execute arithmetic operations significantly faster than data can be transferred from main memory. Therefore, the organization of data in memory is a critical factor in the performance of numerical workloads. High performance depends on temporal locality, where recently accessed data is reused, and spatial locality, where neighbouring memory locations are accessed consecutively. These patterns allow cache lines to be used efficiently, reduce accesses to lower memory levels, and improve hardware prefetching.

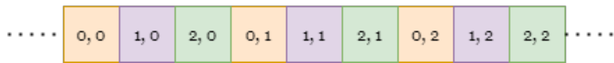

Figure 1: C-order two-dimensional array layout[1]

NumPy is the de facto standard library for scientific array computation in Python [6]. In NumPy[2], arrays are represented using the ndarray structure[3], which combines a memory region with metadata describing the array's shape, data type, and strides. Strides determine how multidimensional indices are mapped to physical memory locations. As shown in Figure 1 and Figure 2, the same two-dimensional array can be stored with different contiguous access directions.

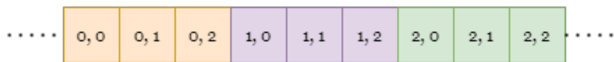

Figure 2: Fortran-order two-dimensional array layout

This distinction is important for EEG workloads. Statistical analysis repeatedly applies tests across the subject dimension for many combinations of biomarkers, frequency bands, and sources. If subject values for a test are stored contiguously, they can be loaded into cache efficiently and processed with fewer memory loads. If they are separated by large strides, each test touches distant memory locations, increasing cache misses, memory traffic, and address-computation overhead.

Modern processors can execute arithmetic operations at a much higher rate than data can often be supplied from memory. Performance therefore depends not only on the number of floating-point operations, but also on the amount of data movement required per operation. This relationship is commonly described by the Roofline model, which relates attainable performance to arithmetic intensity and memory bandwidth [7].

[1]Both memory layout diagrams are sourced from: https://manik.cc/2021/02/25/memory-order/

[2]https://numpy.org/

[3]https://numpy.org/doc/stable/dev/internals.html

3.3 SIMD

SIMD instructions allow a single instruction to operate on multiple values simultaneously. For example, a modern AVX2 register can process eight 32-bit floating-point values in parallel. SIMD is most effective inside compact computational kernels, where the same operation is repeatedly applied to contiguous data. In such cases, compilers and numerical libraries can often apply auto-vectorization, generating SIMD instructions without requiring explicit vector intrinsics in the source code. However, auto-vectorization depends strongly on predictable control flow and regular memory access patterns. Irregular indexing, strided access, or data-dependent branches can prevent vectorization or require less efficient gather operations and temporary data rearrangement.

3.4 Parallel Execution

Parallelism provides a second source of speedup. EEG statistical analysis is naturally parallelizable because many tests are independent: different biomarkers, sources, or frequency bands can often be evaluated concurrently across multiple cores with limited synchronization. Parallel execution is however often constrained by memory behaviour and coordination overhead. Threads have to compete for shared cache capacity and memory bandwidth, while shared data structures may require locking that serializes part of the computation. [8] As a result, more threads do not necessarily produce proportional speedup. Efficient parallel execution therefore requires layouts and data structures that preserve locality, reduce cache contention, and avoid unnecessary locking.

3.5 Research Question

This research is guided by the following main research question:

How can memory-aware data layout optimization and native execution improve the performance of mass-univariate, cohort-based EEG statistical inference in NBT, and how do these changes affect cache locality, SIMD vectorization, and parallel scalability?

To answer this question, the project considers the following subquestions:

- **RQ 1:** To what extent can memory-aware data layout optimization reduce the runtime and scaling behaviour of mass-univariate EEG statistical inference?
- **RQ 2:** How do vectorized Python execution and native compiled Rust execution compare in terms of runtime, scalability, and implementation trade-offs?
- **RQ 3:** How do the optimized implementations affect processor-level behaviour, including cache locality, SIMD utilization, and parallel CPU utilization?

4 Methodology

4.1 Computational Optimization Strategy

Memory Layout Transformation

The baseline NBT data layout is:

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

This representation stores biomarker values contiguously in memory. While suitable for feature-oriented operations, it is inefficient for cohort-based statistical testing, where traversal of the subject dimension dominates execution time. The tensor is therefore transformed into

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

This layout places subject values contiguously in memory. As a result, statistical tests can access cohort values using sequential memory reads. To further reduce indirect memory access, the reshuffled tensor is partitioned into separate contiguous regions for each cohort. This allows each statistical comparison to operate directly on two compact subject arrays instead of repeatedly indexing into the original tensor. Although reshuffling introduces a one-time memory copy, the cost is amortized across the large number of statistical tests performed on the transformed dataset.

Parallel Execution

The statistical computations performed for individual biomarker-frequency-source combinations are independent and therefore highly parallelizable. Independent batches can be distributed across multiple CPU threads, with each thread operating on disjoint memory regions. This minimizes contention and avoids synchronization overhead during computation. As a result, only a final reduction step is required to combine results. By assigning contiguous memory regions to individual threads, cache locality is further improved while maintaining high CPU utilization across available processing cores.

SIMD Vectorization

SIMD (Single Instruction Multiple Data) execution enables modern processors to perform the same arithmetic operation on multiple values simultaneously. The transformed memory layout makes sure that subject values are stored contiguously, making them suitable for vectorized execution.

5 Technical Implementation

Three implementations are evaluated in this work: the baseline NBT implementation, an optimized Python implementation, and a Rust implementation exposed through a Foreign Function Interface (FFI). The optimized Python implementation improves performance by restructuring the computation to leverage NumPy more effectively. The Rust implementation applies the same general optimization principles, but moves the performance-critical computation into custom native code to obtain greater control over memory layout and parallel execution. Both implementations are tested against the baseline for numerical equivalence.

5.1 Motivation for Rust

The optimized Python implementation demonstrates how far performance can be improved by making better use of NumPy's vectorized operations. Rust was selected for the optimized backend because it provides native execution performance while allowing explicit control over data layout and memory access patterns. Unlike Python, Rust code is compiled ahead of time and does not rely on interpreter dispatch during the statistical computation. Compared with C or C++, Rust additionally provides strong memory-safety guarantees through its ownership and borrowing model. The Rust implementation therefore complements the optimized Python version: NumPy is used to evaluate the benefit of vectorization within Python, while Rust is used to evaluate the additional benefit of native implementation with explicit memory-layout and parallelization control.

5.2 Computation Kernel

The Rust backend is organized around a computation kernel with two stages: data reshuffling with cohort extraction, followed by parallel statistical computation. In the first stage, the input tensor is transformed from the original NBT layout into a representation optimized for cohort-based statistical testing. Subject values are stored contiguously, and the selected cohorts are copied into compact buffers. This avoids repeated indirect indexing and provides a regular memory access pattern for the statistical kernel.

In the second stage, statistical tests are performed for each biomarker-frequency-source combination. Since these tests are independent, they can be distributed across CPU threads using Rayon[4], a data-parallel Rust runtime based on parallel iterators and work stealing. Each worker processes a disjoint subset of the transformed data and writes results into preallocated output arrays. This avoids shared mutable state in the inner computation loop and limits synchronization to the final result aggregation. The pseudocode algorithms for this implementation are provided in Appendix Section C.

5.3 Python Integration

The Rust implementation is exposed to Python using PyO3[5] and packaged using Maturin[6]. This design is inspired by previous work on accelerating tensor computations by moving performance-critical kernels from Python into Rust, while preserving a Python-

[4]https://github.com/rayon-rs/rayon

[5]https://github.com/pyo3/pyo3

[6]https://github.com/pyo3/maturin

facing interface [9]. In the same way, the optimized backend can be imported as a normal Python module and integrated into the existing NBT workflow. High-level configuration, cohort selection, experiment setup, and pipeline orchestration remain implemented in Python. Only the performance-critical data reshuffling and statistical computation are moved into Rust.

5.4 Optimized Python Implementation

An optimized Python implementation was developed to measure how much performance can be gained without leaving the Python ecosystem. It preserves the existing Python interface but restructures the workload to use NumPy more effectively. Instead of executing many small statistical tests through Python level loops, the data is reshaped and grouped so that larger batches can be processed by NumPy's compiled native kernels. This reduces interpreter overhead, improves opportunities for vectorized execution, and provides an intermediate comparison point between the baseline implementation and the Rust backend.

6 Evaluation

6.1 Benchmark Execution

Each benchmark run executes the full NBT statistical analysis pipeline on a synthetic EEG dataset with configurable dimensions. The number of subjects, sources, frequency bands, and biomarkers can be varied independently, allowing the workload to be scaled systematically. Each benchmark configuration is executed repeatedly. For every configuration, two warm-up runs and five measured runs are performed, after which the mean runtime is reported. The exact benchmarking hardware and software environment is reported in Appendix Section D

6.2 Isolated Process Execution

Hardware performance counters are collected using perf stat[7]. Since perf measures activity at process level, each benchmark is executed in an isolated subprocess. For every run, the configured pipeline and dataset are serialized, reconstructed in a temporary Python script, and executed under perf stat. This setup makes sure that the collected counters correspond to a single benchmark workload. The Python garbage collector is disabled during the measured run to reduce avoidable runtime noise.

6.3 Experimental Variables

The synthetic dataset is parameterized by the number of subjects (S_{sub}), sources (S_{src}), biomarkers (B), and frequency bands (F). For a dataset with S_{src} sources, F frequency bands, and B biomarkers, the number of statistical tests is

$$N_{\text{tests}} = S_{\text{src}} \times F \times B$$

[7]https://perfwiki.github.io/main/

The subject count does not change the number of tests, but it does affect the cost of each test. In the evaluation, F is fixed at 11 and S_{src} at 100 as reasonable baselines. The benchmark then varies $S_{\text{sub}} = [4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]$ and $B = [1, 2, 4, 8, 16, 32, 64]$ Using powers of two to expose scaling behaviour.

6.4 Statistical test type

All benchmark configurations use an unpaired t-test, which compares the means of two independent subject groups. This is representative of EEG cohort-level analysis, such as comparing control and patient groups or two experimental conditions. The evaluation focuses on a single test type to keep the benchmark space manageable.

6.5 Metrics

The benchmark records both runtime and hardware performance counters. Runtime is measured directly around the statistical analysis call and is used as the primary performance metric, since it most closely captures the execution time of the workload under evaluation. The `perf` counters are used to compute derived metrics for processor behaviour, including L1, L2, and L3 cache behaviour, CPU utilization, and SIMD density. The collected perf events are listed in Appendix Section A, and the derived formulas are given in Appendix Section B.

6.6 Runtime scope

Because `perf` operates at process level, its counters are collected over the full subprocess rather than only the statistical kernel. They include interpreter startup, data loading, and other framework overhead. This distinction is important because the statistical kernel can represent only a small fraction of total subprocess wall time. Therefore, runtime is used to compare kernel performance, while the hardware counters are interpreted as general indicators rather than exact measurements of the statistical computation alone.

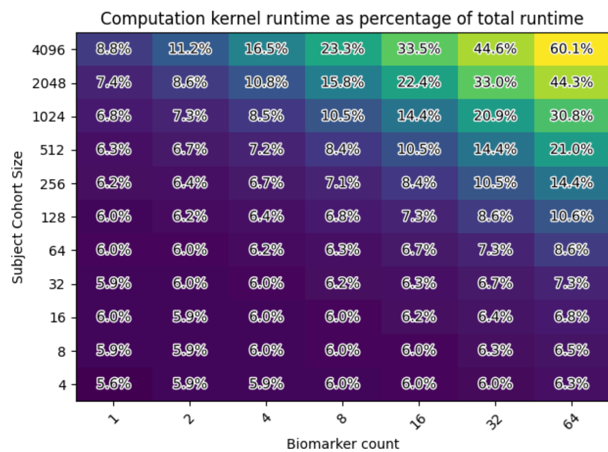

Figure 3: Share of total subprocess wall time spent inside the measured statistical analysis call.

7 Results

7.1 Runtime

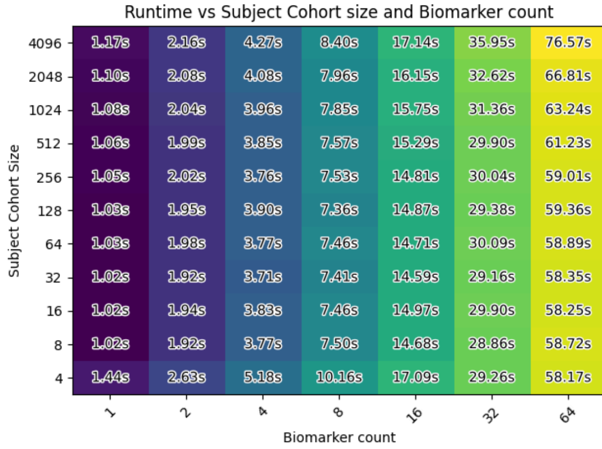

Figure 4: Baseline runtime across subject cohort size and biomarker count. Each cell reports the mean runtime in seconds for one benchmark configuration.

Figure 4 shows the runtime of the baseline implementation across the explored benchmark space. The strongest trend is along the biomarker dimension. For most subject cohort sizes, doubling the biomarker count approximately doubles the runtime. For example, at 128 subjects, runtime increases from 1.03 s for 1 biomarker to 1.95 s for 2 biomarkers, 3.90 s for 4 biomarkers, 7.36 s for 8 biomarkers, this trend continues up to 64 biomarkers.

The effect of subject cohort size is smaller in comparison. Only for 64 biomarkers a meaningful difference starts to occur, from 58.17 s at 4 subjects to 76.57 s at 4096 subjects. This indicates that, in the measured range, the baseline implementation is affected more strongly by the number of statistical tests than by the number of subjects per test.

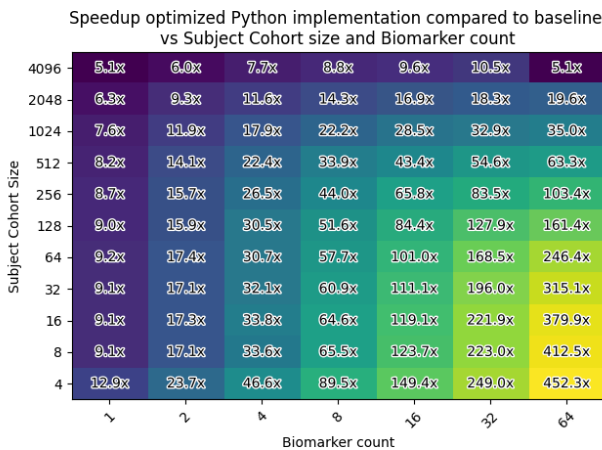

Figure 5: Speedup of the optimized Python implementation relative to the baseline across subject cohort size and biomarker count.

Figure 5 shows the speedup of the optimized Python implementation relative to the baseline. The vectorized implementation outperforms the baseline in every measured configuration. Speedup ranges from 5.1x at 4096 subjects and 1 biomarker to 452.3x at 4 subjects and 64 biomarkers.

The largest speedups are observed at smaller subject cohort sizes and larger biomarker counts. For example, at 4 subjects the speedup increases from 12.9x at 1 biomarker to 452.3x at 64 biomarkers. At 8 subjects, it increases from 9.1x to 412.5x. At higher subject counts, the speedup remains substantial but is less extreme: at 4096 subjects, it ranges from 5.1x to 10.5x for biomarker counts up to 32, and reaches 5.1x at 64 biomarkers.

Overall, the optimized Python implementation achieves its largest speedups for configurations with many biomarkers and smaller subject cohorts.

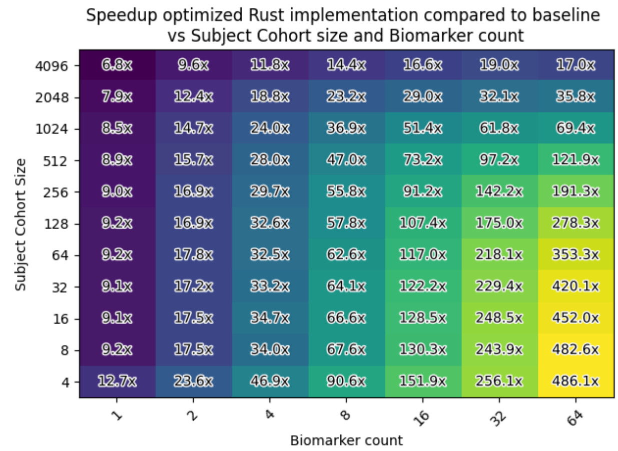

Figure 6: Speedup of the optimized Rust implementation relative to the baseline across subject cohort size and biomarker count.

Figure 6 shows the speedup of the optimized Rust implementation relative to the baseline. The Rust implementation also outperforms the baseline in every measured configuration. Speedup ranges from 6.8x at 4096 subjects and 1 biomarker to 486.1x at 4 subjects and 64 biomarkers.

The same global pattern is visible as for the vectorized Python implementation: speedup generally increases as biomarker count increases, and the largest improvements occur for smaller subject cohorts with many biomarkers.

Compared with the optimized Python implementation, the Rust implementation provides a small additional speedup at larger biomarker counts and medium-to-large subject cohort sizes. However, the difference between the optimized Python and Rust implementations is much smaller than the difference between either optimized implementation and the baseline.

8 Discussion

The heatmaps in the results section provide a broad overview of the benchmark space, but detailed trends are difficult to extract from a two-dimensional visualization. Therefore, the following discussion focuses on the measurements taken at a fixed subject count of 256 while varying the number of biomarkers. This configuration is large enough to show measurable differences between implementations, while still being representative of realistic EEG cohort-level workloads.

8.1 Runtime Scaling

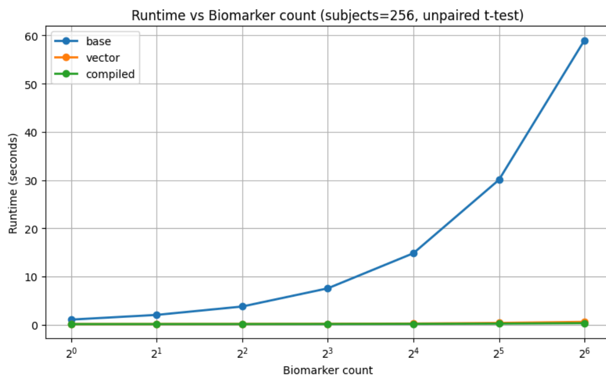

Figure 7: The log scale of the x-axis makes linear the scaling of the

Runtime scaling was examined by fixing the subject cohort size and varying the biomarker count. Since the number of statistical tests is proportional to the number of biomarkers, the baseline implementation is expected to scale approximately linearly with B, where B is the biomarker count. This behaviour is visible in Figure 7: the baseline runtime grows close to linearly over the measured range. Both optimized implementations show substantially weaker runtime growth. Runtime remains comparatively stable for small and medium biomarker counts and increases more noticeably only at larger problem sizes. This suggests that parallelization and vectorization reduce the effective overhead of increasing B, even though the number of statistical comparisons still grows linearly in principle. Therefore, the optimized implementations should not be interpreted as having true $O(1)$ complexity, but rather as having a much smaller constant factor and better practical scaling within the measured range. The optimized Python implementation already removes most of the baseline overhead, while the Rust implementation provides the lowest runtime in most measured configurations, which comes out the most at higher counts.

8.2 Cache Utilization

Figure 8, Figure 9, and Figure 10 show only modest changes in cache behaviour between implementations. Although the optimized implementations achieve large runtime reductions, the measured L1, L2, and L3 metrics do not improve by a comparable amount. This suggests that cache locality alone does not explain the speedup. The memory layout transformation remains important because it enables contiguous access and larger batched operations, but the dominant improvement appears to come from reducing Python-level iteration and executing more work inside optimized native kernels. At larger biomarker counts, cache behaviour is not always better for the optimized implementations, likely because batching and vectorization process more data per operation and can increase pressure on lower cache levels. The runtime results nevertheless improve substantially, indicating that reduced overhead and improved vectorized execution outweigh these cache effects.

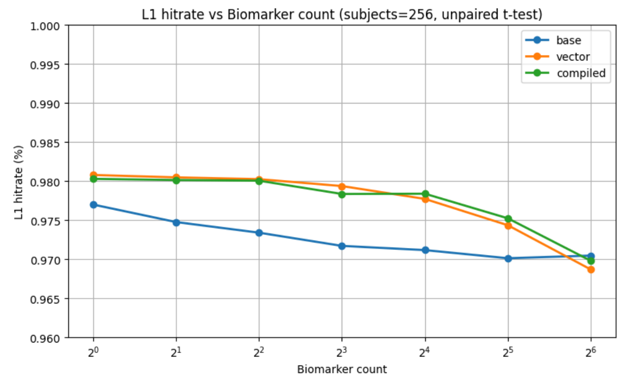

Figure 8: L1 cache behaviour, note the y-axis scale being very small

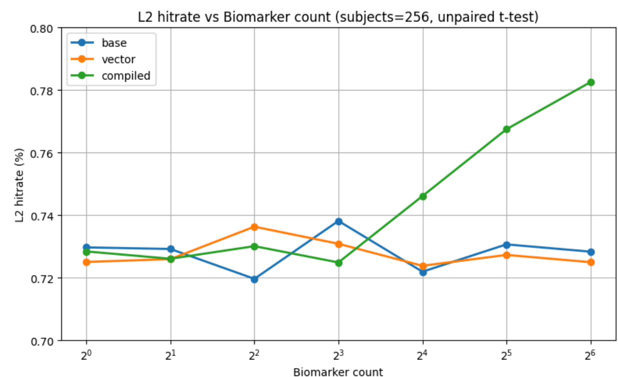

Figure 9: L2 cache behaviour showing compiled implementation outperforming at higher biomarker counts

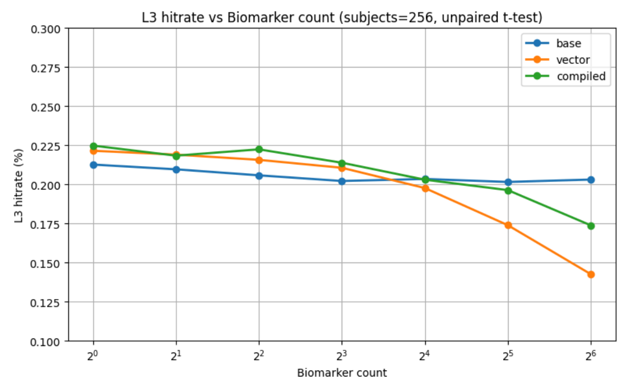

Figure 10: L3 cache behaviour, baseline appears the most consistent

8.3 SIMD Utilization

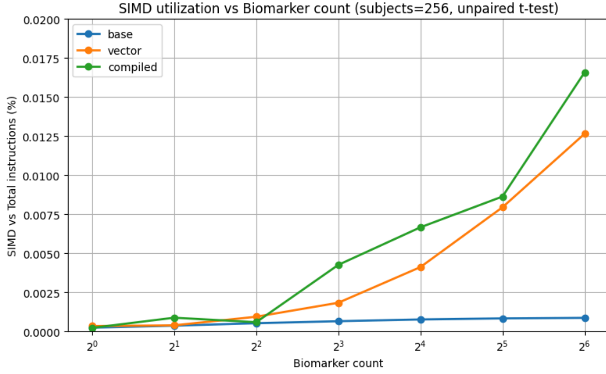

Figure 11: SIMD density at a fixed subject count of 128.

Figure Figure 11 shows that both optimized implementations achieve higher SIMD density than the baseline. This supports the main optimization strategy: by reshaping the computation into larger contiguous operations, more of the workload can be executed using vectorized machine-level instructions.

The baseline implementation performs many small statistical operations through Python-level iteration. Even when individual NumPy or SciPy calls use optimized native code internally, the surrounding loop structure limits how much work is available to each native kernel invocation. The vectorized Python implementation improves this by batching many tests into larger NumPy operations. The Rust implementation further exposes the computation to compiled native code with explicit control over memory layout and parallel iteration.

8.4 Parallelization

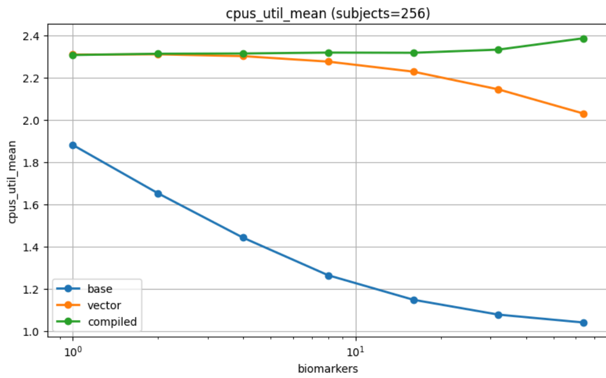

Figure 12: Effective CPU utilization at a fixed subject count of 128.

Figure Figure 12 shows that the optimized implementations make more effective use of available CPU resources than the baseline. This is expected because statistical tests for different biomarker, frequency, and source combinations are independent and can therefore be distributed across multiple cores.

However, the measured CPU utilization remains modest compared with the available hardware parallelism. This should be interpreted with the process level nature of the `perf` measurements in mind. The reported utilization includes both parallel computation and serial parts of the benchmark process, such as interpreter startup, data preparation, memory allocation, reshuffling, and result aggregation.

According to Amdahl's Law [8], the maximum speedup from parallel execution is limited by the fraction of execution that remains serial. Since the measured process includes both serial and parallel regions, the reported CPU utilization should be treated as a conservative estimate of the parallelism achieved inside the statistical kernel itself.

8.5 Rust compared with optimized Python

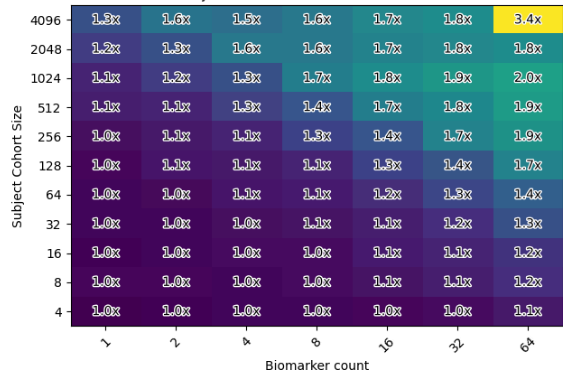

Figure 13: Speedup of the Rust implementation relative to the vectorized Python implementation.

Figure Figure 13 compares the Rust implementation directly against the vectorized Python implementation. The Rust backend provides additional speedup in several configurations, but the difference is smaller than the improvement from the baseline to the vectorized implementation.

This indicates that the largest bottleneck in the baseline is the repeated Python-level expression of the workload rather than the arithmetic cost of the statistical test itself. Once the computation is batched into larger NumPy operations, much of the available speedup is already achieved. The Rust implementation still provides benefits through native compilation, explicit memory layout control, and parallel execution, but these improvements are applied after the main bottleneck has already been reduced by offloading much of the work to NumPy's compiled kernels.

The results suggest that the optimized implementations improve performance through a combination of reduced interpreter overhead, larger batched operations, increased SIMD usage, and better use of parallel execution. Cache locality contributes by enabling these changes, but the cache metrics do not indicate that cache behaviour alone explains the observed runtime reduction.

8.6 Limitations

Several limitations should be considered when interpreting these results.

Statistical Test Type

The evaluation focuses on a single statistical test: the unpaired t-test. This keeps the benchmark space manageable and allows implementations to be compared under identical conditions. However, NBT may support additional statistical procedures with different computational and memory-access characteristics. The results therefore primarily describe the behaviour of the optimized pipeline for this specific class of mass-univariate statistical testing.

Measurement Methodology

The hardware performance counters are collected at process level using `perf`. These measurements include the full benchmark subprocess, including Python interpreter startup, deserialization, data loading, memory allocation, data preparation, reshuffling, and result handling. As shown in Figure 3, the measured statistical call can represent only a small fraction of the total subprocess runtime, especially for faster optimized configurations. Some observed counter values may therefore be influenced by data loading and setup behaviour rather than by the statistical computation itself. This limits how precisely CPU utilization, SIMD density, and cache behaviour can be attributed to the optimized kernel.

Hardware Dependence

The measurements were collected on a single hardware configuration Table 3. Cache sizes, memory bandwidth, SIMD support, core count, frequency scaling, and CPU microarchitecture can all influence the relative performance of the implementations. As a result, the absolute runtimes and some hardware-counter trends and therefore conclusions may differ on other systems.

9 Conclusions and Future Work

This research investigated whether memory-aware optimization can improve mass-univariate EEG statistical analysis in the Neurophysiological Biomarker Toolbox while preserving its Python-based workflow. The baseline NumPy implementation was compared with a vectorized Python implementation and a Rust backend exposed through PyO3. The baseline scaled primarily with biomarker count, increasing from roughly one second at one biomarker to around one minute at 64 biomarkers, with a maximum runtime of 76.57 s. Both optimized implementations substantially reduced this cost: the vectorized Python implementation reached speedups of up to 452.3x, while the Rust backend reached up to 486.1x. Most of the improvement came from replacing repeated Python-level iteration with larger batched NumPy operations. The Rust backend provided the best performance in many configurations, but its improvement over vectorized Python was smaller than expected, suggesting that much of the available speedup can already be achieved through better vectorization and offloading within Python. Profiling showed increased SIMD density and CPU utilization, while cache metrics improved only modestly. Overall, the results indicate that memory-aware restructuring is effective not simply because it improves cache hit rates, but because it enables larger contiguous computational kernels, better SIMD usage, reduced overhead, and more efficient parallel execution.

9.1 Future Work

Fine-grained benchmarking

Future work should include more fine-grained benchmarking of the statistical kernel itself. Profiling is currently down at a process level, code level profiling would make it possible to attribute instruction count, SIMD usage, cache behaviour, memory bandwidth, and CPU utilization more directly to the optimized computation.

Handwritten SIMD

The current implementation mainly relies on compiler auto-vectorization and optimized library kernels. Future work could explore handwritten SIMD kernels for the statistical tests to gain more control over vector register usage, reductions, and memory loads. This may improve performance further, but would increase implementation complexity and reduce portability.

Additional statistical tests

The benchmark space could be expanded beyond the unpaired t-test. Other statistical procedures may have different computational structure, memory-access behaviour, and opportunities for vectorization or parallelization. Evaluating additional tests would show whether the observed speedups generalize across the wider NBT statistical pipeline, or whether the benefits are specific to this particular test type.

Streaming and chunked execution

Future work could investigate streaming or chunked execution for larger EEG datasets. The current optimized implementations assume that the relevant data can be materialized in memory before computation since that is the case in NBT. Processing the dataset in smaller batches could reduce peak memory usage and make the approach more scalable for workloads that exceed available memory.

GPU acceleration

The workload consists of many independent statistical tests, making it a possible candidate for GPU acceleration. This would require careful batching and memory-layout design however, since transfer overhead between CPU and GPU memory may dominate for smaller workloads.

10 Responsible Research

10.1 Ethical considerations

This research did not directly involve human participants, and no experiments on human subjects were conducted as part of this project. The datasets used for the first iteration of testing before synthetic data was produced originates from previously conducted EEG studies and were provided by the Vrije Universiteit Amsterdam in a preprocessed and anonymized form. No personally identifiable information was available to the author. Improving EEG processing performance can reduce the computational cost of large scale analyses and help support research into neurological disorders. However, neurological data remains sensitive and should be handled in accordance with ethical, legal, and scientific standards. The results of such analyses should not be used for discriminatory purposes or decisions that violate individual rights or privacy.

10.2 Reproducibility

The source code of the Neurological Biomarker Toolbox is not currently publicly available. Because the implementation developed in this project depends closely on internal NBT interfaces, the project code cannot be published independently at this stage. To improve reproducibility, the development environment was defined using Nix, a reproducible package manager and build system [10]. While the required tools can be installed without Nix, doing so weakens the guarantee that the same compiler versions, dependencies, and build configuration are used. The final benchmark uses synthetic data, so it does not depend on access to the original EEG datasets. However, full reproducibility remains limited by hardware-dependent `perf` counters and noisy memory measurements, which are affected by CPU architecture, background activity, OS configuration, frequency scaling, and other environment-specific factors.

10.3 AI usage disclosure

During this project, large language model tools were used to support grammar revision, code review, library API exploration, and understanding parts of the NBT code structure. These tools were not used as autonomous coding agents, and all generated suggestions were reviewed and validated by the author. The use of such tools does not remove the author's responsibility for the correctness and integrity of the work presented.

11 Acknowledgements

I would like to express my sincere gratitude to my supervisor, Ricardo Marroquim, and to my co-supervisor, Arthur-Ervin Avramiea, for their guidance, support, and valuable feedback throughout this project. I would also like to thank the Vrije Universiteit Amsterdam for providing the opportunity to conduct this research and for supporting collaboration within their EEG research environment.

12 References

[1] M. Allahbakhshi, A. Sadri, and S. O. Shahdi, "Diagnosis of Parkinson's Disease Using EEG Signals and Machine Learning Techniques: A Comprehensive Study." [Online]. Available: https://arxiv.org/abs/2405.00741

[2] R. Cassani, M. Estarellas, R. San-Martin, F. J. Fraga, and T. H. Falk, "Systematic Review on Resting-State EEG for Alzheimer's Disease Diagnosis and Progression Assessment," *Disease Markers*, vol. 2018, no. 1, p. 5174815, 2018, doi: https://doi.org/10.1155/2018/5174815.

[3] M. Adamou, T. Fullen, and S. L. Jones, "EEG for Diagnosis of Adult ADHD: A Systematic Review With Narrative Analysis," *Frontiers in Psychiatry*, p. 5174815, 2020, doi: https://doi.org/10.3389/fpsyt.2020.00871.

[4] J. Wang, J. Barstein, L. E. Ethridge, M. W. Mosconi, Y. Takarae, and J. A. Sweeney, "Resting state EEG abnormalities in autism spectrum disorders," *Journal of Neurodevelopmental Disorders*, vol. 5, no. 1, p. 24, 2013, doi: 10.1186/1866-1955-5-24.

[5] D. Spronk, M. Arns, K. Barnett, N. Cooper, and E. Gordon, "An investigation of EEG, genetic and cognitive markers of treatment response to antidepressant medication in patients with major depressive disorder: A pilot study," *Journal of Affective Disorders*, vol. 128, no. 1, pp. 41–48, 2011, doi: https://doi.org/10.1016/j.jad.2010.06.021.

[6] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011, doi: 10.1109/MCSE.2011.37.

[7] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, doi: 10.1145/1498765.1498785.

[8] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," pp. 483–485, 1967, doi: 10.1145/1465482.1465560.

[9] K. Harding and D. M. Dunlavy, "Improving Runtime Performance of Tensor Computations using Rust From Python." [Online]. Available: https://arxiv.org/abs/2510.01495

[10] E. Dolstra, M. De Jonge, E. Visser, and others, "Nix: A Safe and Policy-Free System for Software Deployment.," in *LISA*, 2004, pp. 79–92.

Appendix

A. Perf flags

Perf Event	Description
`instructions`	Total number of retired instructions executed during the benchmark.
`cycles`	Total CPU cycles consumed by the workload.
`task-clock`	Aggregate CPU execution time across all threads, used to estimate CPU utilization.
`duration_time`	Wall-clock execution time measured by perf.
`l1-dcache-loads`	Number of L1 data cache load accesses.
`l1-dcache-load-misses`	Number of L1 data cache load misses.
`l1-icache-loads`	Number of L1 instruction cache accesses.
`l1-icache-load-misses`	Number of L1 instruction cache misses.
`l1_data_cache_fills_all`	Total number of L1 data cache line fills from lower levels of the memory hierarchy.
`l1_data_cache_fills_from_memory`	Number of L1 data cache line fills originating directly from main memory (DRAM).
`l2_cache_accesses_from_ic_misses`	L2 cache accesses triggered by L1 instruction cache misses.
`l2_cache_hits_from_ic_misses`	L2 cache hits resulting from L1 instruction cache misses.
`l2_cache_accesses_from_dc_misses`	L2 cache accesses triggered by L1 data cache misses.
`l2_cache_hits_from_dc_misses`	L2 cache hits resulting from L1 data cache misses.
`cache-references`	Total last-level cache (LLC) references recorded by the processor.
`cache-misses`	Total last-level cache misses recorded by the processor.
`fp_ret_sse_avx_ops.all`	Retired floating-point SIMD operations executed through SSE and AVX vector instructions.

B. Computed metrics

Formula	Description
$\mathrm{L1}_{\mathrm{hitrate}} = \frac{\mathrm{HitRate}_{\mathrm{L1I}} + \mathrm{HitRate}_{\mathrm{L1D}}}{2}$	Average of instruction and data L1 cache hit rates; reflects locality of reference.
$\mathrm{L2}_{\mathrm{hitrate}} = \frac{\mathrm{HitRate}_{\mathrm{L2I}} + \mathrm{HitRate}_{\mathrm{L2D}}}{2}$	Measures how effectively L2 cache captures misses from L1 caches.
$\mathrm{L3}_{\mathrm{hitrate}} = \frac{\text{Cache Misses}}{\text{Cache References}}$	Miss ratio at last-level cache; lower is better (despite name, this is a miss rate).
$\mathrm{CPU}_{\mathrm{util}} = \frac{\text{Task Clock}}{\text{Wall Clock Time}}$	Estimates effective CPU usage; values > 1 indicate multi-core parallelism.
$\mathrm{SIMD}_{\mathrm{density}} = \frac{\text{SIMD Operations}}{\text{Instructions}}$	Fraction of SIMD floating-point operations; higher means better vectorization.

Table 2:

C. Pseudo code Rust implementation

Algorithm 1: Data Reshuffling and Cohort Extraction

```
1:  procedure SHUFFLE_AND_SPLIT(data, indices_a, indices_b)
2:
3:      ▷ From S_sub × S_src × F × B
4:
5:      shuffled ← shuffle(data, [B, F, S_src, S_sub]])
6:      ▷ To B × F × S_src × S_sub
7:
8:      A ← select_by_idx(shuffled, 3, indices_a)
9:      B ← select_by_idx(shuffled, 3, indices_b)
10:
11:     return (A, B)
12: end
```

Algorithm 2: Statistical Comparison Computation

```
1:  procedure COMPUTE(data, indices_a, indices_b, test_type)
2:      (A, B) ← shuffle_and_split( data, indices_a, indices_b)
3:      N_test ← S_src × F × B
4:
5:      ▷ Allocate result arrays
6:      p_values ← array(N_test)
7:      significance ← array(N_test)
8:      differences ← array(N_test)
9:      effect_sizes ← array(N_test)
10:
11:     ▷ Iterator distributes workload over threads
12:     for each (b, f, s) combination do
13:         data_a ← A[b, f, s, :]
14:         data_b ← B[b, f, s, :]
15:
16:         (p_value, difference_group, effect_size) ← run_test(test_type, data_a, data_b)
17:
18:         p_values[b,f,s] ← result.p_value
19:         significance[b,f,s] ← p_value < 0.05
20:         differences[b,f,s] ← difference_group
21:
22:         if has_effect(test_type) then
23:             effect_sizes[b,f,s] ← effect_size
24:         end
25:     end
26:
27:     return (p_values, significance, differences, effect_sizes)
28: end
```

D. Benchmark environment

Property	Value
Processor	AMD Ryzen 7 5800H
Microarchitecture	AMD Zen 3
Physical/Logical Cores	8/16
L1 Cache	256 KiB (64KB/core)
L2 Cache	4 MiB (512KB/core)
L3 Cache	16 MiB (Shared)
NUMA Nodes	1
SIMD Extensions	AVX, AVX2, FMA
Operating System	NixOS 26.05 (Yarara) x86_64
Kernel	Linux 6.18.23
Host	Lenovo IdeaPad 5 Pro 16ACH6
Package Manager	Nix
Python version	3.13
Rust version	rustc 1.97.0-nightly (ff9a9ea07 2026-05-13)

Table 3: Hardware and software specifications used for benchmarking.