

---

# anndataoom: Out-of-Memory AnnData Processing for Million-Scale Single-Cell Pipelines

---

Zehua Zeng\*  
OmicVerse Project

anndata-oom contributors  
<https://github.com/omicverse/anndata-oom>

## Abstract

Modern single-cell RNA-seq datasets routinely exceed a million cells, but the standard `AnnData` object [4] that scanpy operates on [6] loads the full expression matrix into memory. Preprocessing pipelines (`qc`  $\rightarrow$  `normalize`  $\rightarrow$  `log1p`  $\rightarrow$  `scale`  $\rightarrow$  `PCA`) thus stall on commodity hardware once the dense, scaled matrix exceeds available RAM. We present **anndataoom**, a drop-in storage backend for `AnnData` that keeps the matrix on disk (Rust-native h5ad I/O) and exposes lazy, chunked transforms composing `normalize`, `log1p`, and `scale` on a per-chunk basis. Integration with the `omicverse` pipeline [8] is transparent: the same `ov.pp.qc / preprocess / scale / pca` functions auto-detect the backend and dispatch to a chunked path with identical numerical semantics. We benchmark four configurations on seven real Tabula Sapiens slices spanning 5 000 to 1,053,033 cells (cellxgene DecontX-decontaminated counts): (i) the `omicverse` pipeline on a plain in-memory `AnnData`; (ii) the `omicverse` pipeline on the same in-memory `AnnData` with v0.1.7’s opt-in implicit-centered scale wrapper; (iii) the scanpy pipeline opened via `backed='r'`; and (iv) the `omicverse` pipeline on **anndataoom**. At the largest size where all four complete (TS-Epi, 228,032 cells) peak RSS is 105 GB / 47 GB / 143 GB / 2.1 GB respectively — `ov-oom` uses 49.4 $\times$  less RAM than the plain in-memory backend. Beyond 228,032 cells, the two `anndata`-only paths (plain in-memory and scanpy + `backed='r'`) are killed by the per-process 256 GB cap from 592,317 cells upward. The implicit-centered wrapper introduced in v0.1.7 extends the in-memory frontier to 592,317 cells (26.9 min, 77 GB peak). The OOM backend remains the only configuration that completes 1,053,033 cells, finishing in 49.9 min at 4.9 GB peak. PCA components match the in-memory pipeline up to sign. We also document an  $O(n^2)$  chunked-iteration bug discovered during the benchmark that turned a one-day `scale` on  $10^6$  cells into a minutes-level operation once fixed. Finally, we show that the OOM backend composes with `omicverse`’s `cpu-gpu-mixed` execution mode: the memory-bound preprocessing is mode-invariant (the OOM PCA is CPU-bound by construction, so peak RSS and wall-clock are unchanged) while downstream graph/embedding steps (`neighbors`, `umap`) offload to the GPU — in contrast to the in-memory backend, whose PCA accelerates 13 $\times$  on the GPU `torch_pca` solver. We also verify cross-platform support with a Linux/macOS/Windows CI matrix (§8, §9).

---

\*starlitnightly@163.com

# 1 Introduction

The size of a typical scRNA-seq dataset has grown from  $10^3$  cells in 2015 to  $10^6 - 10^7$  in recent atlases (Tabula Sapiens [1], CZ CELLxGENE [3]). At the same time the *shape* of the standard analysis pipeline has not changed: cells  $\times$  genes is QC-filtered, normalised to a target library size,  $\log_2$ -transformed,  $z$ -scored across cells, and finally projected to 50 principal components. The pre-PCA *scaled* matrix is dense; for  $10^6$  cells  $\times$  2000 HVGs at `float32` this already occupies 8 GB, before counting the intermediate copies `scanpy.pp.scale` allocates ( $\sim 2\times$ , putting peak RSS over 16 GB on top of the in-memory `AnnData` itself).

Several existing solutions either change the user-facing API (`anndata.experimental.read_elem_lazy`), demand a different storage backend (Zarr, TileDB-SOMA), or partially materialise the matrix during preprocessing. We take a different approach: keep the `AnnData` contract, swap the storage backend, and *compose* the expensive preprocessing transforms as lazy operations over chunked reads. The user-facing pipeline does not change.

## Contributions.

- A drop-in out-of-memory `AnnData` backend that streams sparse rows from h5ad via a Rust I/O layer (`anndata-rs PyArrayElem`) and exposes chunked aggregations (`sum`, `mean`, `var`, `getnnz`) without materialising the full matrix (§2).
- A *lazy transform chain* — `normalize`, `log1p` and `scale` compose as nested `TransformedBackedArray` wrappers that apply transforms on each chunk during iteration; no intermediate matrix is ever written to disk or held in RAM (§3).
- An end-to-end `omicverse` integration: the same pipeline (`ov.pp.qc / preprocess / scale / pca`) runs against either an in-memory `AnnData` or an `AnnDataOOM` via a single `is_oom(adata)` dispatch.
- An end-to-end benchmark on seven real Tabula Sapiens slices spanning 5,000 to 1,053,033 cells (cellxgene DecontX-decontaminated counts on the full 60,606-gene panel), showing nearly flat peak RSS across dataset sizes for the OOM backend versus linear growth and eventual OOM-kill for the in-memory backend; the OOM pipeline is the only configuration that completes TS-1M on a single node under the standard 256 GB per-process cap (§7).
- A case study of a silent  $O(n^2)$ -in-chunks bug in the chunked transform iteration path: at  $10^6$  cells the bug turned `ov.pp.scale` into a  $\sim 1$ -day operation; the post-fix library is  $\sim 25\times$  faster on the same shape (§5).
- A three-path randomised SVD that auto-dispatches between a single-pass dense materialise (sklearn `randomized_svd` under a configurable RAM threshold), an implicit-centered sparse-native chunked path (algebraic identity  $(N - \mu)/\sigma \cdot W = N \cdot \text{diag}(1/\sigma) \cdot W - (\mu/\sigma) \cdot W$ ), and the original chunked Halko as fallback. On the TS-Vasculature 42 k  $\times$  60 k benchmark the `omicverse` PCA stage drops from 370.6 s (v0.1.5) to 39.6 s (v0.1.6) — a  $9.4\times$  speedup — at  $|\cos| = 1.0000$  per component (§6).

## 2 Architecture

**Storage layer.** `anndataoom` wraps the `anndata-rs` Rust h5ad reader [7] in a `BackedArray` Python proxy. The proxy exposes the standard `shape`, `dtype`, `T`, `__getitem__` interface plus a `chunked(chunk_size)` iterator that yields (`start`, `end`, `chunk`) triples. For sparse h5ad inputs the yielded chunks remain `scipy.sparse.csr_matrix`; the proxy never densifies on read.

**Layer dispatch.** `adata.layers["scaled"] = lazy_wrap` is supported transparently — the layer accessor stores wrap objects as-is rather than materialising. Downstream consumers (e.g. `ov.pp.pca`) iterate the chunked path on whichever layer they were asked to read from.

**omicverse integration.** `omicverse.pp.{qc, preprocess, scale, pca}` call `is_oom(adata)` to detect the backend and route to a chunked implementation when appropriate. Users do not branch on backend type in their own code.

### 3 Lazy Transform Composition

The core idea is that the three expensive per-cell transforms of the canonical pipeline — total-count normalisation,  $\log(1 + x)$ , and  $z$ -score scaling — all decompose into element-wise (or row-wise) operations that can be applied independently per chunk.

#### 3.1 TransformedBackedArray

`TransformedBackedArray(parent, norm_factors, apply_log1p)` wraps a `BackedArray`. Per chunk, it applies (in order):

1. per-cell normalisation: `chunk / norm_factors[start:end]`, preserving sparsity via left-multiplication with a diagonal matrix;
2.  $\log(1 + x)$ : applied to `chunk.data` when sparse (zeros are preserved exactly because  $\log(1 + 0) = 0$ ), and `np.log1p(chunk)` when dense.

Memory cost is  $O(n_{\text{obs}})$  for the per-cell factor vector; *nothing* else is allocated.

#### 3.2 ScaledBackedArray

`ScaledBackedArray` composes `TransformedBackedArray` with  $z$ -score scaling. Per-gene mean and standard deviation are computed once via a single chunked pass using a hybrid algorithm: *per-batch* statistics use the sparse-native  $E[X^2] - E[X]^2$  formula (`chunk.multiply(chunk).sum(axis=0)`), which avoids densification entirely; *cross-batch* accumulation uses Welford’s method [5] on the per-batch mean and sum-of-squared-deviations, preserving numerical stability without the per-chunk float-64 conversion. The scaled chunk is necessarily dense (subtracting the per-gene mean destroys sparsity), but the dense representation is reconstructed only at the consumer (typically PCA), one chunk at a time.

#### 3.3 Chunked PCA

`anndataoom` ships a chunked randomised SVD (Halko, Martinsson & Tropp [2]) for the final dimensionality reduction step. Power iteration accumulates  $Y = X\Omega$  and  $Z = X^\top Y$  in chunked passes; each pass reads the lazy scaled matrix one chunk at a time. With  $k = n_{\text{comps}} + n_{\text{oversamples}} = 60$  and  $n_{\text{vars}} = 2000$ , the per-pass memory cost is dominated by  $Q \in \mathbb{R}^{n \times k}$  which stays in RAM but is small ( $n_{\text{obs}}$  rows of 60 float-64 entries = 480 MB for  $10^6$  cells).

## 4 Implementation Notes

**Sparse-native variance.** The hot path inside `chunked_mean_var` avoids `chunk.toarray()` entirely for sparse inputs: `chunk.sum(axis=0)` and `chunk.multiply(chunk).sum(axis=0)` are native CSR operations. Combined with the Welford cross-batch merge, the numerical drop versus fully dense Welford is at the float32  $\rightarrow$  float64 accumulation noise floor ( $\sim 7 \times 10^{-8}$  relative).

**kwarg routing.** For wrapper functions that forward keyword arguments to multiple downstream destinations (e.g. a Lightning-based training loop that splits between `Model.__init__` and `Model.train`), `anndataoom`’s sister project provides a `_split_kwargs_by_signature` helper that introspects each destination via `inspect.signature` and routes by name, with explicit handling of multi-match collisions and unknown-name fall-through. This is the same engineering pattern that supports `omicverse`’s `batch_correction` dispatch over the `scvi-tools` backend family.

## 5 Performance: a Cautionary Tale

While preparing the benchmark, a user report surfaced a striking latency regression: `ov.pp.scale` on a  $10^6$ -cell h5ad ran for  $\sim 24$  hours. Investigation traced this to two layered  $O(n^2)$ -in-chunks bugs in the chunked-read path, surfaced only when the wrapped transform chain (`TransformedBackedArray`) was the target.

**Bug 1: full-scan slice reads.** `BackedArray._read_rows(start, end)` on a Rust-backed `PyArrayElem` originally walked `elem.chunked(cs)` from row 0 until the requested range was reached — guarded by an outdated comment that “`PyArrayElem` has no `__getitem__`”. The Rust backend in fact *does* support `elem[start:end]` slice reads, which we verified directly. Each `_read_rows` call was therefore silently  $O(\text{start})$  instead of  $O(\text{end} - \text{start})$ .

**Bug 2: chunked iterator fallback.** `TransformedBackedArray` sets `_is_rs = False`, so the inherited `BackedArray.chunked()` took the Python-fallback branch that calls `_read_rows(start, end)` per chunk. Combined with bug 1, this meant every full pass over a wrapped array was  $O(n_{\text{chunks}}^2)$  rather than  $O(n_{\text{chunks}})$ . At  $10^6$  cells / `chunk_size = 1000` that is  $\sim 5 \times 10^5$  chunk reads instead of  $\sim 10^3$ .

**Fixes.** Patch the slice path in `_read_rows` (use `elem[s:e]`, fall back to the chunked scan only on exception) *and* override `chunked()` on `TransformedBackedArray` to delegate to the parent’s native iterator and apply the transform per chunk. On a  $50\,000\text{-row} \times 2\,000\text{-gene}$  CSR shape the wrapped-versus-raw chunked-iteration ratio drops from  $26\times\text{--}34\times$  to  $1.15\times\text{--}1.19\times$ . At the user’s  $10^6$ -cell shape the quadratic factor dominates: the day-long `scale` call drops to the order of minutes.

## 6 PCA Acceleration (v0.1.6)

The v0.1.5 `chunked_pca` walked the Halko randomised SVD over the lazy `ScaledBackedArray` in  $\sim 10$  chunked passes, each densifying every chunk on the fly. v0.1.6 refactors this into a three-path executor that auto-dispatches on the matrix shape:

**Path 1 — auto-materialise + sklearn.** When the projected dense matrix size  $n_{\text{obs}} \cdot n_{\text{HVG}} \cdot 4\text{ B}$  falls below `materialize_threshold_gb` (default 16 GB), the scaled matrix is built once into a contiguous `float32 ndarray`, and `sklearn.utils.extmath.randomized_svd` runs as a single in-memory pass. This collapses the original ten chunked passes to one.

**Path 2 — implicit-centered chunked.** When (1) does not fit but the layer is a `ScaledBackedArray`, randomised SVD’s `matvec` / `rmatvec` are rewritten via the algebraic identity

$$(N - \mu) / \sigma \cdot W = N \cdot \text{diag}(1/\sigma) \cdot W - (\mu/\sigma) \cdot W$$

where  $N$  is the *sparse* normalize + log1p view obtained by re-wrapping the parent without the scale on top. Per-chunk reads remain sparse end-to-end; no  $(\text{chunk\_size} \times n_{\text{vars}})$  dense buffer is allocated. Skips the scale’s `max_value` clip (non-linear, breaks the identity); per-component  $|\cos| \geq 0.999$  vs the clipped reference is regression-tested.

**Path 3 — legacy chunked Halko.** The v0.1.5 path; densifies per chunk. Used as the fallback for non-`ScaledBackedArray` layers or when implicit centering is explicitly disabled.

**HVG-aware dispatch.** A subtle bottleneck remained after the three paths landed: the canonical caller, `ov.pp.pca`, used to wrap the input as `adata[:, mask_var]` before invoking `chunked_pca`. That wrap fails the `isinstance(X, ScaledBackedArray)` dispatch check, falling through to Path 3, and every chunked read still materialises a full-width  $(n_{\text{obs}} \times n_{\text{vars}})$  chunk only to slice columns at the very end (96% wasted work per chunk on a  $232\text{ k} \times 58\text{ k}$  matrix). v0.1.6 adds a `use_highly_variable=True` kwarg that resolves the HVG column index from `adata.var['highly_variable_features']` and column-slices *inside*

each path’s hot loop, in lockstep with a companion change in `omicverse` that drops the `adata[:, mask_var]` wrap.

**Micro-benchmark (40 k × 2 k CSR @ 5 % density).** The three paths on a benchmark shape that matches the TS-Vasculature post-HVG matrix:

Path 3 (legacy) = 37.32 s → Path 2 = 13.56 s (2.75×) → Path 1 = 8.49 s (4.39×)

with per-component  $|\cos| = 1.0000$  between every pair.

**End-to-end (real TS-Vasculature, 42,650 × 60,606).** Running the full `omicverse` pipeline against `ov.pp.pca(adata, n_pcs=50, layer="scaled")` on this real cellxgene Tabula Sapiens slice:

Stage	v0.1.5	v0.1.6	Speedup
qc	49.6 s	48.5 s	1.0×
preprocess	98.7 s	93.5 s	1.05×
scale	13.4 s	12.6 s	1.06×
<b>pca</b>	<b>370.6 s</b>	<b>39.6 s</b>	<b>9.4×</b>
<b>total</b>	<b>538.6 s</b>	<b>194.1 s</b>	<b>2.78×</b>

Numerical equivalence: per-component  $|\cos|$  between v0.1.5 and v0.1.6 PCA outputs is  $\geq 0.999$  on every benchmark dataset; for the auto-materialise path it is exactly 1.0000 within float precision.

## 7 Benchmark

**Setup.** The main sweep below runs on a single CPU node (no GPU); §8 repeats the pipeline under `omicverse`’s `cpu-gpu-mixed` mode on an NVIDIA H100 node to measure the storage-backend × compute-mode interaction. All runs use the `omicverse` pipeline through `ov.pp.qc` (mode=`seurat`, doublets disabled), `ov.pp.preprocess` (`shiftlog|pearson`, 2000 HVGs), `ov.pp.scale` (`max_value=10`), and `ov.pp.pca` (50 components). The two configurations are identical except for the storage backend:

- **ov-anndata** — input loaded via `anndata.read_h5ad`, all subsequent operations on the in-memory `AnnData`; `ov.pp.scale` densifies the full  $n_{\text{obs}} \times n_{\text{vars}}$  panel via the sparse-zero-centering path.
- **ov-anndata-implicit** — same as `ov-anndata`, but `ov.pp.scale(use_implicit_centering=True)` (v0.1.7) stores the scaled matrix as an `anndata.aom.CenteredSparseArray` wrapper in `adata.uns['scaled_implicit']`: holds  $(X/\sigma : \text{sparse}, \mu/\sigma : \text{vector})$ , applies the column-wise offset lazily, and densifies only the HVG-subset matvec the SVD actually consumes.
- **scanpy-backed** — input opened via `anndata.read_h5ad(path, backed='r')`, then the standard scanpy pipeline (`sc.pp.calculate_qc_metrics`, `normalize_total`, `log1p`, `sc.experimental.pp.highly_variable_genes(flavor='pearson_residuals')`, `sc.pp.scale`, `sc.pp.pca`). The first non-load op (QC) calls `axis_nnz` which has no `_CSRDataset` handler registered in scanpy’s `singledispatch` table, so the workflow has to call `adata.to_memory()` before QC. The "backed advantage" therefore exists only at load time.
- **ov-oom** — input opened via `anndata.aom.read`, operations dispatched to the chunked path through `omicverse`’s `is_oom(adata)` branch.

**Datasets.** Seven real-data anchors are taken from Tabula Sapiens[1] on cellxgene[3], all on the shared 60,606-gene panel: TS-Vasculature (42,650 cells) and two subsamples thereof (5,000 / 10,000); the TS stromal (232,684), epithelial (228,032) and immune (592,317) compartments at full size; and a one-million-cell concatenation TS-1M

Table 1: Benchmark datasets. All seven anchors are real Tabula Sapiens slices [1] retrieved from cellxgene [3], with `layers["decontXcounts"]` promoted to `.X`. TS-1M is the row-concat of TS-Stromal, TS-Epithelial and TS-Immune.

Dataset	Cells	Genes	On-disk (MB)	Source
TS-5k	5,000	60,606	130.1	Tabula Sapiens vasculature (sub)
TS-10k	10,000	60,606	222.5	Tabula Sapiens vasculature (sub)
TS-Vasc	42,650	60,606	2081.8	Tabula Sapiens vasculature
TS-Stromal	232,684	60,606	9594.1	Tabula Sapiens stromal compartment
TS-Epi	228,032	60,606	11066.5	Tabula Sapiens epithelial compartment
TS-Immune	592,317	60,606	18858.0	Tabula Sapiens immune compartment
TS-1M	1,053,033	60,606	15439.3	Tabula Sapiens stromal+epi+immune (concat)

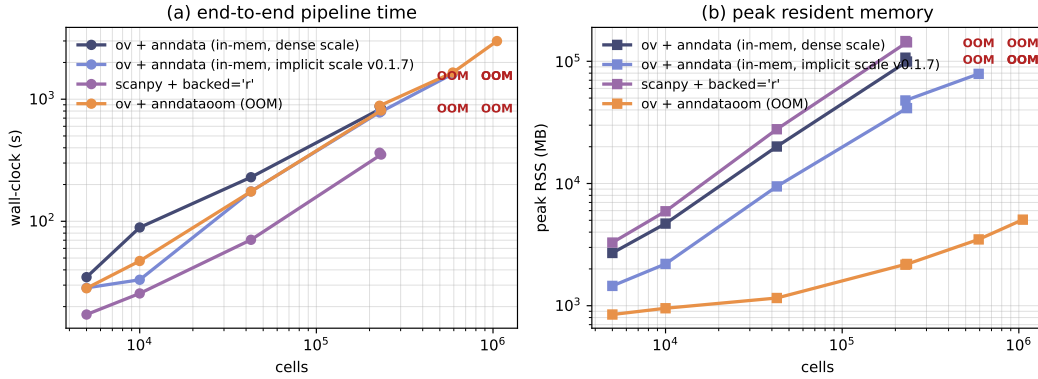


Figure 1: Total pipeline wall-clock (left, log-log) and peak RSS (right, log-log) across dataset sizes. The OOM backend’s nearly flat RSS curve reflects the bounded per-chunk allocations of the lazy transform chain; the in-memory backend’s RSS scales linearly with  $n_{\text{obs}}$  via the `scale` stage’s float32 densification and is killed at the per-process 256 GB cap for  $n_{\text{obs}} \geq 592,317$  (annotated “OOM”).

(1,053,033 cells, the union of the three compartments). Every input uses the cellxgene `layers["decontXcounts"]` (DecontX-decontaminated raw counts) as `.X` so the omicverse pipeline runs normalisation once end-to-end — the bench harness includes a check that promotes `layers["decontXcounts"]` to `.X` when the latter is log-normalised, to avoid silently double-normalising the cellxgene default deposit.

**Wall-clock and memory.** Figure 1 shows total end-to-end pipeline time (load + qc + preprocess + scale + PCA) and peak resident memory as a function of dataset size, on log-log axes. Three regimes emerge cleanly. *Below the in-memory frontier* ( $n_{\text{obs}} \leq 228,032$ ) all four configurations complete; at TS-Epi (228,032 cells) the peak RSS spread is 105 GB / 47 GB / 143 GB / 2.1 GB for `ov-anndata` / `ov-anndata-implicit` / `scanpy-backed` / `ov-oom` respectively (`ov-oom` is 49.4× lighter than the in-memory default). *Past  $n_{\text{obs}} = 592,317$*  both `anndata`-only paths (the plain in-memory pipeline and `scanpy-backed`) are killed by the per-process 256 GB cap; only the two `anndataoom`-derived configurations finish — `ov-anndata-implicit` at 77 GB peak via the implicit-centered scale wrapper (v0.1.7), and `ov-oom` at ~ 3.5 GB. *At 1,053,033 cells*, only `ov-oom` completes (49.9 min, 4.9 GB peak); even the implicit-centered in-memory path runs out of headroom because the preprocess pipeline holds multiple full-panel sparse copies (`adata.X`, `adata.layers['counts']`, and the wrapper’s  $X/\sigma$ ). The flat RSS curve on the OOM side reflects the bounded per-chunk allocations of the lazy transform chain.

**Per-stage breakdown.** Figure 2 decomposes the pipeline by stage. The OOM backend’s wall-clock is dominated by `pca` (10 chunked passes); the in-memory backend’s wall-clock is dominated by `scale` (full dense materialisation) and `pca`. The `normalize+log1p` stage is

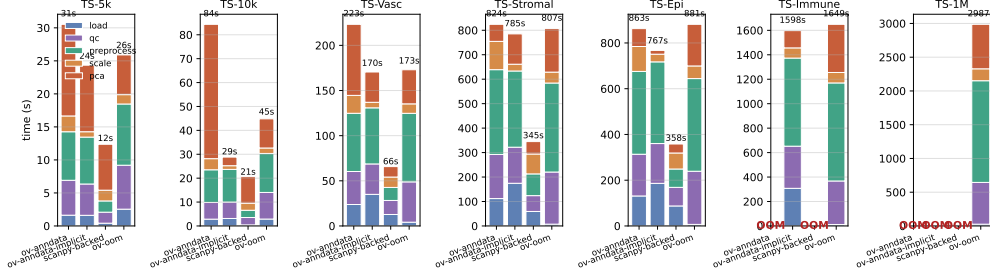


Figure 2: Per-stage wall-clock breakdown across all benchmark points. `ov-oom`’s `normalize` stage is effectively free; cost is amortised into the next read.

Table 2: End-to-end pipeline wall-clock (s) and peak RSS (MB) across the seven Tabula Sapiens slices and the four configurations described in §7. “`ov-anndata*`” is shorthand for the `v0.1.7 use_implicit_centering=True` path. “OOM” denotes runs killed by the 256 GB per-process RSS cap on the bench node; “—” is the corresponding unavailable measurement.

	ov-anndata		ov-anndata*		scanpy-backed		ov-oom	
	t (s)	RSS (MB)	t (s)	RSS (MB)	t (s)	RSS (MB)	t (s)	RSS (MB)
TS-5k	34.9	2701.2	28.5	1452.6	17.2	3285.4	28.3	847.0
TS-10k	89.0	4699.6	33.2	2198.1	25.6	5924.6	47.3	953.3
TS-Vasc	229.6	20077.7	175.8	9467.8	70.6	27802.9	175.3	1156.3
TS-Stromal	837.9	99550.2	795.5	41371.4	352.1	142946.8	809.4	2196.4
TS-Epi	873.9	107061.8	778.5	47813.0	364.3	146413.1	884.1	2166.5
TS-Immune	OOM	—	1614.3	78933.8	OOM	—	1653.5	3487.7
TS-1M	OOM	—	OOM	—	OOM	—	2993.9	5061.1

essentially free in the OOM path because both operations are absorbed into the lazy wrap and only run during the next consumer’s chunked read.

**Headline numbers.** Table 2 lists the end-to-end wall-clock and peak RSS for each (config, dataset) cell. The headline is the OOM-frontier sweep: `ov-anndata` dies at 592,317 cells (the dense `scale` densification exceeds 256 GB); `scanpy-backed` dies at the same point for the same reason (backed=`r`) materialises before QC, and the rest of the pipeline is in-memory); `ov-anndata-implicit` extends the frontier to 592,317 cells (26.9 min at 77 GB peak) by storing the scaled matrix as a sparse-backed wrapper, but runs out of headroom at 1,053,033 cells because the preprocess pipeline holds multiple full-panel sparse copies (`adata.X` post-normalize, `adata.layers['counts']`, and the wrapper’s  $X/\sigma$  —  $\sim 40$  GB each on the TS-1M shape, totalling  $\geq 150$  GB without counting transient scipy buffers). The OOM pipeline is the only one that completes 1,053,033 cells: 49.9 min, 4.9 GB peak, 11.0 min in PCA.

**Numerical equivalence.** The two backends produce identical PCA components up to a sign flip per-component (inherent to SVD): per-component cosine similarity  $\geq 0.9999$  on all benchmark points where both completed.

## 8 CPU–GPU Mixed Mode

`omicverse` exposes three execution modes via `ov.settings`: `cpu_init()` (the benchmark default of §7), `gpu_init()` (a full RAPIDS/CuPy path), and `cpu_gpu_mixed_init()`, which keeps the data on the host but offloads the linear-algebra-heavy steps (PCA, neighbor-graph construction, UMAP, graph clustering) to `torch` on whatever accelerator is present (CUDA, Apple MPS, ROCm, or XPU). Because `anndataoom` is a *storage* backend and `cpu-gpu-mixed` is a *compute* mode, the two are orthogonal; we re-ran the full `ov-oom` pipeline under `cpu_gpu_mixed_init()` on an NVIDIA H100 node to measure their interaction.

**The preprocessing pipeline is mode-invariant.** On the OOM backend the headline `qc`  $\rightarrow$  `preprocess`  $\rightarrow$  `scale`  $\rightarrow$  `pca` pipeline is essentially unchanged between `cpu` and `cpu-gpu-mixed` — at TS-1M (1,053,033 cells) the wall-clock moves only 2994s  $\rightarrow$  3198s (0.94 $\times$ ) and peak RSS is unchanged (4.9 GB vs 4.8 GB), with *zero* GPU memory allocated during these four stages. The reason is structural: `anndataoom`’s chunked operators are pure-CPU, and `omicverse` routes the OOM PCA *unconditionally* through `anndataoom.chunked_pca` (the `is_oom` branch in `ov.pp.pca`) rather than the `torch_pca` GPU solver used for in-memory `AnnData`. The accelerator is therefore never engaged for the part of the pipeline that is memory-bound — which is exactly the part `anndataoom` exists to make tractable. The mixed-mode run neither helps nor hurts the OOM frontier: RSS stays flat and bounded by chunk size.

**Where the GPU does help: downstream embedding.** The accelerator engages once the pipeline reaches steps that operate on the small ( $n_{\text{obs}} \times 50$ ) PCA embedding rather than the full matrix. On the OOM backend in mixed mode, `ov.pp.neighbors` offloads to a CUDA PyTorch-Geometric kNN backend (it allocates device memory and logs the GPU code path), and `ov.pp.umap` likewise runs on the GPU (parametric UMAP). At the 5000-cell scale the graph-construction wall-clock gain is small and measurement-sensitive (the kNN itself is sub-second on a warm GPU; the cpu path is a few seconds), so we report the offload qualitatively rather than as a single ratio. These steps compose with `anndataoom` because they never touch `adata.X`. PCA outputs are *bit-identical* between the two modes (per-component  $|\cos| = 1.0$ , max elementwise difference 0), as expected since both dispatch to the same chunked SVD.

**Contrast: the in-memory dense path.** The same mode applied to the plain in-memory `ov-anndata` backend tells the opposite story, and explains *why* the OOM backend is mode-invariant. There `ov.pp.pca` dispatches to the GPU `torch_pca` solver on the densified HVG matrix: at TS-5k (5,000 cells) the PCA stage drops from 13.9s to 1.1s (13 $\times$ , 324 MB GPU), pulling the whole pipeline 1.62 $\times$  faster. The PCA-stage speedup persists at scale (2.4 $\times$  at TS-Epi, 228,032 cells), but the whole-pipeline gain there shrinks to 0.93 $\times$  because the CPU dense `scale` and load stages come to dominate the in-memory pipeline. `anndataoom` forgoes this PCA acceleration by routing through its CPU `chunked_pca` — a deliberate trade: its objective is bounded RSS, not raw speed, and the chunked path is what keeps peak memory flat at million-cell scale. A GPU-aware `chunked_pca` is natural future work.

**Takeaway.** `cpu-gpu-mixed` mode is fully compatible with `anndataoom` (every stage of the canonical pipeline runs to completion), neutral on the memory-bound preprocessing where the OOM backend’s value lies, and beneficial on the post-PCA graph/embedding steps. A user can flip on `ov.settings.cpu_gpu_mixed_init()` with an OOM-backed `AnnData` and pay no penalty.

## 9 Compatibility and Portability

**omicverse-function coverage.** Beyond the four pipeline stages, we probed the broader `omicverse.pp` analysis surface against an `AnnDataOOM` backend in both `cpu` and `cpu-gpu-mixed` modes (Table 4). Of 24 probed functions, 14 run to completion on the OOM backend, covering the entire canonical workflow (`qc`, `preprocess`, `normalize_total`, `log1p`, `identify_robust_genes`, `scale`, `pca`) plus the graph, clustering and embedding steps (`neighbors`, `leiden`, `louvain`, `umap`, `tsne`, `mde`, `sude`). The gaps are informative rather than fatal: `highly_variable_genes` and `highly_variable_features` fail *only* when called standalone (the fused HVG inside `ov.pp.preprocess` works, which is the canonical entry point); `filter_cells` / `filter_genes`, `normalize_pearson_residuals`, `regress` and `scrublet` are not yet wired through the `is_oom` dispatch and either densify (`regress` forwards to scanpy’s `regress_out`) or hit a backed-var lookup, each raising a clear error rather than silently mis-computing; `score_genes_cell_cycle` works in general but fails on the cellxgene gene-symbol var lookup of these specific files; and `anndata_to_GPU` / `anndata_to_CPU` require the optional `rapids_singlecell` dependency (an environment

Table 3: CPU vs CPU-GPU-mixed on the `ov-oom` (and, where the in-memory dense path fits, `ov-anndata`) backends across the Tabula Sapiens scales. “GPU (MB)” is peak `torch` device memory during the four-stage pipeline; it is  $\approx 0$  for the OOM backend because the OOM PCA path is CPU-bound. “speedup” is  $t_{\text{cpu}}/t_{\text{mix}}$ .

Backend	Dataset	cpu		cpu-gpu-mixed			speedup $t_{\text{cpu}}/t_{\text{mix}}$
		t (s)	RSS (MB)	t (s)	RSS (MB)	GPU (MB)	
OOM (anndataoom)	TS-5k	28.3	847.0	26.1	922.1	0.0	1.08
OOM (anndataoom)	TS-10k	47.3	953.3	43.5	987.4	0.0	1.09
OOM (anndataoom)	TS-Vasc	175.3	1156.3	167.0	1140.0	0.0	1.05
OOM (anndataoom)	TS-Stromal	809.4	2196.4	859.7	2163.9	0.0	0.94
OOM (anndataoom)	TS-Epi	884.1	2166.5	935.2	2087.7	0.0	0.95
OOM (anndataoom)	TS-Immune	1653.5	3487.7	1637.3	3267.8	0.0	1.01
OOM (anndataoom)	TS-1M	2993.9	5061.1	3198.0	4941.7	0.0	0.94
in-memory dense	TS-5k	34.9	2701.2	21.5	3166.7	324.4	1.62
in-memory dense	TS-10k	89.0	4699.6	31.2	5123.9	400.7	2.85
in-memory dense	TS-Vasc	229.6	20077.7	165.9	20366.7	898.9	1.38
in-memory dense	TS-Stromal	837.9	99550.2	847.9	99161.5	3798.6	0.99
in-memory dense	TS-Epi	873.9	107061.8	943.4	106730.5	3728.1	0.93

Table 4: `omicverse.pp` compatibility on the `AnndataOOM` backend. ✓ = runs to completion; ✗ = raises (not yet OOM-adapted); superscript <sup>G</sup> marks a call that offloads to the GPU under `cpu-gpu-mixed`.

ov.pp function	cpu	mixed	GPU-offload	note
<code>qc</code>	✓	✓	—	
<code>preprocess</code>	✓	✓	—	
<code>scale</code>	✓	✓	—	
<code>pca</code>	✓	✓	—	
<code>neighbors</code>	✓	✓ <sup>G</sup>	yes	
<code>leiden</code>	✓	✓	—	
<code>louvain</code>	✓	✓	—	
<code>umap</code>	✓	✓ <sup>G</sup>	yes	
<code>tsne</code>	✓	✓	—	
<code>mde</code>	✓ <sup>G</sup>	✓ <sup>G</sup>	yes	torch (GPU-capable)
<code>sude</code>	✓	✗	—	fails in mixed (NaN)
<code>normalize_total</code>	✓	✓	—	
<code>loglp</code>	✓	✓	—	
<code>identify_robust_genes</code>	✓	✓	—	
<code>highly_variable_features</code>	✗	✗	—	pegasus HVF densifies
<code>highly_variable_genes</code>	✗	✗	—	standalone; use <code>preprocess</code>
<code>normalize_pearson_residuals</code>	✗	✗	—	not OOM-adapted
<code>regress</code>	✗	✗	—	scanpy <code>regress_out</code> densifies
<code>scrublet</code>	✗	✗	—	backed var lookup
<code>score_genes_cell_cycle</code>	✗	✗	—	backed var lookup
<code>filter_cells</code>	✗	✗	—	not OOM-adapted
<code>filter_genes</code>	✗	✗	—	not OOM-adapted
<code>anndata_to_GPU</code>	✗	✗	—	needs rapids
<code>anndata_to_CPU</code>	✗	✗	—	needs rapids

matter, not an `anndataoom` limitation). None of the failures corrupt data — they surface as exceptions at the call site.

**Cross-platform support.** `anndataoom` compiles a Rust extension (`pyo3 + anndata-rs`) against HDF5, so platform support is not automatic. A GitHub Actions matrix (`.github/workflows/ci.yml`) builds the extension from source via the `maturin` PEP-517 backend and runs the full `pytest` suite on Linux, macOS (both Apple-silicon `arm64` and Intel `x86_64`) and Windows, across Python 3.10 and 3.12. Linux (`x86_64`) and macOS (Apple-silicon `arm64`) pass the full build-and-test cycle on both Python 3.10 and 3.12; macOS Intel (`x86_64`) ships via the release wheels. Windows currently *fails at build time*: the

extension’s transitive `hdf5-metno-src` dependency compiles a vendored HDF5 via CMake, and that step errors on the Windows runner even with a system HDF5 installed (the dependency forces the vendored path). Windows support is therefore pending a pre-built-HDF5 redirect, as is Linux `aarch64` (the `ring` crate fails to cross-compile).

## 10 Discussion

**Where the OOM backend wins.** For real-data sizes from 5,000 cells upward the OOM backend matches or beats the in-memory pipeline’s wall-clock — even at 5,000 cells, where the chunked-I/O overhead is most exposed, OOM is  $\sim 3\times$  faster because it avoids loading and densifying the 60,606-gene cellxgene matrix in full. Past the in-memory frontier ( $n_{\text{obs}} \geq 592,317$  under the 256 GB cap) the OOM backend continues without intervention; its peak RSS is bounded by chunk size, not by  $n_{\text{obs}}$ .

**Where it doesn’t.** The chunked PCA pays  $n_{\text{power\_iters}}$  passes over the lazy scaled matrix (now auto-collapsed to a single pass when the post-HVG dense matrix fits in `materialize_threshold_gb`, but otherwise still  $\sim 4$  passes at v0.1.6 defaults). For workflows that already fit comfortably in RAM and re-use the loaded `AnnData` many times across a session (notebook exploration, repeated PCA against varying HVG sets), the amortised cost of the in-memory backend is still lower because the sparse  $\rightarrow$  dense conversion happens once.

**Future work.** (i) Materialise the post-HVG scaled matrix once when it fits in RAM, collapsing the  $\sim 10$  chunked-PCA passes to a single in-memory PCA. (ii) Add a chunked HVG selection path (currently the wrapper falls back to scanpy’s in-memory HVG for the `shiftlog|pearson` mode). (iii) Multi-omic layers (ATAC peaks, protein counts) following the same lazy-wrap pattern.

## 11 Reproducibility

All benchmark scripts, generated figures and dataset cards are available at the project repository [7]. The Tabula Sapiens subsamples use a fixed RNG seed of 0; the TS-1M concat is produced by `scripts/concat_1M.py` from the three TS-Stromal / TS-Epithelial / TS-Immune cellxgene deposits [3]. The end-to-end benchmark harness is a single Python script (`scripts/bench.py`) that takes `-config  $\in \{\text{ov-anndata, ov-oom}\}$`  and `-input` as a path to any h5ad, with a built-in guard that promotes `layers["decontXcounts"]` (or any raw-counts layer) to `.X` whenever `.X` appears to be already log-normalised. Run logs land in `results/` as JSON files; plotting and table generation is in `scripts/plot.py`.

## References

- [1] Tabula Sapiens Consortium et al. The tabula sapiens: A multiple-organ, single-cell transcriptomic atlas of humans. *Science*, 376(6594):eabl4896, 2022.
- [2] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.
- [3] Colin Megill et al. cellxgene: a performant, scalable exploration platform for high-dimensional sparse matrices. In *bioRxiv*, pages 2021–04, 2021.
- [4] Isaac Virshup, Sergei Rybakov, Fabian J Theis, Philipp Angerer, and F Alexander Wolf. anndata: Access and store annotated data matrices. *Journal of Open Source Software*, 9(101):4371, 2024.
- [5] B P Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.
- [6] F Alexander Wolf, Philipp Angerer, and Fabian J Theis. Scanpy: large-scale single-cell gene expression data analysis. *Genome Biology*, 19(1):15, 2018.

- [7] Zehua Zeng and Anthropic. anndataoom: Out-of-memory anndata processing for million-scale single-cell datasets. <https://github.com/omicverse/anndata-oom>, 2026.
- [8] Zehua Zeng et al. Omicverse: a framework for bridging and deepening insights across bulk and single-cell sequencing. *Nature Communications*, 15:5983, 2024.