Quik 2.0: Efficient Large-Scale DNA Barcode Calling

Steffen Schüler
Martin Luther University Halle-Wittenberg, Germany

Antonia Schmidt
Martin Luther University Halle-Wittenberg, Germany

Matthias Müller-Hannemann
Martin Luther University Halle-Wittenberg, Germany

Abstract

DNA barcodes are used as unique identifiers in high-throughput sequencing technologies with applications in areas such as single cell analysis, spatial transcriptomics and DNA data storage. Given a set of barcodes and a set of reads, each containing a barcode, the task of barcode calling is to assign each read to its respective barcode. This is challenging in applications involving large barcode sets and high rates of base insertion, deletion and substitution errors. Naive solutions require the calculation of pairwise distances between each barcode and read. As this is infeasible for modern applications with millions of barcodes and billions of reads, much work has been done during the previous years in accelerating this task. In 2026, Uphoff et al. introduced the barcode calling tool *Quik* based on k-mer filtering and pseudo-distances. They demonstrated that it is faster than state-of-the-art tools by several orders of magnitude. Here, we present *Quik* 2.0, which is faster than the original release by a factor of up to 56 and scales well to multiple GPUs. We discuss several algorithmic design choices that led to this speedup. In large-scale experiments with 10^6 barcodes, we can now process approximately 300 million reads per hour on a GPU server equipped with four GPUs. In addition, we show that unfiltered barcode calling approaches can only slightly improve the accuracy at the cost of a vastly increased running time. Finally, we introduce more fine-grained assignment rejection criteria to achieve a better trade-off between precision and acceptance rate. To help users select suitable rejection parameters for real-world experiments, we propose an automatic calibration procedure that optimizes parameters for specific barcode and read sets.

2012 ACM Subject Classification Applied computing → Bioinformatics; Computing methodologies → Parallel algorithms; Applied computing → Computational genomics

Keywords and phrases DNA barcode calling, k-mer filtering, GPU computing, algorithm engineering, spatial transcriptomics

Digital Object Identifier 10.4230/LIPIcs.WABI.2026.14

Supplementary Material *Software (Source Code)*: `https://github.com/uni-halle/quik` [20] archived at `swh:1:dir:948d8ae0e0b66260e91d201ea0b5f6f732aa4eff`

1 Introduction

DNA barcodes are short unique sequences of nucleotides added to DNA fragments to identify samples in pooled sequencing experiments. They have become an essential tool in modern genomics, supporting applications such as single-cell analysis [10], spatially resolved transcriptomics [13, 23], and emerging technologies like DNA-based information storage [2]. Barcodes can encode samples in multiplex experiments, cell lines for single-cell transcriptomics or locations in spatial transcriptomics experiments.

Advances in sequencing technologies now allow experiments to generate vast numbers of reads, each associated with a barcode. In practice, these barcodes are rarely observed without errors: synthesis imperfections and sequencing artifacts introduce substitutions, insertions, and deletions. Recovering the original barcode from such corrupted observations is known

as *barcode calling* and constitutes a key step in downstream analysis. From an algorithmic perspective, barcode calling can be formulated as a nearest-neighbor search problem with respect to a suitable distance measure on DNA sequences. The basic algorithmic task is as follows: Given a set of barcodes and a set of reads, determine a barcode of minimum distance for each read.

Related Work. If the expected error rate is very low, one can simply look for exact barcode matches, which is very fast and often used in practice. Error-correcting codes such as those from Hawkins et al. [9] also allow fast barcode calling, but impose severe limitations on the number of barcodes and the error rate. Modern techniques such as photolithographic microarray synthesis enable the synthesis of millions of barcodes needed for spatial transcriptomics studies at reasonable cost [4, 15]. However, their error rates may exceed 20% per base, as reported by Lietard et al. [12]. This makes naive approaches inapplicable. Recent work has shown that randomly generated barcode sets of adequate length can remain distinguishable even under substantial noise, suggesting new design principles for large-scale experiments with practically unlimited barcode sets [18].

As explained above, the central idea of many barcode calling approaches is to determine for each read a *most similar* barcode with respect to a given distance measure such as the Hamming distance [6, 14], the edit or Levenshtein distance [11, 1, 8], or the Sequence-Levenshtein (SL) distance [5]. The latter is similar to the Levenshtein distance, but allows either sequence to be truncated at the end without penalty. While the Hamming distance does not account for insertions or deletions ("indels") in the read, the Levenshtein distance and the SL distance do so. In general, the basic strategy is to use an all-to-all distance computation to identify the best match. Tools such as Dorado [16], Flexiplex [7] and Barbell [3] perform all-to-all distance computations for demultiplexing barcodes in Oxford Nanopore sequencing applications, in which barcode sets are usually small.

However, directly comparing every read against all candidate barcodes is computationally prohibitive in applications with millions of barcodes and billions of reads. This has led to the development of more scalable approaches, including data structures for efficient distance computation, clustering-based heuristics, and multi-stage filtering techniques that prune the search space before applying more expensive comparisons. Zorita et al. [24] propose the calculation of all pairwise edit distances using prefix trees and several optimizations of the classical Needleman-Wunsch algorithm as well as clustering ("starcode"). Tambe and Pachter [21] develop the barcode-calling approach "sircel" and demonstrate its efficacy for small barcode sets and short barcodes.

To cope with large barcode sets, Press introduced RandomBarcodes [18], which implements a filtering approach called triage that identifies promising barcodes by considering the occurrence and corresponding positions of trimers. Uphoff et al. developed Quik [22], which pre-filters the barcodes using a k-mer filter. Both tools achieve a significant speedup compared to exhaustive all-to-all comparisons and run on GPUs. Columba, a general-purpose read aligner developed by Renders et al. [19], was not designed for barcode calling, yet has proven effective for this purpose. It implements an exact search for hits with up to 13 errors sped up by clever search schemes and performance enhancement techniques.

Despite these improvements, achieving both high accuracy and computational efficiency remains a significant challenge, particularly in scenarios involving millions of barcodes and high error rates. A recent benchmark study by Poma-Soto et al. [17] evaluated barcode calling tools for such scenarios, including RandomBarcodes, Quik and Columba. It concluded that Quik outperforms the other tools in terms of speed, accuracy and scalability on real sequencing data from spatial transcriptomics.

Contribution. In this paper, we introduce Quik 2.0, which significantly improves upon the original version. Our achievements are as follows.

- First, we carefully reengineer the k-mer based filtering approach of Quik to better exploit the capabilities of GPUs and enable multi-GPU support. In large-scale experiments with 10^6 barcodes, we achieve a speed-up factor of 16 on a single GPU and a speed-up factor of 56 on a GPU server with four GPUs. On such a GPU server, we reach a throughput of more than 80,000 reads per second.
- Second, we provide an implementation without filtering, i.e., a version which exhaustively compares each read with each barcode. Unsurprisingly, this unfiltered barcode calling method is orders of magnitudes slower than the filtering approaches. In fact, we observed that it is more than 500 times slower in our experiments. However, this slow method is still useful to validate the filtering approaches. It turns out that the unfiltered method has only a slightly better trade-off between precision and acceptance rate.
- Third, we reconsider the final phase of barcode calling. In the original version of Quik, the user has to choose an application-dependent rejection parameter D: if the distance of a read to the barcode candidate exceeds the threshold D, this read is rejected. Such a rejection rule is necessary since in practice some reads may be corrupt due to high error rates or experimental artifacts making it impossible to detect a barcode. In this work, we revisit this simple rejection criterion and propose several more sophisticated alternatives. In particular, we consider not only the distance to the best candidate, but also the distance to the second best. Based on these two distance values, we derive more complex rejection criteria allowing more fine-grained trade-off between precision and acceptance rate.

  In practice, the choice of appropriate rejection criteria highly depends on the barcode and read set and is thus nontrivial. We propose a method to flexibly and automatically calibrate suitable rejection criteria for each data set. In large-scale synthetic experiments, we find that more sophisticated rejection criteria and their optimization can improve the trade-off between precision and acceptance rate.

Structure. The remainder of this paper is structured as followed: In the following Section 2, we revisit essential ideas of Quik [22], at a level sufficient to understand the algorithmic principles. In Section 3, we discuss several algorithmic improvements and present associated speedups. Afterwards, in Section 4, we discuss how much the unfiltered calling approach can improve the accuracy of the barcode calling at the cost of a vastly increased running time. In Section 5, we present our method to automatically determine criteria to reject falsely assigned barcodes. Finally, in Section 6, we summarize our findings.

2 Quik Barcode Calling

In this section, we briefly summarize essential ideas of Quik as introduced by Uphoff et al. [22]. Quik requires two files including n barcodes and m reads, respectively. While the barcodes are assumed to have a fixed length ℓ, the read lengths may vary. Quik assumes, however, that the reads have already approximately been trimmed to the relevant area in which the barcode is located.

Preprocessing. Quik first applies a preprocessing of the barcodes. Depending on a parameter k, the length of k-mers, it initially constructs subsets $L(x, j)$ of the barcode indices $1, \ldots, n$. For each possible k-mer $x \in \{A, C, G, T\}^k$ and k-mer start position $j \in$

$\{0, \ldots, \ell - k\}$ within a barcode, the associated subset $L(x, j)$ contains exactly the indices of the barcodes in which x occurs as a substring starting at position j. This preprocessing can be done efficiently in a single sweep over the barcodes.

Main loop. In its original design, Quik processes the reads independently of each other in parallel, associating *exactly one* CUDA thread to each read. Each thread *sequentially* runs the following steps:

1. First, it computes a so-called *pseudo-distance* between its associated read r and all barcodes (see Algorithm 1). This pseudo-distance is always non-positive, with zero as the largest possible value. The smaller the value of the pseudo-distance, the more similar is the respective barcode to the read. Hence, we are only interested in barcodes with *negative* pseudo-distance. Thus, Algorithm 1 compiles and returns a set C of all barcode indices with negative pseudo-distance, together with the array P of all pseudo-distances.
2. Next, each thread identifies from the candidate set C the $c = 100$ barcodes with minimum pseudo-distance to r. This has been implemented by a single sweep over C and insertion-sorting the barcodes into a reduced candidate list C' of at most c elements, ordered by their pseudo-distance. As c is very small, the new list C' can be stored in registers that are private to each CUDA thread. In doing so, the construction of C' is sufficiently fast and requires only a single sweep over the candidate set C.
3. In the third step, each thread calculates the SL distance between its associated read r and the c barcodes in its reduced candidate list C'. There are several approaches to calculate the SL distance. As the barcode length $\ell = 34$ is very small in our application, we found that the fastest way is the classical dynamic programming approach, which runs $\Theta(\ell^2)$ steps and uses $\Theta(\ell)$ extra registers. From the candidate barcodes of C', we select one with *minimum* SL distance and assign it to r. In case of a tie, we assign r to a barcode with smaller pseudo-distance.
4. Finally, each thread evaluates some rejection criterion to decide whether to accept the assignment of the associated read r to the previously selected barcode with minimum SL distance, or to reject it. In the original implementation of Quik, a read r is *rejected* if its SL distance is larger than some predefined rejection threshold D given by the user.

Problems and imperfections of Quik 1.0. Although Uphoff et al. demonstrated the superiority of their approach to that of Press, the original design of Quik as described above was not explicitly designed to run on GPUs. In particular, the following aspects within the first step of the main loop are suboptimal for a GPU implementation:

1. As the arrays P of pseudo-distances have n elements each (and n is about one million in our application), these arrays do *not* fit into the private or shared GPU memory spaces and must thus be stored in the *global* memory of the GPU. As each thread writes to its own array P and therefore to a *different* area of the global memory, the writing accesses to P in Line 11 of Algorithm 1 are uncoalesced and thus slow, as individual memory requests from different threads cannot be combined.
2. A similar problem occurs with the arrays C of candidate barcodes. These must also be stored in the global memory, as the number of barcodes with negative pseudo-distance cannot be bounded by a small constant in advance. As a consequence, accesses to these arrays are uncoalesced, too.

Algorithm 1 The original algorithm for the computation of the k-mer pseudo-distance between a fixed read r and all barcodes as presented by Uphoff et al. [22].

Input: Read r, barcode count n, barcode lists $L(x, j)$
Output: Candidate set $C \subseteq \{1, \ldots, n\}$ and array $P \in \mathbb{Z}^n$ of pseudo-distances
Parameters: Length ℓ of barcodes and reads, length k of k-mers, $d_{\max} = 5$

```
1  P ← 0^n        // pseudo-distance array of length n, initialized with zero
2  C ← ∅          // indices of barcodes with nonzero pseudo-distance
3  for each possible k-mer starting position i ∈ {0, ..., |r| − k} in the read r do
4      x ← r[i..i + k − 1]                          // k-mer starting at position i
5      j_start ← max(0, i − d_max)
6      j_end ← min(ℓ − k, i + d_max)
7      for each possible k-mer starting position j ∈ {j_start, ..., j_end} do
8          for each barcode index b ∈ L(x, j) do
9              if P[b] = 0 then
10                 C ← C ∪ {b}
11             P[b] ← P[b] + |i − j| − ℓ
12 return C and P
```

3. As each CUDA thread sequentially processes the lists $L(x, j)$ in Line 8 of Alg. 1 (also stored in the global GPU memory due to their size), the threads of the same CUDA block again access *different* memory locations in an uncoalesced way.
4. Furthermore, as the individual lists $L(x, j)$ have different lengths, the CUDA threads of the same CUDA block suffer from thread divergence and are limited by the length of the longest list within the CUDA block.

In summary, a main problem of the original Quik implementation were a lot of uncoalesced (and thus slow) accesses to the global memory.

3 Algorithmic Improvements

To overcome these limitations, we made several adjustments to the original algorithm, from which we are going to present some instructive ones. For each improvement step, we performed a benchmark experiment to evaluate its effect on Quik's total running time. The speedup techniques discussed here are specific for Quik but follow general principles that can be summarized as follows:

- Exploit the shared GPU memory whenever you can.
- When accessing the global GPU memory, use a coalesced memory access pattern. In doing so, memory requests from threads are combined into a small number of large, efficient memory transactions, thereby making better use of the available memory bandwith.
- Make sure that all threads have enough work.

Although these principles are well-known in the context of GPU computing, it is not yet clear how to apply them to the algorithms used in Quik.

Benchmark experiment. To assess the effect of the optimization steps, we re-ran the benchmark experiment as described by Uphoff et al [22]. In this experiment, a set of $n = 10^6$ random barcodes of length $\ell = 34$ is used, as Press showed that barcode sets of this size

and length remain distinguishable even under high error rates [18]. From these barcodes, $m = 10^6$ synthetic reads, each also of length $\ell = 34$, are generated with an error probability of 20% per base, distributed evenly between substitutions, insertions, and deletions.

In our benchmark experiment, we used Quik's 4-mer filter approach to assign the reads to the barcodes. Quik offers other filtering approaches but Uphoff et al. demonstrated that $k = 4$ gives the highest accuracy. For this reason, we focused on optimizing the 4-mer filter. We ran the program on a Linux system with four AMD EPYC 7662 64-Core processors (thus giving a total of 256 logical CPU cores), 1 TB of main memory, and four NVIDIA A100-SXM4 GPUs, each having 6912 CUDA cores and 40 GB of GPU memory. As the original version of Quik does not support the use of multiple GPUs, only one of them is used until the very last step. For each optimization step, we determined the average wall clock time of three independent runs as our performance measure.

Step 1: Omit the candidate set C. As discussed in the previous section, the construction of the barcode set C with negative pseudo-distance induces an uncoalesced memory access pattern. In fact, our benchmark experiment showed that it is faster to omit the construction of C and, in the second step of the main loop, to scan *all* barcodes while identifying the ones with smallest pseudo-distance. This gives a speedup factor of about 2.17 to the original version of Quik.

Step 2: Introducing parallelism at the read level. Instead of assigning a single thread to each read, we associate with each read a whole *block* of threads. The exact number t of threads per block is a parameter that can be optimized for the fine tuning of the algorithm. In the following steps, we experimented with different values of t ranging from 32 to 256 and individually selected a value of t that minimizes the running time. Introducing multiple threads per read gives us the opportunity to parallelize various parts of the main loop:

- In the first phase of the main loop, we parallelize Line 8 of Algorithm 1 by evenly distributing the elements from $L(x, j)$ between the t threads of the same block. These threads can now read *adjacent* elements from $L(x, j)$, leading to a coalesced (and thus faster) memory access pattern.
- There are many ways to parallelize the second phase of the main loop, in which we need to construct a reduced candidate set C' of $c = 100$ barcodes with smallest pseudo-distances. For the moment, we stay with the sequential insertion-sort-like construction scheme as described above, only using one thread of each block. We will discuss techniques for parallelization in a few moments.
- In the third phase of the main loop, we parallelized the computation of the SL distances by distributing the candidate barcodes from C' evenly between the threads such that each thread computes about c/t SL distances. As all threads of the same block perform *exactly* the same computations on different data, this is ideal for SIMD processing units like GPUs.

Implementing this step actually slows down the running time by a factor of 0.73 (relatively to the previous step) but is a necessary prerequisite to exploit the shared memory in the following steps.

Step 3: Exploiting the shared memory. The original version of Quik does not make use of the shared GPU memory, which is known to be several times faster than the global memory, but is very limited in size (typically only about 100 KB per shared multiprocessor). As discussed above, the hindering problem in storing the pseudo-distance array P in the shared memory is the large number n of barcodes in our application.

To overcome this problem, we partition the barcodes into chunks of $n' < n$ barcodes, and process these chunks sequentially. For each chunk, the threads associated to a read r compute the pseudo-distances between r and just the barcodes of the current chunk. Choosing n' as a small constant (we used $n' = 2048$), the pseudo-distance array P easily fits into the shared memory. There are several consequences to this modification:

- Accessing the pseudo-distance array P is now *much* faster.
- The precomputed lists $L(x, j)$ used to compute the pseudo-distances must now be restricted to the individual barcode chunks. Thus, in the preprocessing step, we now compute a whole collection of lists $L(x, j)$, one for each chunk.
- In contrast to the general description of the main loop above, we now cannot strictly separate the first two phases. Instead, these phases are interleaved: Each block of threads maintains its candidate list C' of at most c elements, which is updated while processing the barcode chunks.

Implementing this idea gives a speedup of 2.32 relatively to the previous step, now leading to an absolute speedup factor of $2.17 \times 0.73 \times 2.32 = 3.67$ in comparison to the original version.

Step 4: Parallel update of the candidate lists. In the second step of the main loop, we must identify the $c = 100$ barcodes with smallest pseudo-distance. As discussed above, an insertion sort like approach has been used to create a reduced candidate list C' of barcode indices, ordered by their pseudo-distance. So far, the creation of this list has been done sequentially by one thread of each CUDA block, while all other threads are stalled. However, we can easily use a parallel approach to insert a barcode b with pseudo-distance q into the candidate list C':

- First, all threads of the same block use (sequential) binary search to identify a position $p \in \{1, \ldots, c\}$ within C' to which the new element needs to be inserted to keep the barcode indices ordered by their pseudo-distance. This needs additional $\Theta(\log c)$ steps, which is a very small constant as $c \leq 100$.
- Second, all threads of the same block in parallel move the elements of the sub-list $C'[p, c]$ one position to the right, thereby making room for the new element (b, q) at position p. As threads of the same warp perform in lock-step, this needs only very few accesses to registers.
- Finally, one thread of each block inserts the new element (b, q) at position p of C'.

Implementing this idea gives a small speedup of 1.09 in comparison to the previous step and thus an absolute speedup of $2.17 \times 0.73 \times 2.32 \times 1.09 = 4.01$.

Step 5: Simultaneous access to multiple lists. As a drawback of breaking the barcodes into small chunks, the lists $L(x, j)$ of barcode indices associated to each chunk are often very short (only containing a handful of barcode indices). This severely limits the performance when traversing these lists in Line 8 of Algorithm 1.

To overcome this problem, the threads of each block now access *multiple* lists $L(x, j)$ at once. For this purpose, we concatenated the lists $L(x, j)$ in ascending order of j into one large list $L(x)$. When creating this list, we augment each element of $b \in L(x, j)$ by the associated starting position j. Thus, $L(x)$ is a long list of pairs (b, j) of barcodes b (more precisely, their indices) and starting position j such that for each element $(b, j) \in L(x)$ the k-mer x is a substring of the barcode b starting at position j. For each value of j, we additionally store the start and end position of the original list $L(x, j)$ within $L(x)$. Thus, we can efficiently identify the start and end positions of each subset $L(x, j)$ within $L(x)$.

Algorithm 2 The algorithm for the computation of the k-mer pseudo-distance between a fixed read r and all barcodes of a chunk as described in Step 5.

Input: Read r, barcode count n', barcode lists $L(x, j)$
Output: Array $P \in \mathbb{Z}^{n'}$ of pseudo-distances

$P \leftarrow 0^{n'}$
for each possible k-mer starting position $i \in \{0, \ldots, |r| - k\}$ in the read r **do**
  $x \leftarrow r[i..i + k - 1]$
  $j_{\text{start}} \leftarrow \max(0, i - d_{max})$
  $j_{\text{end}} \leftarrow \min(\ell - k, i + d_{max})$
  $L' \leftarrow$ sub list from $L(x)$ associated to $L(x, j_{\text{start}})$ to $L(x, j_{\text{end}})$
  for each $(b, j) \in L'$ **in parallel do**
    $P[b] \leftarrow P[b] + |i - j| - \ell$ `// Atomic Add!`
return P

Algorithm 2 formalizes and implements this idea, which leads to a speedup of 4.02 relative to the previous step, and to a total speedup of $2.17 \times 0.73 \times 2.32 \times 1.09 \times 4.02 = 16.10$.

Step 6: Use multiple GPUs. Finally, we make use of multiple GPUs by evenly distributing the reads among the GPUs. Although this is no trivial task from the perspective of software design, it is still algorithmically straightforward and gives a speedup that is almost linear to the number of GPUs. In our benchmark experiment with 4 GPUs, we found a speedup of about 3.5, leading to a total speedup of $2.17 \times 0.73 \times 2.32 \times 1.09 \times 4.02 \times 3.50 = 56.35$ to the original version of Quik.

Experimental Results

We repeated our benchmark experiment for other values of $k \in \{5, 6, 7\}$, and for the two-step approaches described by Uphoff et al. [22]. These approaches first use a 7-mer filter to assign reads to barcodes, and afterwards run a 4-mer (resp. 5-mer) filter on the subset of the still unassigned reads. Uphoff et al. showed that these approaches give similar accuracy as the pure 4-mer (resp. 5-mer) filters, while lowering the running time.

Table 1 shows the running time and resulting speedup factors for Quik's filtering variants after our optimization. We want to highlight some important observations:

- The 4-mer and 5-mer filter now have the lowest running times from all filtering variants. This stands in contrast to the observations of Uphoff et al., where it was shown that the running time of the original version of Quik declines with growing values of k.
- While profiting from the algorithmic optimization, all of the pure k-mer filter approaches are now faster than the two-step approaches. This is due to the additional overhead of identifying the unassigned reads and re-compiling them compactly. Again, this is in contrast to the observations from Uphoff et al.
- Unfortunately, the speedup induced by our optimization steps declines rapidly with growing values of k. However, that doesn't have to concern us, since the 4-mer filter is the one with the largest accuracy (which has been demonstrated by Uphoff et al., and which will also be shown later in Table 2). Thus, the 4-mer filter is our primary concern.

We conclude that the 4-mer filter is now the best filtering variant as it gives the highest accuracy while also being among the fastest.

Table 1 Total running time (in seconds) for our benchmark experiment, processing 10^6 reads. Speedup (relative to the original version of Quik as described by Uphoff et al.), and throughput (in reads per second) for the different filtering modes of Quik.

Filter	Runtime	Speedup	Throughput
4-mer	11.7	56.35	83,097
5-mer	10.8	20.95	92,712
6-mer	14.8	5.48	65,167
7-mer	18.8	2.29	53,204
7+4-mer	24.4	2.28	40,386
7+5-mer	25.5	3.72	41,326

4 Unfiltered Barcode Calling

Quik's key idea as presented by Uphoff et al. was to avoid expensive all-to-all SL computations between reads and barcodes by identifying a small candidate set C' of at most $c = 100$ barcodes, and to restrict the SL computations to this set. In doing so, the total number of SL computations is reduced from mn to cm and thus to $\Theta(m)$. However, if the true barcode is not among the c candidates, there is no chance to correctly assign the associated read.

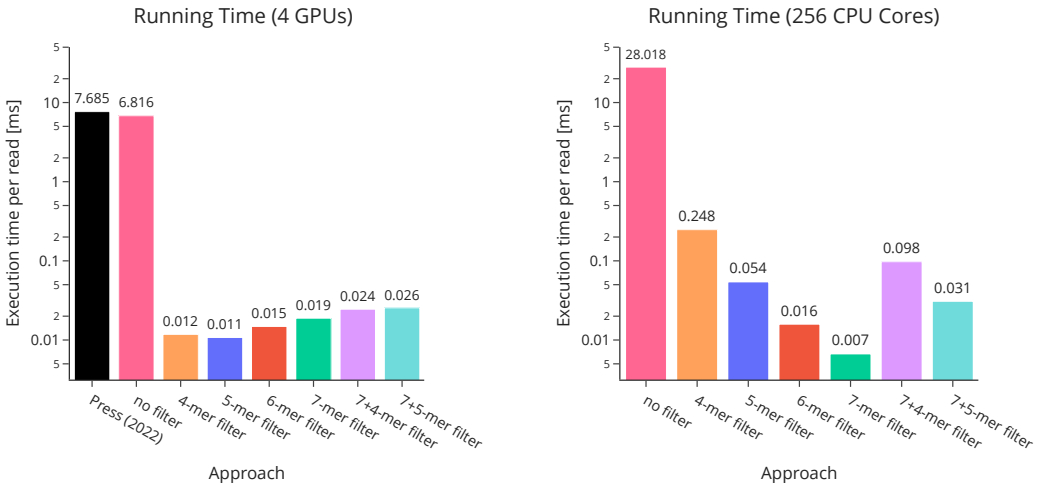
This motivates us to find out how much more precise our read assignment might be if we did not use a filter at all. In this section, we present the *unfiltered* calling approach, which does not make use of the pseudo-distance filter but uses an efficient way of all-to-all SL computations. We show how much this improves Quik's accuracy and study how much running time it costs.

Algorithmic Approach

We start by giving a few ideas on how to implement the all-to-all SL computations *efficiently*. Like the k-mer filter approaches of Quik, the unfiltered barcode calling approach is designed to run efficiently on GPUs.

- We assign to each read r a block of $t = 64$ CUDA threads, between which we distribute the barcodes evenly. Each thread then *sequentially* computes the SL distances between r and its subset of about n/t barcodes. Similar as discussed in the previous section, all threads run exactly the same sequence of operations on different data, which is perfect for GPUs.
- During the computations of the SL distances, each thread stores within its private memory the index of a barcode with minimum SL distance from its subset. After all threads have computed their minimum SL distance, we use an in-block reduction to determine the minimum SL distance within $\Theta(\log t)$ additional steps.
- We optimize the access to the global memory by using a carefully designed data structure for the barcodes that allows coalesced memory accesses when reading the barcodes.

We gain an algorithm that runs a total number mn of SL computations, each requiring $\Theta(\ell^2)$ sequential operations. In contrast, the filtered approaches require only cm SL computations in the third phase of the main loop. Thus, the total number of SL computations when using no filter is increased by a factor of n/c. In our benchmark experiment with $n = 1,000,000$ and $c = 100$, this gives us a rough estimation of a $10,000$-fold increase of the running time when omitting the filter.



(a) Running time using 4 NVIDIA A100 GPUs. **(b)** Running time using 4 AMD EPYC 7662 CPUs.

Figure 1 Normalized running time (ms per read) of Quik's filtering approaches in our benchmark experiment. The ordinates use a logarithmic scale. In Figure 1b, the bar for Press is missing as his program does not work without a GPU.

Running Time on GPUs

To assess the running time of our unfiltered barcode calling approach in practice, we repeated the benchmark experiment from the previous section. We compared the running time of the unfiltered approach with Quik's k-mer based filtering approaches, and with the filtering approach of Press (2022) [18]. Figure 1a shows the results.

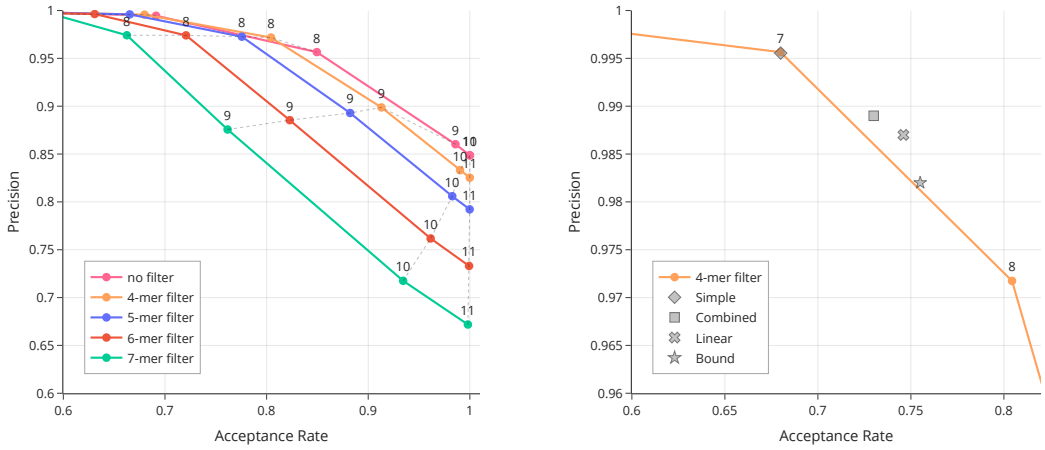
We want to highlight some essential observations.

- The unfiltered calling approach is in practice slower than the 4-mer filter by a factor of 568, which fortunately is far less than the slowdown factor of 10,000 discussed above. This is due to the fact that a significant proportion of running time in Quik's k-mer filtering approaches is used for the first and second phase of the main loop, which we do not have in the unfiltered approach.
- To our greatest surprise, our unfiltered approach is even slightly *faster* than the Random-Barcodes filtering approach suggested by Press (2022). From our perspective, this is due to Press' non-optimal parallelization strategy, which processes the reads *sequentially* and only parallelizes the search for their most similar barcodes. In contrast, Quik processes the *reads* in parallel.

Our experiment demonstrates that a careful GPU implementation of all-to-all SL computations can be a feasible option in practical applications, even with millions of barcodes and reads.

Running Time on Multi-Core CPUs

For systems without a GPU, Quik offers a fallback mode that uses OpenMP to enable classical CPU parallelism. For a better interpretation of the GPU running times discussed so far, we repeated the benchmark experiment on our multi-core CPU using 256 OpenMP threads. The results are shown in Figure 1b.



(a) Comparison of the filter strategies for different rejection thresholds D.

(b) Zoomed view: 4-mer-filter with optimized rejection functions from Section 5.

Figure 2 Precision vs acceptance rate for different values of the threshold distance D on our benchmark experiment. For each filtering approach and each value of $D \in [7, 11]$, we gain one point. The points associated to the same value of D are connected via a dashed line.

- Concerning the k-mer filtering approaches, we confirm the findings from Uphoff et al.: The running time of the pure k-mer filters steadily decreases with growing k, and the two-step filter approaches are still faster than the pure 4-mer and 5-mer filters.
- Using GPUs, the 4-mer filter is about 20 times faster than its CPU equivalent. This speedup declines with growing values of k. For $k = 7$, it would in fact be better to use the CPU implementation. This is consistent with our previous findings, as we saw that the speedup effect of the algorithm optimization declines with k.
- The unfiltered calling approach is about 4.1 times slower on the CPU system.

Our observations indicate that the GPU implementation of Quik is in most situations faster and thus more relevant in practical applications. However, in applications where the limited accuracy of the 7-mer filter is sufficient, it might in fact be faster to run Quik on classical multi-core CPUs.

Accuracy

To analyze how much the accuracy of the barcode calling profits from using all-to-all distance computations, we repeated the synthetic experiment from Uphoff et al. In this experiment, the authors used different values of D to reject reads if their SL distance is larger than D. Varying the rejection threshold D allows the user to control the number of accepted reads and, consequently, the accuracy of the resulting barcode assignment. Generally, the larger the chosen rejection threshold, the more reads will be accepted and the larger will be the number of accepted reads assigned to a wrong barcode.

To quantify the accuracy of Quik's calling approaches, we calculated two metrics: the *acceptance rate*, i.e. the number of accepted reads divided by the total number of reads, and the *precision*, i.e. the number of correctly accepted reads divided by the total number of accepted reads. Note that the precision can only be calculated in synthetic experiments with a known ground truth, whereas the acceptance rate can also be calculated in real-life

Table 2 Acceptance rate and precision (in percent) for each rejection threshold $D \in [7, 11]$.

	Acceptance rate					Precision				
Approach	7	8	9	10	11	7	8	9	10	11
no filter	69.1	84.9	98.6	99.9	100	99.5	95.7	86.0	84.9	84.9
4-mer filter	68.0	80.4	91.3	99.0	99.9	99.6	97.2	89.9	83.3	82.5
5-mer filter	66.5	77.6	88.2	98.3	99.9	99.6	97.3	89.3	80.6	79.2
6-mer filter	63.1	72.1	82.3	96.1	99.9	99.6	97.4	88.5	76.2	73.3
7-mer filter	58.8	66.2	76.2	93.4	99.8	99.6	97.4	87.6	71.8	67.2

applications. These metrics were calculated for all values of D between 7 and 11, and for each filtering approach. The results are shown numerically in Table 2 and graphically in Figure 2a.

- We observe that the unfiltered calling approach reaches a significantly higher acceptance rate than the filtering approaches do for the same level of the rejection threshold. For example, when using a rejection threshold of $D = 8$, the unfiltered approach achieves an acceptance rate of 84.9 %, while the 4-mer filtering approach only reaches 80.4 %.
- However, the precision is slightly lower than that of the k-mer filtering approaches. For our example of $D = 8$, the no-filter approach achieves a precision of 95.7 %, while the 4-mer filter has 97.2 %.

From our experiments, we conclude that the unfiltered calling approach may be useful in situations where a large acceptance rate is essential and the running time not a limiting problem. However, when aiming for a high precision, a k-mer filtering approach and a small value of D is the better choice.

5 Read Rejection

In the previous section, the accuracy of Quik has been measured by defining a rejection threshold D and evaluating the acceptance rate and the precision dependent on D in a synthetic experiment. This nicely shows the trade-off between these measures (see Figure 2a). In practice, however, we lack a ground truth, so the precision is unknown. Therefore, it is unclear how to select the appropriate rejection criteria to achieve the desired level of precision. So far, Quik has left the choice of rejection completely to the user. While this gives users ultimate control, it makes Quik less user-friendly. Our aim is to identify a way to determine an appropriate rejection function suitable to be included as the default setting in Quik.

Other tools like Columba [19], Flexiplex [7] and RandomBarcodes [18] also use a rejection threshold D to the best barcode candidate. Columba leaves the choice of threshold completely to the user, while Flexiplex [7] implements a default value that is tailored to the main intended use case of demultiplexing barcodes for Oxford Nanopore. Press [18] derives appropriate thresholds by estimating the distribution of Levenshtein distances between two random strings and statistically computing the minimum distance between a random string and a set of random barcodes, depending on the set's size. This estimation must be repeated for different barcode lengths, alphabet sizes and distance measures. Nevertheless, this aids the user in finding an appropriate threshold for the use cases considered. However, this threshold may be insufficient in certain cases: Consider a read with the same distance $d \leq D$ to two barcode candidates A and B. This read will be assigned to either, even though there is no way to determine the correct assignment. This naturally leads to a second rejection

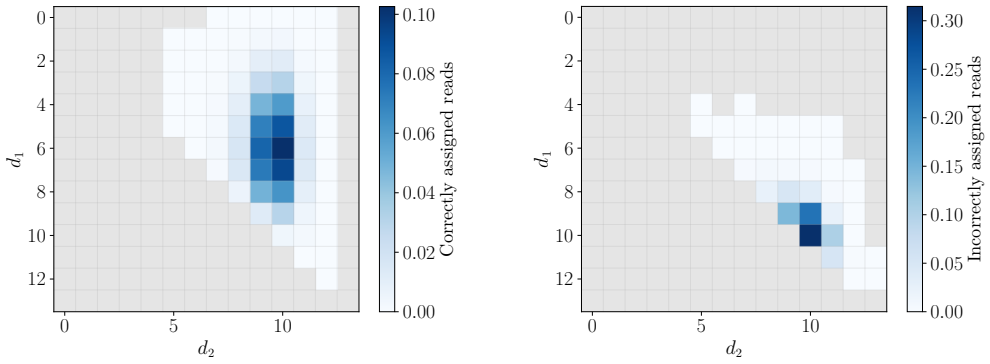


Figure 3 Read frequency of each pair of distances (d_1, d_2) to the best and second-best barcode candidates of correctly assigned reads (left) and incorrectly assigned reads (right).

strategy: Let d_1 be the distance to the best and d_2 the distance to the second best barcode candidate. Reject if d_1 is larger than D or if the distance difference $d_2 - d_1$ is less than a second threshold T. Similar ideas are implemented in Dorado [16], Flexiplex [7], and Barbell [3].

To visualize the motivation for rejection functions, we consider the frequency of simulated reads for each pair (d_1, d_2) after barcode calling. We split the read sets into correctly and incorrectly assigned reads based on the ground truth (Figure 3). The objective is to partition both matrices into accepted and rejected regions in such a way that as many correctly assigned reads as possible are accepted and as many incorrectly assigned reads as possible are rejected. This motivates us to consider linear rejection functions. A read should be rejected if $d_1 + \alpha \cdot d_2 > \beta$ holds for suitable values of the parameters α and β. See Table 3 for an overview of the rejection functions. The following subsections formalize these ideas, propose approaches for determining optimized parameters and then evaluate and compare the different rejection methods.

5.1 Rejection Optimization

First, we need to define the objective of the rejection optimization. Recall Figure 3. Our aim is to divide both matrices into accepted and rejected parts such that, in the accepted region, the number of correctly assigned reads is maximized, while the number of incorrectly assigned reads is minimized. In real experiments, however, we do not know the ground truth and thus cannot use any information on the true assignments to split the read set in the same way. However, we can *simulate* the matrix of incorrectly assigned reads by calling a set of randomly generated strings, as incorrect assignments are basically random. These randomly generated strings do not correspond to any barcode and thus should be rejected. We optimize the parameters of our rejection functions by maximizing the number of accepted reads under the condition that we achieve a pre-defined minimum rate r_S of rejected random strings. By increasing r_S, we expect that the rate of incorrectly accepted reads decreases. In our experiments, we set $r_S = 95\%$.

We propose the following approach for the parameter optimization: Given a barcode set and a read set, use Quik to call the reads and aggregate the results to a matrix R in which each entry R_{d_1,d_2} corresponds to the number of reads with distance d_1 to the first and d_2 to the second best barcode candidate. Next, randomly generate a set Z of strings of the same length as the reads and use Quik to call these strings. Aggregate the results to a matrix S with entries S_{d_1,d_2} which denote the number of strings with distance d_1 to the first and d_2 to

Table 3 Rejection functions.

Name	Rejection Criteria	Optimization Approach
Simple	$d_1 > D$	Enumeration for $D \in \{0, ..., \ell\}$
Combined	$d_1 > D \vee d_2 - d_1 < T$	Enumeration for $D \in \{0, ..., \ell\}, T \in \{0, ..., \ell\}$
Linear	$d_1 + \alpha \cdot d_2 > \beta$	Integer Linear Program (Gurobi)
Upper Bound	$x_{d_1,d_2} = 0$	Integer Linear Program (Gurobi)

the second best barcode candidate. Using R, we can compute the number of accepted reads for any choice of function parameters. Likewise, we use S to compute the rate of rejected random strings for the same choice of parameters.

To determine an optimal parameter choice for each rejection function, we follow different strategies. The simple and the combined rejection thresholds can be optimized by complete enumeration as the range of possible values of the parameters D and T is small. The linear function is modeled as an integer linear program (ILP) as follows. Let L be the maximum distance of a read or random string to its second closest barcode.

$$\max \sum_{d_1} \sum_{d_2} R_{d_1,d_2} \cdot x_{d_1,d_2} \tag{1}$$

$$\text{s.t.} \sum_{d_1} \sum_{d_2} S_{d_1,d_2} \cdot (1 - x_{d_1,d_2})/|Z| \geq r_S \tag{2}$$

$$x_{d_1,d_2} = 1 \Leftrightarrow d_1 + \alpha \cdot d_2 \leq \beta \qquad \forall d_1, d_2 \in \{0, ..., L\} \tag{3}$$

$$x_{d_1,d_2} \in \{0, 1\} \qquad \forall d_1, d_2 \in \{0, ..., L\} \tag{4}$$

$$\alpha, \beta \in \mathbb{Q} \tag{5}$$

The variables x_{d_1,d_2} indicate whether to accept an assignment with the corresponding distances ($x_{d_1,d_2} = 1$) or to reject it ($x_{d_1,d_2} = 0$). Objective (1) maximizes the number of accepted reads, while Constraint (2) ensures a minimum rate of rejected random strings. Constraints (3) force the indicator variables to correspond to the linear function as a border between accepted and rejected region. The implications are implemented using the Big M method. See Appendix A.1 for the full ILP formulation.

Additionally, we define an upper bound ILP that can freely choose which pairs of distances (d_1, d_2) to reject. It is implemented as a similar model as for the linear rejection function, but omitting Constraints (3) and Variables (5). The solution of the bound ILP could also serve as a special type of rejection function. However, it depends more heavily on the simulated random strings and is therefore more susceptible to statistical error. For example, it may accept reads with very large d_1 and d_2 because the probability of these values occurring for random strings is low. We expect the other rejection functions to be more robust.

5.2 Experimental Setup

As a proof of concept, we performed the following experiment. We re-used the set of $n = 10^6$ random barcodes of length $\ell = 34$. We applied the error model from Uphoff et al. with total error rates of 15%, 20%, 25%, and 30% to generate read sets of $m = 5 \cdot 10^6$ reads of length $\ell = 34$ each. We generated a set of 10^6 random strings of length ℓ for each read set. Each set was called using 4-mer-filtering. The rejection function optimizations were performed using either enumeration or the Python API of Gurobi 13.0.1. In all cases, the running time for the optimization required just fractions of a second and is therefore negligible. After determining optimal rejection functions, we evaluated acceptance rate and precision.

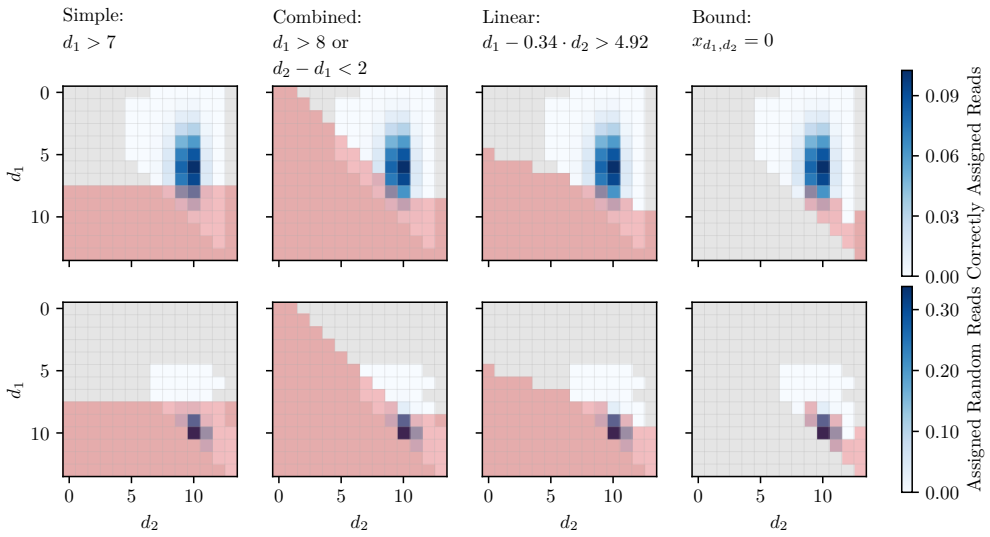


Figure 4 Overview over the optimal rejection functions for the correctly assigned reads of the set with 20% error rate (top) and the random strings (bottom). The rejected entries are shown in red.

Table 4 Acceptance rate and precision for each bound type and each read set (given as the error rate in percent) for $r_S = 95\%$.

	Acceptance rate				Precision			
Set	Simple	Combined	Linear	Bound	Simple	Combined	Linear	Bound
15	89.2	91.0	92.0	92.4	99.9	99.7	99.7	99.5
20	68.0	73.0	74.6	75.5	99.6	98.9	98.7	98.2
25	42.0	49.4	50.9	52.1	98.8	96.4	96.2	95.1
30	21.1	28.6	29.6	30.7	97.0	90.1	89.9	88.2

5.3 Results

Figure 4 shows the optimal rejection functions for each rejection type for the read set with 20% error rate. See Appendix A.2 for the corresponding function parameters. The simple rejection threshold is quite strict, rejecting most of the random strings, but also removing a significant proportion of the correctly assigned strings yielding an acceptance rate of 68%. The combined approach increases the acceptance rate to 73% by setting a higher threshold for d_1 and additionally restricting the distance difference. This only slightly decreases the precision. The linear rejection function further improves the acceptance rate to 74.6%. It comes close to the acceptance rate of the upper bound ILP, which reaches 75.5%. The resulting pairs of precision and acceptance rate are shown in Figure 2b. The newly proposed rejection functions allow a more fine-grained trade-off. Due to the optimization, we find several new non-dominated pairs of precision and acceptance rate.

Table 4 shows the results for all read sets. The acceptance rate increases from simple over combined to linear, with the linear rejection function being very close to the upper bound. This suggests that more complex rejection functions are more advantageous than simple ones. Conversely, the precision is highest for the simple rejection threshold. The upper bound has the lowest precision, which may be undesirable in some cases. We also see that the results

depend on the error rates used for read generation: For a relatively low rate of 15%, the acceptance rate is at about 90% with a precision of almost 100%, regardless of the rejection type. For the highest tested error rates of 30%, the acceptance rate is only between 20% and 30% with a precision of 88% to 97%. Naturally, it is hard to find the corresponding barcode for such high error rates. In this case, we would expect around 11 errors per read of length 34. Therefore, only a few reads with far fewer than the expected number of errors would have a chance of being assigned correctly. We cannot expect high acceptance rate and precision in these situations. Consequently, 21% acceptance rate at 97% precision with a stricter setting, or 30% acceptance rate at 90% precision seem like reasonable trade-offs. We summarize some key observations:

- The combined and linear rejection functions are superior to the simple one in terms of acceptance rate, and still have an acceptable precision for most use cases.
- The combined rejection thresholds are easy to compute and nearly as good as the linear one, and thus suitable to be included in Quik.
- As the simple one tends to be more strict, the precision is higher.
- For higher error rates, both acceptance rate and precision naturally decrease.
- For very high error rates (30%), a choice must be made whether to prioritize acceptance rate or precision.

This three-step approach of random string generation, calling and optimization is highly flexible and can be applied to any read and barcode set to automatically obtain optimized rejection criteria tailored to the specific data *without* any knowledge of the ground truth. The combined rejection function is particularly easy to compute and close to the upper bound. We therefore consider this to be a valuable addition to Quik.

6 Summary

In this paper, we presented several ideas that proved useful to improve the barcode calling software Quik. First, we discussed algorithmic optimization techniques that accelerated the program by a factor of up to 56, while maintaining a high accuracy. Subsequently, we demonstrated that an unfiltered calling approach may give a slightly larger acceptance rate at the cost of a smaller precision and vastly increased running time. Our experiments demonstrate that the k-mer filters are to be favored over the unfiltered approach in almost all practical situations. Finally we presented a mechanism to automatically determine rejection criteria optimized to specific barcode and read sets. This greatly increases Quik's usability, as users no longer need to think about an appropriate rejection threshold.

Our improvements are not only instructive from an algorithmic standpoint, but also set new standards for high-precision barcode calling at an unprecedented speed of over 90,000 reads per second. Our tool is freely available at `https://github.com/uni-halle/quik`.

References

1 Daniel Ashlock and Sheridan K. Houghten. DNA error correcting codes: No crossover. In *2009 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, pages 38–45. IEEE, March 2009. doi:10.1109/CIBCB.2009.4925705.

2 James L Banal, Tyson R. Shepherd, Joseph Berleant, Hellen Huang, Miguel Reyes, Cheri M. Ackermann, Paul C. Blainey, and Mark Bathe. Random access DNA memory using boolean search in an archival file storage system. *Nature Materials*, 21:1272–1280, 2021. doi:10.1038/s41563-021-01021-3.

3 Rick Beeloo, Ragnar Groot Koerkamp, Xiu Jia, Marian J. Broekhuizen-Stins, Lieke van IJken, Els M. Broens, Aldert Zomer, and Bas E. Dutilh. Barbell resolves demultiplexing and trimming issues in nanopore data. *bioRxiv*, 2025. doi:10.1101/2025.10.22.683865.

4 Jürgen Behr, Timm Michel, Maya Giridhar, Santra Santhosh, Arya Das, Hamed Sabzalipoor, Tadija Kekić, Etkin Parlar, Igor Ilić, Gisela Ibáñez-Redín, Erika Schaudy, Jory Lietard, Thomas Schletterer, Max Funck, and Mark Somoza. An open-source advanced maskless synthesizer for light-directed chemical synthesis of large nucleic acid libraries and microarrays. *ChemRxiv*, 2024(0222), 2024. doi:10.26434/chemrxiv-2024-j4c90.

5 Tilo Buschmann and Leonid V. Bystrykh. Levenshtein error-correcting barcodes for multiplexed DNA sequencing. *BMC Bioinformatics*, 14(1):272, September 2013. doi:10.1186/1471-2105-14-272.

6 Leonid V. Bystrykh. Generalized DNA Barcode Design Based on Hamming Codes. *PLOS ONE*, 7(5):e36852, May 2012. doi:10.1371/journal.pone.0036852.

7 Oliver Cheng, Min Hao Ling, Changqing Wang, Shuyi Wu, Matthew E Ritchie, Jonathan Göke, Noorul Amin, and Nadia M Davidson. Flexiplex: a versatile demultiplexer and search tool for omics data. *Bioinformatics*, 40(3):btae102, March 2024. doi:10.1093/bioinformatics/btae102.

8 Brant C. Faircloth and Travis C. Glenn. Not All Sequence Tags Are Created Equal: Designing and Validating Sequence Identification Tags Robust to Indels. *PLOS ONE*, 7(8):e42543, 2012. doi:10.1371/journal.pone.0042543.

9 John A. Hawkins, Stephen K. Jones, Ilya J. Finkelstein, and William H. Press. Indel-correcting DNA barcodes for high-throughput sequencing. *Proceedings of the National Academy of Sciences*, 115(27):E6217–E6226, 2018. doi:10.1073/pnas.1802640115.

10 Allon M. Klein, Linas Mazutis, Ilke Akartuna, Naren Tallapragada, Adrian Veres, Victor Li, Leonid Peshkin, Weitz David A, and Marc W. Kirschner. Droplet barcoding for single-cell transcriptomics applied to embryonic stem cells. *Cell*, 161(5):1187–1201, 2015. doi:10.1016/j.cell.2015.04.044.

11 Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

12 Jory Lietard, Adrien Leger, Yaniv Erlich, Norah Sadowski, Winston Timp, and Mark M Somoza. Chemical and photochemical error rates in light-directed synthesis of complex DNA libraries. *Nucleic Acids Research*, 49(12):6687–6701, 2021. doi:10.1093/nar/gkab505.

13 Yang Liu, Mingyu Yang, Yanxiang Deng, Graham Su, Archibald Enninful, Cindy C. Guo, Toma Tebaldi, Di Zhang, Dongjoo Kim, Zhiliang Bai, Eileen Norris, Alisia Pan, Jiatong Li, Yang Xiao, Stephanie Halene, and Rong Fan. High-spatial-resolution multi-omics sequencing via deterministic barcoding in tissue. *Cell*, 183(6):1665–1681.e18, 2020. doi:10.1016/j.cell.2020.10.026.

14 Eli Lyons, Paul Sheridan, Georg Tremmel, Satoru Miyano, and Sumio Sugano. Large-scale dna barcode library generation for biomolecule identification in high-throughput screens. *Scientific reports*, 7(1):1–7, 2017.

15 Timm Michel, Jürgen Behr, Hamed Sabzalipoor, Gisela Ibáñez-Redín, Jory Lietard, Thomas Schletterer, Max Funck, and Mark M. Somoza. Color-corrected and high-contrast catadioptric relay for maskless high-resolution nucleic acid photolithography. *Opt. Express*, 33(8):17068–17084, April 2025. doi:10.1364/OE.557937.

16 Oxford Nanopore Technologies plc. Dorado v1.4.0, 2026. URL: https://github.com/nanoporetech/dorado/.

17 Franco Poma-Soto, Hanne Van Droogenbroeck, Brecht Soulliaert, Maya Giridhar, Jürgen Behr, Hamed Sabzalipoor, Mark Somoza, Pieter Mestdagh, and Jo Vandesompele. Benchmarking DNA barcode decoding strategies under high error rates. *BMC Bioinformatics*, June 2026. doi:10.1186/s12859-026-06540-x.

18 William H Press. Fast trimer statistics facilitate accurate decoding of large random DNA barcode sets even at large sequencing error rates. *PNAS Nexus*, 1(5):pgac252, 2022. doi:10.1093/pnasnexus/pgac252.

19 Luca Renders, Lore Depuydt, Travis Gagie, and Jan Fostier. Columba: fast approximate pattern matching with optimized search schemes. *Bioinformatics*, 41(12):btaf652, December 2025. doi:10.1093/bioinformatics/btaf652.

20 Schüler Steffen, Riko Uphoff, and Antonia Schmidt. Quik. Software, version 2.0., swhId: swh:1:dir:948d8ae0e0b66260e91d201ea0b5f6f732aa4eff (visited on 2026-08-12). URL: https://github.com/uni-halle/quik, doi:10.4230/artifacts.27619.

21 Akshay Tambe and Lior Pachter. Barcode identification for single cell genomics. *BMC Bioinformatics*, 20(1):32, January 2019. doi:10.1186/s12859-019-2612-0.

22 Riko Corwin Uphoff, Steffen Schüler, Ivo Grosse, and Matthias Müller-Hannemann. Fast barcode calling based on k-mer distances. *PNAS Nexus*, 5(2):pgag001, January 2026. doi:10.1093/pnasnexus/pgag001.

23 Johannes Wirth, Nina Huber, Kelvin Yin, Sophie Brood, Simon Chang, Celia P. Martinez-Jiminez, and Matthias Meier. Spatial transcriptomics using multiplexed deterministic barcoding in tissue. *Nature Communications*, 14(1523), 2023. doi:10.1038/s41467-023-37111-w.

24 Eduard Zorita, Pol Cuscó, and Guillaume J. Filion. Starcode: sequence clustering based on all-pairs search. *Bioinformatics*, 31(12):1913–1919, June 2015. doi:10.1093/bioinformatics/btv053.

A Read Rejection

A.1 Rejection Optimization

In Section 5, we briefly sketched an ILP for the optimization of the linear rejection function. Using this formulation as is leads to numerical problems when the parameters α and β are represented as floating point numbers. Representing $\alpha = \frac{\alpha_n}{\alpha_d}$ and $\beta = \frac{\beta_n}{\beta_d}$ as rational numbers with $\alpha_n, \alpha_d, \beta_n, \beta_d \in \mathbb{Z}$ only partially solves the problem, since with missing bounds on these numbers it remains unclear how to choose the parameter M required to model the implications in Constraints (3) with the Big M method. Therefore, we further restricted these numbers to reach an accuracy of 1% which we consider as sufficient for our purposes. More precisely, we fixed the denominators α_d and β_d to 100. The full ILP is given as:

$$\max \sum_{d_1} \sum_{d_2} R_{d_1,d_2} \cdot x_{d_1,d_2} \tag{6}$$

$$\text{s.t.} \sum_{d_1} \sum_{d_2} S_{d_1,d_2} \cdot (1 - x_{d_1,d_2})/|Z| \geq r_S \tag{7}$$

$$d_1 \cdot \alpha_d \cdot \beta_d + \alpha_n \cdot \beta_d \cdot d_2 \leq \beta_n \cdot \alpha_d + M \cdot (1 - x_{d_1,d_2}) \qquad \forall d_1, d_2 \in \{0, ..., L\} \tag{8}$$

$$d_1 \cdot \alpha_d \cdot \beta_d + \alpha_n \cdot \beta_d \cdot d_2 \geq \beta_n \cdot \alpha_d - M \cdot x_{d_1,d_2} + 1 \qquad \forall d_1, d_2 \in \{0, ..., L\} \tag{9}$$

$$\alpha_d = \beta_d = 100 \tag{10}$$

$$-\alpha_d \leq \alpha_n \leq \alpha_d \tag{11}$$

$$-\beta_d \cdot L \leq \beta_n \leq \beta_d \cdot L \tag{12}$$

$$x_{d_1,d_2} \in \{0, 1\} \qquad \forall d_1, d_2 \in \{0, ..., L\} \tag{13}$$

$$\alpha_n, \beta_n \in \mathbb{Z} \tag{14}$$

$$M = L \cdot \alpha_d \cdot \beta_d + L \cdot \beta_d \cdot \alpha_d + L \cdot \beta_d \cdot \alpha_d \tag{15}$$

Inequalities (8) and (9) correspond to Constraints (3) in Section 5. Constraints (10)-(12) are necessary to restrict the variable values for choosing an appropriate value for M.

A.2 Further Results

Table 5 shows the parameters for the optimized rejection functions from Section 5.3. Please note that these parameters highly depend on the settings, particularly the barcode length, set size, alphabet, and distance measure (SL vs. Levenshtein distance). This makes it impossible to derive any universally applicable parameters. The generation of random strings takes into account the distribution of bases and read lengths. Both the read set and the random string set are called with the same settings (the same barcode set and the same distance measure).

To illustrate the dependence of the parameters on the setting, we evaluated a slightly varied experiment: The same barcode set of 10^6 barcodes and the same error model were used. However, the reads were *not* trimmed to constant length, resulting in reads of *varying* lengths, depending on the number of insertions and deletions in each one. The reads were called with Quik using the Levenshtein distance instead of the SL distance. Table 6 shows the results. The optimal parameters clearly differ between Tables 5 and 6. Thus, the optimized parameters should always be tailored to the specific data set.

Table 5 Optimized parameters for each bound type and each read set (given as the error rate in percent) for $r_S = 95\%$.

	Simple	Combined		Linear	
Set	D	D	T	α	β
15	7	8	2	−0.34	4.92
20	7	8	2	−0.34	4.92
25	7	8	2	−0.11	6.99
30	7	8	2	−0.34	4.92

Table 6 Optimized parameters for each bound type and each read set (given as the error rate in percent) for $r_S = 95\%$ with Levenshtein distance.

	Simple	Combined		Linear	
Set	D	D	T	α	β
15	8	9	2	−0.59	2.87
20	8	9	2	−0.59	2.87
25	8	9	2	−0.59	2.91
30	8	9	1	−0.33	5.70