Minimising Data-Layout and Copy Overhead

A Memory-Management Study of an EEG Biomarker Pipeline

Sem Lelie[1]
Supervisor(s): Ricardo Marroquim[1]**, Arthur Avramiea**
[1]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 21, 2026

Name of the student: Sem Lelie
Final project course: CSE3000 Research Project
Thesis committee: Ricardo Marroquim, Arthur Avramiea, **[Examiner name TBD]**

Abstract

Electroencephalography (EEG) biomarker pipelines are usually assumed to be limited by statistical computation, but much of their cost is data movement: copying, reshaping, and indexing arrays through high-level Python abstractions. We profile the data-reshaping (*reduce*) stage of the Neurophysiological Biomarker Toolbox, a Python EEG-analysis framework, and find it performs no arithmetic at all: its cost is an eager deep copy of the per-subject container, which also doubles the stage's peak memory by allocating a full duplicate of the data.

We evaluate three memory-management strategies (zero-copy array views, layout pinning, and lazy materialisation) against an unmodified baseline, verifying that every variant reproduces the baseline's statistical outputs exactly. Zero-copy views remove the duplicate, cutting reduce-stage peak memory from gigabytes to near zero; this lowers worst-case (tail) latency and, under the parallel load of a cohort sweep, lifts throughput by up to 4.4× by keeping concurrent workers out of swap. Layout pinning and lazy materialisation act only when subjects have repeated sessions, where lazy materialisation cuts reduce-stage peak memory by two orders of magnitude.

The reduce stage is thus effectively eliminated as a cost. The end-to-end speedup is a more modest 1.3×, bounded not by the optimisation but by a separate, arithmetic-bound statistics step that lies outside this paper's scope and which we flag as the natural next target. The practical recommendation is to eliminate eager deep copies first: a small change that removes the memory doubling and, under parallel load, keeps a cohort sweep out of swap.

1 Introduction

EEG analysis pipelines are more computationally expensive than they appear. Electroencephalography (EEG) produces high-dimensional time-series data across subjects, recording sessions, channels, and frequency bands [1, 2], and a single analysis workflow repeatedly segments, filters, decomposes, and aggregates these recordings into biomarkers [3]. Exploratory studies compound the problem by sweeping many parameter combinations across cohorts of thousands of subjects, so even small per-operation overheads accumulate into substantial end-to-end delays and large memory footprints. Memory is often the sharper constraint: when several runs share one machine during a parameter sweep, a single stage that briefly holds two copies of its data can be enough to exhaust the available RAM and force the system to swap, so a pipeline that is usually fast can occasionally stall for an order of magnitude longer. We summarise this behaviour by the *tail* latencies T_{p95} and T_{p99}, the runtimes below which 95% and 99% of runs complete. Such unpredictable stalls matter in practice: they make a pipeline hard to schedule and provision for.

Much of this overhead comes from data movement rather than mathematics. In scientific Python, data-transformation steps are often written with defensive deep copies and eager array reshaping to guarantee correctness during development; while robust, these choices are costly at scale. Performance on modern hardware is increasingly bounded by memory bandwidth rather than arithmetic throughput [4, 5, 6], so the layout, stride patterns, and allocation cost of multidimensional arrays can dominate a stage even when the numerical work is delegated to optimised libraries such as NumPy [7]. A deep copy is two times as costly: it also allocates a full duplicate of the data, doubling the stage's peak memory. Lazy evaluation is a well-established remedy for unnecessary materialisation [8], yet the practical magnitude of these costs in real multi-stage EEG toolchains remains largely unquantified.

This paper isolates and quantifies them within the Neurophysiological Biomarker Toolbox (NBT), a Python framework for computing and statistically comparing EEG biomarkers across clinical cohorts. We study the full pipeline, but an initial profiling pass (Section 4) identifies the data-reshaping *reduce* stage as the one whose cost is purely structural data movement, and therefore the one addressable by memory-management strategies, so we concentrate our optimisations there and ask three research questions. **RQ1:** how much of the reduce stage's time and peak memory is spent on data-layout and memory-management operations (deep copies, reshapes, remappings, allocations) rather than on arithmetic? **RQ2:** which of these operations dominates, and how does that dominance scale with cohort size and with the number of recording sessions per subject? **RQ3:** can targeted memory-management strategies, namely zero-copy array views, contiguous-layout pinning, and lazy materialisation, reduce runtime and peak memory while preserving the pipeline's numerical outputs exactly?

We answer these questions empirically: we profile a baseline NBT variant with direct wall-clock timing, relate the observed costs to the bytes moved, and evaluate three modular optimisation strategies and their combinations, verifying numerical equivalence on the full pipeline output (p-values, effect sizes, group differences, and significance masks) so that any speedup provably preserves the scientific result.

We make three contributions. First, we give a clean, instrumentation-free decomposition of the NBT pipeline that locates where its cost lies. It shows the reduce stage to be entirely structural data movement, and therefore the stage that memory management can address, while the larger statistics cost is arithmetic-bound; we then quantify the reduce stage in detail and characterise how it scales. Second, we evaluate three memory-management strategies and their combinations, each reported in the regime where it actually applies, documenting where each one helps, where it is inert, and where it backfires. Third, we provide a reproducible benchmarking harness and practical guidance for prioritising these optimisations, and we identify the un-vectorised statistics step, exposed once the reduce-stage cost is removed, as the principal target for turning these stage-level gains into end-to-end ones.

The remainder covers background (Section 2), methodology (Section 3), results (Section 4), responsible research (Section 5), discussion (Section 6), and conclusions (Section 7).

2 Background

This section introduces the concepts the optimisations build on: the NBT biomarker pipeline and its reduce step, and the memory-layout and copy costs that govern performance in scientific Python.

2.1 The EEG Biomarker Pipeline and NBT

Clinical and neuroscientific studies compare EEG-derived *biomarkers* (scalar or spectral per-subject summaries such as band-power or connectivity measures) between patient groups and healthy controls [1, 3]. Such studies now routinely involve tens of thousands of recordings [9], placing real computational pressure on the analysis infrastructure.

A typical pipeline applies three stages to a cohort (Figure 1): a *filter step* that selects the subjects in each clinical group; a *reduce step* that re-indexes the biomarker arrays from a flat per-recording layout into a per-subject array; and a *statistics step* that applies group-comparison tests (e.g. unpaired t-tests or Wilcoxon rank-sum tests) to every (channel, frequency) cell. Here M is the number of *recording sessions* per subject (how many times

each subject is recorded): once ($M = 1$) in some studies, several times ($M \geq 2$, e.g. before and after a treatment) in others. A cohort of N subjects then holds $N \cdot M$ recordings, one per session. Crucially, M shapes the *data*, not the statistics: each test still compares the subjects across the two clinical groups, so its power is governed by N, and it runs identically for any M. A larger M only adds a session axis, i.e. more cells to test.

This shape change is what our optimisations target. At $M = 1$ the data is already one row per subject, so the reduce step merely re-labels it; at $M \geq 2$ it must *expand* the recordings into a per-subject, per-session array, allocating and filling a new buffer. That buffer is what two of our three strategies act on, so they are inert when subjects are recorded only once. We target the reduce step because its cost is purely structural data movement; as Section 4 shows, however, the most expensive stage overall is the statistics step, to which we return in the discussion.

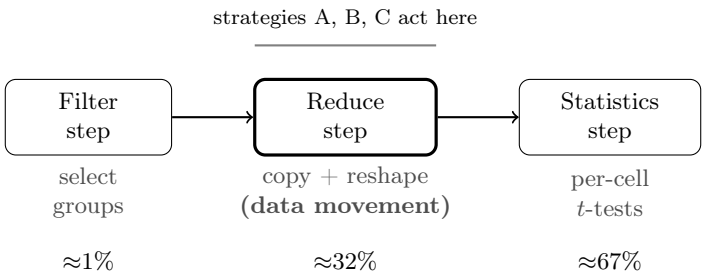

Figure 1: The NBT statistics pipeline. Percentages are each stage's share of baseline end-to-end runtime at $N = 10{,}000$, $M = 2$ (Table 2). Our strategies act on the reduce step, whose cost is pure data movement; the dominant statistics step is left unchanged.

NBT realises this pipeline as a Python reimplementation of an earlier MATLAB-based toolbox. Its core data structure is a per-subject container holding labelled n-dimensional arrays (`xarray.DataArray` objects) indexed by recording filename, alongside a table of subject metadata (condition, cohort, recording session). The baseline copies the entire container at every pipeline stage boundary with a recursive deep copy, a design choice inherited from correctness constraints during development.

2.2 Memory Layout and Copy Costs in Scientific Python

NumPy stores multidimensional arrays in either C-contiguous (row-major) or Fortran-contiguous (column-major) order [7]. SciPy and NumPy statistical kernels expect C-contiguous inputs, so passing an F-order buffer triggers an implicit copy at the library boundary, a hidden latency penalty [10].

A recursive deep copy duplicates every object in the graph, including each array's underlying numeric buffer. For a container of N subjects with C channels and F frequency bins, this allocates $\mathcal{O}(N \cdot C \cdot F)$ new bytes per call (linear in the data, since C and F are fixed), and, because it transiently holds *two* copies of the buffer, doubles the stage's peak memory; its wall-clock cost is therefore bounded by memory bandwidth rather than arithmetic. A *zero-copy view*, by contrast, wraps the existing buffer without duplicating it, at $\mathcal{O}(1)$ cost per array regardless of N [11]. *Lazy materialisation* defers constructing intermediate arrays until their values are first accessed, avoiding allocations for results that are later discarded or never read [8, 12]. Applied to the $M \geq 2$ expansion buffer, a lightweight descriptor records

the intended shape and index mapping, and the concrete buffer is built only if and when a downstream stage reads it.

3 Methodology

We follow an empirical, quantitative systems-evaluation approach, because purely theoretical algorithmic analysis cannot capture the hidden costs of memory bandwidth, allocation, and array layout in interpreted Python pipelines. The methodology runs in four sequential phases (baseline profiling, hotspot analysis, candidate implementation, and controlled evaluation) and closes with the hypotheses it is designed to test. Throughout, we keep *time* (runtime) and *space* (peak memory) as separate axes, because the strategies affect them differently.

3.1 Phase 1: Baseline Profiling and Bottleneck Identification

We profile the unmodified pipeline at the stage level (filter, reduce, statistics) to locate the dominant stage, then decompose the reduce stage into its sub-operations (container copy, coordinate remap, and, at $M \geq 2$, the expansion-buffer fill). *Time* is measured with direct, instrumentation-free wall-clock timing (`time.perf_counter`) per stage and per sub-operation. We deliberately avoid call-tracing profilers such as `cProfile`, whose per-call probe disproportionately inflates call-heavy operations such as a recursive deep copy and distorts both absolute times and the speedup ratios between variants. *Space* is measured separately, as the Python-heap delta (`tracemalloc`) at small N and the OS resident-set delta (`psutil`) at large N.

3.2 Phase 2: Hotspot Analysis and Cost Modelling

We trace how the dominant cost scales with cohort size N and recording sessions per subject M, and pair the empirical behaviour with two separate models: a *time* model that relates runtime to the bytes moved (so a stage is bounded by memory bandwidth rather than arithmetic), and a *space* model that relates peak memory to the buffers allocated. Keeping the two distinct is essential, since a strategy can cut one while leaving the other unchanged. This directs the optimisations at structural inefficiencies (bytes moved, buffers allocated) rather than superficial symptoms. The resulting per-strategy time and space complexities are derived in Appendix B.

3.3 Phase 3: Candidate Strategies

Guided by the Phase 2 hotspots, we design three modular memory-management strategies, each enableable independently:

- **Zero-Copy Array Views (Strategy A):** the eager recursive deep copy is replaced by a shallow container copy and zero-copy array views that share the underlying buffer, with per-recording data read directly from the underlying NumPy buffers. This removes the duplicate allocation at every stage boundary, most significantly in the reduce stage. *Targets both space (no duplicate buffer) and time (no copy).*
- **Memory-Layout Pinning (Strategy B):** the expansion buffer, built only when subjects have repeated sessions, is allocated in Fortran order for contiguous column-wise

assignment, then converted to C-contiguous layout (`np.ascontiguousarray`) before the statistical kernels. *Targets fill time; has nothing to act on at $M = 1$.*

- **Lazy Materialisation (Strategy C):** the same expansion buffer is represented as a lightweight descriptor (shape, stride, and index mapping) and the concrete array is allocated only on first access. *Targets space; like Strategy B, acts only at $M \geq 2$.*

To isolate each strategy and test pairwise interactions, we form the complete 2^3 factorial of the three strategies, giving seven candidate implementations plus the baseline (Table 1).

Candidate	**Label**	**A** (zero-copy)	**B** (layout)	**C** (lazy)
Baseline	baseline			
Cand. 1	zero_copy	✓		
Cand. 2	layout_pin		✓	
Cand. 3	lazy	✓		✓
Cand. 4	combined	✓	✓	
Cand. 5	layout_lazy		✓	✓
Cand. 6	all_three	✓	✓	✓
Cand. 7	lazy_only			✓

Table 1: The eight configurations: baseline plus the 2^3 factorial of strategies A (zero-copy), B (layout), C (lazy). **B and C act only at $M \geq 2$**; at $M = 1$ only A does work.

The baseline relied on deep copies to keep each stage's data separate; instead, we treat the shared arrays as read-only, so no stage overwrites data that another stage still needs and no aliasing bugs can arise. We keep an explicit copy only where an external library requires its own contiguous array.

3.4 Phase 4: Controlled Evaluation and Correctness Verification

3.4.1 Experimental Setup and Data Generation

Experiments ran on a single local machine (Intel Core i7-13700H, 16 GB RAM, Windows 11) in a locked Python 3.12 environment with stable releases of NumPy, SciPy, and `xarray`. Because real EEG datasets at the scale needed to stress the memory subsystem are unavailable under privacy constraints, we use a synthetic per-subject container with realistic frequency-domain dimensions ($C = 64$ channels, $F = 50$ frequency bins, tensor shape $(N, 64, 50)$), split evenly between two clinical conditions and parameterised by the number of recording sessions M. Buffers are filled from fixed-seed random normal distributions, giving bit-for-bit reproducibility.

3.4.2 Metrics and What Each Candidate Is Measured On

Each candidate is evaluated on two independent axes plus correctness:

- **Time:** reduce-stage wall-clock runtime T and speedup $S = \mu_{\text{baseline}}/\mu_{\text{candidate}}$ (full pipeline where stated). Each configuration is run 30 times for the significance tests; the descriptive scaling and stage-breakdown tables use fewer repetitions, as stated per table. We report the mean. Tail latency T_{p95}, T_{p99} captures worst-case spikes.

- **Space:** peak memory of a single reduce-stage call, as the Python-heap delta (`tracemalloc`) or resident-set delta (`psutil`) at large N, in MB.
- **Correctness:** full-pipeline output equivalence to the baseline (below).

Each candidate is measured in the regime where its strategies are active. At $M = 1$ only Strategy A does work, so we report it against the baseline (Section 4.2); the B/C-only configurations act as setup-overhead controls. At $M \geq 2$ all three strategies engage and we evaluate the full factorial (Section 4.3). Single-session timing is collected single-process to avoid swap artefacts from coexisting large copies.

3.4.3 Concurrency

A cohort sweep typically runs several pipelines at once on one machine, so we also stress the *space* axis under parallel pressure: we run $W \in \{1, 2, 4\}$ reduce steps concurrently (one per process via `ProcessPoolExecutor`, each subprocess generating its own synthetic dataset) at three cohort sizes, recording the total wall-clock to finish all W. This isolates how each configuration's per-worker memory footprint affects parallel throughput.

3.4.4 Statistical and Correctness Verification

Because execution times are right-skewed (a few slow runs stretch the upper tail), we test runtime differences with the **Wilcoxon signed-rank test** (a test that ranks the paired differences and does not assume a normal distribution) on the paired baseline−candidate differences (H_0: median difference = 0, $\alpha = 0.05$) [13]. Scientific validity is verified by re-running the *full* pipeline (filter → reduce → statistics) for every candidate and checking that all statistical outputs (p-values, effect sizes, group differences, and significance masks) match the baseline with `numpy.testing.assert_allclose` (`rtol=1e-5`, `atol=1e-8`). This exercises the statistics-step arithmetic, so equivalence covers the actual computation and not merely the reshaped arrays.

3.5 Hypotheses

- **H1 (time + space): within the reduce stage, data-layout and copy operations, not arithmetic, drive the cost.** The reduce stage performs re-indexing and copying but no statistics, so its runtime and peak memory should be almost entirely attributable to data movement [11].
- **H2 (space, then time): zero-copy views yield the largest reductions in reduce-stage peak memory and runtime** for copy-heavy single-session workloads, by removing the duplicate allocation at near-zero cost [11].
- **H3 (time): layout pinning reduces per-kernel runtime for cache-sensitive kernels.** Testable only where the expansion buffer is built, i.e. at $M \geq 2$; we evaluate it there (Section 4.3) rather than at $M = 1$, where the F-order path never executes [10].
- **H4 (space): lazy materialisation lowers peak memory by avoiding the expansion-buffer allocation.** Like H3 this is only meaningful at $M \geq 2$, and is evaluated there.

4 Experimental Setup and Results

We evaluate on the synthetic benchmark of Section 3.4. All timings are instrumentation-free wall-clock means (Section 3.1), with the run count n given per table, and peak memory is measured separately. We first locate the dominant cost and characterise how it scales (RQ1, RQ2), then evaluate the strategies in each session regime and under concurrency, and finally confirm that none alters the scientific output (RQ3). Per-scale tables are in Appendix A.

4.1 Where the Cost Lies and How It Scales

Table 2 times the three stages end to end at $N = 10{,}000$, $M = 2$, for the baseline and the all-strategies configuration. The reduce stage is about a third of the runtime and the statistics step two thirds; our strategies cut the reduce stage from 7.24 s to 0.075 s (97×), but the statistics step, which we do not modify, dominates the total, so the end-to-end speedup is only 1.27×. The optimised statistics time is slightly higher because the lazy strategy shifts the expansion-buffer construction downstream into it.

Stage	**Baseline (s)**	**Optimised (s)**	**% of base**
Filter	0.28	0.21	1.2%
Reduce	7.24	0.075	31.8%
Statistics	15.25	17.58	67.0%
Total	**22.76**	**17.86**	**100%**

Table 2: End-to-end stage breakdown at $N = 10{,}000$, $M = 2$ ($n = 3$). "Optimised" is the all-strategies configuration (Cand. 6).

Zooming in, direct sub-operation timing shows the reduce stage performs no arithmetic: its cost is a container deep copy plus a coordinate remap, both pure data movement. This answers RQ1: essentially 100% of the stage is data-layout and copy work. The deep copy, linear in data size, dominates at scale (≈88% at $N = 100{,}000$); the remap dominates only for small cohorts (Table 8). The stage therefore scales about linearly in N, from ≈17 ms at $N = 1{,}000$ to 0.63 s at $N = 100{,}000$, with peak memory tracking the raw buffer (24 MB to 2.5 GB) as the copy transiently holds a duplicate (Table 9); there is no super-linear regime (an earlier apparent one was a profiler artefact, Section 3.1). The copy's wall-clock is, however, acutely memory-pressure sensitive (the same ∼0.7 s copy can exceed 10 s once the working set spills toward swap); we quantify this under concurrency below.

4.2 Single-Session Regime: Zero-Copy

At $M = 1$ only Strategy A (zero-copy) executes; B and C gate on $M \geq 2$, so the eight configurations collapse into those with zero-copy and those without (Table 1). Table 3 reports one representative of each group; the full per-scale data is in Table 7.

The runtime benefit of zero-copy is real but modest and N-dependent: marginal at $N = 1{,}000$ (the deep copy is cheap there) and ≈4–9× at $N = 100{,}000$ (the spread reflecting memory pressure). The robust gain is in memory: zero-copy removes the doubling, cutting reduce-stage peak from 49.1 MB to 0.16 MB at $N = 1{,}000$ (306× by `tracemalloc`) and from ≈2.5 GB to essentially zero at $N = 100{,}000$ (Table 10). Configurations without A keep the full footprint, and adding the inactive B/C machinery is not free: it makes them slower than the baseline at small N (Cand. 6 0.56×, Cand. 2 0.43× at $N = 1{,}000$).

Configuration	$N = 1{,}000$ (ms)	S	$N = 100{,}000$ (s)	S
Baseline	17.3	1.00×	0.633	1.00×
Cand. 1: zero-copy (A)	15.7	1.10×	0.154	4.1×
Cand. 6: all three (A+B+C)	30.8	0.56×	0.146	4.3×
Cand. 2: layout pin (B)	40.0	0.43×	0.962	0.66×

Table 3: Reduce-stage runtime and speedup S at a single session (clean, isolated, $n = 3$). One representative per group is shown; Cands. 3, 4, 5, 7 are omitted (the A-containing 3, 4 track Cand. 1, the non-A 5, 7 the baseline). Speedups are memory-pressure sensitive: Cand. 1 ranged ≈4–9× at $N = 100{,}000$.

Significance and tail latency. At $N = 1{,}000$ the runtime reduction is significant (Wilcoxon signed-rank, $p \approx 1.3 \times 10^{-7}$, $n = 30$), whereas the layout-pin control is not ($p = 0.27$). Eliminating the deep copy also lowers *tail* latency by a factor that grows with N and tracks the mean reduce-time speedup, the isolated distributions being tight (Table 4): the 99th-percentile reduce time is essentially unchanged at $N = 1{,}000$ (the deep copy is cheap there), then ∼3× lower at $N = 10{,}000$, ∼7× at $N = 31{,}622$, and ∼8× at $N = 100{,}000$ under clean isolated conditions. Like the mean speedup this factor is memory-pressure sensitive: the reduce-stage speedup itself spans ∼4–9× at $N = 100{,}000$ across runs (Table 3), so these isolated tails stay sub-second, while the order-of-magnitude worst case appears only under memory pressure (Section 4.4).

N	Config	Mean	T_{p95}	T_{p99}
1,000	Baseline	17.8 ms	18.9 ms	22.1 ms
1,000	Zero-copy	13.7 ms	15.0 ms	17.2 ms
10,000	Baseline	59.4 ms	63.9 ms	65.1 ms
10,000	Zero-copy	17.8 ms	20.0 ms	21.6 ms
31,622	Baseline	215 ms	228 ms	231 ms
31,622	Zero-copy	29.1 ms	30.7 ms	30.9 ms
100,000	Baseline	639 ms	689 ms	702 ms
100,000	Zero-copy	79.1 ms	84.3 ms	85.0 ms

Table 4: Reduce-stage tail latency at a single session ($M = 1$). Clean isolated protocol: the two configurations interleaved per iteration, warmup discarded, fresh container per run, no profiler, single thread ($n = 30$; $n = 15$ at $N = 100{,}000$). Absolute times are sensitive to machine load, so the per-N reduction factor, not the raw milliseconds, is the robust quantity.

4.3 Multi-Session Regime ($M \geq 2$)

At $M \geq 2$ the reduce stage must build the $(\mathrm{ch}, \mathrm{fr}, M, N)$ expansion buffer, iterating over recordings one at a time, and the picture inverts: this buffer, not the container copy, now dominates the stage's time and memory (Table 5).

Three findings follow. **(i)** Zero-copy alone (Cand. 1) only halves peak memory (997 MB): it removes the source duplication but still builds the output buffer eagerly. **(ii)** Lazy materialisation is decisive: deferring the buffer drops peak heap to 7.65 MB (195×, Cands. 3, 6) and keeps the working set flat as M grows from 1 to 3 while every eager variant grows with M (Table 11). **(iii)** Lazy pays off only combined with zero-copy: lazy alone (Cand. 7) cuts peak heap 3.0× (to 505 MB), whereas lazy with zero-copy (Cand. 3) reaches 195× (7.65 MB).

Configuration	Strat.	Peak (MB)	Reduce (s)
Baseline	–	1494.1	7.222
Cand. 1 zero-copy	A	996.8	1.162
Cand. 2 layout	B	2453.9	2.233
Cand. 3 lazy	A+C	7.65	0.062
Cand. 4 combined	A+B	1956.9	2.128
Cand. 5 lay.+lazy	B+C	504.6	0.146
Cand. 6 all three	A+B+C	7.65	0.060
Cand. 7 lazy only	C	504.6	0.142

Table 5: Reduce-stage peak heap and runtime at $N = 10{,}000$, $M = 2$ ($n = 7$). "Strat." lists the active strategies (A zero-copy, B layout, C lazy).

Layout pinning speeds the eager fill (Cand. 2 is 3.2× faster than the baseline) but raises peak memory by holding two buffers, and on the zero-copy path adds only slowdown (Cand. 4 at 2.13 s vs Cand. 1 at 1.16 s).

End to end, the full-pipeline $M = 2$ speedups are modest: 1.27× for the all-strategies configuration and 1.23–1.37× across the seven candidates at $N = 10{,}000$ (every candidate faster than the baseline on all 30 paired runs, so all $p \approx 1.9 \times 10^{-9}$, Wilcoxon signed-rank, $n = 30$).

4.4 Behaviour Under Concurrency

A cohort sweep typically runs several pipelines at once on one machine, so we measure behaviour under parallel memory pressure. Table 6 reports the wall-clock to finish W concurrent reduce steps (one per process). At $N = 100{,}000$ the baseline degrades sharply as workers are added (6.6 s at $W = 1$ to 110.7 s at $W = 4$): each worker's deep copy doubles its footprint, and four together exceed the machine's 16 GB and force swapping. Zero-copy, holding no duplicate, finishes the same workload in 25.1 s (4.4×). The advantage is 1.5× at $N = 31{,}622$ and nil at $N = 10{,}000$, where four footprints still fit in RAM.

N	Config	W=1	W=2	W=4
10,000	Baseline	3.16	3.26	3.54
10,000	Zero-copy	3.05	3.21	3.71
31,622	Baseline	3.75	4.37	6.81
31,622	Zero-copy	3.81	4.14	4.62
100,000	Baseline	6.61	30.14	110.69
100,000	Zero-copy	7.43	6.72	25.09

Table 6: Total wall-clock (s) to finish W concurrent reduce steps, one per process (best of two repetitions). Totals include process start-up and data generation, so the per-reduce advantage is larger still.

4.5 Correctness

Scientific equivalence was checked on the *full* pipeline: for each of the seven candidates we ran filter → reduce → statistics and compared all four statistical outputs (p-value, effect size, group difference, significance mask) against the baseline at `rtol=1e-5`, `atol=1e-8`, across three seeds, two cohort sizes, and $M \in \{1, 2, 3\}$. This exercises the statistics-step arithmetic

(the per-cell t-test loop) and the multi-session expansion path, so the claim covers the actual computation and not merely the reshaped arrays. **All 126 checks ($7\times3\times2\times3$) passed**, and a separate aliasing spot-test confirmed that mutating a candidate's output never propagates to the baseline. No candidate alters the scientific result.

5 Responsible Research

Reproducibility is supported by fixed random seeds, identical parameter configurations, and a locked software environment (Section 3.4), so the protocol can be replicated on comparable hardware. Each optimised variant is additionally verified to produce output numerically equivalent to the unmodified baseline (Section 4.5). Because of an institutional non-disclosure agreement, neither the NBT pipeline nor the real EEG dataset underlying it (provided by a project collaborator) can be released publicly; however, all performance claims are reproducible on synthetic data that mimics real-world EEG datasets, and the dataset generator and benchmarking harness are available on request, subject to the NDA. Residual hardware and external-library effects on absolute latency are reported as limitations (Section 6.6).

This work analyses only the computational performance of an existing pipeline and processes no new human data; the real EEG dataset is deleted after use, and the optimisations alter only memory layout and allocation, leaving the statistical computations and scientific interpretation unchanged. In accordance with TU Delft guidelines, generative AI (Claude) was used solely to improve clarity, grammar, and academic style, with the author retaining full responsibility for the content, accuracy, and originality of this work.

6 Discussion

The results answer the three research questions but also reframe them: the reduce stage turns out to be the wrong place to look for the pipeline's largest cost and the right place to look for its most easily removable one. We draw out that distinction, synthesise what the strategies do and do not buy, and locate the remaining headroom before stating the limitations.

6.1 The reduce stage is addressable, not dominant

The decomposition answers RQ1 cleanly: the reduce stage performs no arithmetic at all, only a deep copy of the container and a coordinate remap, both pure data movement, with the deep copy reaching $\approx$88% of the stage at $N = 100{,}000$ and the remap dominating only for small cohorts. Because the stage contains no arithmetic, there is no irreducible compute to work around (H1): a memory-management change can speed up essentially the whole stage. That is unusual for an optimisation target, which normally has some computation that cannot be removed.

This has to be read against the end-to-end picture, because the reduce stage is not the pipeline's largest cost. The statistics step is, at roughly twice the reduce stage (Section 6.4). We targeted the reduce stage not because it dominates the runtime but because its cost is purely structural, and so can be removed without perturbing any number the pipeline produces. Fully removable yet only a third of the runtime, a tension we return to in the end-to-end analysis.

For RQ2, the deep copy is $\mathcal{O}(N \cdot C \cdot F)$, linear in the data since C and F are fixed, and both wall-clock and peak memory track it. One correction is worth recording: an apparent "super-linear" jump between $N = 31{,}622$ and $N = 100{,}000$ in earlier drafts was a measurement artefact, not an algorithmic one: partly call-tracing instrumentation, which inflates recursive deep copies by an order of magnitude, and partly memory pressure pushing large copies toward swap. Under clean, profiler-free timing the growth is linear. That the artefact appeared at all is itself telling: the deep copy is bound by memory traffic [5], so its wall-clock depends sharply on how close the working set sits to physical RAM, the theme the rest of this section turns on.

6.2 Memory, not runtime, is the robust win

Zero-copy views remove the $\mathcal{O}(N \cdot C \cdot F)$ duplicate outright, since the new wrapper shares the buffer [11], and the two benefits this produces are of very different quality. The runtime speedup is real but soft: it grows with N (it removes exactly the copy that grows with N) yet swings between roughly four- and nine-fold at $N = 100{,}000$ depending on memory pressure. The peak-memory reduction is large and stable: from 49.1 MB to 0.16 MB at $N = 1{,}000$, and $\approx$2.5 GB removed at $N = 100{,}000$. This is the deep copy's hidden second cost, and it is the result we would stake the central hypothesis (H2) on.

Memory matters more than the isolated runtime number suggests, because it is what survives realistic load. A single $N = 100{,}000$ copy transiently holds the original and the duplicate, so one baseline worker needs $\approx$5 GB; four such workers exceed the test machine's 16 GB and force the operating system to swap, which is where the baseline's reduce time collapses to 110.7 s. Zero-copy keeps each worker's footprint near zero and finishes the same four-worker sweep in 25.1 s. The modest isolated speedup and the 4.4$\times$ parallel one are the same effect at different memory pressures: under the parallel load that cohort sweeps actually impose, the memory saving *is* the speedup, and it is also what removes the order-of-magnitude worst-case tail.

6.3 Where the strategies help, are inert, or backfire

Zero-copy, established above as the workhorse, is also the foundation at $M \geq 2$; the other two strategies are conditional refinements on it. Lazy materialisation earns its keep only when subjects have repeated sessions, where the eager expansion buffer becomes the dominant cost; deferring that buffer cuts reduce-stage peak heap by 195$\times$ and holds it flat as M grows, confirming H4 in that regime. On its own, lazy still helps (3$\times$), but it cannot remove the source deep copy that only zero-copy eliminates, so it is a refinement on top of zero-copy rather than an alternative to it.

Layout pinning is the cautionary case. At a single session its Fortran-order path never runs, so it merely retains the deep copy and adds setup overhead, finishing slower than the baseline. At $M \geq 2$ it does speed the eager fill (3.2$\times$) by handing the kernels contiguous inputs [10], but it pays with a second copy of the expansion buffer, and on the memory-efficient zero-copy or lazy path that speed-up turns into a net slowdown. So H3 gets only narrow support: pinning accelerates exactly the eager fill that the better strategies avoid building at all. The single best general-purpose configuration is therefore zero-copy plus lazy, which wins on memory in both regimes and never regresses on the optimised path.

A methodological point sits underneath this. Two of the three strategies are inert at $M = 1$, so a sweep that benchmarked all of them in that regime would have logged layout

pinning and lazy as pure overhead and missed their real behaviour. Separating the regimes is what lets B and C be judged where they actually run, a precaution that applies to any optimisation that only switches on under certain conditions, not only these.

6.4 The end-to-end ceiling: the statistics step

Within its scope the intervention is a near-total success: the reduce stage, the explicit target of this work, drops by 97× in runtime (7.24 s to 0.075 s) and by more than two orders of magnitude in peak memory, effectively erasing it as a cost. The end-to-end figure is necessarily more modest, and it should not obscure the magnitude of that stage-level result. At $N = 10{,}000$ with two sessions the reduce stage is only 32% of the runtime against the statistics step's 67%, so removing it entirely lifts the total by 1.27×, a ceiling set by Amdahl's law, not a shortfall in the optimisation. The statistics step remains the larger cost because it is dominated by arithmetic rather than data movement, so the memory-management strategies had nothing to act on there to begin with. The small end-to-end gain is thus the flip side of a complete success on the targeted stage: with the structural cost gone, what remains is almost entirely the un-vectorised statistics step, now clearly exposed as the next bottleneck.

Concretely, that arithmetic is an independent t-test run in a Python loop over every (channel, frequency) cell, with per-cell indexing a smaller contributor (15–22% of the loop), a computational inefficiency, not a data-movement one. Zero-copy is the one strategy that still transfers: the statistics step takes its own deep copy of the container, which the same shallow-copy change would remove, saving another full duplicate. But the principal lever is to vectorise the per-cell loop, replacing thousands of Python-level test calls with a single call over the whole grid. Zero-copy to remove that remaining deep copy, together with a vectorised statistics kernel, is the route to an end-to-end speedup well beyond the present 1.3×, and the primary target for future work (Section 7).

6.5 Practical recommendations

The first priority for developers of NumPy-based pipelines is to find and remove eager deep copies in data-reshaping stages: replacing a recursive deep copy with shallow copies and zero-copy views is a small, low-risk change (given that downstream code treats the data as read-only) that eliminates the memory doubling and yields a runtime gain that grows with data size. Layout and deferred-evaluation transforms should be applied only where they actually run; at a single session they are pure overhead, and layout pinning in particular trades memory for fill speed, so neither belongs in an unconditional code path. Where a stage builds a large intermediate buffer only under some conditions (here, the multi-session expansion), lazy materialisation layered on zero-copy is the right tool. Finally, the whole pipeline should be profiled before any end-to-end win is claimed, since a large stage-level gain can vanish into a stage that dominates the total, as the statistics step does here.

6.6 Limitations

The reduce stage's wall-clock is sensitive to memory pressure, and early profiler-instrumented measurements were misleading (Section 6.1); we therefore report single-session runtime speedups as ranges over $n = 3$ profiler-free runs and rest the main conclusions on the far more stable memory reductions rather than on fine runtime margins. Externally, all experiments ran on one 16 GB machine, so absolute latencies and the precise point at which

parallel pressure forces swapping are hardware-specific, and the synthetic data does not capture the irregular per-subject shapes and missing sessions of real EEG recordings. We keep reduce-stage and end-to-end metrics distinct throughout, because a large stage-level gain need not imply a large end-to-end one (Section 6.4), and we establish correctness by numerical equivalence of all four statistical outputs across 126 full-pipeline checks rather than by auditing downstream clinical decisions, though the deterministic floating-point operations make divergence unlikely. Finally, tail latency is characterised for the single-session reduce stage (Table 4); the multi-session evaluation reports reduce-stage heap and runtime up to $N = 10{,}000$ rather than full tail distributions across the factorial.

7 Conclusions and Future Work

7.1 Conclusions

This paper profiled the data-reshaping (reduce) stage of the Neurophysiological Biomarker Toolbox and evaluated three memory-management strategies and their combinations under controlled, profiler-free conditions, asking how much of the stage is data movement, how that cost scales, and whether targeted strategies can cut it without changing the pipeline's scientific output.

The stage performs no arithmetic: its cost is a deep copy of the per-subject container plus a coordinate remap, both pure data movement. The deep copy scales linearly in the data, $\mathcal{O}(N \cdot C \cdot F)$, dominates the stage at scale ($\approx$88% at $N = 100{,}000$), and doubles peak memory by allocating a full duplicate of the biomarker buffer; an apparent super-linear regime in earlier measurements proved to be a profiler-and-swap artefact, not a property of the algorithm. With no computational fraction inside the stage, a memory-management change can address essentially all of it.

That paid off, with the benefit concentrated on memory and on the regime where each strategy runs. At a single session, zero-copy views remove the duplicate allocation (cutting reduce-stage peak heap by 306$\times$ at $N = 1{,}000$ and $\approx$2.5 GB at $N = 100{,}000$, with a runtime gain that grows with cohort size), while layout pinning and lazy materialisation are inert and add only setup overhead. With repeated sessions, where the stage builds an expansion buffer, lazy materialisation becomes decisive (195$\times$ smaller peak heap, held flat as the session count grows) and layout pinning merely trades memory for fill speed. All 126 full-pipeline equivalence checks passed, so none of this costs any scientific fidelity. The reduce stage is, in effect, eliminated: cut 97$\times$ in runtime and to a near-constant memory footprint. The end-to-end gain is a more modest 1.2–1.4$\times$, but that is a ceiling set by Amdahl's law and by the statistics step, whose dominant cost is genuine arithmetic rather than data movement, a kind of cost these strategies are not built to address, not a shortfall in the optimisation itself.

Two lessons generalise beyond NBT. A deep copy's worst cost is usually its memory rather than its time: doubling a stage's footprint is what, under parallel load, separates a cohort sweep that fits in RAM from one that swaps, so removing eager deep copies belongs first in any optimisation effort, ahead of layout or evaluation-order tuning. And a stage-level speedup means little until it is weighed against the whole pipeline: here an orders-of-magnitude reduction in one stage barely moves the total, because another stage dominates it.

7.2 Future Work

The clear next target is that statistics step. Its dominant cost is an independent t-test run in a Python loop over every (channel, frequency) cell; replacing the loop with a single vectorised test over the whole grid would remove thousands of per-cell calls and is the most promising route to a genuine end-to-end speedup. Paired with the zero-copy and deferred-evaluation discipline developed here (streaming the comparison directly over the lazy descriptor so the full $(\mathrm{ch}, \mathrm{fr}, M, N)$ buffer is never materialised at once), it is the natural continuation of this work. Three further directions follow:

- **Wider concurrency and repetition sweep:** the concurrency results (Section 4.4) already show the baseline's memory doubling degrading disproportionately under parallel load; extending the sweep to more worker counts and larger repetition counts would map the swap threshold precisely and tighten the single-session runtime intervals.
- **Hardware generalisability:** all experiments used one 16 GB machine, so reproducing them on systems with different memory capacities and cache hierarchies would establish how hardware-specific the swap threshold and speedup ratios are.
- **Transfer to other pipelines:** eager deep copies over array-backed containers are not NBT-specific, so testing whether the same zero-copy fix applies to comparable neuroimaging or scientific-Python pipelines would broaden the impact.

References

[1] F. Mushtaq *et al.*, "One hundred years of EEG for brain and behaviour research," *Nature Human Behaviour*, vol. 8, no. 8, pp. 1437–1443, 2024.

[2] A. Chaddad, Y. Wu, R. Kateb, and A. Bouridane, "Electroencephalography signal processing: A comprehensive review and analysis of methods and techniques," *Sensors*, vol. 23, no. 14, p. 6434, 2023.

[3] H. Zhang, Q.-Q. Zhou, H. Chen, X.-Q. Hu, W.-G. Li, Y. Bai, J.-X. Han, Y. Wang, Z.-H. Liang, D. Chen, *et al.*, "The applied principles of eeg analysis methods in neuroscience and clinical neurology," *Military Medical Research*, vol. 10, no. 1, p. 67, 2023.

[4] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, pp. 81:1–81:14, ACM, 2019.

[5] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[6] U. Drepper, "What every programmer should know about memory," tech. rep., Red Hat, Inc., 2007.

[7] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. Del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[8] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, pp. 130–136, 2015.

[9] D. Trübutschek *et al.*, "EEGManyPipelines: A large-scale, grassroots multi-analyst study of electroencephalography analysis practices in the wild," *Journal of Cognitive Neuroscience*, vol. 36, no. 2, pp. 217–224, 2024.

[10] M. Kandemir, J. Ramanujam, and A. Choudhary, "Improving cache locality by a combination of loop and data transformations," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 159–171, 1999.

[11] 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.

[12] G. Zhang and X. Shen, "Best-effort lazy evaluation for Python software built on APIs," in *Proceedings of the 35th European Conference on Object-Oriented Programming*, ECOOP '21, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[13] P. Virtanen *et al.*, "SciPy 1.0: Fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[14] J.-W. Hong and H. T. Kung, "I/O complexity: The red-blue pebble game," in *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing (STOC '81)*, pp. 326–333, ACM, 1981.

[15] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 285–297, IEEE, 1999.

A Supplementary Data

This appendix collects the per-scale tables referenced in Section 4. All timings are instrumentation-free wall-clock measurements (Section 3.1).

A.1 Clean Per-Scale Reduce-Stage Tables (single session)

Table 7 reports clean wall-clock reduce-stage latency for the baseline and three representative configurations across input sizes. Each value is the mean of $n = 3$ isolated runs. The zero-copy configurations (Cands. 1 and 6) flatten the deep-copy cost, while the layout-pin configuration (Cand. 2) retains the deep copy and sits above the baseline. Cands. $3, 4, 5, 7$ follow the same grouping (Section 4.2).

Table 8 decomposes the baseline reduce stage into its deep-copy and remap components, Table 9 gives the baseline runtime and peak memory against the data-buffer size, and Table 10 reports peak memory per configuration. These support Sections 4.1 and 4.2.

N	**Baseline (s)**	**Cand. 1 (A)**	**Cand. 2 (B)**	**Cand. 6 (A+B+C)**
10	0.0124	0.0161	0.0291	0.0284
31	0.0124	0.0160	0.0317	0.0310
100	0.0129	0.0156	0.0334	0.0283
316	0.0136	0.0151	0.0352	0.0298
1,000	0.0173	0.0157	0.0400	0.0308
3,162	0.0259	0.0170	0.0534	0.0343
10,000	0.0551	0.0198	0.0956	0.0386
31,622	0.1584	0.0325	0.2331	0.0614
100,000	0.6325	0.1539	0.9623	0.1456

Table 7: Clean reduce-stage wall-clock (s) versus N at a single session (mean of $n = 3$ isolated runs, no profiler). Corresponding peak resident-set deltas are ≈0 for the zero-copy configurations (Cands. 1, 6) and track the buffer size for the baseline and Cand. 2 (Table 9).

N	**deep copy (s)**	**remap (s)**	**copy share**
1,000	0.005	0.011	32%
10,000	0.040	0.016	71%
31,622	0.126	0.027	82%
100,000	0.603	0.086	88%

Table 8: Reduce-stage decomposition at a single session, by direct wall-clock timing (mean of $n = 3$ isolated runs). The stage is entirely data movement: a deep copy plus a coordinate remap, no arithmetic. The deep copy's share rises with N because it is linear in the data size while the remap grows sub-linearly.

N	**reduce (s)**	**peak RSS (MB)**	**buffer (MB)**
1,000	0.017	24	25
10,000	0.055	248	248
31,622	0.158	787	784
100,000	0.633	2482	2480

Table 9: Baseline reduce-stage runtime and peak resident-set delta versus cohort size (single session, isolated single-process runs, $n = 3$). Peak memory tracks the raw data-buffer size ($N \cdot C \cdot F \cdot 8$ bytes), confirming the copy is memory-bound and approximately linear in N.

Configuration	N=1,000	N=100,000
Baseline	24.4 MB	2482 MB
Cand. 1 zero-copy (A)	≈0	≈0
Cand. 2 layout pin (B)	24.7 MB	2484 MB
Cand. 6 all three	≈0	≈0

Table 10: Reduce-stage peak resident-set delta at a single session (`psutil`). Any configuration containing zero-copy eliminates the duplicate allocation, while configurations retaining the deep copy allocate the full buffer. The $N = 1{,}000$ heap figures by `tracemalloc` are 49.1 MB (baseline) vs 0.16 MB (zero-copy), a 306× reduction.

A.2 Multi-Session: Peak Heap vs. Session Count

Table 11 gives the full data behind the M-scaling claim in Section 4.3: reduce-stage peak heap at a fixed $N = 1{,}000$ as the number of recording sessions per subject grows. Lazy+zero-copy (Cands. 3, 6) stays essentially flat, while every eager variant grows with M.

Configuration	**Strat.**	M=1	M=2	M=3
Baseline	–	25.0	149.8	224.4
Cand. 1 zero-copy	A	0.18	99.8	149.5
Cand. 2 layout	B	25.0	245.6	368.2
Cand. 3 lazy	A+C	0.18	0.91	1.11
Cand. 4 combined	A+B	0.18	195.9	293.6
Cand. 5 lay.+lazy	B+C	25.0	50.6	75.7
Cand. 6 all three	A+B+C	0.18	0.91	1.11
Cand. 7 lazy only	C	25.0	50.6	75.7

Table 11: Reduce-stage peak heap (MB) versus number of recording sessions M at $N = 1{,}000$ ($n = 7$). Lazy+zero-copy (Cands. 3, 6) stays flat as M grows; every eager variant scales with M. At $M = 1$ Cand. 7 equals the baseline exactly, confirming that lazy without zero-copy degenerates to the baseline when the expansion path is not entered.

B Time and Space Complexity of the Reduce Stage

This appendix derives the analytical cost of each strategy, keeping *time* and *space* separate. Because the reduce stage performs no arithmetic, its operations are bound by the volume of data they move rather than by computation, so we count cost in data moved, following the I/O-complexity perspective of Hong and Kung [14] and the ideal-cache model of cache-oblivious algorithms [15]; for bandwidth-bound bulk operations this volume is also proportional to wall-clock time [5]. We treat the per-recording payload (C channels and F frequency bins) as parameters rather than constants, precisely so the layout and materialisation effects, which act on exactly this payload, remain visible in the asymptotics (in our experiments $C = 64$ and $F = 50$ are fixed).

Notation. A cohort has N subjects with M recording sessions each, giving $R = N \cdot M$ recordings; each recording carries a $C \times F$ biomarker payload, so the dominant array holds $\Theta(NMCF)$ elements. The baseline copies the whole container with a recursive deep copy at every stage boundary (Section 2); at $M \geq 2$ the reduce step additionally allocates an expansion buffer of $\Theta(NMCF)$ elements and fills it by iterating over the R recordings, whereas at $M = 1$ it only re-labels the existing arrays ($\Theta(R)$ index work, no payload copy). Table 12 summarises the resulting per-strategy time and space complexity in each regime; the discussion below works through its entries.

Discussion. Two improvements are asymptotic in the payload factor $C \cdot F$, not merely constant-factor. *Zero-copy (A)* removes the container deep copy, so at $M = 1$ both time and peak space fall from $\Theta(NCF)$ to $\Theta(N)$: the stage no longer touches the $C \cdot F$ payload at all, only the $\Theta(R)$ index. At $M \geq 2$, however, A still allocates the eager expansion buffer, so its complexity stays $\Theta(NMCF)$. It only removes the *source* duplication, lowering the constant (the drop from 1494 MB to 997 MB in Table 5). *Lazy materialisation (C)*,

		$M = 1$		$M \geq 2$	
Variant	**Strat.**	**Time**	**Space**	**Time**	**Space**
Baseline	–	$\Theta(NCF)$	$\Theta(NCF)$	$\Theta(NMCF)$	$\Theta(NMCF)$
Zero-copy	A	$\Theta(N)$	$\Theta(N)$	$\Theta(NMCF)$	$\Theta(NMCF)$
Layout pin	B	$\Theta(NCF)$	$\Theta(NCF)$	$\Theta(NMCF)$	$\Theta(NMCF)^{\dagger}$
Lazy	A+C	$\Theta(N)$	$\Theta(N)$	$\Theta(NM)$	$\Theta(NM)$

Table 12: Reduce-stage time and peak-space complexity, by strategy and session regime, with the per-recording payload $C \cdot F$ kept as a parameter. At $M = 1$ the expansion path is not entered, so Strategy B reduces to the baseline (its extra setup is constant overhead) and Strategy C is inert (alone it equals the baseline; with A it equals zero-copy). †Layout pinning shares the baseline's asymptotic class but carries a larger constant: it transiently holds the Fortran-order buffer *and* its C-contiguous conversion (`np.ascontiguousarray`), roughly doubling the expansion-buffer footprint. The lazy figures are for the reduce stage; the deferred $\Theta(NMCF)$ buffer is materialised later, in the statistics step.

paired with A, replaces the eager buffer with a descriptor storing $\Theta(R) = \Theta(NM)$ index metadata and a shared view of the source, lowering the reduce-stage peak from $\Theta(NMCF)$ to $\Theta(NM)$: it drops the entire $C \cdot F$ payload from the stage's footprint (consistent with the 195× reduction to 7.65 MB). Crucially, lazy alone (without A) leaves the source deep copy in place, so its reduce-stage peak remains $\Theta(NMCF)$ despite deferring the buffer, which is why C earns its keep only in combination with A (Table 11). *Layout pinning (B)* changes neither asymptotic class: it trades the baseline's $\Theta(R)$ Python-level fill loop for a single vectorised assignment (a constant-factor time win on the eager path) at the price of a second $\Theta(NMCF)$ contiguous buffer, raising peak memory. For completeness, the unmodified statistics step runs an un-vectorised Python loop over all $\Theta(C \cdot F)$ (channel, frequency) cells, each performing a group comparison over $\Theta(N)$ subjects, for $\Theta(NCF)$ time dominated by per-cell interpreter overhead; this stage, not the reduce stage, dominates end-to-end runtime (Section 4.1).