RESEARCH

Alevin: An integrated method for dscRNA-seq quantification

Avi Srivastava[1], Tom Sean Smith[2], Ian Sudbery[3] and Rob Patro[1*]

*Correspondence: rob.patro@cs.stonybrook.edu
[1]Department of Computer Science, Stony Brook University, New York, USA
Full list of author information is available at the end of the article

Abstract

We introduce alevin, an efficient pipeline for gene quantification from dscRNA-seq (droplet-based single-cell RNA-seq) data. Alevin is an *end-to-end* quantification pipeline that starts from sample-demultiplexed `FASTQ` files and generates gene-level counts for two popular droplet-based sequencing protocols (drop-seq [1], and 10x-chromium [2]). Importantly, alevin handles all processing internally, avoiding reliance on external pipeline programs, and the need to write large intermediate files to disk. Alevin adopts efficient algorithms for cellular-barcode whitelist generation, cellular-barcode correction, lightweight per-cell UMI deduplication and quantification. This integrated solution allows alevin to process data much faster (typically ~ 10 times faster) than other approaches, while also working within a reasonable memory budget. This enables full, end-to-end analysis for single-cell human experiment consisting of ~ 4500 cells with 335 Million reads with 13G of RAM and 8 threads (of an Intel Xeon E5-2699 v4 CPU) in 27 minutes.

Keywords: dscRNA-seq; Cellular Barcode; UMI; Quantification

1 Introduction

There has been a steady increase in the throughput of single-cell RNA-seq (scRNA-seq) experiments, with droplet-based protocols (dscRNA-seq) facilitating experiments assaying tens of thousands of cells in parallel. Three of the most widely-used dscRNA-seq protocols: drop-seq [1], inDrop [3] and 10x-chromium [2] use two separate barcodes which require appropriate processing to obtain accurate quantification estimates.

First, cellular barcodes (CBs) are used to enable pooling of cells for sequencing and their subsequent separation *in silico*. This relies upon a greater pool of possible CBs being available than the number of cells input into the system, ensuring each cell is likely to be labeled by a unique barcode. Thus, correct data processing requires the identification of the true cell barcodes corresponding to distinct cells, so that the reads can be segregated or grouped by them. This whitelisting task is made more complex by the amplification of background RNA, for droplets which either do not contain a cell (ambient CBs) or contain lysed or otherwise damaged cells [2, 4].

Since the RNA content of a single cell is low, and the mRNA capture rate is only around 5-10% [5] , many rounds of PCR are typically performed prior to sequencing [1]. Accurate quantification therefore requires avoiding counting multiple reads that arise from PCR duplicates of the same initial molecule. The identification of PCR duplicates is enabled via Unique Molecular Identifiers (UMIs), which barcode each unique molecule [6].

Unfortunately, both CBs and UMIs are subject to errors that occur during sequencing and PCR amplification. This makes the accurate deconvolution of this information *in silico* a non-trivial task. If not accounted for, these errors can propagate to the quantification estimates and downstream analyses [7]. Various methods have been proposed for whitelisting [2, 8, 9], correcting sequencing errors in CBs and UMIs [7, 10], deduplication of UMI tags inferred to be duplicates [7], and quantification of gene expression at the per-cell level [2, 7].

However, a considerable drawback of existing pipelines is the fact that the solutions for solving these problems tend to be designed and implemented independently, and are not optimized for performing dscRNA-seq quantification. While this affords flexibility and modularity, it can significantly increase processing time, especially since the most common form of inter-process communication is through the writing and reading of intermediate files on disk. Though pipelines using lightweight methods for mapping are available [11], they focus on alternative forms of downstream analysis, and may not be ideal for extracring gene-level counts from tagged-end data, since independent per-equivalence-class level deduplication of UMIs appears to considerably under-collapse PCR duplicates (see Table S2). Moreover, the time requirements for current lightweight methods are driven up considerably by the reliance on pre-processing steps to segregate the initial raw data into a per-cell format. This approach generates one or more intermediate input files per-cell, and can be inefficient when scaling up with the increasing number of cells being assayed in dscRNA-seq experiments.

Here, we describe an end-to-end quantification pipeline which takes as input sample-demultiplexed `FASTQ` files, separated based on the illumina sample-index, and performs the intermediate steps (optimized specifically for single-cell sequencing data) of barcode whitelist generation, CB correction, and UMI sequence correction and deduplication, resulting in gene-level read counts for each cell in the library. This per-cell count matrix is the most common input format for downstream tools, and can be directly used for tasks such as clustering (Seurat)[12], cell-type identification [13], and pseudo-time analysis (Monocle)[14]. We call this unified pipeline alevin. It is integrated into the salmon quantification tool [15] as of release 0.10.0, and is available at `https://github.com/COMBINE-lab/salmon` and via bioconda [16].

2 Methods

2.1 Initial whitelisting and barcode correction

The first step of many existing scRNA-seq processing pipelines is to extract cell barcode and UMI sequences, adding this information to the header of the sequenced read or saving it in temporary files. This approach, while versatile, can create many intermediate files on disk for further processing, which can be time and space-consuming.

Alevin begins with sample-demultiplexed `FASTQ` files, and implements a two-pass algorithm. In the first pass, alevin quickly iterates over the barcode file only, and tallies the frequency of all observed barcodes (regardless of putative errors). We denote the collection of all observed barcodes as $\mathcal{B}$, with b_i being the count of

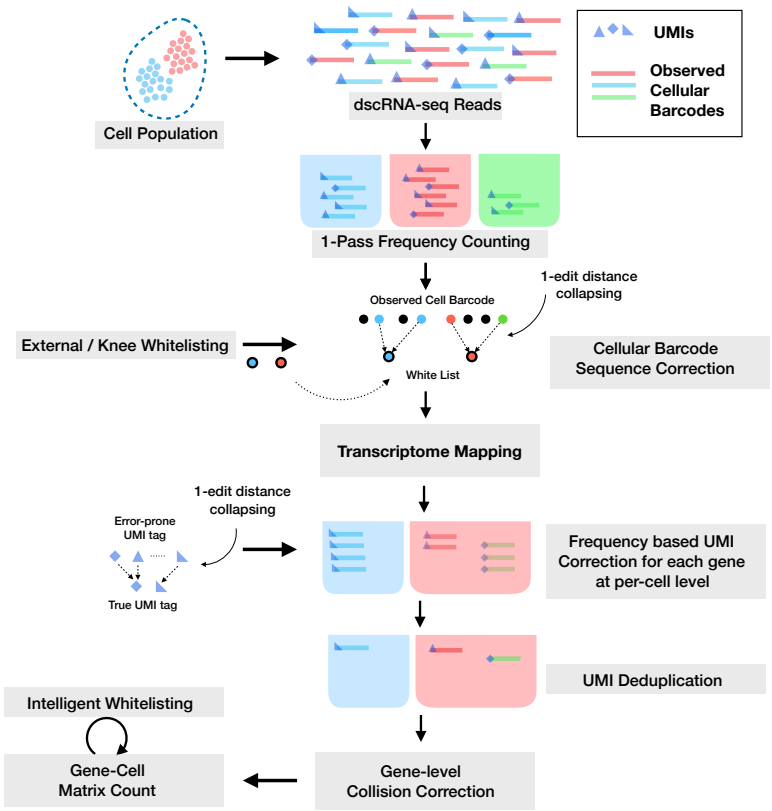

Figure 1: An overview of the pre-processing and quantification workflow for the alevin pipeline. Alevin starts with `FASTQ` input with CB and UMI information and first generates the frequency distribution of the CBs (shown by the red, green, and blue buckets) without any sequence correction. Whitelist CBs (shown as only the blue and red bucket on left) for the CB sequence correction can be externally provided, or alevin can generate a whitelist using a Gaussian-kernel based "knee" method. The next stages generate the transcript-level mappings for the reads, corrects and deduplicate reads at the gene-level (as explained in Section 2.2). After the deduplication of reads using UMIs, alevin can be instructed to generate gene-level counts for each cell (ignoring gene-ambiguous equivalence classes). Optionally, alevin also implements a naïve Bayes-based intelligent whitelisting procedure to classify low-quality and likely-correct CBs based on a set of features available after initial estimation of gene-expression and other features.

barcode i. Alevin's first task is to determine a set of putative whitelisted barcodes that are inferred to derive from valid cells and for which quantification estimates will ultimately be generated. When the data has been previously processed by another pipeline, a whitelist may already be available. When a whitelist is not available, alevin uses a two-step procedure for calculating the whitelist. An initial whitelist is produced using the procedure explained below, to select CBs for initial quantification. This whitelist is then refined again, once per-cell level quantifications are available (see section 2.3)

Traditional dscRNA-seq pipelines determine the putative whitelist by analyzing the cumulative distribution of barcode frequencies, and finding the knee in this

curve [1, 2]. Those barcodes occurring after the knee constitute the whitelist. We follow this convention and use a Gaussian kernel to estimate the probability density function for the barcode frequency and select the local minimum corresponding to the "knee" to determine the initial putative whitelist, $\mathcal{W}$. In the case of a user-provided whitelist, $\mathcal{W}$ is the fixed final whitelist.

Next, we wish to consider those barcodes in $\mathcal{E} = \mathcal{B} \setminus \mathcal{W}$. Specifically, we want to determine, for each non-whitelisted barcode, whether its corresponding reads should actually be assigned to some barcode in $\mathcal{W}$, or whether this barcode represents some other type of noise or error (e.g., ambient RNA, lysed cell, etc.) and its associated reads should be discarded. The approach of alevin is to determine, for each barcode $h_j \in \mathcal{E}$, the set of whitelisted barcodes to which h_j could be associated. We call these the putative labels of h_j — denoted as $\ell(h_j)$. Inspired by the criteria used by previous pipelines [1], we consider a whitelisted barcode w_i to be a putative label for some erroneous barcode h_j if $d(h_j, w_i) = 1$, where $d(\cdot, \cdot)$ is the Levenshtein distance. Rather than applying traditional algorithms for computing the relevant distances directly, we exploit the fact that barcodes are relatively short, and therefore have a reasonably small edit-distance ball of radius 1. We iterate over each $w_i \in \mathcal{W}$ and perform a query in $\mathcal{B}$ for all sequences within an edit distance of 1 from w_i. Let $Q(w_i, H)$ be the set of barcodes from $\mathcal{E}$ that are at an edit distance of 1 from w_i. For each $h_j \in Q(w_i, H)$, we append w_i as putative label for the erroneous barcode h_j.

Once all whitelisted barcodes have been processed, each element in $\mathcal{E}$ will have zero or more putative labels. If an erroneous barcode has more than one putative label, we prioritize substitutions over insertions and deletions. If this does not yield a single label, ties are broken randomly. If no candidate is discovered for an erroneous barcode, then this barcode is considered "noise", and its associated reads are simply discarded.

2.2 Mapping reads and UMI de-duplication

After labeling each barcode, either as noise or as belonging to some whitelisted barcode, alevin maps the sequenced reads to the target transcriptome [17]. Reads mapping to a transcript (or multi-mapping to a set of transcripts) are categorized hierarchically, first based on the label of their corresponding cellular barcode, and then based on their unique molecular identifier (UMI). At this point, it is then possible to deduplicate the reads based on their mapping and UMI.

The process of read deduplication involves the identification of duplicate reads based on their UMIs and alignment positions. Since most amplification occurs prior to fragmentation in the library preparations used in the 10x Chromium protocol[18], the alignment position of a given read is not straightforward to interpret with respect to deduplication, as the same unique initial molecule may yield reads with different alignment coordinates[1]. A further consideration is that UMIs can contain sequence errors. Achieving the correct deduplication thus requires proper consideration of the

[1]We note that whether the majority of amplification occurs pre or post-fragmentation can be protocol specific and can suggest different strategies for UMI deduplication. Here, however, we are primarily concerned with the 10X Chromium protocols, dominated by pre-fragmentation amplification.

available positional information and possible errors. We account for sequence errors by utilizing the "directional" method introduced in UMI-Tools [7] (as defined below) to deal with errors in the UMIs. Let $\mathcal{U}_i$ be the set of UMIs observed for gene i. A specific UMI $u_n \in \mathcal{U}_i$, observed f_n times in gene i, is considered to have arisen by PCR or sequence error if there exists $u_m \in \mathcal{U}_i$ such that $d(u_n, u_m) = 1$ and $f_m > 2f_n + 1$, where $d(\cdot, \cdot)$ is, the Levenshtein distance. Only UMIs that could not have arisen as an error under this model are retained. However, as discussed below, this approach may over-collapse UMIs if there exists evidence that similar UMIs (i.e. UMIs at an edit distance of 1 or less) must have arisen from different transcripts, and hence, distinct molecules.

As previously proposed in [11], the equivalence class [15, 19–24] that gives rise to a fragment encodes some positional information, by means of encoding the set of transcripts to which the fragment is mapped. This can be used to avoid over-collapsing UMI tags that are likely to result from different molecules by considering UMIs as distinct per equivalence class. However, in its simplest form, this deduplication method is prone to reporting a higher number of distinct UMIs than likely exist. This is because reads from different positions along a single transcript can give rise to different equivalence classes, so that membership in a different equivalence class is not, alone, sufficient evidence that a read must have derived from a distinct (pre-PCR) molecule (see Figure S1 and Section S1.2.2 for more details). This deters us from directly using such a UMI collapsing strategy for deriving gene-level counts, though it may be useful for other types of analysis.

Alevin implements two modes of operation for UMI

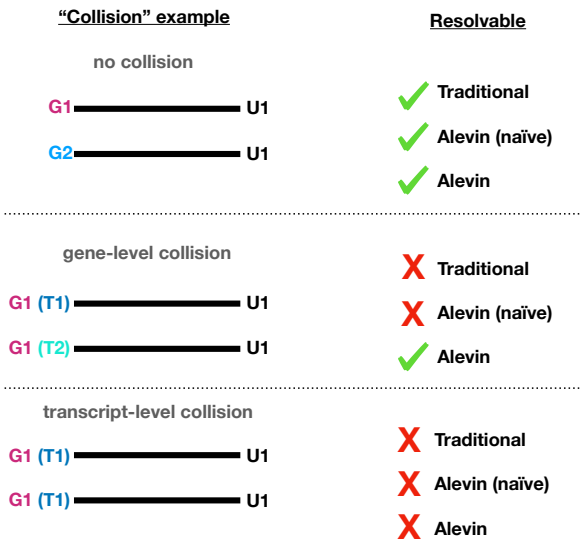

Figure 2: This figure illustrates examples of various classes of UMI collisions. The first row (top to bottom) demonstrates a case when we observe the same UMI (U1) being used to tag transcripts from two separate genes (G1 and G2). Here, all methods are able to correctly assess that these instances of U1 are not PCR duplicates. In the center row, we observe the same UMI deriving from two (sequence-distinct) transcripts of the same gene. Here, purely gene-level methods fail to resolve this collision, while alevin's default strategy can. Finally, in the bottom row, we observe a UMI collision within a single transcript. That is two different copies (molecules) of the same transcript have been tagged with the same UMI. This resolution cannot be resolved by any of the methods. These cases are shown top-to-bottom in order of their likelihood. Though possible, the situation presented in the third row is *highly*-unlikely, especially given current sequencing depths.

deduplication. The default mode makes use of transcript-level information derived from equivalence classes, but also attempts to avoid under-collapsing UMIs that are likely duplicates.

2.2.1 Baseline: Gene-level Deduplication

In this mode, alevin mimics most common, alignment-based, gene-level deduplication methods. It considers all UMIs from the same gene simultaneously, working under the assumption that in a given cell, observing the same UMI being attached to two or more isoforms of the same gene is a very unlikely event. This gene-level deduplication is achieved by aggregating the UMI counts from all single-gene equivalence classes for a given gene, and then performing deduplication using the "directional" method (discussed above). This mode is activated by providing the `--naive` option to alevin.

2.2.2 Default: Collision Correction

A potential drawback of gene-level deduplication is that it necessarily involves discarding transcript-level evidence. In this case, such evidence is encoded in the equivalence classes. Thus, gene-level deduplication provides a conservative approach and works on the assumption that it is highly-unlikely to observe molecules that are transcripts of the same gene but are nonetheless tagged with a similar UMI (within an edit distance ball of radius 1 of another UMI from the same gene). However, entirely discarding transcript-level information will mask true UMI collisions to some degree, even when there is direct evidence that the similar UMIs must have arisen from distinct transcripts. This is especially true when using an error-aware deduplication approach, and as sequencing depth increases. The *default* mode of alevin attempts to overcome this problem by first deduplicating UMIs for each gene as described above. Subsequently, it attempts to identify and correct for possible UMI-collisions for which there is direct evidence based on the transcript-level information.

While performing UMI deduplication on each gene, alevin first identifies the minimum number of transcripts which can parsimoniously explain all the equivalence-classes associated with it. Specifically, for each gene g having set of equivalence classes g_e where, $g_e = \{e_1, e_2, ...e_n\}$ and each $e_x = \{t_1, t_2, ...t_m\}$ is a set of transcripts (called labels of the equivalence classes), we retain the minimum set of transcripts $\{t_i..t_j\}$ which can explain the full set of equivalence classes g_e. We say that a transcript *hits* an equivalence class if it appears in the label of this equivalence class.

This problem is clearly equivalent to minimum hitting set, and so the optimization version of this problem is NP-HARD [25]. However, if the size of the problem instance is small, it can still be solved exactly, and otherwise, a greedy algorithm yields an $O(\log N)$ approximation [26]. We let $T_e = \{t_{i_1}, t_{i_2}, \ldots, t_{i_k}\}$ be the minimum set of transcripts that hit all equivalence classes for g_e.

In the next phase, alevin modifies the labels of all the equivalence classes by removing transcripts not present in T_e, and corrects for potential over-estimation of the UMI counts (shown in Figure 2) by avoiding obvious cases of double counting. Specifically, any UMI present in more than one equivalence class explained by a

single transcript from the hitting set is very likely the result of a UMI collision, and we increment the count of distinct UMIs assigned to this gene by 1 for each such collision. We observe in most of the datasets in this manuscript an increase of $\sim$ 1% of the total UMIs reported after correcting for collisions.

2.3 Final whitelisting

Many existing tools for whitelisting CBs, such as Cell-Ranger [2] and Sircel[8] perform whitelisting only once. As discussed above, both tools rely on the assumption that the number of times a CB is observed is sufficient to identify the *correct* CBs, i.e. those originating from droplets containing a cell. However, as observed by Petukhov et. al [10], there is considerable variation in sequencing depth per-cell, and some droplets may contain damaged or low-quality cells. Thus, true CBs may fall below a simple knee-like threshold. Similarly, erroneous CBs may lie above the threshold. Petukhov et. al [10] proposed that instead of selecting a single threshold, one should treat whitelisting as a classification problem and segregate CBs into three regions; high-quality, low-quality and uncertain / ambiguous. Here, high-quality refers to the CBs which are deemed to be definitely correct, and low-quality are the CBs which are deemed to most likely not arise from valid cells. A classifier can then be trained on the high and low-quality CBs to classify the barcodes in the ambiguous region as either high or low-quality. We adopt this approach in alevin, using our knee method's cutoff to determine the ambiguous region. Specifically, we divide everything above the knee threshold into two equal regions; high-quality valid barcodes (upper-half), denoted by $\mathcal{H}$, and ambiguous barcodes (lower-half), denoted by $\mathcal{L}$. To learn the low-quality region, we take $n_l = \max(0.2 \cdot |\mathcal{H}|, 1000)$ barcodes below the knee threshold.

We used a slightly different set of features than Petukhov et al.[10], which we believe may capture the differences between high and low-quality cells better (see Section S1.1). We chose to use a naïve Bayes classifier to perform classification, since we observed no clear difference between multiple ML methods (not shown), and the naïve Bayes classifier yields classification probabilities which are easy to interpret. Our final set of whitelisted CBs are those classified as high-confidence. We observe (see Table 1) that the number of high-confidence cells predicted by alevin are in close proximity to the count of cells predicted by Cell-Ranger, but there are non-trivial differences.

	alevin	Cell-Ranger
neurons_900	1,290	931
neurons_2000_fixed	2,469	1,920
neurons_2000	1,889	2,022
t_4k	4,192	4,538
neuron_9K	8,669	9,128

Table 1: Number of whitelisted cellular barcodes for alevin and Cell-Ranger

3 Results

3.1 Comparing alevin to existing approaches

To demonstrate the improved runtime and memory usage of alevin, we analyzed 6 10x Chromium datasets [27], comprising approximately 1000—10000 mouse brain

cells and human T cells, with approximately 50-400 million reads per sample. Each dataset was processed with alevin with default settings (as in Section S1.5) and the runtime and memory usage was compared with that observed for the Cell-Ranger pipeline [2] (v2.1.1) and a custom pipeline using STAR(v2.4.2a) [28], featureCounts(v1.4.6) [29] and UMI-tools (v0.5.3) [7], which represents a commonly used alignment-based approach, which we refer to as *naïve* [30].

The runtime for alevin is roughly an order of magnitude faster than Cell-Ranger and *naive* (Figure 3). For example, with the `neuron_9K` dataset, alevin took 26 minutes to process the 383M reads to a gene-level quantification estimates, compared to 440 minutes for Cell-Ranger and 289 for *naive* (all using up to 20 threads, see Figure S4 and Section S1.3 for discussion of the effect of different numbers of threads).

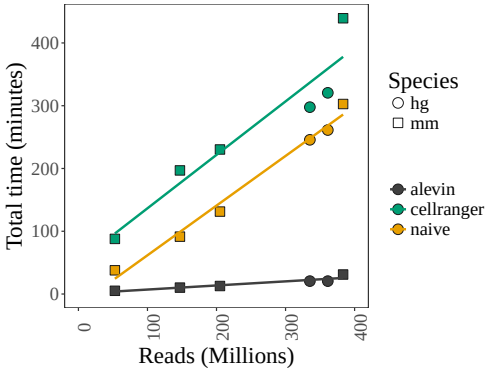

Figure 3: Wall-Clock run time for alevin, Cell-Ranger and *naive* on two species (human and mouse) on varying number of total raw reads.

As expected, we observed a near-linear increase in runtime for all pipelines with respect to the number of input reads (Figure 3). In contrast, memory usage for alevin increased linearly with respect to the number of cells. Linear modeling indicated that for the mouse samples, alevin memory usage is approximately 5.3 GB + 1.5 GB per 1000 cells (at the per-cell read depth present in these samples). Maximum memory usage for *naive* was invariant (19 and 21 GB for mouse and human respectively) across the range of inputs tested, being the amount of memory used by the STAR index. Since Cell-Ranger is optimized to use whatever memory is available, a comparison of memory usage is less informative. Cell-Ranger requires a minimum of 16 GB memory to run all pipeline stages. However, in order to ensure that the extra run time observed was not due to limitations in available memory, we allocated 120GB (see Section S1.3 for details).

Finally, we assessed the gene-level quantification estimates produced by these different methods. Since we don't have a ground truth expression to test against, we compared the estimates from each pipeline, as an initial confirmation that alevin generates similar gene-level quantification estimates to existing popular approaches. Given that Cell-Ranger and *naive* are more similar in their approach, being based upon alignment to the genome using STAR, we expected the quantifications from these two pipelines to be the most similar.

We consider two main metrics in our comparisons. First, we compare the expressed mean absolute relative difference (eMARD) of the alevin-default, alevin-naive, and *naive* pipelines with Cell-Ranger (shown in Figure 4) across all the cells and stratified by experiment. The eMARD is defined as:

$$\text{eMARD} = \frac{1}{|\mathcal{W}|}\sum_{j=1}^{|\mathcal{W}|}\text{eMARD}_j, \text{ where } \text{eMARD}_j = \frac{1}{\sum_{i=1}^{|\mathcal{G}|} 1_{x_i+y_i>0}}\sum_{i=1}^{|\mathcal{G}|}\frac{|x_i-y_i|}{x_i+y_i}, \quad (1)$$

where $\mathcal{W}$ is the set-intersection of the CBs used by two compared tools (which is same since here we made use of the external CB whitelists) and $\mathcal{G}$ is the set of genes. Traditionally, bulk RNA-seq experiments are compared by considering the full set of genes available in the reference (and considering division by $x_i + y_i = 0$ as 0), but the sparsity in single-cell experiments, due to relatively-low number of expressed genes per-cell, makes it hard to compare two tools in this way. We chose to use eMARD (Equation (1)) for comparing the genes reported as expressed by at least one of the tools being compared. We have also calculated the expressed gene agreement matrix for both the alevin and *naive* methods in the human and mouse datasets (Section S1.4). These matrices show the genes determined to be "expressed" (under the very permissive condition of yielding at least a single read) by the alevin and *naive* methods compared with Cell-Ranger. We find the same trend observed under the eMARD metric. Specifically, we note that all methods yield similar results, with the agreement being largest between Cell-Ranger and *naive*— as is expected given that they make use of the same alignments (produced by aligning reads to the genome with STAR) and use quite similar methods for deduplication.

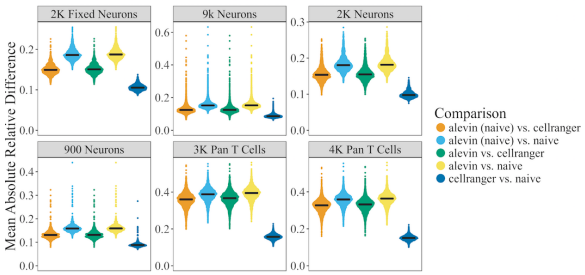

Figure 4: Expressed Mean Absolute Relative Difference (eMARD) based comparison of alevin-default, alevin-naive, with Cell-Ranger and *naive* pipelines.

We also attempted to compare alevin to a kallisto (v0.44)-based pipeline [31] (which we refer to as kb-pipe), but such a comparison poses a number of difficulties. First, kb-pipe is currently designed for Chromium v1 chemistry rather than the v2 chemistry for which alevin has been primarily developed. Second, and more importantly, the output of the pipelines is not of the same basic form. While alevin, by default, produces a gene-level count matrix, the kb-pipe method produces transcript compatibility counts [11] (i.e. equivalence class counts) as output.

Therefore, we made a more rudimentary comparison between alevin and kb-pipe on a single version 1 chemistry sample. We refrained from attempting to normalize across the fundamentally different output types, and instead chose to benchmark only the time and memory requirements of these tools when processing this dataset. We observed a ~ 4 fold improvement in running time and ~ 8 fold improvement in the memory usage for alevin (Section S1.2).

4 Discussion and Conclusion

We have shown that the alevin achieves substantial runtime advantages compared to current approaches, whilst the quantification estimates are quite similar to those of common alignment-based pipelines. For small and medium-scale experiments, alevin also uses less memory. This brings single-cell analysis in reach of those with only moderate compute resources, and enables rapid re-analysis of dscRNA-seq

data for e.g, meta-analyses. This has been achieved in two ways: first by the use of a lightweight, efficient, and transcript-aware UMI-deduplication algorithm, and second by streamlining of the process into a single framework that avoids the use of disk to store intermediate results. Nonetheless, alevin is very much an active and evolving package for dscRNA-seq quantification and processing, and there are a number of opportunities for improvement that we are pursuing.

4.1 Methodological considerations and comparisons

There are many areas of on-going methodological development within alevin. Here, we discuss some areas of active consideration and development.

One area for potential improvement is the process of handling erroneous CBs. Specifically, rather than trying to *solve* the CB assignment problem upstream of quantification, one can imagine probabilistically (or *soft*) assigning the erroneous CBs (i.e., fractionally) to two or more whitelisted CBs. We have considered and attempted this approach, but in our experiments, we have largely focused on 10x data and observed that it is quite "clean" with respect to erroneous barcodes (i.e., most erroneous barcodes have at most 1 putative label). In contrast, Drop-seq (another dscRNA-seq method), exhibited a much higher rate of ambiguity, and more refined probabilistic models for barcode labeling deserve further consideration. In addition, where the CB sequence is random (e.g [1, 3]), there is a small but not insignificant probability of two cells receiving a CB which is within one edit distance. As such, it may be worth exploring the use of additional filters to avoid erroneously assigning CBs from droplets containing cells, but which nonetheless fall below the initial knee threshold, to another CB. Moreover, it is likely worth exploring more sophisticated approaches for identifying and removing ambient RNA. Here, work along the lines of the recently-introduced SoupX [4] and EmptyDrops [32] looks particularly promising.

Another major area of consideration is how best to deduplicate reads using UMI information. In particular, in protocols where most of the fragmentation occurs after amplification, it is not straightforward to identify the group of potential duplicate reads in which to assess the UMIs, as true duplicate reads may contain sequences from different portions of the same gene. For this reason, UMI-tools [7] can consider all reads mapping to the same gene as potential duplicates. However, this approach discards relevant transcript-level information contained in the read mapping and tends to over-collapse UMIs. For example, if there are reads sharing a UMI and mapping to the same gene, but which very likely derived from distinct isoforms of this gene, the UMI-tools approach will falsely collapse these reads.

On the other hand, the approach taken during single-cell processing in kb-pipe uses equivalence classes [15, 19–24] as the basis for UMI de-duplication (reads from the same cell, that resides within the same equivalence class are considered duplicates if they share the same UMI). However, unlike UMI-tools, this pipeline only identifies exact duplicate UMI tags, and does not consider possible errors in the UMI sequence. Furthermore, this approach implicitly makes use of the transcript-level information contained within the equivalence classes, but can potentially under-collapse UMIs (e.g., in the case where a given UMI occurs in multiple equivalence classes that correspond to the same gene, and where the reads may have derived from positions compatible with different fragments drawn from the same isoform).

Here we have taken an intermediate approach, and perform UMI deduplication at the level of genes, but then use transcript-level information — derived from a minimum hitting set of transcripts over the equivalence classes belonging to a gene — to avoid over collapsing UMIs that have clear evidence of deriving from distinct transcripts. We observe that simply collapsing UMIs per-equivalence class, even if one considers the 1 edit-distance neighborhood instead of simply exact duplicates, can likely result in overestimating distinct UMIs by a factor of close to 2 (Section S1.2.2). Nonetheless, complicated situations can exist where our formulation does not yield a clear, optimal resolution for collapsing the UMI network (for example, when an equivalence class is hit by multiple elements of the hitting set). Therefore, we are interested in exploring further refinements and additions to our current approach. Specifically, we are currently exploring how to properly integrate our UMI deduplication methodology to best handle deduplication in cases where multi-mapping reads span multiple genes.

4.2 Future directions

Alongside the areas of development discussed above, we are in the process of performing more detailed assessments to demonstrate the value of the two-step whitelisting approach implemented in alevin. Finally, we also implemented, within alevin, a simplified form of the bulk RNA-seq generative model which allows using an optimization algorithm to resolve the ambiguity of reads that map between multiple genes, rather than discarding gene-ambiguous reads as is currently done. However, some of the basic assumptions of common models for bulk RNA-sequencing does not hold in tagged-end data (e.g., general length effects). In general, determining the appropriate generative model for dscRNA-seq data is still an active area of research.

While developing alevin, we have run the software on many experiments, ranging from hundreds of cells to over then thousand cells. Yet, as discussed above, the number of cells assayed per "experiment" is rapidly increasing, with some recent datasets assaying over a million cells. If presented as one, monolithic experiment (one pair of FASTQ files), the memory requirements for alevin on such data can be a bottleneck. However, most large experiments consist of collections of individual samples, which can be processed largely independently, and in which barcodes may be reused (for example, the largest experiment of which we are currently aware — assaying ~ 1.3 Million mouse cells [33] — is actually combination of 133-individual samples). We imagine two ways to handle such large datasets within the alevin framework:

- When large experiments consist of multiple individual samples, then alevin can, in theory, be run independently on each such sample. Subsequently, the count matrices, which consist of disjoint subsets of cells, can be merged together into a larger gene-by-cell count matrix (accounting, of course, for barcode that may be reused for different cells between samples). We note that one must still be careful when performing the aggregation of the count matrices, since one might have to correct for potential confounding factors (e.g. performing batch correction). How to most accurately and efficiently handle such aggregation is an active area of alevin's development.

- We also note that, after the initial whitelisting step (which, itself, takes very little memory even for millions of cells), it is possible to have alevin process input files in a multi-pass fashion. Here, some subset of barcodes will be processed on each pass over the file, allowing one to limit the total memory used by alevin in exchange for the extra time required to make multiple passes over the input files. Integrating this feature into alevin is conceptually straightforward and a development priority, though it is not yet complete.

These parts of alevin are still under heavy development, though they pose a promising direction for future research. We envisage that alevin will continue to evolve and improve as we receive feedback and further suggestions from the community.

Author's contributions

AS, TS, IS and RP devised the methods, AS and RP implemented the methods, AS, TS, IS and RP devised and carried out the experiments, all authors wrote and approved the manuscript.

Acknowledgments

The authors wish to acknowledge all the members of COMBINE-lab for their support and valuable feedback, and wish to especially acknowledge Hirak Sarkar for his help in crafting Figure 1 of the manuscript. The authors would also like to thank Mike Love for comments and suggestions made in the initial development of the work.

Author details

[1]Department of Computer Science, Stony Brook University, New York, USA. [2]Cambridge Centre for Proteomics, Department of Biochemistry, University of Cambridge, CB2 1GA Cambridge, UK. [3]Sheffield Institute for Nucleic Acids, Department of Molecular Biology and Biotechnology, The University of Sheffield, Sheffield, UK.

References

1. Macosko, E.Z., Basu, A., Satija, R., Nemesh, J., Shekhar, K., Goldman, M., Tirosh, I., Bialas, A.R., Kamitaki, N., Martersteck, E.M., *et al.*: Highly parallel genome-wide expression profiling of individual cells using nanoliter droplets. Cell **161**(5), 1202–1214 (2015)
2. 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 **8**, 14049 (2017)
3. Klein, A.M., Mazutis, L., Akartuna, I., Tallapragada, N., Veres, A., Li, V., Peshkin, L., Weitz, D.A., Kirschner, M.W.: Droplet barcoding for single-cell transcriptomics applied to embryonic stem cells. Cell **161**(5), 1187–1201 (2015)
4. Young, M.D., Behjati, S.: Soupx removes ambient rna contamination from droplet based single cell rna sequencing data. bioRxiv (2018). doi:10.1101/303727. https://www.biorxiv.org/content/early/2018/04/20/303727.full.pdf
5. Svensson, V., Natarajan, K.N., Ly, L.-H., Miragaia, R.J., Labalette, C., Macaulay, I.C., Cvejic, A., Teichmann, S.A.: Power analysis of single-cell rna-sequencing experiments. Nature methods **14**(4), 381 (2017)
6. Islam, S., Zeisel, A., Joost, S., La Manno, G., Zajac, P., Kasper, M., Lönnerberg, P., Linnarsson, S.: Quantitative single-cell rna-seq with unique molecular identifiers. Nature methods **11**(2), 163 (2014)
7. Smith, T., Heger, A., Sudbery, I.: Umi-tools: modeling sequencing errors in unique molecular identifiers to improve quantification accuracy. Genome research **27**(3), 491–499 (2017)
8. Tambe, A., Pachter, L.: Barcode identification for single cell genomics. bioRxiv, 136242 (2017)
9. Zhao, L., Liu, Z., Levy, S.F., Wu, S.: Bartender: a fast and accurate clustering algorithm to count barcode reads. Bioinformatics (2017)
10. Petukhov, V., Guo, J., Baryawno, N., Severe, N., Scadden, D., Kharchenko, P.V.: Accurate estimation of molecular counts in droplet-based single-cell rna-seq experiments. bioRxiv, 171496 (2017)
11. Ntranos, V., Kamath, G.M., Zhang, J.M., Pachter, L., David, N.T.: Fast and accurate single-cell rna-seq analysis by clustering of transcript-compatibility counts. Genome biology **17**(1), 112 (2016)
12. Satija, R., Farrell, J.A., Gennert, D., Schier, A.F., Regev, A.: Spatial reconstruction of single-cell gene expression data. Nature biotechnology **33**(5), 495 (2015)
13. Usoskin, D., Furlan, A., Islam, S., Abdo, H., Lönnerberg, P., Lou, D., Hjerling-Leffler, J., Haeggström, J., Kharchenko, O., Kharchenko, P.V., *et al.*: Unbiased classification of sensory neuron types by large-scale single-cell rna sequencing. Nature neuroscience **18**(1), 145 (2015)
14. Qiu, X., Mao, Q., Tang, Y., Wang, L., Chawla, R., Pliner, H.A., Trapnell, C.: Reversed graph embedding resolves complex single-cell trajectories. Nature methods **14**(10), 979 (2017)
15. Patro, R., Duggal, G., Love, M.I., Irizarry, R.A., Kingsford, C.: Salmon provides fast and bias-aware quantification of transcript expression. Nature methods **14**(4), 417 (2017)
16. Grüning, B., Dale, R., Sjödin, A., Rowe, J., Chapman, B.A., Tomkins-Tinch, C.H., Valieris, R., Köster, J., et al.: Bioconda: A sustainable and comprehensive software distribution for the life sciences. bioRxiv, 207092 (2017)
17. Srivastava, A., Sarkar, H., Gupta, N., Patro, R.: Rapmap: a rapid, sensitive and accurate tool for mapping rna-seq reads to transcriptomes. Bioinformatics **32**(12), 192–200 (2016)

18. 10x-Genomics Single-Cell 3'-V2 Kit. https://teichlab.github.io/scg_lib_structs/data/CG000108_AssayConfiguration_SC3v2.pdf
19. Turro, E., Su, S.-Y., Gonçalves, Â., Coin, L.J., Richardson, S., Lewin, A.: Haplotype and isoform specific expression estimation using multi-mapping rna-seq reads. Genome biology **12**(2), 13 (2011)
20. Mezlini, A.M., Smith, E.J., Fiume, M., Buske, O., Savich, G.L., Shah, S., Aparicio, S., Chiang, D.Y., Goldenberg, A., Brudno, M.: ireckon: simultaneous isoform discovery and abundance estimation from rna-seq data. Genome research **23**(3), 519–529 (2013)
21. Patro, R., Mount, S.M., Kingsford, C.: Sailfish enables alignment-free isoform quantification from rna-seq reads using lightweight algorithms. Nature biotechnology **32**(5), 462 (2014)
22. Bray, N.L., Pimentel, H., Melsted, P., Pachter, L.: Near-optimal probabilistic rna-seq quantification. Nature biotechnology **34**(5), 525 (2016)
23. Zhang, Z., Wang, W.: Rna-skim: a rapid method for rna-seq quantification at transcript level. Bioinformatics **30**(12), 283–292 (2014)
24. Ju, C.J.-T., Li, R., Wu, Z., Jiang, J.-Y., Yang, Z., Wang, W.: Fleximer: Accurate quantification of rna-seq via variable-length k-mers. In: Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics, pp. 263–272 (2017). ACM
25. Bernhard, K., Vygen, J.: Combinatorial optimization: Theory and algorithms. Springer, Third Edition, 2005. (2008)
26. Vazirani, V.V.: Approximation Algorithms. Springer, ??? (2013)
27. 10x-Genomics v2 Chemistry Data. https://support.10xgenomics.com/single-cell-gene-expression/datasets
28. Dobin, A., Davis, C.A., Schlesinger, F., Drenkow, J., Zaleski, C., Jha, S., Batut, P., Chaisson, M., Gingeras, T.R.: Star: ultrafast universal rna-seq aligner. Bioinformatics **29**(1), 15–21 (2013)
29. Liao, Y., Smyth, G.K., Shi, W.: featurecounts: an efficient general purpose program for assigning sequence reads to genomic features. Bioinformatics **30**(7), 923–930 (2013)
30. http://hemberg-lab.github.io/scRNA.seq.course/construction-of-expression-matrix.html
31. Single-Cell based Kallisto's Notebook. https://github.com/pachterlab/scRNA-Seq-TCC-prep (2016)
32. Lun, A., Riesenfeld, S., Andrews, T., Gomes, T., Marioni, J., et al.: Distinguishing cells from empty droplets in droplet-based single-cell rna sequencing data. bioRxiv, 234872 (2018)
33. E18 mouse brain cells. https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSM2453144 (Jan 12, 2017)
34. Köster, J., Rahmann, S.: Snakemake—a scalable bioinformatics workflow engine. Bioinformatics **28**(19), 2520–2522 (2012)
35. CGAT-core, a python library. http://cgat-core.readthedocs.io/en/latest/

Supplementary Methods

S1.1 Whitelist Classification

In the implementation of Petukhov et al.[10], a kernel density estimation classifier was trained using features which described the number of reads per UMI, UMIs per gene, the fraction of intergenic reads, non-aligned reads, the fraction of lowly expressed genes and the fraction of UMIs on lowly expressed genes. In addition, a maximum allowable mitochondrial read content was set for a CB to be classified as "high-quality". Whilst these features enabled the authors to build a classifier which efficiently separated "high-quality" cells from "low-quality" cells, we believe it may be possible to improve this set of features. Specifically, most of these features would be expected to correlate with the number of reads or UMIs per CB. Thus, the classifier is encouraged towards simply learning the feature attributes associated with higher read depth, when in fact one wants it to learn the feature attributes associated with high-quality cells. We therefore used a slightly different set of features which we believe may better capture the differences between high and low-quality cells:

1. Fraction of reads mapped
2. Fraction of mitochondrial reads
3. Fraction of rRNA reads
4. Duplication rate
5. Mean gene counts post de-duplication
6. The maximum correlation of gene-level quantification estimates with the high-quality CBs. (Optionally activated by `--useCorrelation` flag.)

S1.2 10x Version-1 Chemistry

S1.2.1 Resource Utilization

Currently, kallisto handles scRNA-Seq processing as part of a pipeline that also makes uses of a number of preprocessing scripts for formatting and preparing the input [31]. We refer to this entire pipeline as kb-pipe. For dscRNA-seq data, the scripts appear to currently process only v1 Chromium chemistry. Therefore, in order to compare alevin with kb-pipe, we wrote a wrapper around alevin to make the comparison on a v1 chemistry dataset possible. We note that this wrapper transforms data on-the-fly, and, itself, can be considerably optimized. We used the dataset of Zheng et al. [2], and observe (Table S1) ~ 4 fold improvement in running time and ~ 8 fold improvement in the memory usage of the tools favoring alevin.

Table S1: Wall clock run time of alevin and kb-pipe for the PBMC 10x (v1 chemistry) data.

	alevin	kb-pipe
Time (Minutes)	21.23	83.34
Memory (GB)	7.14	54.94

S1.2.2 Effect of deduplication methodology

The direct comparison of kb-pipe (whose final output is a collection of per-cell equivalence class counts) with more traditional pipelines considered in this paper (whose output is a cell-by-gene count matrix) is not straightforward. In fact, even the best approaches for downstream processing of these different quantification output types could differ considerably. However, to set a baseline suggesting how per-equivalence class deduplication of UMIs would perform if used to directly obtain gene-level counts, we have implemented an equivalence class-level UMI deduplication mode for alevin. In this mode (triggered by the `--eqClassLevel` flag), alevin performs UMI deduplication independently in each equivalence class. However, it still corrects for PCR and sequencing errors in UMIs via the directional approach. Since both the default and equivalence class-level modes generate a gene-level count matrix (while discarding gene-ambiguous equivalence classes) they can be compared. In the PBMC v1 10x data [2], we have observed ~ 2 times (Table S2) more UMIs reported by the independent equivalence class-based deduplication approach as compared to alevin's default approach (we provide an illustrative description of why this is likely to occur in Figure S1). Here, it is very likely that most of these differences are the result of UMIs deriving from the same pre-PCR molecule failing to be collapsed by the independent equivalence class-level deduplication approach. Thus, while this approach may be adequate, or even desirable, for equivalence class-level analysis (as is suggested downstream of kb-pipe), it does not seem, itself, ideal for producing gene-level UMI counts.

Table S2: Reported number of (deduplicated) UMIs for the same 10x Chromium v1 dataset [2], when using different UMI deduplication procedures.

	default	equivalence class-level
#UMIs	$9,247,187$	$17,034,101$

S1.3 Alevin: Runtime and Memory

We compared the runtime and memory usage of alevin by varying the number of cells, the number of reads, and the number of threads. We observe large differences in the runtimes of various tools across datasets from 900-9k cells and ~ 50 million to ~ 400 million reads (Figure S2). We also benchmarked the memory usage of the tools, as shown in Section S1.3, across the same set of datasets (measurements were made using `/usr/bin/time`). We observed that the memory usage of alevin scales linearly with the number of cells ($\sim$ 1.5GB of memory for every 1000 cells) and the number of reads ($\sim$ 5Mb of memory for every million reads), in addition to the salmon reference transcriptome index ($\sim$ 6GB for mouse or human). The memory usage of the *naive* pipeline is approximately equal to the size of the STAR reference index ($\sim$ 18.5GB). Memory usage profiling of Cell-Ranger cannot be achieved using standard command-line profiling commands such as `/usr/bin/time` since Cell-Ranger starts sub-processes which are not properly profiled by this instrumentation. For the profiling of Cell-Ranger run time presented here, we allocated 120 GBs of memory for the top-level process by providing the option `--localmem 120`. By submitting a single Cell-Ranger job to a dedicated cluster node and taking snapshots of the complete memory usage on the node, we estimate that the maximal

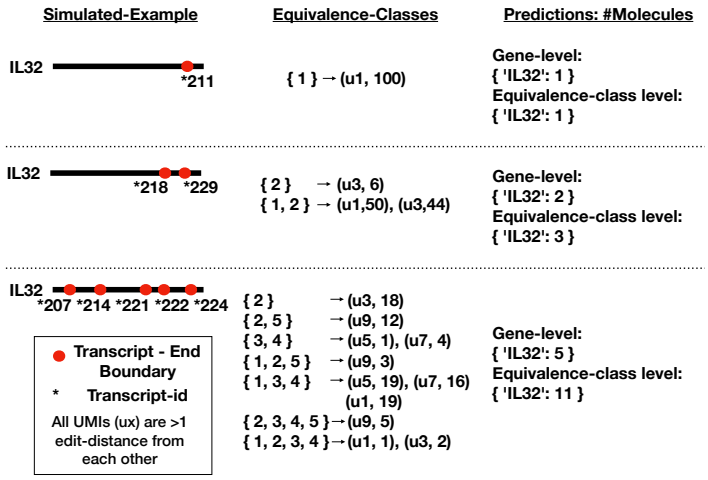

Figure S1: A simulated example demonstrates how treating equivalence classes individually during UMI de-duplication can lead to under-collapsing of UMIs compared to gene-level methods (especially in protocols where the majority of cDNA amplification occurs prior to fragmentation). In this example, annotated transcript sequences of the human Interleukin 32 (IL32) gene is used as a source of read simulation. Each simulation assumes (top to bottom) various numbers of expressed transcript(s). Reads are randomly sampled from the 3'-end of the annotated-transcript(s) according to a realistic fragment length distribution, where exon overlap induces the corresponding equivalence classes of each fragment. Every simulation assumes 1 (pre-PCR) molecule is expressed for each transcript, identifiable by a unique-id (UMI) ux. Within each simulation, all UMIs are at an edit distance > 1, and hence not collapsible. In the first simulation, having only a single expressed transcript, both methods correctly report a UMI count of 1. In the second and third simulations, where there are 2 and 5 expressed transcripts respectively, the deduplication methods report different number of molecules. Specifically, as the rate of splicing (and hence the number of equivalence classes) increases, so too does the number of distinct UMIs reported. In the bottom row, this method reports 11 distinct UMIs, while the gene-level method reports (in this case correctly) 5 distinct UMIs.

memory usage was $\sim$ 80 GB, even with the smallest 900 cell dataset (Section S1.3) . Although not a precise profile of memory usage, this captures, roughly, the memory usage of the Cell-Ranger pipeline under this configuration. This measurement is somewhat self-referential, since we have specified the amount of memory which Cell-Ranger should be allowed to use. However, this ensures that the additional runtime observed for Cell-Ranger is not due to limitations in available memory.

Alevin is designed to make efficient use of multiple threads, though the optimal number of threads can depend on many factors like the speed of the underlying disk and the size of the raw input and output matrix to be written. We ran alevin using a varying number of threads to observe how the runtime and memory usage scale as a function of the number of threads. On our testing systems, we observed that 8-10 threads appear to be optimal (Section S1.3). Each additional thread appears to require $\sim$ 27 MB of memory (Section S1.3).

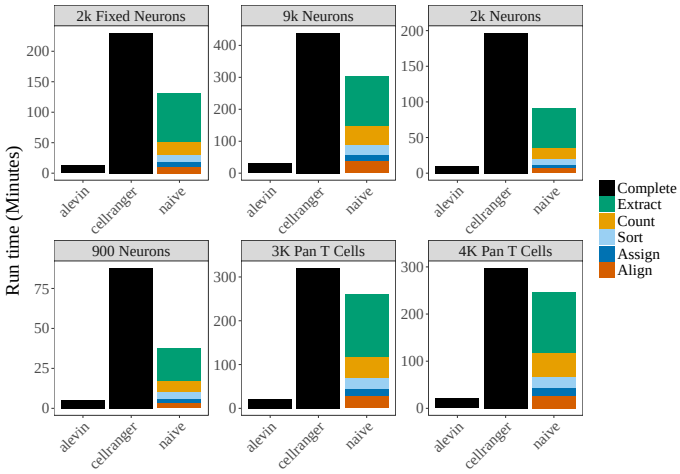

Figure S2: Wall-Clock run time for alevin, Cell-Ranger and *naive* on two species (human and mouse) and varying number of total reads and cells. For *naive* the run time for each stage was measured separately

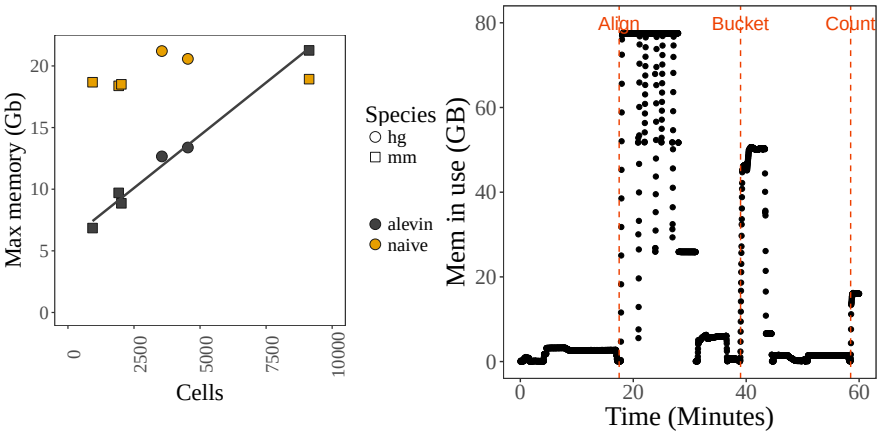

Figure S3: (left). Memory Usage comparison for alevin and *naive*. Cell-Ranger is not included as its memory usage is determined by the user. (right). Memory usage of Cell-Ranger for 900 cell datasets by taking per-second snapshot of the memory usage on a dedicated node. The start of various stages in the pipeline are indicated by dashed red lines

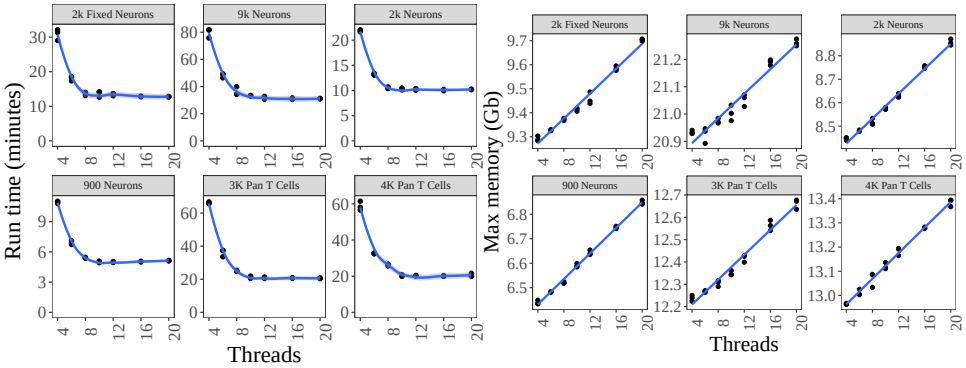

Figure S4: (left) Runtime and (right) Memory usage comparison of alevin across various datasets and monotonically increasing the number of threads.

S1.4 Expressed gene agreement matrices for alevin and *naive* compared with Cell-Ranger

Table S3: Expressed gene agreement matrix between Cell-Ranger and alevin/ *naive*, based on the number of expressed genes, averaged over all cells, and stratified by experiment.

	Human-t_4k	
	Cell-Ranger (present)	Cell-Ranger (absent)
alevin (present)	1112.53	422.89
alevin (absent)	79.14	56 673.45
	Cell-Ranger (present)	Cell-Ranger (absent)
naive (present)	1111.86	92.10
naive (absent)	79.81	57 004.23
	Human-t_3k	
	Cell-Ranger (present)	Cell-Ranger (absent)
alevin (present)	1176.81	522.82
alevin (absent)	91.26	56 497.11
	Cell-Ranger (present)	Cell-Ranger (absent)
naive (present)	1181.98	106.10
naive (absent)	86.09	56 913.84
	Mouse-neuron_9k	
	Cell-Ranger (present)	Cell-Ranger (absent)
alevin (present)	2641.87	255.55
alevin (absent)	65.75	50 415.83
	Cell-Ranger (present)	Cell-Ranger (absent)
naive (present)	2602.95	96.95
naive (absent)	104.67	50 574.43
	Mouse-fixed_neurons_6days_2000	
	Cell-Ranger (present)	Cell-Ranger (absent)
alevin (present)	2883.29	277.78
alevin (absent)	145.99	50 071.94
	Cell-Ranger (present)	Cell-Ranger (absent)
naive (present)	2882.36	118.17
naive (absent)	146.93	50 231.55
	Mouse-neurons_2000	
	Cell-Ranger (present)	Cell-Ranger (absent)
alevin (present)	2613.20	293.76
alevin (absent)	109.69	50 362.35
	Cell-Ranger (present)	Cell-Ranger (absent)
naive (present)	2595.57	98.44
naive (absent)	127.33	50 557.67
	Mouse-neurons_900	
	Cell-Ranger (present)	Cell-Ranger (absent)
alevin (present)	3321.31	333.72
alevin (absent)	85.38	49 638.60
	Cell-Ranger (present)	Cell-Ranger (absent)
naive (present)	3270.65	112.91
naive (absent)	136.04	49 859.41

S1.5 Machine Configuration and Pipeline Replicability

10x v1 chemistry benchmarking has been scripted using Snakemake [34] and performed on an Intel(R) Xeon(R) CPU (E5-2699 v4 @2.20GHz with 44 cores and 56MB L3 cache) with 512GB RAM and a 4TB TOSHIBA MG03ACA4 ATA HDD running Ubuntu 16.10.

10x v2 chemistry benchmarking has been scripted using CGATCore (`https://github.com/cgat-developers/cgat-core`). The full pipeline and analysis using [35] is available at `https://github.com/COMBINE-lab/alevin-paper-pipeline`. (configuration may be necessary to run on a cluster).