AHAS: Adaptive Hybrid Accelerator for ScRNA

Shivaritha Sakthi Rengasamy

***Abstract*—Sparse matrix–dense matrix multiplication (SpMM) is a key kernel in single-cell RNA sequencing (scRNA-seq) analysis, where expression matrices are extremely sparse and irregular. CPU-only implementations handle sparsity well but leave GPU throughput unused, while naïvely offloading sparse matrices to GPUs often becomes memory-bandwidth bound and underutilizes the device. We present a sparsity-aware, hybrid CPU/GPU SpMM framework for real 10x scRNA data that combines PIM-inspired data reduction, 2D tiling, tile-density prediction, and a hybrid execution path. First, we apply an offline, PIM-style filter that removes low-variance genes and reduces nnz in (X) by up to 50% while preserving highly variable ("rare") genes. The filtered matrix is then partitioned into 64×64 tiles; for each tile we estimate density and classify it as dense (5%) or sparse. Dense tiles are packed and routed to the GPU, where they are computed with cuBLAS SGEMM, while sparse tiles are kept on the CPU and processed with a CSR SpMM kernel using OpenMP. Row and column permutations further improve data locality and tile density. We evaluate this design on multiple real scRNA-seq datasets and show that the combination of PIM-style filtering and hybrid CPU/GPU tiling reduces data movement, adapts to heterogeneous sparsity patterns, and improves end-to-end SpMM performance relative to a CPU-only baseline.**

***Index Terms*—scRNA-seq, SpMM, sparsity, hybrid CPU–GPU, tiling, PIM-inspired filtering**

I. Introduction

SINGLE-CELL RNA sequencing (scRNA-seq) has enabled large-scale profiling of gene expression at single-cell resolution, producing matrices with tens of thousands of genes and thousands to tens of thousands of cells [1]. These gene–cell matrices are extremely sparse, often below 10% density, and the nonzeros are distributed in highly irregular, biologically driven patterns [1]. Sparse matrix–dense matrix multiplication (SpMM) on this data, for example to compute $Y = XW$ for downstream dimensionality reduction or neural models, is both compute- and data-movement intensive, and tends to be dominated by memory traffic rather than raw floating-point throughput [2], [3].

Conventional SpMM implementations sit at two extremes. CPU-only implementations using CSR can handle irregular sparsity well but do not exploit the massive parallelism available on modern GPUs, while naïvely mapping the entire sparse matrix to a GPU as if it were dense underutilizes the hardware due to low operational intensity, irregular memory access, and kernel-launch and host–device transfer overheads [2], [3]. This tension is particularly pronounced for scRNA-seq, where sparsity is heterogeneous not only across datasets but also within a single matrix; some regions are relatively dense and GPU-friendly, while others are extremely sparse and better suited to a CPU sparse kernel [1]. A single, uniform execution strategy (CPU-only or GPU-only) cannot adapt to these local patterns and often leaves either performance or resources on the table.

We address this challenge with an adaptive hybrid SpMM framework that combines PIM-inspired filtering, 2D tiling, and per-tile CPU/GPU scheduling. At a high level, we first apply an offline, PIM-style filter that operates at the data-source level: using variance across cells, we identify highly variable genes and remove low-variance genes that are likely to be noise or housekeeping. This reduces the number of nonzeros and the dimensionality of X while explicitly preserving rare, informative genes. We then partition the (optionally filtered) matrix into 64×64 tiles and compute a simple density metric per tile. Tiles whose density exceeds a threshold (5% in our experiments) are classified as dense and routed to a cuBLAS SGEMM path on the GPU; tiles below the threshold are treated as sparse and computed on the CPU using a CSR SpMM kernel with OpenMP. Row and column permutations further improve the effectiveness of this strategy by clustering nonzeros into denser tiles, increasing the fraction of work that maps efficiently to the GPU.

Prior work in genomics has shown that pushing compute closer to storage with PIM can significantly reduce data movement for long-read pipelines [4]–[7], while sparse accelerators for DNNs and GNNs co-design tiling, dataflow, and formats to match fixed hardware [3], [8]. In contrast, our goal is to target single-cell RNA SpMM specifically and to make simple, per-tile runtime decisions about both where to execute (CPU vs GPU) and how much data to keep (via PIM-style filtering) for this workload. The result is a unified software PIM emulator and hybrid CPU/GPU SpMM path that adapts to the heterogeneous sparsity patterns seen in real scRNA-seq datasets, and exposes the performance–data-reduction trade-offs needed to reason about future near-memory or PIM implementations.

II. Related Work

Recent work on sparse acceleration and genomics motivates our design from both the storage and compute sides. PIM-Quantifier, GenPIP, and related in-memory genome accelerators target long-read genomics pipelines where I/O and intermediate data movement dominate latency and energy; they push compute and filtering closer to storage using SSD- or ReRAM-based PIM, and show that cutting bytes near data can yield large end-to-end speedups over CPU/GPU baselines [4]–[7]. Sparse accelerators such as SparseMap, as well as recent surveys of deep-learning hardware for heterogeneous platforms, instead focus on sparse linear algebra and DNNs: they co-search or co-design dataflow, tiling, and sparse formats to match a fixed accelerator, or add a small number of online modes to adapt to changing sparsity while reducing DRAM traffic [2], [3], [8]. Together, these systems demonstrate two recurring themes: (1) real speedups come from matching

sparse format and tiling to the hardware, and (2) moving less data, either via compression, pruning, or near-memory filtering, is as important as raw FLOPs.

However, these designs are either storage-centric genomics accelerators or offline-planned sparse accelerators; none of them targets single-cell RNA-seq SpMM specifically, nor do they expose a GPU design that makes per-tile, run-time decisions about both where to execute and how much data to keep for this workload. In particular, prior work does not report, in one place, the joint effect of tile-wise scheduling and data reduction on bytes moved, format-conversion overhead, and GPU utilization for scRNA matrices. Our work fills this gap by combining a lightweight, PIM-style gene filter with a sparsity-aware tiler and a hybrid CPU/GPU SpMM path, and by evaluating the full cost/benefit trade-off on realistic scRNA workloads.

III. Motivation

Accelerating sparse matrix–dense matrix multiplication for single-cell RNA-seq (scRNA-seq) runs into two coupled bottlenecks: data movement and extreme, irregular sparsity. Modern scRNA-seq datasets produce gene–cell matrices with tens of thousands of genes and thousands of cells, yet less than 10% of the entries are nonzero [1]. Moving such matrices around in compressed sparse formats (indices, indptr, and values) is expensive; for low operational intensity (FLOPs per byte), the time spent transferring and unpacking sparse structures can dominate the time spent doing arithmetic, especially on GPUs [2], [3]. Irregular, index-driven accesses also break coalescing and caching on the GPU, so even after paying the transfer cost the device is often memory-bound and underutilized [2].From the hardware's perspective, this appears as low GPU utilization: the streaming multiprocessors are often idle not because there is no work to do, but because they are stalled waiting for sparse data and index structures to arrive from memory.

At the same time, sparsity in scRNA is not uniform. Some regions of the matrix, such as broadly expressed housekeeping genes, are relatively dense and well suited to dense-like GPU kernels, while other regions, such as rare cell-type–specific markers, are extremely sparse and are better handled on the CPU with a CSR SpMM kernel [1]. Treating the entire matrix uniformly, either as a CPU-only sparse computation or as a single GPU offload, ignores this heterogeneity. Dense tiles can end up stuck on the CPU, leaving GPU throughput idle, while extremely sparse tiles pay GPU transfer overhead for very little compute. Poor locality from scattered nonzeros further degrades cache behavior on both CPU and GPU. These combined effects motivate an approach that first reduces data before it ever reaches the accelerator (for example using a PIM-style filter that drops low-variance genes) and then adapts execution at fine granularity, routing each region of the matrix to the most appropriate engine, CPU or GPU, based on its local sparsity and data-movement cost.

IV. Key Insights

This work explores how to make sparse matrix multiplication for single-cell RNA sequencing data behave more like a first-class citizen on heterogeneous hardware, rather than an afterthought bolted onto dense GPU kernels. The core idea is to combine PIM-inspired data reduction with a sparsity-aware tiler and a simple runtime decision rule that lets the system decide, tile by tile, whether the CPU or GPU is the better place to run SpMM.

First, we introduce a PIM-inspired, offline filtering stage that reduces the size of the scRNA matrix while explicitly preserving biologically important structure. Using a variance-based, highly-variable-gene (HVG) criterion, we classify genes into three groups: high-variance "rare" genes that are retained, low-variance "noise" genes that are dropped, and zero-variance genes that are also removed [1]. The filter operates on the original 10x Genomics H5 files, builds a new 10x compatible matrix that preserves barcodes and metadata, and emits both a filtered X and a small metadata file with the counts of rare, noise and housekeeping genes. On the dataset where we applied it, this PIM-style stage removed roughly half of the nonzeros in X while keeping all HVGs by construction, so the downstream compute operates on a smaller but biologically faithful problem.

Second, we design and implement a tiled hybrid SpMM pipeline that can route work between CPU and GPU at tile granularity. The matrix X is partitioned into 64 by 64 tiles in the gene by cell space, and for each tile we compute a simple density metric. Tiles whose density exceeds a fixed threshold of 5 percent are treated as dense and packed into temporary dense buffers that are sent to the GPU and multiplied with W using cuBLAS SGEMM. Tiles below this threshold remain in CSR form on the CPU and are executed by an OpenMP parallel CSR SpMM kernel. To make this routing more effective, we apply row and column permutations that cluster nonzeros and increase the fraction of tiles that are dense enough to benefit from the GPU. The result is a single C++ and CUDA codebase that can run in two modes: a CPU-only baseline and a tiled hybrid mode on the PIM-filtered X.

Third, we provide an end-to-end experimental characterization on several real 10x scRNA datasets, and we treat that characterization itself as a contribution. For each dataset, we measure how PIM filtering changes nnz, how tiling changes the mix of dense and sparse tiles, and how the two configurations compare in wall clock time. We also compute operational intensity and place each configuration on a roofline-style plot, which lets us connect changes in FLOPs and bytes to changes in runtime. This analysis surfaces both success cases and failure modes. On some datasets, filtering plus hybrid execution delivers speedup over the CPU baseline, because nnz reduction and better tile density outweigh tiling and launch overheads. On others, especially where almost all tiles are dense and tile counts are high, GPU kernel launch overhead dominates and the hybrid path can be slower than a single well tuned CPU CSR SpMM. These results lead to several practical insights: extreme sparsity and heterogeneity make

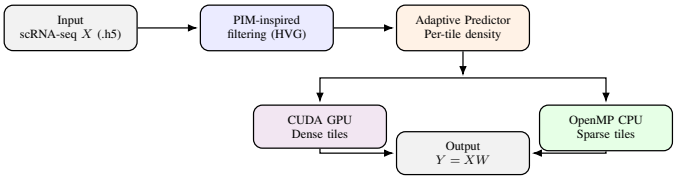

Fig. 1. AHAS workflow: filtering + tile-level routing to CPU/GPU for SpMM.

tile-level routing necessary, data reduction amplifies the benefit of hybrid execution but must pay for its own overhead, and fine-grained GPU offload is limited by launch costs and would benefit from tile batching or adaptive tile sizes.

V. Experimental Setup

All experiments were run on a Google Colab instance equipped with an NVIDIA L4 GPU and a multi-core Intel Xeon CPU (the exact CPU model may vary across sessions). On the CPU side, the sparse SpMM kernel is parallelized using OpenMP with 16 threads, as reported by the runtime. The software environment consists of Ubuntu 22.04 (Colab runtime), with `g++` and `nvcc` used as the C++ and CUDA compilers, targeting the C++17 standard and enabling `-O3` optimization together with `-fopenmp` for multithreaded execution. CUDA and cuBLAS are provided by the Colab installation under `/usr/local/cuda`, and all datasets are read from and written to disk using h5 1.10.x via the `libh5` and `libh5_cpp` libraries. Hybrid execution is enabled with a `-DUSE_CUDA` build flag: when this flag is set, dense tiles are offloaded to the GPU and computed using cuBLAS SGEMM, while sparse tiles are executed on the CPU using a CSR-based SpMM kernel with OpenMP.

All inputs are stored as 10x Genomics h5 files. Each dataset contains a sparse count matrix X in the standard 10x CSC layout on disk and a dense weight matrix W stored as an h5 dataset. When a dataset is loaded, X is converted to CSR format in memory for computation, while W remains dense. The output matrix Y is written back to h5 for later analysis. We use several real single-cell RNA-seq datasets that vary in both size and sparsity: the smallest matrices have on the order of 3×10^4 genes and 3×10^3–10^4 cells, with roughly 1.5×10^7 nonzeros, while the largest matrix has over 1.3×10^5 genes and thousands of cells, with roughly 2.5×10^7 nonzeros. Across these datasets, the fraction of nonzeros in X is between about 6% and 15%. In all cases, rows of X correspond to genes, columns correspond to cells, and W maps genes into a 32-dimensional output space.

Tiling and hybrid execution are controlled via compile-time parameters in `config/hw_config.h`. In all experiments, the matrix is partitioned into 64×64 tiles (`TILE_ROWS = 64`, `TILE_COLS = 64`) in the gene–cell space. For each tile, we compute a tile density, and tiles whose density is at least 5% (`DENSE_TILE_THRESHOLD = 0.05`) are classified as dense and routed to the GPU, while tiles below this threshold are treated as sparse and kept on the CPU. We evaluate two execution configurations. Throughout the paper, "Baseline" refers to this CPU-only CSR–dense SpMM configuration with no tiling, no PIM filtering, and no GPU offload. The *Baseline-CPU* configuration converts X to CSR and runs a single CSR–dense SpMM on the CPU, without any tiling or GPU offload. The *PIM+Hybrid* configuration uses 64×64 tiling and density-based routing, sending dense tiles to cuBLAS on the GPU and sparse tiles to the OpenMP CSR kernel on the CPU, but operates on a PIM-filtered version of X where low-variance genes have been removed offline using a highly variable gene (HVG) criterion.

The decision rule is simple: for each 64×64 tile we compute density as $\mathrm{nnz}/(64\cdot64)$. If density $\geq\tau$ with $\tau=0.05$, the tile is marked dense and packed into a temporary dense buffer that is multiplied with W on the GPU using cuBLAS SGEMM; otherwise the tile is marked sparse and kept in CSR form on the CPU, where it is processed by the OpenMP CSR SpMM kernel. The 5% threshold was chosen empirically as the smallest value that avoids sending extremely sparse tiles to the GPU on these datasets.

For each dataset and configuration, we measure the end-to-end SpMM wall-clock time from loading X and W to writing Y, and report speedups relative to the Baseline-CPU configuration. For PIM-filtered runs, we additionally record data-reduction statistics such as the change in $\mathrm{nnz}(X)$ and the size of Y. These measurements are used to compute operational intensity (FLOPs per byte moved), which feeds into a roofline-style analysis of how tiling and PIM filtering change the balance between computation and data movement. For all unfiltered runs, correctness is checked by comparing the computed Y against a provided reference Y_{check} with a small absolute and relative error tolerance.

VI. Results

We evaluated the prototype on four real 10x scRNA-seq matrices that span 3.1×10^8 to 3.7×10^8 total entries with only 5.8–14.6% nonzeros (Table I). All reported speedups and roofline points labeled "Baseline" use this CPU-only CSR SpMM as the reference. The two implementations (CPU baseline and hybrid with PIM filtering) produced numerically consistent outputs; elementwise comparisons against a reference SpMM passed with a relative tolerance of 10^{-3} and absolute tolerance of 10^{-5}, confirming that tiling, permutation, and mixed CPU/GPU execution do not introduce detectable numerical error.

Key insight: The workloads are extremely sparse but numerically stable, and every optimization stage preserves correctness within standard floating-point tolerance.

Figure 2 shows how the offline PIM-style filter reduces the number of nonzeros in X for each dataset: nonzero counts fall by roughly half on three datasets and by about 86% on the largest matrix, so the effective sparse workload shrinks substantially before it reaches the accelerator. Because Y has the same number of rows as X, the Y-row plot in Figure 3 shows a matching reduction on the output side, with around 90% of rows removed on most datasets and over 97% on the highly filtered case.

TABLE I
DATASET SPARSITY SUMMARY.

Dataset	Dimensions (genes×cells)	NNZ	Total Elements	Density	Sparsity
d2	33,696×9,263	26,407,826	312,123,648	8.46%	91.54%
d3	38,606×6,704	14,971,913	258,814,624	5.78%	94.22%
d4	134,920×2,711	24,511,186	365,978,120	6.70%	93.30%
d5	36,706×2,979	16,028,697	109,428,174	14.64%	85.36%

Table I — Dataset sparsity summary

TABLE II
PIM REDUCTION SUMMARY.

Dataset	Dimensions (genes×cells)	Data Points Reduced	Data Reduction %	File Size Change	NNZ Reduced	NNZ Reduction %
d2	33,696×9,263	26,003,276	49.09	-189.94	12,938,039	49.01
d3	38,606×6,704	12,768,074	42.40	-214.81	6,313,125	42.17
d4	134,920×2,711	42,945,845	86.41	28.44%	21,143,416	86.26
d5	36,706×2,979	14,955,715	46.17	-219.08	7,320,338	45.67

Table II — PIM reduction summary

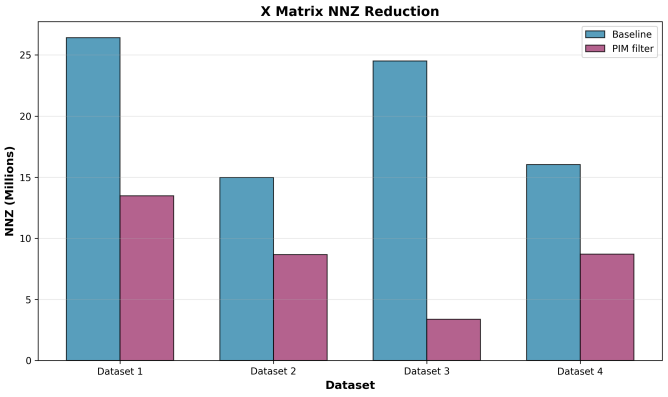

Fig. 2. Nonzero reduction in X after offline PIM-style (HVG) filtering.

Key insight: HVG-based PIM filtering consistently cuts both input and output problem size, turning some scRNA matrices into 2–6× smaller SpMM problems without dropping the high-variance genes.

The tiler and predictor then map this reduced work to CPU and GPU tiles with very different mixes across datasets. Two datasets are almost entirely CPU-heavy, with more than 98% of tiles classified as sparse and kept on the CPU; one dataset has a mixed profile where about 14% of tiles are dense; and the densest dataset has every tile classified as dense and sent to the GPU. In practice, the mixed and fully GPU cases pay a high price in CPU/GPU coordination and thousands of small kernel launches, while the CPU-heavy cases behave more predictably despite doing more work on the host.

Key insight: Hybrid execution is most effective when the predictor sends only a modest dense tail to the GPU and keeps the rest on the CPU, and it is least effective when many small dense tiles must be launched individually on the GPU.

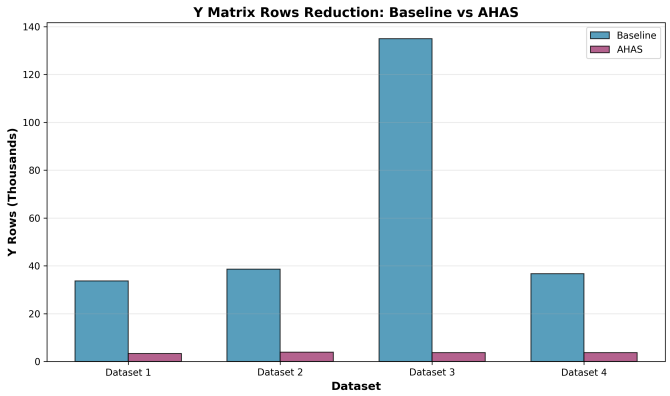

Fig. 3. Y row reduction induced by filtering (output rows correspond to retained genes).

End-to-end timing reflects this balance between filtering benefit and orchestration cost. Even though the PIM filter removes 42–86% of the nonzeros, the current prototype spends substantial time in Python-side preprocessing, tile packing, and GPU kernel launches, so filtered hybrid runs are between 2.7× and 25× slower than a single well-tuned CPU CSR SpMM on these problem sizes. The best case is the dataset with 86% nnz reduction and mostly CPU tiles, where reduced memory traffic partially offsets overhead, while the worst cases are the mixed and fully GPU workloads that are dominated by launch and marshalling costs.

Key insight: In its current form, the framework proves that PIM-style filtering reduces work, but also shows that tiling and launch overhead must be reduced or batched before those reductions translate into wall-clock speedup.

Figure 4 reports operational intensity (FLOPs per byte) before and after PIM filtering. Baseline CPU runs cluster around 6.8–7.6 FLOPs/Byte, while the AHAS configurations drop to roughly 2.4–4.1 FLOPs/Byte because both FLOPs and bytes fall, but bytes do not decrease as quickly as FLOPs. Figure 5 places these points on an aggregated roofline plot: baseline points lie in the compute-bound region but well below the ideal CPU line, and the AHAS points move down and to the right, reflecting lower operational intensity and additional overhead that is not captured by FLOPs or bytes alone.

Across all datasets, this drop in operational intensity correlates with the slowdown in Table 3: because both the baseline and AHAS points stay in the compute-bound region, the AHAS configurations sit farther from the roofline and achieve lower effective GFLOP/s at these problem sizes. In the current prototype, fixed overheads from filtering, packing, and per-tile kernel launches dominate, so the reduction in FLOPs and bytes does not yet translate into faster end-to-end runtime; however, these overheads are essentially constant per tile, so for larger matrices or batched workloads the same reductions in work and data movement would be amortized more effectively and are expected to compensate, or even outweigh, the overhead.

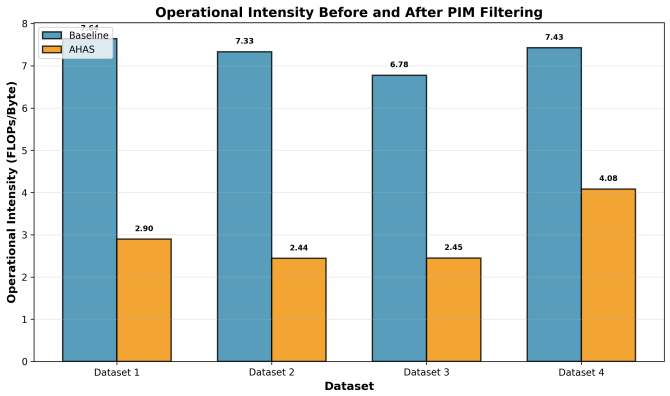

Fig. 4. Operational intensity (FLOPs/Byte) before and after PIM filtering.

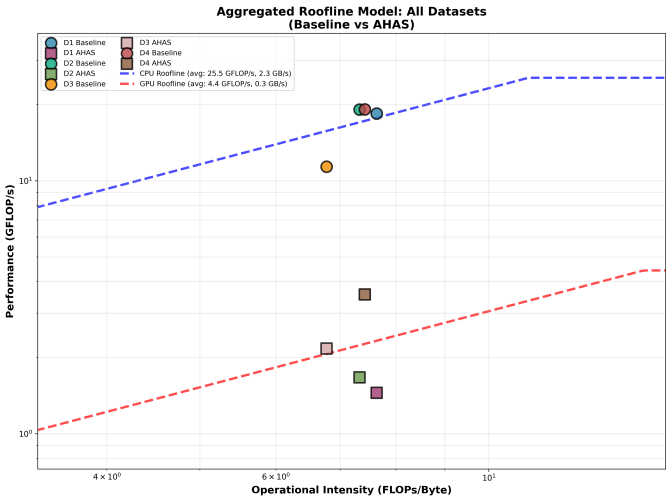

Fig. 5. Aggregated roofline view of baseline and AHAS configurations.

Key insight: The roofline view confirms that the system is not limited by peak compute or bandwidth but by software overhead, pointing directly to optimizations like tile batching.

In principle, moving to lower operational intensity shifts a kernel toward the memory-bound side of the roofline, where performance is limited by bandwidth rather than compute. In our prototype, however, the drop in operational intensity comes together with extra software overhead (Python-side filtering, tile packing, and many small kernel launches) that is invisible to FLOP/byte counts, so the AHAS points move down more than they move right on the roofline. As a result, reduced operational intensity in this work correlates with worse end-to-end runtime, which highlights that removing software overhead is more important than changing the compute–bandwidth balance.

VII. Conclusions

The evaluation shows that adaptive hybrid acceleration for scRNA SpMM is viable, but its benefit is tightly coupled to the balance between data reduction and orchestration overhead. PIM-style filtering based on highly variable genes consistently removes 40–80% of nonzeros while preserving rare, informative genes, so from a biological and algorithmic perspective the preprocessing step is sound. Tile-level density prediction and CPU/GPU routing also work as intended: CPU-heavy configurations incur little coordination cost, while mixed and fully GPU workloads expose the overhead of managing many small dense tiles.

TABLE III
Comprehensive performance metrics (tile mix, reductions, and timing).

Category	Metric	d2	d3	d4	d5
CPU workload	Sparse Tiles	6,636 (86.4%)	6,300 (98.4%)	2,451 (98.3%)	0 (0%)
CPU workload	CPU Workload %	86.4%	98.4%	98.3%	0%
GPU workload	Dense Tiles	1,049 (13.6%)	105 (1.6%)	43 (1.7%)	2,726 (100%)
GPU workload	GPU Workload %	13.6%	1.6%	1.7%	100%
GPU workload	GPU Overhead	High	Low	Low	Very High
Reduction / Movement	NNZ Reduction	49.01%	42.17%	86.26%	45.67%
Reduction / Movement	Row Reduction	90.0%	90.0%	97.3%	90.0%
Reduction / Movement	Memory Traffic Change	-34.6% (increase)	-73.8% (increase)	62.0% (decrease)	1.2% (minimal)
Timing	Baseline Time (ms)	91.6	50.1	137.9	53.6
Timing	PIM Filtering Time (ms)	2069.8	1235.9	370.9	659.8
Timing	Speedup vs Baseline	0.04× (22.6× slower)	0.04× (24.7× slower)	0.37× (2.7× slower)	0.08× (12.3× slower)

Table III — Comprehensive performance metrics

At the same time, the current prototype makes clear that software overhead, not raw compute or bandwidth, is the main limiter. Per-tile kernel launches, tile packing, and Python-side filtering costs are large enough that even the best case, with roughly 86% nnz reduction and substantial memory-traffic savings, does not yet outperform a well-tuned CPU CSR baseline. The roofline analysis reinforces this picture by placing all configurations well below the hardware ceilings, even though they remain in the compute-bound regime.

These findings point directly to future work: batching multiple tiles into each GPU call, adapting tile sizes to local density, and tuning the dense-tile threshold per dataset so that the predictor avoids pathological “all GPU” cases. More broadly, the study supports the idea that extremely sparse, heterogeneous scientific workloads require data-aware, adaptive execution strategies rather than a single global choice of CPU or GPU. The combination of PIM-inspired filtering, sparsity-aware tiling, and hybrid execution explored here provides a concrete starting point for such designs in scRNA analysis and in other domains that share similar sparsity patterns.

REFERENCES

[1] Q. Su, Y. Long, F. Duan, and Q. Lian, "A dual-metric framework for quantifying the biological fidelity of scrna-seq pipelines," *bioRxiv*, Nov. 2025, preprint.

[2] S. Chandrasekaran, H. Xu, O. Villa, and N. Sakharnykh, "Predicting sparsity and load balance for high performance spmm on gpus," in *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, Denver, CO, USA, Nov. 2023, pp. 1–13.

[3] C. Silvano *et al.*, "A survey on deep learning hardware accelerators for heterogeneous hpc platforms," *ACM Computing Surveys*, vol. 57, no. 11, p. 286, Jun. 2025.

[4] F. Zhang, S. Angizi, N. A. Fahmi, W. Zhang, and D. Fan, "Pim-quantifier: A processing-in-memory platform for mrna quantification," in *Proc. 58th ACM/IEEE Design Automation Conf. (DAC)*, San Francisco, CA, USA, Dec. 2021, pp. 43–48.

[5] H. Mao, R. Yao, D. Kim, A. Pedram, and H. Kim, "Genpip: In-memory acceleration of genome analysis via tight integration of basecalling and read mapping," in *Proc. 56th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Toronto, ON, Canada, Oct. 2023, pp. 1–15.

[6] F. Byloos *et al.*, "In-memory acceleration of raw signal genome analysis for real-time pathogen identification," in *Proc. 39th Int. Conf. Supercomputing (ICS)*, Turin, Italy, Jun. 2025, pp. 1–12.

[7] R. Hadidi, A. Danis, A. Shafiee, and H. Kim, "A holistic framework for using processing-in-memory in modern computer systems," arXiv preprint, Jul. 2023.

[8] S. Moghadamfar, M. Wang, J. G. Steffan, and A. Moshovos, "Sparsemap: A sparse tensor accelerator," arXiv preprint, Aug. 2025.

APPENDIX A
CHALLENGES AND LESSONS LEARNED

Working directly with real 10x h5 files turned out to be more involved than expected. I first had to understand how the matrix, feature, and barcode groups were organized, then repeatedly convert the h5 contents into explicit matrices offline to sanity-check that the simulator was consuming the right data. Numerical validation also surfaced floating-point subtleties: IEEE-754 non-associativity meant that parallel GPU results did not bit-match the sequential CPU baseline, so I had to design tolerance-based checks rather than exact equality. The permutation and tiling logic was another source of friction; getting row and column reordering correct required starting with a 5×5 toy example and only then scaling to the full scRNA matrices. A practical lesson from the project was knowing when to stop digging into a "clever" but brittle approach: in a few cases it was faster and safer to step back, redesign a simpler method, and re-implement than to keep chasing tricky corner cases in the original design.

A recurring challenge was keeping the hybrid pipeline efficient while preserving transparency for debugging and profiling. Dense-tile packing and GPU execution introduced per-tile launch and marshalling costs, and the HVG filtering step was implemented offline rather than fused into the runtime path. These choices made it much easier to validate correctness and isolate performance effects, but they also exposed how quickly coordination overhead can dominate when many small tiles are routed to the GPU.

APPENDIX B
SIMULATION OPTIMIZATION

To keep the simulator trustworthy while optimizing it, I added incremental checkpoints and reference paths rather than trying to tune everything at once. The first checkpoint was a pure CPU baseline that performed CSR SpMM and wrote out Y, which served as the reference for all later experiments. Next, I added a separate CUDA-only dense-tile SpMM path that used cuBLAS SGEMM without tiling or PIM, and verified that its outputs matched the CPU baseline under the IEEE-754 tolerance criteria. Only after those two ends of the spectrum were validated did I integrate the full tiled hybrid pipeline, with per-tile density prediction and CPU/GPU routing, and again compared against the same reference. This staged approach allowed me to attribute discrepancies and performance changes to specific components (I/O, tiling, permutation, routing, or CUDA) and prevented "mystery bugs" from being buried in the end-to-end system.

APPENDIX C
CODE DOCUMENTATION

The full implementation of AHAS (Adaptive Hybrid Accelerator for scRNA SpMM) is organized as a set of modular components that mirror the pipeline stages evaluated in this report: dataset ingestion, optional PIM-style filtering, tiling and prediction, hybrid CPU/GPU SpMM execution, and post-processing analysis. The repository is structured so that each directory corresponds to a specific responsibility in the system, making it easier to validate correctness and attribute performance costs to individual stages.

Datasets. The experimental inputs are public 10x Genomics datasets spanning different sizes and sparsity profiles, including mouse splenocytes, melanoma, PBMC (granulocytes removed), and A549 lung carcinoma with CRISPR perturbations. Each dataset is represented as an h5 file in the standard 10x layout, and the pipeline produces an output matrix Y along with logs that record timing and data-movement statistics.

GitHub: https://github.com/Shivaritha22/Adaptive-Hybrid-Accelerator-for-Sparse-Genomics-PIM-CPU-GPU-