MDPI

Article

ScaleSC - a superfast and scalable single cell RNA-seq data analysis pipeline powered by GPU

Wenxing Hu [1,†], Haotian Zhang [1,†], Jake Gagnon [1], Zhengyu Ouyang [2], Yuka Moroishi [1], Yu Sun [1], Shaolong Cao [1], and Baohong Zhang [1,*]

[1] Research Department, Biogen, Inc. 225 Binney St, Cambridge, MA 02142, USA
[2] Data Science, BioInfoRx Inc, Madison, WI 53719, USA
* Correspondence: baohong.zhang@biogen.com
† These authors contributed equally to this work.

Abstract: The rise of large-scale single-cell RNA-seq data has introduced challenges in data processing due to its slow speed. Leveraging advancements in GPU computing ecosystems, such as CUPY, and building on the NVIDIA genomics team's rapid-singlecell (RSC) package, we developed scaleSC, a GPU-accelerated solution for single-cell data processing. scaleSC delivers over a 20x speedup through GPU computing and significantly improves scalability, handling datasets of 10–40 million cells by overcoming memory bottlenecks—far surpassing RSC's capacity of processing only 1 million cells. Additionally, we resolved discrepancies between GPU and CPU algorithm implementations to ensure consistency. In addition to core optimizations, we developed new advanced tools for marker gene identification, reference mapping, and more, with GPU-optimized implementations seamlessly integrated. Designed for ease of use, the scaleSC package is compatible with Scanpy workflows, requiring minimal adaptation from users. The scaleSC package (https://github.com/interactivereport/scaleSC) promises significant benefits for the single-cell RNA-seq computational community.

Keywords: single cell; scRNA-seq; scanpy; Seurat

Citation: Lastname, F.; Lastname, F.; Lastname, F. Title. *Journal Not Specified* **2024**, *1*, 0. https://doi.org/

Received:
Revised:
Accepted:
Published:

Copyright: © 2025 by the authors. Submitted to *Journal Not Specified* for possible open access publication under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Recent advances of single cell RNA sequencing (scRNA-seq) technology enable researchers to study the gene expression profiles of individual cells. Unlike bulk RNA sequencing, which averages gene expression across a population of cells, scRNA-seq captures the heterogeneity within a cell population, offering unprecedented insights into cell types, states, and dynamic biological processes.

Spatial transcriptomics (ST) is another innovative technology that combines gene expression profiling with spatial information, allowing researchers to understand how cellular interactions and tissue organization influence gene expression. Emerging techniques integrate single-cell resolution with spatial precision, such as 10x Genomics Visium HD and Xenium, enabling researchers to resolve cellular-level insights within the tissue landscape. These approaches provide a multidimensional view of tissues, bridging molecular and structural biology.

Both scRNA-seq and ST operate at single-cell resolution, producing exceptionally large scale datasets. ScRNA-seq datasets are highly dimensional, encompassing both numerous observations, such as millions of cells, and a vast array of features, such as 30,000 genes. Analyzing scRNA-seq data is a complex, multi-step process aimed at extracting meaningful biological insights from the transcriptomes of individual cells while removing unwanted noise and batch effects. The data analytical workflow includes quality control (QC), normalization, dimensionality reduction, batch effect correction, clustering, cell type annotation, and various downstream analyses. These steps help mitigate unwanted noise and extract biologically relevant signals from cleaned cells and cell types.

The generated scRNA-seq data, typically delivered as a cell-by-gene matrix, is massive due to the high dimensionality of 20-50k genes and millions of cells. This makes the analytical steps computationally intensive in term of both space (memory) and time cost. As a result, data analysis of large scale scRNA-seq dataset requires high-performance servers with huge memory and is slow. To address computational challenge, specialized tools have been developed to manage the scalability of the data and reduce computational costs. These tools focus on reducing memory usage, parallelizing workflows, and optimizing algorithms for efficient, high-throughput analysis. For example, HDF5 https://www.hdfgroup.org/solutions/hdf5/, a dedicated package for fast and flexible I/O for large and complex data [1], is used to support hierarchical meta data structures and out-of-memory operations. Anndata [2] is a dedicated tool designed to effectively manage large cell×gene matrices. Both 'in memory' and 'on disk' data loading are supported in Anndata. Scipy [3] supports sparse matrices to further reduce memory storage and speed up computing operations, by leveraging the high sparsity of cell×gene matrices. Highly variable genes (HVG) [4,5] are used for feature dimension reduction, reducing from 20-50k features to 1-5k, by excluding genes with small variations. Principal component analysis (PCA) is used subsequently to further reduce feature dimension from 1-5k to 20-200.

Scanpy [6], https://scanpy.readthedocs.io/en/stable/, built on Anndata, is an one-stop package for single cell data analysis, including preprocessing, cell clustering, data visualization, etc. Scanpy utilized the efficient methods and data structures aforementioned, and incorporated other essential scRNA-seq data processing steps, e.g., normalization, quality control, batch effect removal, and clustering. With these, scanpy are becoming one of the most popular packages for scRNA-seq data analysis.

Despite of the efficient methods, algorithms, and programming, data analysis using Scanpy is still slow due to the extremely high dimensionality of cells. Since scRNA-seq data is stored as matrices, an natural idea is to use GPU to handle the computing in parallel because one GPU card contains thousands of computational units, i.e., CUDA cores. A team from NVIDIA developed a GPU-based scRNA-seq data analytical package, namely rapids-single cell (RSC) [7] https://github.com/scverse/rapids_singlecell, which is 10+ times faster than Scanpy. However, the capacity of RSC is restricted to approximately 1 million cells, which limits its utility. This limitation is due to 1) memory limit of GPU cards, 2) the GPU-based sparse matrix class, i.e., cupyx in CUPY [8], only supports int32 indexing, 3) some of the methods/algorithms are memory-consuming. The weak scalability of RSC limits its usage. Smaller datasets don't need RSC because conventional computational tools can finish within hours. On the other hand, larger datasets need RSC, but it cannot run because they exceed the limit of 1 million cells.

In this work, we developed a new GPU-based package, scaleSC, which solved the memory limitations of RSC. ScaleSC is 20-100 times faster than Scanpy and can accommodate data of up to 20-40 million cells. More detailed descriptions of scaleSC can be found in Section 2.

2. Materials and Methods

Building on the foundation of RSC, which already implemented most of the essential scRNA-seq data processing steps in GPU version, we developed scaleSC with several improvements, including 1) fixing the result discrepancy between Scanpy and RSC, 2) solving the scalability bottlenecks, and 3) optimizing algorithms.

2.1. Discrepancy removal

The output results are different between Scanpy and RSC. Scanpy uses CPU-based scientific computing libraries like *Numpy*, *Scipy*, and *Sklearn*, while RSC is mainly based on the GPU-accelerated *Rapids* AI ecosystem. Scanpy and RSC implement the same algorithm slightly differently in order to optimize the different computational architectures between CPU and GPU. We refer to this difference as "system variance". In addition to system variance, floating errors also contribute to differences in results, and we refer this as

"numerical variance". Although neither "system variance" nor "numerical variance" is incorrect, the difference in results leads to inconsistencies in data processing, posing challenges for reproducibility. Numerical variance is unavoidable but less significant compared to system variance and can be mitigated by increasing precision settings. This section focuses primarily on how scaleSC eliminates system variance.

2.1.1. PCA discrepancy

Principle component analysis (PCA) is a critical step in single-cell RNA-seq pipeline, which efficiently projects high dimensional single-cell count data into low dimensional space for proceeding downstream analysis. PCA typically consists of two steps:

- **Standardization**: The count value of each gene in each cell X_{ij} is normalized to a mean of 0 and a deviation of 1 as following,

$$Z_{ij} = \frac{X_{ij} - \mu_i}{\sigma_i + \epsilon}$$

  where Z_ij is the normalized count, μ_i is the mean of gene i over all cells, σ_i is the deviation of gene i over all cells, ϵ is a constant to avoid divided by zero.
- **Eigen-Decomposition**: An Eigen-Decomposition has performed on the covariance matrix. The eigenvectors represent the directions (principal components), and the eigenvalues represent the magnitude of the variance along those directions.

$$Z^T Z v_i = \lambda_i v_i$$

  where v_i is the i-th principal component and λ_i is the corresponding eigenvalue of the i-th eigenvector.

The differences are raised by the sign of eigenvectors that is not uniquely defined, flipping the signs does not affect their mathematical validity, but this will result in a different visualization and make the following analysis not reproducible, for example, batch removal algorithm *Harmony* takes principle components as inputs. As what we found, *Scanpy* internally call PCA function from *Sklearn*, which aligns the direction by flipping the sign of v_i such that the largest absolute value in v_i is always positive, the correction can be formulated as below.

$$\tilde{v_i} = sign(v_{ik})v_i$$

where k is the index of the largest absolute value. Instead of calling PCA from *Sklearn*, *rsc* relies on *Rapids* suite that doesn't correct it automatically, which makes the discrepancy appear. To further address this issue, one more step for correction of principle components will be introduced in *ScaleSC*.

2.1.2. Harmony discrepancy

Harmony is a powerful method designed to integrate single-cell datasets from multiple batches or experiments while preserving the biological variation and removing batch effects. It operates on low-dimensional embeddings, typically generated by PCA, then initializes clusters by KMeans based on embeddings and repeatedly adjusts them until convergence. We found the discrepancy comes from KMeans clustering and randomness caused by Random Number Generator(RNG) even the random seed is the same. More specifically, CPU typically generates random numbers sequentially using algorithms like Linear Congruential Generators (LCG), and is deterministic when initialized with the same seed, while GPU generates random numbers in parallel without any guarantee of ordering. This property leads to the uncertainty of algorithms involving randomness, like KMeans. To ensure scaleSC obtains the same initial clusters, the starting points of KMeans need to be matched up, we modified the original Python implementation *harmonypy* in two ways, The first approach is replacing GPU implementation of KMeans with CPU implementation entirely, this minimizes the code changes and we call it 1-step modification. The second approach is

switching the calculation of starting centroids to CPU then feed them to GPU-accelerated KMeans, we found this way can maximally leverage GPU capacity compared to "1-step" modification, and this refers to 2-step modification.

2.1.3. Neighbor discrepancy

Finding the nearest neighbors given the *Harmony* outputs is critical for performing following Leiden clustering. We try to construct clusters based on the same neighborhood graph and cell connectivities. We observe that Scanpy and RSC define two different graphs given the same input. After carefully checking the implementation of both Scanpy and RSC, we found they implement the neighbor algorithm in two different ways. RSC applies exact k Nearest Neighbor (k-NN) algorithm in parallel using *cuML*, whereas Scanpy approximates k-NN using NNdescent[9], an algorithm for approximating nearest neighbor search in high-dimensional spaces. To further validate this, we force Scanpy to use exact k-NN and finally obtain the same results as RSC, thus scaleSC will use the same algorithm as RSC to call exact k-NN instead of approximation.

By integrating improvements listed above, scaleSC can generate the same results as Scanpy at most of steps, however, even if with these discrepancies resolved, some steps involving random algorithms may still not guarantee to produce the exactly same results, such as Leiden, UMAP, and t-SNE. These randomnesses are mostly introduced in visualization, and won't affect the quantitative analysis conclusions.

2.2. Scalability

Frequent interaction with the count matrix can't be avoided in the single-cell RNA-seq preprocessing pipeline, loading the whole matrix into memory during runtime is infeasible. Due to its high sparsity (typically $> 90\%$ non-zero values), equivalent sparse formats are preferred for storage and computing on both CPU and GPU. However, there are two bottlenecks prevent us from actually using GPU:

- **GRAM Bottleneck**: Most of high-end GPUs in the market have limited memory. Even like the most advanced NVIDIA A100 has only $80G$, which is insufficient for one-time loading for dataset even with sparse matrix supported.
- **Int32 Bottleneck**: *CuPy* as the backend of sparse matrix on GPU, one notable limitation is the integer type used for indexing, which is commonly 32-bit integers, indicates it can only store non-zero elements up to $2^{31} - 1$, or approximately 2.1 billion. This is not enough for extremely large single-cell dataset, says a dataset with 5×10^6 cells, 2×10^4 genes, and a sparsity of 0.9, the number of non-zero elements is much larger than that upper limit. And there is no solution to solve it directly so far.

To workaround with extremely large dataset, we adopt the chunking approach to split the dataset into chunks along cells, so that each data chunk won't reach the Int32 bottleneck. In addition, this also give us flexibility to process data chunks separately across devices, which indicates there is a potential way to process arbitrary large data.

2.3. scaleSC overview

Next, we build scaleSC on the top of Scanpy and RSC so that our method can seamlessly be scaled to tens of millions of cells with GPU acceleration. Like Scanpy and RSC, scaleSC implements a series of necessary methods in the classical single-cell RNA-seq pipeline, including steps as shown in Figure.1:

1. Cell QC: Filter out low-quality cells and genes according to the number of counts.
2. Normalization: Perform log1p normalization on counts by $\log(1 + X)$, X is the count value.
3. HVG: Choose highly variable genes using Seurat v3 [5].
4. PCA: Dimensional reduction using PCA.
5. Batch Correction: Invoke *Harmony* to remove batch effects.
6. Neighbor: Use k-NN for finding k nearest neighbors to build the neighborhood graph.

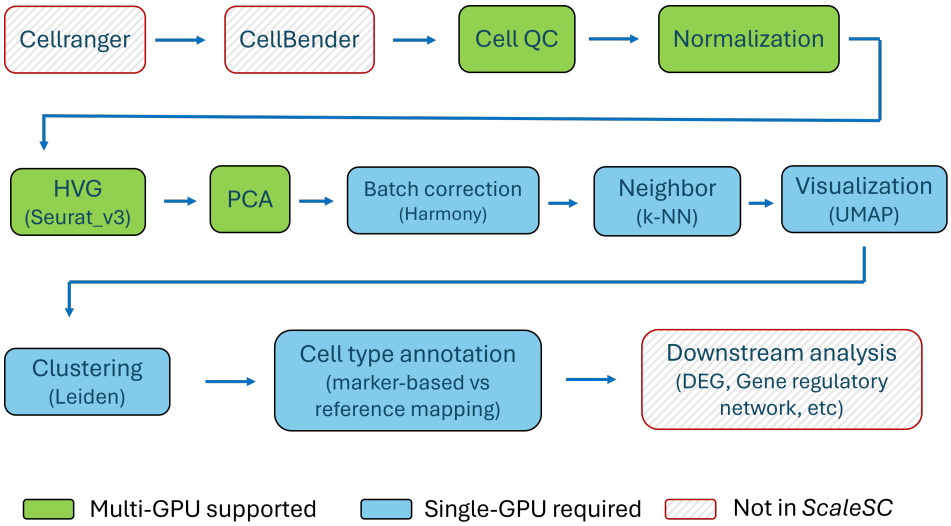

Figure 1. The Workflow of standard scRNA-seq data processing.

7. Visualization: Call UMAP on PCA-reduced matrix for better visualization.
8. Clustering: Call Leiden clustering.
9. Cell Type Annotation: marker-based de novo cell type annotation; reference data based annotation transfer.

Since scaleSC conduct data loading and preprocessing in data chunks, we developed a chunked-data class which is dedicated for chunked data loading and computation. This reader supports three different modes for loading data under various scenarios, depending on users GPU and CPU size. The three data loading modes are described in the next subsections.

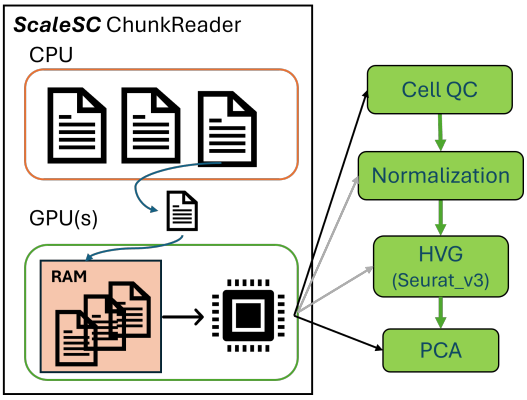

Figure 2. The flowchart Chunk-based data loading and preprocessing.

2.3.1. Time efficient but high memory consumption

In this setting, scaleSC will load all data chunks into GPU memory, In each step, each chunk is taken out to be processed iteratively, then result is merged automatically. Since all chunks are pre-stored in GRAM, there is no overhead for data transfer between host(CPU) and device(GPU) during computing, thus this is the fastest way.

However, this mode is limited by the GRAM, the size of dataset should not be larger than its capacity. To further extend our method to handle larger dataset, multi-GPU support is introduced for pre-loading more chunks across multiple cards, this is because the data transfer rate between GPUs through NVSwitch is also much faster than the rate from CPU to GPU. Noted, multiple cards are for chunk storage only, computation is still done in a single card.

2.3.2. Balanced on both time and memory

Compared to the previous mode, scaleSC won't utilize GPU for storing all chunks, scaleSC preloads all chunks on CPU instead. Then scaleSC will transfer a single chunk to GPU and transfer it back after calculation is done. This mode is limited by the memory size of CPU and also add some overheads for device communication, usually this mode works well in all real applications.

2.3.3. Memory efficient but time-consuming

Those two modes above are based on the assumption that users have enough CPU RAM. For uses with limited resources, scaleSC is allowed to read chunks from disk every time it requires, this definitely involves a lot of IO between disk and host, which makes it the least memory cost but much more time-consuming. Ideally, this mode can work with arbitrarily large dataset.

2.4. Details about steps in scaleSC

In this section, we will introduce the modifications made by scaleSC in some steps to accommodate chunked data then investigate into Harmony for handling much larger batch size.

2.4.1. Cell QC

Within each individual chunk, a subset of cells is retained, while all genes are preserved. This approach facilitates the filtering of cells based on the number of gene counts. In contrast, selecting genes based on the number of cells presents a challenge, as other chunks are not accessible when processing a specific chunk. To address this, an array is maintained to track the number of cells associated with each gene, which is updated incrementally as new chunks are processed. Ultimately, a finalized list of cells and genes is generated following a single iteration over all chunks. This procedure requires traversing all chunks once to ensure the completeness of the filtering process.

2.4.2. HVG

This section outlines the implementation of the HVG selection step, following the methodology described in the Seurat v3 paper [5]. The workflow can be summarized as follows, N represents the number of cells and M represents is the number of genes:

1. Calculation of Gene Means and Variances: Compute the means and variances for each gene using the raw count matrix. This step requires a single iteration over all data chunks to compute the means μ_i of the ith gene and sample variances σ_i of the ith gene(using $N-1$ as the divisor). To achieve this accurately, the CUDA kernel must be modified to ensure compatibility with the Scanpy's results. Noted, using population variance instead of sample variance would lead to discrepancies.

$$
\begin{aligned}
s_{i,k} =& s_{i,k-1} + \sum_{x \in X_{k,i}} x \\
s^2_{i,k} =& s^2_{i,k-1} + \sum_{x \in X_{k,i}} x^2 \\
\mu_i =& \frac{s_{i,K}}{N} \\
\sigma_i =& \frac{N}{N-1}\left(\frac{s^2_{i,K}}{N} - \mu_i^2\right)
\end{aligned}
$$

where K is number of chunks, $X_{k,i}$ is the ith gene column in the k-th chunk, x is the count value, $s_{i,k}$ is sum of count values in the first k chunks, $s^2_{i,k}$ is the squared sum of count values in the first k chunks.

2. Loess Regression in Logarithmic Space: Perform locally weighted regression (loess), using the logarithm of means as the predictor and the logarithm of variances as the response. There is no GPU implementation for now but is fast and efficient enough on CPU.
3. Normalization and Clipping: Normalize the raw counts using the fitted variances and means for each gene, and clip any values exceeding $\sqrt{N}$.

$$x' = max(\frac{x-\mu}{\sigma'}, \sqrt{N})$$

   where x' is the normalized value, μ is the mean of the ith gene before normalization, σ is the fitted variance in Step(2).
4. Repeat Step(1) to recalculate the variance of normalized data, then mark genes with K-largest variances as HVGs.

In HVG, two loops of all chunks are needed: the first loop obtains mean and variance of raw data; the second loop normalizes data with regressed variances.

2.4.3. PCA

Since we are dealing with chunked data, it is not easy to get eigenvectors through SVD, but it would be much easier to obtain them through zero-centered covariance matrix.

$$(X-m)^T(X-m) = X^TX - m^Ts - s^Tm + Nm^Tm$$

where $X \in R^{N\times M}$ represents the entire matrix, $m \in R^M$ is a vector of means, $s \in R^M$ is a vector of summation of count values. And we can calculate X^TX by simple add up all chunks.

$$X^TX = \sum_{k=1}^{K} X_k^T X_k$$

where X_k is the k-th chunk, and $X^T = [X_1^T, X_2^T, \ldots, X_K^T]$. Based on calculated zero-centered covariance matrix, it's easy to obtain the principle components by eigen-decomposition. Similar to the HVG, scaleSC goes over all chunks twice.

2.4.4. Batch Correction

Harmony [10] is a widely used algorithm for batch effects removal. However, during these iterative calculations, the algorithm generates and stores numerous intermediate variables, leading to substantial memory consumption. For datasets with approximately 13 million cells and over 1,000 samples, the original Harmony implementation encounters significant memory issues, often resulting in out-of-memory (OOM) errors. The memory requirement for such datasets exceeds 220 GB, highlighting a critical limitation in its scalability for large-scale single-cell analyses.

Table 1. Some large variables produced during calculation (13 million cells), d denotes the dimension of PC loadings, B denotes the number of samples, C denotes the number of clusters (up to 100).

Variable	Description	Shape	Size
Z_orig	PCA input	(d, N)	2.5G
Z_cos	L2 normalization of input, cosine distances	(d, N)	2.5G
Z_corr	Corrected embeddings	(d, N)	2.5G
Phi	The one-hot encoded batch assignment matrix.	(B, N)	64G
Phi_moe	The one-hot batch assignment with bias in MoE estimation	$(B+1, N)$	64G
Phi_Rk	The one-hot batch assignment for a given cluster	$(B+1, N)$	64G
R	the cluster assignment matrix	(C, N)	5G
dist_mat	Distance matrix	(C, N)	5G
scale_dist	Scaled distance matrix	(C, N)	5G

From the table above, the obvious bottleneck is B, we can observe that when B goes larger, it becomes impossible to hold even such a single large matrix under 32-bit float. However, Phi, Phi_moe and Phi_Rk are extremely sparse containing about N non-zero values, if convert them to sparse matrix, that would save a lot of memory. Based on this idea, a (B, N) dense matrix can be converted to a sparse matrix in only O(N) space ($0.2G$). All algebraic operations involved for converted matrices need to be adjusted accordingly as well.

The original script also used PD.GET_DUMMIES to create the matrix Phi, which takes a lot of memory on CPU as well. To further optimize the memory performance, we generate a sparse dummy matrix directly instead of a huge dense matrix. Besides those sparse matrices, the original script keeps some unused matrices in memory, we removed them and further optimized to save more memory.

After conversion and cleaning, it still fails on $13M$ data. The reason is *Harmony* produces an extremely large intermediate values when it estimates MoE model parameters as following:

$$W = (\Phi_{Rk}\Phi_{moe}^T + \lambda)^{-1} \cdot \Phi_{Rk} \cdot Z_{orig}^T$$

Multiplication of $(\Phi_{Rk}\Phi_{moe}^T + \lambda)^{-1}$ and Φ_{Rk} results in an intermediate matrix with shape as (B, N), which is approximately $64G$ memory space allocated by GPU for temporary storage. A simple trick can be observed to avoid such large temporary storage appeared during calculations: $\Phi_{Rk} \cdot Z_{orig}^T$ won't take too much space so that can be computed first, then multiply by $(\Phi_{Rk}\Phi_{moe}^T + \lambda)^{-1}$.

By combining all of above, scaleSC can smoothly work with up to $15M$ in a single A100 card. The performance of scaleSC is presented in Tables 2-4.

Table 2. The time cost comparison between Scanpy and scaleSC on the 70K human lung data [11]. Time cost.

processing step	Scanpy (seconds)	scaleSC (seconds)	speed X
Data loading	0.9	0.4	2.0
QC	7.5	0.2	31.3
Normalization	1.1	0.0	362.0
HVG	10.5	0.3	38.8
PCA	14.8	0.4	37.0
Harmony	131.4	2.1	61.8
Neighbors	153.5	0.3	570.6
UMAP	87.5	0.2	366.0
Leiden clustering	16.2	6.7	2.4
All	423.4	11.1	38.0

Table 3. The time cost comparison between Scanpy and scaleSC on the 1.3M mouse brain data [4].

processing step	Scanpy (seconds)	scaleSC (seconds)	speed X
Data loading	26.0	16.1	1.6
QC	271.6	4.9	55.6
Normalization	21.3	2.0	10.7
HVG	312.5	8.5	37.0
PCA	269.6	21.7	12.4
Harmony	8818.1	23.5	375.5
Neighbors	336.5	34.1	9.9
UMAP	2404.3	4.4	546.8
Leiden clustering	3793.2	10.9	346.6
All	16253.1	126.0	128.9

Table 4. The time cost comparison between Scanpy and scaleSC on the 13M data.

processing step	Scanpy (seconds)	scaleSC (seconds)	speed X
Data loading	-	405	-
QC	-	686	-
Normalization	-	56	-
HVG	-	419	-
PCA	-	622	-
Harmony	-	628	-
Neighbors	-	163	-
UMAP	-	27	-
Leiden clustering	-	49	-
All	-	3055	-

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Folk, M.; Heber, G.; Koziol, Q.; Pourmal, E.; Robinson, D. An overview of the HDF5 technology suite and its applications. In Proceedings of the Proceedings of the EDBT/ICDT 2011 workshop on array databases, 2011, pp. 36–47.
2. Virshup, I.; Rybakov, S.; Theis, F.J.; Angerer, P.; Wolf, F.A. anndata: Annotated data. *BioRxiv* **2021**, pp. 2021–12.
3. Virtanen, P.; Gommers, R.; Oliphant, T.E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* **2020**, *17*, 261–272.
4. Zheng, G.X.; Terry, J.M.; Belgrader, P.; Ryvkin, P.; Bent, Z.W.; Wilson, R.; Ziraldo, S.B.; Wheeler, T.D.; McDermott, G.P.; Zhu, J.; et al. Massively parallel digital transcriptional profiling of single cells. *Nature communications* **2017**, *8*, 14049.
5. Stuart, T.; Butler, A.; Hoffman, P.; Hafemeister, C.; Papalexi, E.; Mauck, W.M.; Hao, Y.; Stoeckius, M.; Smibert, P.; Satija, R. Comprehensive integration of single-cell data. *cell* **2019**, *177*, 1888–1902.
6. Wolf, F.A.; Angerer, P.; Theis, F.J. SCANPY: large-scale single-cell gene expression data analysis. *Genome biology* **2018**, *19*, 1–5.
7. Nolet, C.; Lal, A.; Ilango, R.; Dyer, T.; Movva, R.; Zedlewski, J.; Israeli, J. Accelerating single-cell genomic analysis with GPUs. *bioRxiv* **2022**, pp. 2022–05.
8. Nishino, R.; Loomis, S.H.C. Cupy: A numpy-compatible library for nvidia gpu calculations. *31st confernce on neural information processing systems* **2017**, *151*.
9. Dong, W.; Moses, C.; Li, K. Efficient k-nearest neighbor graph construction for generic similarity measures. In Proceedings of the Proceedings of the 20th international conference on World wide web, 2011, pp. 577–586.
10. Korsunsky, I.; Millard, N.; Fan, J.; Slowikowski, K.; Zhang, F.; Wei, K.; Baglaenko, Y.; Brenner, M.; Loh, P.r.; Raychaudhuri, S. Fast, sensitive and accurate integration of single-cell data with Harmony. *Nature methods* **2019**, *16*, 1289–1296.
11. Travaglini, K.J.; Nabhan, A.N.; Penland, L.; Sinha, R.; Gillich, A.; Sit, R.V.; Chang, S.; Conley, S.D.; Mori, Y.; Seita, J.; et al. A molecular cell atlas of the human lung from single-cell RNA sequencing. *Nature* **2020**, *587*, 619–625.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.