Fast Gapped k-mer Counting with Subdivided Multi-Way Bucketed Cuckoo Hash Tables

Jens Zentgraf
Department of Computer Science, Saarland University, Saarbrücken, Germany
Center for Bioinformatics, Saarland University, Saarbrücken, Germany
Saarbrücken Graduate School of Computer Science, Saarland University, Saarbrücken, Germany

Sven Rahmann
Department of Computer Science, Saarland University, Saarbrücken, Germany
Center for Bioinformatics, Saarland University, Saarbrücken, Germany

Abstract

Motivation. In biological sequence analysis, alignment-free (also known as k-mer-based) methods are increasingly replacing mapping- and alignment-based methods for various applications. A basic step of such methods consists of building a table of all k-mers of a given set of sequences (a reference genome or a dataset of sequenced reads) and their counts. Over the past years, efficient methods and tools for k-mer counting have been developed. In a different line of work, the use of gapped k-mers has been shown to offer advantages over the use of the standard contiguous k-mers. However, no tool seems to be available that is able to count gapped k-mers with the same efficiency as contiguous k-mers. One reason is that the most efficient k-mer counters use *minimizers* (of a length $m < k$) to group k-mers into buckets, such that many consecutive k-mers are classified into the same bucket. This approach leads to cache-friendly (and hence extremely fast) algorithms, but the approach does not transfer easily to gapped k-mers. Consequently, the existing efficient k-mer counters cannot be trivially modified to count gapped k-mers with the same efficiency.

Results. We present a different approach that is equally applicable to contiguous k-mers and gapped k-mers. We use multi-way bucketed Cuckoo hash tables to efficiently store (gapped) k-mers and their counts. We also describe a method to parallelize counting over multiple threads without using locks: We subdivide the hash table into independent subtables, and use a producer-consumer model, such that each thread serves one subtable. This requires designing Cuckoo hash functions with the property that all alternative locations for each k-mer are located in the same subtable. Compared to some of the fastest contiguous k-mer counters, our approach is of comparable speed, or even faster, on large datasets, and it is the only one that supports gapped k-mers.

2012 ACM Subject Classification Applied computing → Molecular sequence analysis; Applied computing → Bioinformatics; Theory of computation → Bloom filters and hashing; Theory of computation → Data structures design and analysis

Keywords and phrases gapped k-mer, k-mer, counting, Cuckoo hashing, parallelization

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.12

Supplementary Material Our software `hackgap` (**ha**sh-based **c**ounting of ***k***-mers with **gap**s) is available on gitlab under the MIT license.
Software (Source Code): `https://gitlab.com/rahmannlab/hackgap`
archived at `swh:1:dir:2a19b268091ab679be5cc1424153dbdcfdb52433`

Acknowledgements We thank Rajesh Moturu for running preliminary experiments with gapped k-mers.

22nd International Workshop on Algorithms in Bioinformatics (WABI 2022).
Editors: Christina Boucher and Sven Rahmann; Article No. 12; pp. 12:1–12:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Counting k-mers (short pieces of DNA sequences of length k) in reference genomes (FASTA files) or sequenced samples (FASTQ files) is a basic step of modern DNA sequence analysis, and many applications depend on efficient k-mer counting tools. For example, genome assembly is often done using the so-called De Bruijn graph (of which there are many variations), which represents all sufficiently frequent k-mers in the sample, recently even for several values of k [1]. RNA transcript quantification can be done by examining the k-mers in each read and assigning a set of possible genes or transcripts to each read based on k-mer content [2]. The abundance of each species in a metagenomic sample can also be estimated by counting k-mers in the reads and assigning them to (groups of) species [10, 16, 21]. Because k-mer counting is such an ubiquitous task, many methods tools for precisely this purpose have been developed, published and reviewed [5, 8, 6, 12].

The purpose of a k-mer counter is to output a mapping $\mathit{count} : \Sigma^k \supset X \to \{1, \ldots, C\}$, where Σ is the (DNA) alphabet, X is the set of k-mers that are present in the sample under consideration, and C is the maximum implemented count, such that all values $\geq C$ will be reported as C. Typically, $C = 2^v - 1$ if v bits are used to represent counters. The final mapping can be stored in different ways, and each tool uses its own disk-based representation.

A recent review [12], and also our own experience, indicate that KMC3 [8, 5] and Gerbil [6] are among the fastest tools in practice. However, these tools also have their limitations: They are specialized towards k-mer counting and do not offer much other functionality (although they have accompanying complementary tools that allow to perform multiset operations on several output files with counts). They only support contiguous k-mers and do not extend towards gapped k-mers (Section 2.1). This limitation is not easily fixed, because their data structures (and their efficiency) depend on concepts related to *minimizers* and *super-k-mers*, sequences of consecutive k-mers that share the same minimizer (Section 2.2). On the other hand, it has been shown several times that gapped k-mers offer a number of advantages over contiguous k-mers in many applications, in particular as seeds for alignments [17], but also for metagenomic classification [3].

In the above-mentioned application examples (assembly, transcript quantification, metagenomics), having *unique* k-mers (that appear at only one location in the genome, in only one gene or transcript, in only one species' genome) will facilitate the analysis. If there are not enough unique k-mers, the k-mer counts will have to be disentangled to provide accurate estimates in the examples above. On the other hand, if most k-mers are unique, such a step may not be necessary, and one may even be able to ignore non-unique k-mers entirely. Consequently, we focus on $k \in \{21, \ldots, 31\}$, because for these k, most of the k-mers are unique in a mammalian genome, i.e., they occur at only one position in the genome. Using gapped k-mers with specific masks may further increase the number of unique (gapped) k-mers in comparison to contiguous k-mers (see Section 4.4).

Because of the wide applicability of gapped k-mers, it would be helpful to have a method and a tool that allows counting them with comparable efficiency as for contiguous k-mers. The challenge is that the concepts of minimizers and super-k-mers do not directly extend to gapped k-mers. Therefore, we present an alternative methodology, using multi-way bucketed Cuckoo hashing with quotienting. After introducing the necessary background and terminology (Section 2), we present our hashing strategy and its parallelization using subtables (Section 3). We evaluate the approach on several datasets, both reference genomes and large sequence-read datasets, and compare (for the contiguous case) with Gerbil and KMC3 (Section 4).

2 Background and Basic Definitions

We first introduce necessary definitions (k-mers, canonical codes of k-mers, gapped k-mers, masks, shapes and κ-mers) in Section 2.1. Then, we review efficient k-mer counting strategies based on minimizers and super-k-mers (Section 2.2) and explain why these do not easily extend to gapped k-mers. Finally, we introduce Cuckoo hashing (Section 2.3) and its practical variations, a basis for our method.

2.1 Terminology

By $[n]$, we denote the set of integers $\{0, \dots, n-1\}$. Indexing of strings starts at 0 (not at 1). We also write $u[0:k] := u_0 u_1 \dots u_{k-1}$ (slice notation *excluding* the last index).

▶ **Definition 1** (k-mer)**.** *A DNA sequence is a string of any finite length over the DNA alphabet $\Sigma := \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ of size 4. A k-mer (of a sequence s) is any substring of length k.*

There are 4^k different DNA k-mers. However, a DNA molecule u is equivalent to its reverse complement $rc(u)$, i.e., the reversed string with A replaced by T and C replaced by G (and vice versa). Here $rc : \Sigma^k \to \Sigma^k$ maps a k-mer u to its reverse complement. For example, for $k = 5$, the sequence AACTG is equivalent to CAGTT. Therefore, for odd k, there are only $4^k/2$ equivalence classes (DNA molecules). When k is even, a k-mer's reverse complement may be equal to the k-mer itself (e.g., for GATATC), and this happens for exactly $4^{k/2}$ k-mers, so there are $(4^k - 4^{k/2})/2 + 4^{k/2} = (4^k + 4^{k/2})/2$ different DNA molecules of length k. To avoid issues of self-reverse-complementarity, we work with odd values of k.

▶ **Definition 2** (Canonical integer encoding of a k-mer)**.** *Each DNA nucleotide may be encoded with two bits (typically, $\mathtt{A} \mapsto (00)_2 = 0$, $\mathtt{C} \mapsto (01)_2 = 1$, $\mathtt{G} \mapsto (10)_2 = 2$, $\mathtt{T} \mapsto (11)_2 = 3$). This allows encoding a k-mer u uniquely (for fixed k) as a base-4 number $c = enc(u)$ with $0 \le c < 4^k$. To ensure that a k-mer and its reverse complement (the same DNA molecule) are represented by the same integer, we represent both by the* larger *of the two encodings; this is called the* canonical code *$cc(u)$ of the k-mer u: $cc(u) := \max\{enc(u), enc(rc(u))\}$.*

Often, the *smaller* value of the two encodings is used as the canonical code, but our choice avoids confusion with minimizers; see below.

▶ **Example 3.** For $k = 5$, we have $enc(\mathtt{AAGCG}) = (00212)_4 = (00|00|10|01|10)_2 = 38$, and $enc(rc(\mathtt{AAGCG})) = enc(\mathtt{CGCTT}) = (12133)_4 = 415$, so $cc(\mathtt{AAGCG}) = cc(\mathtt{CGCTT}) = 415$.

We have defined k-mers as (contiguous) substrings of a string s. However, we can also "extract" k characters from a larger window of size w, according to a specific pattern, called a *mask*, of *shape* (w, k). The resulting k-tuple of characters is called a *gapped k-mer* (for the given mask), or also a *spaced seed*.

▶ **Definition 4.** *Given two integers $w \ge k \ge 2$, a mask of* shape *(w, k) is a string μ of length w over the alphabet $\{\mathtt{\#}, \mathtt{\_}\}$ that contains exactly k times the character # and $w-k$ times the character _. The positions marked # are called* significant*, and the positions marked _ are called* insignificant *or* gaps*. We call k the* weight *of the mask and w its* width*. The mask μ may equivalently be defined as a bit vector of length w with k ones. It can also be represented as the set κ of the significant positions: $\kappa = \{j : 0 \le j < w \text{ and } \mu_j = \mathtt{\#}\}$.*

The masks we consider are subject to two technical conditions:

Table 1 Different masks of shape $(w, k) = (31, 25)$ and their properties. Column "max d_{H}" refers to the maximum Hamming distance tolerated in a read of length 100 bp, such that there is guaranteed to be *at least one* κ-mer unaffected by the substitutions, even if their positions "conspire" to affect as many κ-mers as possible. Column "intact κ-mers" specifies *how many* κ-mers are guaranteed to be unaffected by substitutions in the worst case at the maximum possible Hamming distance. Column "covered bp" specifies *how many basepairs* intact κ-mers are guaranteed to cover.

ID	mask μ	max d_{H}	intact κ-mers	covered bp
k25 = m1	#########################	3	1	25
m2	####_####_###_###_###_####_####	4	2	32
m3	####_###_###_#####_###_###_####	4	2	32
m4	###_##_#####_#####_#####_##_###	4	3	34

1. We require that the first and last mask positions are significant: $\mu_0 = \mu_{w-1} =$ #; otherwise, we could just shorten the mask and obtain equivalent results, except for boundary effects.
2. We require that the mask is *symmetric*, so we do not need to distinguish (in terms of the mask) between the forward and reverse complementary DNA sequence.

For a given mask μ or equivalent set κ, we say that a string u of length k is a *μ-mer* or *κ-mer* of a sequence s, if $u = (s_{i+j})_{j \in \kappa}$ for some starting position i.

Clearly, a κ-mer of shape (w, k) with a symmetric mask has a canonical code just like a regular k-mer by only considering the significant k positions.

▶ **Example 5.** Consider $(w, k) = (7, 3)$ with $\mu =$ #__#__# or $\kappa = (0, 3, 6)$. Let $s :=$ TACAGATATA. We extract and encode the κ-mers as follows:

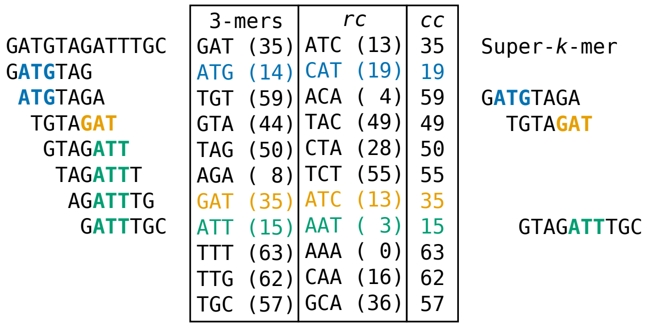
```
TACAGATATA
T__A__T      cc(TAT) = max(enc(TAT), enc(ATA)) = 51
 A__G__A     cc(AGA) = max(enc(AGA), enc(TCT)) = 55
  C__A__T    cc(CAT) = max(enc(CAT), enc(ATG)) = 19
   A__T__A   cc(ATA) = max(enc(ATA), enc(TAT)) = 51
```

Design of masks. For a symmetric mask with odd k, also w must be odd, and the middle position must be significant. For a given shape (w, k) with odd w and k, there are $\binom{(w-3)/2}{(k-3)/2}$ symmetric masks that satisfy our technical constraints (the first, middle and last position must be significant positions; we may freely distribute half of the remainder of the significant positions in the first half of the mask between first and middle position). For example, for $(w, k) = (31, 25)$, there are $\binom{14}{11} = \binom{14}{3} = 364$ different masks to consider.

There is a rich body of literature on the design of good or optimal masks with regard to a given objective function [18, 7, 11], with early works dating back to the first ISMB conference in 1993 [4]. An extensive bibliography on the topic is maintained by Laurent Noé[1].

Table 1 shows the contiguous 25-mer mask and three different $(31, 25)$-shaped masks and some of their properties. For example, if we distribute 4 substitutions (SNVs or sequencing errors) in a read of length 100 in a "conspiring" manner (as to modify as many κ-mers as possible), it is possible to distribute them in a way (equidistantly) to modify all of the contiguous 25-mers. Even at the lower Hamming distance of 3, only a single 25-mer is guaranteed to be unchanged. However, if we use mask m4, independently of the distribution

[1] https://sites.google.com/view/laurentnoe/spaced-seeds



	3-mers	*rc*	*cc*	
GATGTAGATTTGC	GAT (35)	ATC (13)	35	Super-*k*-mer
GATGTAG	ATG (14)	CAT (19)	19	
ATGTAGA	TGT (59)	ACA (4)	59	GATGTAGA
TGTAGAT	GTA (44)	TAC (49)	49	TGTAGAT
GTAGATT	TAG (50)	CTA (28)	50	
TAGATTT	AGA (8)	TCT (55)	55	
AGATTTG	GAT (35)	ATC (13)	35	
GATTTGC	ATT (15)	AAT (3)	15	GTAGATTTGC
	TTT (63)	AAA (0)	63	
	TTG (62)	CAA (16)	62	
	TGC (57)	GCA (36)	57	

Figure 1 Illustration of a string CGTTGATCATTTG, its 7-mers and 3-mers, the canonical codes of the 3-mers, the minimizers for $m = 3$ using the identity permutation $\pi = \mathrm{id}$ and the resulting super-k-mers. We observe that minimizers frequently do not change as we advance the k-mer window; this yields super-k-mers of length up to $2k - m$.

of the 4 substitutions, at least 3 intact κ-mers remain, and at least 34 bp are covered by intact κ-mers. These properties have been determined by combinatorial optimization (unpublished own work).

2.2 Strategies for Efficient k-mer Counting

The k-mer counting tools KMC3 [8] and Gerbil [6] have repeatedly been evaluated to be among the fastest such tools [12]. They rely on the concepts of *minimizers* and *super-k-mers* for efficient counting. Here, we briefly review these ideas. Unfortunately, they do not easily transfer to counting gapped k-mers, as we explain below. Therefore, we propose a different strategy in this work based on a variant of Cuckoo hashing, whose basics we review in the next subsection. We first define the concept of a minimizing m-mer within a k-mer for $m \leq k$.

▶ **Definition 6** (Minimizer). *Let $1 \leq m \leq k$ be two integers. Let $u = u[0:k]$ be a DNA k-mer. Let $V = (v_0, \dots, v_{k-m}) := (u[0:m], u[1:m+1], \dots, u[k-m:k])$ be the $k-m+1$-long sequence of m-mers in u. Let $\pi : [4^m] \to [4^m]$ be a permutation on the set of encoded m-mers (for simplicity, we may think of the identity permutation). Consider the sequence of permuted canonical m-mer codes $P = (p_0, \dots, p_{k-m}) := (\pi(cc(v_0)), \pi(cc(v_1)), \dots, \pi(cc(v_{k-m})))$. We say that v_j is a* minimizer *(or minimizing m-mer) of u if $p_j = \min P$. In other words, a minimizer is an m-mer with the smallest π-value in u.*

The definition above can be relaxed in the sense that instead of a (bijective) permutation π, a hash function can be used, which may result in non-uniqueness of minimizers.

Adjacent k-mers frequently share the same minimizer, which moves through the k-mer from right to left. Such adjacent k-mers with the same minimizer may be grouped into a so-called *super-k-mer* of some length ℓ with $k \leq \ell \leq 2k - m$. This is illustrated in Figure 1.

The strategy of KMC can be described (somewhat simplified) as follows: The input files are read. For each k-mer in sequence, the minimizer is computed, and the k-mer is appended to one of 512 buffers, which is computed as a function of the minimizer (the default number of buffers can be changed by command line). This means that adjacent k-mers are frequently stored in the same buffer, which is cache-efficient. If a buffer is full, it is sorted, uniquified and counted, and then appended to a corresponding temporary file. After all k-mers have been read, each temporary file is again sorted, uniquified and counted. This can be done independently and hence in parallel for each temporary file, subject to memory and CPU

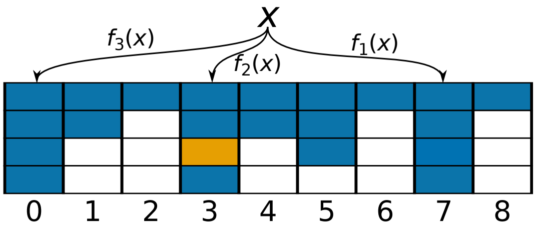


Figure 2 Illustration of multi-way bucketed Cuckoo hashing with $h = 3$ hash functions and bucket size $b = 4$. If all possible $hb = 12$ slots to store a key x are already occupied (blue/orange), a random one of these slots is picked (orange) and removed to make room for x. The removed element is then inserted at an alternative location.

core constraints. Finally, filters are applied to remove too frequent or too rare k-mers, and the remaining k-mers, together with their counts, are written into an output database in a custom binary format.

The strategy of Gerbil is slightly different, but also relies on the cache-friendliness of super-k-mers. The output file of Gerbil is larger but easier to use than that of KMC3: The k-mers and counters are given as byte sequences with as many bytes per k-mer and counter as required.

The purpose of this section is to highlight that the main reason for the speed of some of the fastest existing k-mer counters relies on exploiting the cache-friendliness of grouping many adjacent k-mers into a super-k-mer. In addition, for computing hash values of adjacent contiguous k-mers (or m-mers), a *rolling* hash function may be used, such as ntHash [14], which updates hash values of k-mers in constant time instead of time proportional to k.

Neither idea (super-k-mers, rolling hash functions) is directly applicable if one uses gapped k-mers instead of contiguous ones. this is why these and other tools only support contiguous k-mers.

2.3 Multi-way Bucketed Cuckoo Hashing

The method we propose in Section 3 is not based on minimizers or super-k-mers, but on directly building an in-memory hash table with canonical codes of k-mers as keys and their counts as values. We previously found multi-way bucketed Cuckoo hashing [23, 24] to perform well in terms of both low memory usage (due to a high load factor) and reasonably fast insertion, update and lookup times, so we briefly review their basics.

Consider a hash table of p buckets, where each bucket has space for $b \geq 1$ key-value pairs; we also say that a bucket consists of b slots. Thus, the total capacity of the table is $n := pb$. We use $h \geq 2$ hash functions $f_1, \dots, f_h : \Sigma^k \to [p]$; each hash function maps each k-mer (key) u to one of the p buckets. Thus, in general, there are hb possible slots where a given key may be stored (see Figure 2). This hashing strategy is referred to as h-way bucketed Cuckoo hashing with bucket size b, or (h, b)-*Cuckoo hashing* for short. It is designed to provide fast lookups and acceptable insertion time. The main idea is that the contents of a small bucket ($b \in \{3, \dots, 10\}$) are contained within a single cache line, so searching a bucket will incur only a single cache miss.

Lookup. When searching for a k-mer u with canonical code c, we first examine every slot in bucket $f_1(c)$. If u is not found (but the bucket is full), the search continues in bucket $f_2(c)$, and then in $f_3(c), \dots, f_h(c)$. In this way, a lookup examines between 1 and h buckets, resulting in between 1 and h cache misses. Depending on the load factor of the hash table (number of slots with keys in relation to $n = hb$), many searches only need to examine a single bucket and incur only a single cache miss. For example, we can experimentally observe that with $h = 3$, $b = 6$ and a load factor around 83%, we need to access 1.16 different buckets on average for a successful search; so many keys are stored in their "first" bucket $f_1(c)$.

Insertion by random walk. While lookup is simple and restricted to h different buckets, the insertion of a new element u can be slightly more complex. Let us start with the easy case. If any of the b slots in bucket $f_1(c)$ is available, u is inserted there. Otherwise, we repeat the attempt for buckets $f_2(c), \dots, f_h(c)$. As the table gets fuller, it may happen that all hb slots for u are already occupied. In this case, we pick a random one of these hb keys (call it v) and remove it to insert u in its place (hence the name "Cuckoo" hashing). Now we re-start the insertion process for v instead of u, hoping that one of its hb slots is still free. The process may repeat until we either find a free slot, or we report failure (after a pre-determined number of attempts). When picking a random element to remove to insert v, we ensure not to remove again the just previously inserted u, but to pick a different slot.

Alternatively, if all keys are known in advance, it is possible (but computationally expensive) to compute an optimal placement, such that the average number of bucket accesses for successful searches is minimized [25].

Properties of (h, b)-Cuckoo hashing. Walzer [22] analyzed the theoretical load limits of (h, b)-Cuckoo hashing. For classical Cuckoo hashing ($h = 2$ and $b = 1$), the load limit is 1/2 (meaning that if the hash table is only slightly more than 50% full, it will not be possible to insert all keys), but the load limit for (3,4)-Cuckoo hashing is above 99.9%. With random walk insertion (using a finite number of steps), these limits cannot be achieved, and performance degrades as the actual load approaches the limit, so a "safety margin" of 5% should be used in practice. Thus in practice, to obtain high loads, $h = 3$ hash functions suffice, even for small bucket sizes, and in this work, we only use $h = 3$.

3 A k-mer Counter Based on (h, b)-Cuckoo Hashing with Subtables

3.1 Design Goals for Counter Data Structures

A data structure that supports counting the number of occurrences of keys (here, integer-encoded k-mers) must support the following operations very efficiently:

- insertion of a new key with counter value 1,
- test whether a given key is present,
- retrieval of the counter value of a given key if present, or zero otherwise,
- update of the counter value (typically, increment by 1) for a given key.

In contrast, deletions of existing keys need not be supported. Since (h, b)-Cuckoo hash tables (see Section 2.3) support the required operations quite efficiently, they form a good basis for designing a k-mer counter.

Other desirable properties of the data structure are

- the ability to grow dynamically (if the final number of keys to be stored is initially unknown),
- its support of parallel writes, as the sizes of datasets often range in the hundreds of gigabytes.

Our current implementation does not support dynamic growth, but in most applications, a good estimate of the number of keys is known can be quickly estimated [15].

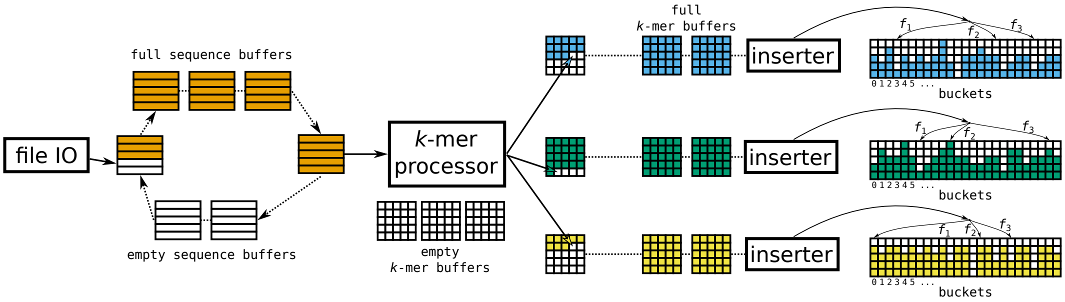


Figure 3 Producer-consumer architecture: An I/O thread reads a large chunk of an input file into an available sequence buffer. A producer thread (the k-mer processor) computes the canonical code c of each k-mer in the sequence buffer, and determines into which subtable t it shall be inserted, by using a simple invertible hash function g_0, such that $(t, s) = (g_0(c) \bmod T,\ g_0(c) /\!\!/ T)$. The quotient s is placed into a k-mer buffer that only thread t will read for insertion into subtable $t \in [T]$.

Supporting parallel write access to hash tables from different threads is non-trivial if one wants to avoid data races. One possibility is to use locks (of individual buckets or larger regions of the hash table), although this approach is rather costly in terms of computational overhead. A different, less costly, option is to design lock-free approaches using atomic CPU operations such as CAS (compare-and-swap), as was implemented for the Jellyfish k-mer counter [13]. However, these atomic CPU instructions operate on entire memory words (e.g., a 64-bit integer). As we use bit packing to save space (see Section 3.3 below), it is possible that one key/value pair extends over several (memory-aligned) words in our hash table implementation; so we opted for a different approach that offers additional advantages.

3.2 Parallelizing Key Insertions into (h, b)-Cuckoo Hash Tables

Given T threads, the key idea is to divide the hash table into T independent subtables, such that each subtable is served (written to) by a specific single thread.

As we read a (gapped) k-mer u from the sequences in the dataset, a *producer thread* computes its canonical code $c = cc(u)$ and, using a simple *bijective* hash function g_0, partitions it into a subtable number $t = g_0(c) \bmod T \in \{0, 1, \ldots, T-1\}$ and a *subkey* $s = g_0(c) /\!\!/ T$, where $/\!\!/$ denotes integer floor division. Given (t, s), we can recover $g_0(c) = s \cdot T + t$, and since g_0 is bijective, we can recover c. For this to work efficiently, g_0 must be easily invertible (see below).

The producer thread inserts the computed s into a buffer that will be read only by the *consumer thread* for subtable t. Several such buffers exist for each consumer thread, and the producer fills them in a round-robin fashion, marking a buffer as "ready-to-read" if it is full and can be processed by the consumer. It then selects a new (ready-to-write) buffer for consumer t to fill next. If no ready-to-write buffer is available, the producer must wait, stalling the whole counting process. Conversely, if a consumer thread has inserted all keys from one buffer, it marks it as "ready-to-write" for the producer and picks the next "ready-to-read" buffer that is available for it (or waits for one to become available). This producer-consumer approach is shown in Figure 3.

The "ready-to-read" and "ready-to-write" messages are sent by atomic volatile stores into and loads from a small integer array accessible from all threads. If a thread has to wait, it keeps monitoring the states of all of its buffers and consumes CPU time, but in an energy-saving state executing the `pause` instruction between checks.

Table 2 Double quotienting: A k-mer u's canonical code c is partitioned into a subtable number t and the first "quotient", the subkey s. The subkey is then partitioned (for each hash function h_j into the bucket address $f_j(s)$ and the second "quotient" $q_j(s)$. Only this quotient $q_j(s)$ needs to be stored (together with j) at address $f_j(s)$ in subtable t in order to reconstruct c.

$$\text{key } u \in \Sigma^k \to \text{ canonical code } c \in [4^k] \quad \longleftrightarrow \quad (t, s) = (g_0(c) \bmod T,\, g_0(c) /\!\!/ T)$$

$$\text{subkey } s \quad \longleftrightarrow \quad (f_j(s),\, q_j(s),\, j) = (g_j(s) \bmod p,\, g_j(s) /\!\!/ p,\, j) \text{ for } j = 1, \dots, h$$

Hash functions. The h functions $f_1, \dots, f_h$ that select the possible buckets for a k-mer u are functions of u's subkey s, and they map each possible subkey onto the address space of a subtable $[p_{\text{sub}}]$. Each subtable contains p_{sub} buckets, so the full hash table contains $p = Tp_{\text{sub}}$ buckets.

For the f_j, we use the same structure as for the initial hash function f_0: In general, we take $f_j(s) = g_j(s) \bmod p_{\text{sub}}$ with a function g_j that is bijective on $[2^\sigma]$, where σ is the bit width of subkeys. We also compute the quotient $q_j(s) = g_j(s) /\!\!/ p_{\text{sub}}$, so we can recover s from its address (bucket number) $f_j(s)$ and quotient $q_j(s)$ if $g_j(s)$ is efficiently invertible.

For the bijective functions g_j, $j = 1, \dots, h$ we use affine functions of the form

$$G_{a,b}(s) := [a \cdot (\text{rot}(s) \text{ xor } b)] \bmod 2^\sigma,$$

where rot performs a cyclic rotation by half the bit width of s, and b is a σ-bit offset, and a is an odd multiplier. Picking a "random" hash function means picking random values for a and b. Such a $G_{a,b}$ is invertible on $[2^\sigma]$ because a is odd; so $\gcd(a, 2^\sigma) = 1$. Therefore, there exists a unique multiplicative inverse a' such that $aa' = 1 \pmod{2^\sigma}$. This inverse a' can be pre-computed efficiently using the extended Euclidean algorithm. It follows that, given $i = f_j(s)$ and $q = q_j(s)$, we can recover s by first computing $x = q \cdot p_{\text{sub}} + i$ and then $s = G^{-1}_{a,b}(x) = [(a' \cdot x) \bmod 2^\sigma] \text{ xor } b$.

3.3 Space-Saving Techniques

As we do not use temporary files and do only in-memory computations, we must take care not to waste space unnecessarily. Our main ingredients for space efficiency are

1. a high load of the hash table,
2. bit packing,
3. storing quotients instead of full keys in the hash table.

A high load rate is achieved by choosing large enough h (number of hash functions) and b (bucket size); we already mentioned that $h \geq 3$ and $b \geq 4$ allow loads above 99.9%. By "bit packing" we mean that we view the hash table as a bit array instead of a byte or uint64 array: If a key-value pair needs 31 bits, we only store 31 bits and do not round up to 32 bits. This sometimes requires reading a single value from two adjacent memory cells, but the CPU efficiently handles bit manipulations. The quotienting technique reduces the storage requirement of every single key-value pair and is explained below.

Double quotienting. Instead of storing the full key-value pair (canonical k-mer code c and its counter) for each k-mer, we only store a part of the key that, however, allows to reconstruct the exact value of c. This is possible because part of the information is stored in the address (subtable t, bucket i) itself. As described above and visualized in Table 2, we first split the canonical code c into (t, s), so we only need to store s within subtable t (which requires fewer bits than the full c). Because s is a quotient ($s = g_0(c) /\!\!/ T$), this is referred to as *quotienting*. The same strategy is applied again to s where we compute the bucket numbers $i_j = g_j(s) \bmod p_{\text{sub}}$ and the quotients $q_j = g_j(s) /\!\!/ p_{\text{sub}}$ for $j = 1, \dots, h$. Because we use h hash functions, we must also store *which* hash function j we used (2 bits for $h = 3$).

Importantly, quotienting is only possible when using efficiently invertible hash functions $g_0, g_1, \dots, g_h$, as described above.

▶ **Example 7.** We store 25-mers (50 bits each) and counters between 0 and 255 (8 bits each), requiring 58 bits per key-value pair. (Technically, 49 bits suffice for storing canonical 25-mers, using a minimal perfect hash function that maps the valid canonical 50-bit codes to $[2^{49}]$. Such functions are folklore for odd k and have been communicated to us at DSB 2022 by Roland Wittler for even k.)

Assume that we use $T = 9$ subtables, each with $p_{\text{sub}} = 83\,333\,335$ buckets of size $b = 4$, for a total of $p = Tp_{\text{sub}} = 750\,000\,015$ buckets and $n = bp = 3\,000\,000\,060$ slots. Then, $t \in [9] = \{0, \dots, 8\}$, and each subkey $s < 4^k/T$ can be represented in 47 bits (46.83 in theory). Moreover, $f_j(s) \in [p_{\text{sub}}]$, and each quotient $q_j(s) < 2^{47}/p_{\text{sub}}$ can be represented in $\lceil \log_2(2^{47}/p_{\text{sub}}) \rceil = 21$ bits.

Instead of 50 bits per key, we only need to store 21 bits per key, plus 2 bits to remember the hash function j (1, 2, or 3), plus the 8 bits for counters up to 255. So we need 31 bits (instead of 58) per key-value pair.

The more buckets we have, the fewer bits per k-mer are needed (when p_{sub} doubles, we have 1 bit less in the quotient). Using smaller values of b also increases p_{sub} (for a constant number of slots in the hash table). However, using smaller b also decreases the load threshold and requires storing more keys at their second and third hash choices.

3.4 Just-in-Time Compilation

An important aspect of our implementation (not of the methodology itself) is that we just-in-time compile all methods of the hash table *after* we know its parameters. This is achieved using Python as our main implementation language, together with the `numba` package [9] that allows compiling a subset of Python at run-time.

As we invoke affine hash functions of the form $c \mapsto (a(c \text{ xor } b)) \bmod p$ billions to trillions of times, using mod and integer division ($/\!\!/$) operations, there is a significant difference in running time if the modulus or divisor p is a variable or a compile-time constant. If it is a compile-time constant, as in our case, the compiler can replace the general and expensive `divmod` operations by specialized faster code for the given value of p. The same optimization potential exists for integer-encoding a gapped k-mer with a fixed mask that becomes known after the program starts but before compilation.

4 Computational Experiments

4.1 Datasets and Experimental Setup

We present results on different datasets, which comprise the recent telomere-to-telomere (t2t) human genome [19], and several whole-genome sequencing samples from humans and pigs. Sources and statistics are given in Table 3. The datasets were chosen because they exhibit different properties.

Table 3 Datasets for evaluation and their properties (Ref.: literature reference; Gbp: Giga basepairs in the dataset; size [GB]: total file size on disk for the dataset; distinct-25: number of distinct 25-mers in the dataset; for gapped k-mers, the number is higher). URLs for obtaining the data are given in Appendix A.

Name	Ref.	type	Gbp	size [GB]	distinct-25
t2t human genome	[19]	1 `.fasta.gz`	3.117	0.916	2 391 456 540
Göttingen minipigs (10 minipig samples)	[20]	2 `.fastq` (each sample)	36.788 to 42.540	126.034 to 145.458	2 892 672 435 to 3 501 660 208
GIAB Ashkenazim trio	[26]	12 `.fastq.gz`	314.554	220	14 297 937 824

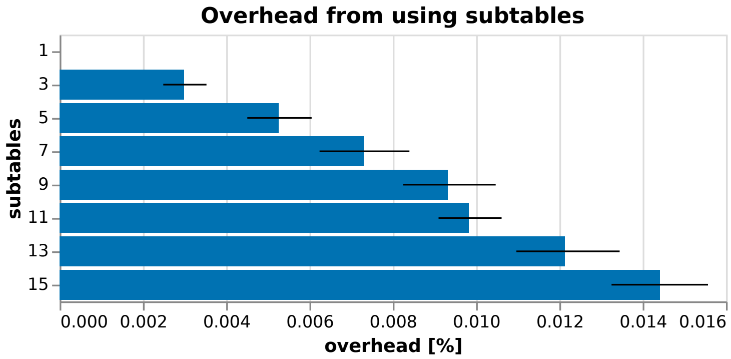


Figure 4 Load overhead from using different numbers of subtables (means and standard deviations from 20 random samples of hash functions). All overheads are less than $0.02\% = 2 \cdot 10^{-4}$.

- The t2t human genome is a small dataset in gzipped FASTA format with only approx. 2.4 billion distinct 25-mers, most of which occur only once.
- The 10 individual Göttingen minipig WGS samples are of moderate size (15x coverage, 2x100 bp Illumina paired-end, uncompressed FASTQ) and their 25-mers and counts conveniently fit into main memory. Many k-mers occur multiple times, as expected for 15x coverage samples, and in each individual, there is a limited number of rare k-mers (e.g., singletons). Each individual was counted separately.
- The GIAB Ashkenazim trio dataset consists of three individual humans from one family (mother, father, child), mate-pair sequenced with 2x125 bp. We counted (gapped) 25-mers from all files simultaneously, resulting in a table that just fits into 64 GB of main memory (14.3 billion 25-mers in 55 GB).

We use $k = 25$ and different symmetric masks of shape $(k, w) = (25, 31)$ with 6 insignificant positions (see Table 1). To evaluate k-mer uniqueness, we also report results on more different values of k that are suitable for mammalian genomes.

We compared KMC3 [8], Gerbil [6] and our tool `hackgap` on a PC workstation with an AMD Ryzen 9 5950X (4.9 GHz) processor with 16 cores (32 threads due to hyperthreading) and 64 GB of DDR4 RAM (3200 MHz, CL22). During evaluations, other activities on the workstation were suspended. All files (inputs, outputs and temporary files) were stored on an internal 16 TB HDD (SATA 3.3 with 6 Gbit/s, 7200 rpm). The exact command lines for calling each tool can be found in Appendix B.

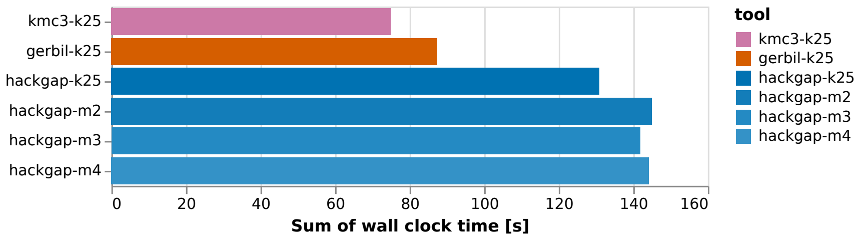


Figure 5 Wall clock times to count 25-mers in the t2t genome for different tools, and, using our hackgap tool only, for different $(w, k) = (31, 25)$-shaped masks m_2, m_3, m_4 (cf. Table 1).

4.2 Load Variation among Subtables

We first consider the question whether the partitioning of a large hash table into subtables leads to uneven loads in the subtables, i.e., whether using simple affine invertible hash functions as described above results in some of the subtables being (much) fuller than others. The performance of parallel k-mer counting is limited by the load of the fullest subtable.

Therefore, we chose 20 sets of random hash functions and distributed the 25-mers of the t2t genome into different numbers of subtables. We measured the *overhead*, which is the ratio between the size of the largest subtable and the mean size of all subtables, expressed in per cent above 100%. (A ratio of 1.0007 would be an overhead of 0.07%.) We expect a trend towards larger overheads for more subtables, just because of random fluctuations. The results in Figure 4 confirm this expectation, and they also show that the overhead is extremely small ($\leq 0.016\% = 1.6 \cdot 10^{-4}$) for up to 15 subtables.

4.3 Results on the t2t Genome

We counted the 25-mers in the t2t genome (from its gzipped FASTA file) using KMC3, Gerbil and hackgap and measured the wall clock running time. The reference contains a relatively small number of distinct 25-mers (2.4G), and most of the 25-mers occur only once (see also Section 4.4). The final hash table fits into 12.5 GB of memory (with a higher fill rate, it can even fit into 9 GB); each tool was given 16 GB of memory to work. Both KMC3 and Gerbil were given up to 16 cores, but did not use them for most of the time. We used 5 subtables for hackgap (6 threads).

The results can be seen in Figure 5. KMC3 was fastest, with Gerbil being almost as fast. Our tool hackgap took almost twice as long, but is the only one that supports gapped k-mers, which do not take much longer to count. One reason for the (comparatively) slower performance of hackgap may be that the FASTA reader is currently implemented in pure Python and not very efficient. Performance is better on FASTQ files (see Section 4.5 below).

4.4 Effect of Masks on Unique k-mers

As discussed in the introduction, unique k-mers (that appear only once in a reference genome and thus uniquely identify a location in the genome) are most helpful for alignment-free methods. An even stronger notion is the one of *strongly unique* k-mers; these are unique k-mers who additionally have no Hamming-distance 1 neighbor in the genome. Formally, a genomic k-mer $u \in \Sigma^k$ is strongly unique if $count(u) = 1$ and $d_{\text{Hamming}}(u, v) \geq 2$ and $d_{\text{Hamming}}(u, rc(v)) \geq 2$ for all genomic k-mers $v \neq u$. Strong uniqueness of u can be checked by computing the canonical codes of all $3k$ Hamming-distance-1 neighbors of u and querying the hash table for them, but there are also faster methods [24].

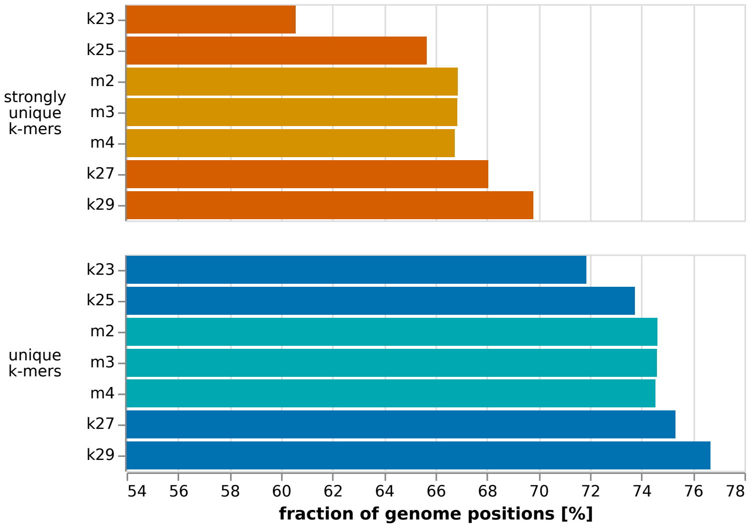


Figure 6 Fraction of genomic starting positions of *unique* and *strongly unique* k-mers begin, for different values of k and a selection of $(w, k) = (31, 25)$-shaped masks.

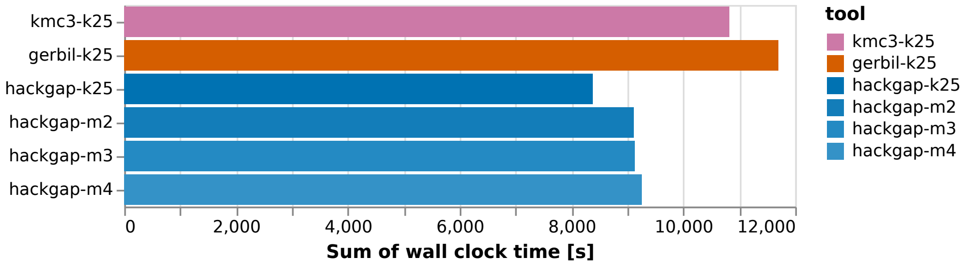


Figure 7 Wall clock times to count 25-mers in whole genome samples of 10 Göttingen minipigs (FASTQ, uncompressed) for KMC3 and Gerbild (given 16 GB RAM and up to 24 threads), and, using our `hackgap` tool only (given 16 GB RAM and using 5 subtables or 7 threads), for different $(31, 25)$-shaped masks, as given in Table 1.

We investigated the effect of varying k and using gapped k-mers with different masks (Table 1) on the fraction of positions in the genome where we find (strongly) unique k-mers. The results are shown in Figure 6. Clearly, the fraction of unique and strongly unique k-mers increases with k. Moreover, the three $(31, 25)$-shaped masks achieve a higher fraction of (strongly) unique genome positions than for the contiguous 25-mers (but less than 27-mers).

Currently, `hackgap` is the only k-mer counter that efficiently supports identifying strongly unique gapped k-mers.

4.5 Results on Minipig Whole Genome Samples

We counted 25-mers in each of 10 Göttingen minipig whole genome samples (unzipped FASTQ files) using KMC3, Gerbil and `hackgap` and measured the wall clock running time. Each sample had an average coverage of approximately 15x and thus contained many repeated k-mers, but also several unique k-mers (due to sequencing errors or technical artefacts).

The final hash table fits into 14 GB of memory for each minipig sample, so each tool was given 16 GB of memory. Both KMC3 and Gerbil were given up to 24 threads, but did not use them for most of the time. We used 5 subtables for `hackgap` (7 threads).

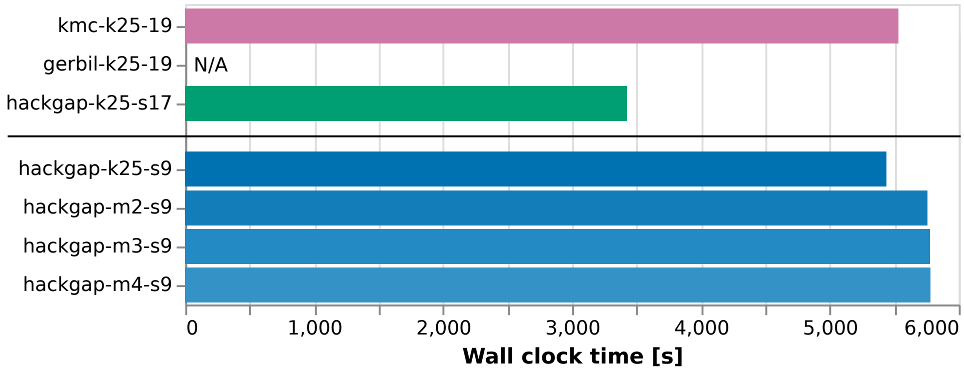


Figure 8 Wall clock times to count 25-mers in a combined whole genome sample of 3 members of an Ashkenazim family (gzipped FASTQ) for different tools, and, using our `hackgap` tool only, for different $(31, 25)$-shaped masks, as given in Table 1. Each tool was given 55 GB of RAM. On all of our test machines, Gerbil exited with an "Unknown error".

The results can be seen in Figure 7. Surprisingly, `hackgap` was the fastest tool overall and also on each single dataset. Counting gapped k-mers takes slightly longer for `hackgap` than counting contiguous k-mers (due to some optimizations not being applicable), but was still faster than KMC3 or Gerbil. This makes `hackgap` the fastest k-mer counter on medium-sized FASTQ files (whose k-mer hash table fits into main memory) and the only one that can count gapped k-mers efficiently.

On the other hand, due to our "busy waiting" (spin lock) implementation, our approach currently consumes unnecessarily much CPU time (details in Table 4) in Appendix C).

4.6 Results on a Large Human Whole Genome Sample

We furthermore picked a large dataset, the Genome-in-a-bottle Ashkenazim trio (three individuals of one family, HG002, HG003, HG004) and counted 25-mers for all three samples together. The resulting hash table barely fits into main memory (with the operating system and necessary systems processes running) and needs 55 GB. The KMC output files even need 68 GB. Gerbil was unable to process the dataset and exited with an "Unknown Error" on all of our test machines.

The results can be seen in Figure 8. With 17 subtables (19 active threads), we count everything in less than 1 hour (even less than 3500 s) wall-clock time, where KMC3 needs 5500 s. When we used gapped 25-mers, we do not benefit from more than 9 subtables (11 threads; the k-mer encoder becomes the bottleneck), but we achieve running times comparable to that of KMC3 with 19 threads. Using gapped k-mers needs approximately 5% more time than using contiguous k-mers.

We additionally ran KMC3 in "memory-only" mode (on a different server with 1 TB of memory). This mode does not write temporary files to disk, and on this data set saved 14% of the running time, but required 433 GB of RAM, whereas our approach is "memory-only" by design, and the allocated 55 GB are sufficient.

5 Discussion and Conclusion

Summary of contributions. We have demonstrated that, in contrast to common belief, efficient k-mer counting does not need to depend on the concepts of minimizers and super-k-mers. Three-way subdivided bucketed Cuckoo hash tables can show the same performance, if used with appropriate parameters. Having such an alternative paves the way for efficient gapped k-mer counting, for which we provide one of the first efficient tools written in partially jit-compiled Python. Surprisingly, it can even be slightly faster than two of the fastest known k-mer counters (KMC3 and Gerbil) on large gzipped FASTQ datasets.

Using our approach, the running time for gapped k-mers is consistently slightly slower than for contiguous k-mers (by approximately 5% to 10%). This observation can be explained by the more involved integer encoding of a sequence of gapped k-mers compared to a sequence of contiguous k-mers, where two adjacent k-mers overlap by $k-1$ characters and efficient bit-shifting rolling hash functions can be (and are) used. However, this appears to be a relatively small cost for the gained ability to work with gapped k-mers. Also, the increase in time may be relatively small because the mask is a compile-time constant for the jit-compilation during runtime, the compiler can unroll the loop over the significant positions.

On a technical level, we have parallelized Cuckoo hash tables by subdividing them into subtables and designing the hash functions such that all possible locations of a key lie in the same subtable. We also showed that the keys are evenly distributed among the subtables (for an odd number of subtables), even with very simple affine (and even linear) hash functions.

Advantages and limitations of different approaches. Our implementation currently has a number of limitations, some of which we hope to remove in the near future.

Tools like KMC3 and Gerbil that use temporary files whose size is only limited by the available hard drive space do not need to know the number of distinct k-mers beforehand. Our approach, however, needs to allocate a hash table of sufficient size from the beginning; so we need at least a good estimate of the number of distinct k-mers to store. This may be an issue on a completely unknown dataset, but on a mammalian genome dataset, we can usually get a good estimate from the FASTQ file sizes. An alternative is to use a fast cardinality estimation tool like ntCard [15].

Our implementation is currently limited to $k \leq 32$ because we internally use 64-bit integers. For most applications in genomics, this should be sufficient, but in particular Gerbil efficiently supports much larger values of k.

The fact that our method is not disk-based (i.e., writes no temporary files) can be seen as advantage or disadvantage: The number of distinct k-mers we can store is limited by the main memory size. (The FASTQ datasets can be huge, as long as the number of distinct k-mers stays bounded.) In contrast, both KMC and Gerbil use large temporary space on the hard drive (KMC up to 350 GB for the GIAB dataset). While this is not a problem with today's drive sizes, it contributes to drive wear, which may especially affect SSDs which allow a limited number of writes during their lifetime.

If the dataset contains several input files, both KMC and Gerbil may read them in parallel. Our file IO is currently implemented in pure Python and becomes a bottleneck. We observed that we can feed up to 17 threads with contiguous k-mers, but only 9 threads with gapped k-mers, where our k-mer processor must do more work to compute canonical codes from the input sequence buffer. On the other hand, our input can be any stream, such as an anonymous pipe, while KMC3 and Gerbil only accept regular files.

Perspectives. We believe that the subdivided Cuckoo hashing approach may be further improved, as suggested above. The most important bottleneck currently is FASTA and FASTQ file I/O, which is solved much better in KMC3 and Gerbil and one of many reasons of their high speed. Further ongoing work concerns the implementation of pre-filtering steps to catch a large number of rare k-mers, which consume a large amount of memory in our hash table but typically are of no interest to the user. Lastly, we are investigating the possibility of developing a hybrid method between Cuckoo hashing and minimizers and how the concept of minimizers may be transferred to gapped k-mers.

References

1 A. Bankevich, A. V. Bzikadze, M. Kolmogorov, D. Antipov, and P. A. Pevzner. Multiplex de Bruijn graphs enable genome assembly from long, high-fidelity reads. *Nat Biotechnol*, February 2022.

2 N. L. Bray, H. Pimentel, P. Melsted, and L. Pachter. Near-optimal probabilistic RNA-seq quantification. *Nat. Biotechnol.*, 34(5):525–527, May 2016. Erratum in Nat. Biotechnol. 34(8):888 (2016).

3 Karel Břinda, Maciej Sykulski, and Gregory Kucherov. Spaced seeds improve k-mer-based metagenomic classification. *Bioinformatics*, 31(22):3584–3592, July 2015. doi:10.1093/bioinformatics/btv419.

4 Andrea Califano and Isidore Rigoutsos. FLASH: a fast look-up algorithm for string homology. In Lawrence Hunter, David B. Searls, and Jude W. Shavlik, editors, *Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology, Bethesda, MD, USA, July 1993*, pages 56–64. AAAI, 1993. URL: http://www.aaai.org/Library/ISMB/1993/ismb93-007.php.

5 S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, May 2015.

6 M. Erbert, S. Rechner, and M. Müller-Hannemann. Gerbil: a fast and memory-efficient k-mer counter with GPU-support. *Algorithms Mol Biol*, 12:9, 2017.

7 Lars Hahn, Chris-André Leimeister, Rachid Ounit, Stefano Lonardi, and Burkhard Morgenstern. rasbhari: Optimizing spaced seeds for database searching, read mapping and alignment-free sequence comparison. *PLoS Comput. Biol.*, 12(10), 2016. doi:10.1371/journal.pcbi.1005107.

8 M. Kokot, M. Dlugosz, and S. Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, September 2017.

9 Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a LLVM-based python JIT compiler. In Hal Finkel, editor, *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015*, pages 7:1–7:6. ACM, 2015. doi:10.1145/2833157.2833162.

10 J. Lu, F. P. Breitwieser, P. Thielen, and S. L. Salzberg. Bracken: estimating species abundance in metagenomics data. *PeerJ Computer Science*, 3:e104, 2017. doi:10.7717/peerj-cs.104.

11 Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinform.*, 18(3):440–445, 2002. doi:10.1093/bioinformatics/18.3.440.

12 Swati C Manekar and Shailesh R Sathe. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*, 7(12), October 2018. giy125. doi:10.1093/gigascience/giy125.

13 Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, January 2011. doi:10.1093/bioinformatics/btr011.

14 H. Mohamadi, J. Chu, B. P. Vandervalk, and I. Birol. ntHash: recursive nucleotide hashing. *Bioinformatics*, 32(22):3492–3494, November 2016.

15 H. Mohamadi, H. Khan, and I. Birol. ntCard: a streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, 33(9):1324–1330, May 2017.

16 A. Müller, C. Hundt, A. Hildebrandt, T. Hankeln, and B. Schmidt. MetaCache: context-aware classification of metagenomic reads using minhashing. *Bioinformatics*, 33(23):3740–3748, December 2017.

17 L. Noé. Best hits of 11110110111: model-free selection and parameter-free sensitivity calculation of spaced seeds. *Algorithms Mol Biol*, 12:1, 2017.

18 Laurent Noé. Best hits of 11110110111: model-free selection and parameter-free sensitivity calculation of spaced seeds. *Algorithms Mol. Biol.*, 12(1):1:1–1:16, 2017. doi:10.1186/s13015-017-0092-1.

19 S. Nurk, S. Koren, A. Rhie, M. Rautiainen, A. V. Bzikadze, A. Mikheenko, M. R. Vollger, N. Altemose, L. Uralsky, A. Gershman, S. Aganezov, S. J. Hoyt, M. Diekhans, G. A. Logsdon, M. Alonge, S. E. Antonarakis, M. Borchers, G. G. Bouffard, S. Y. Brooks, G. V. Caldas, N. C. Chen, H. Cheng, C. S. Chin, W. Chow, L. G. de Lima, P. C. Dishuck, R. Durbin, T. Dvorkina, I. T. Fiddes, G. Formenti, R. S. Fulton, A. Fungtammasan, E. Garrison, P. G. S. Grady, T. A. Graves-Lindsay, I. M. Hall, N. F. Hansen, G. A. Hartley, M. Haukness, K. Howe, M. W. Hunkapiller, C. Jain, M. Jain, E. D. Jarvis, P. Kerpedjiev, M. Kirsche, M. Kolmogorov, J. Korlach, M. Kremitzki, H. Li, V. V. Maduro, T. Marschall, A. M. McCartney, J. McDaniel, D. E. Miller, J. C. Mullikin, E. W. Myers, N. D. Olson, B. Paten, P. Peluso, P. A. Pevzner, D. Porubsky, T. Potapova, E. I. Rogaev, J. A. Rosenfeld, S. L. Salzberg, V. A. Schneider, F. J. Sedlazeck, K. Shafin, C. J. Shew, A. Shumate, Y. Sims, A. F. A. Smit, D. C. Soto, I. Sović, J. M. Storer, A. Streets, B. A. Sullivan, F. Thibaud-Nissen, J. Torrance, J. Wagner, B. P. Walenz, A. Wenger, J. M. D. Wood, C. Xiao, S. M. Yan, A. C. Young, S. Zarate, U. Surti, R. C. McCoy, M. Y. Dennis, I. A. Alexandrov, J. L. Gerton, R. J. O'Neill, W. Timp, J. M. Zook, M. C. Schatz, E. E. Eichler, K. H. Miga, and A. M. Phillippy. The complete sequence of a human genome. *Science*, 376(6588):44–53, April 2022.

20 C. Reimer, C. J. Rubin, A. R. Sharifi, N. T. Ha, S. Weigend, K. H. Waldmann, O. Distl, S. D. Pant, M. Fredholm, M. Schlather, and H. Simianer. Analysis of porcine body size variation using re-sequencing data of miniature and large pigs. *BMC Genomics*, 19(1):687, September 2018.

21 Z. Sun, S. Huang, M. Zhang, Q. Zhu, N. Haiminen, A. P. Carrieri, Y. Vázquez-Baeza, L. Parida, H. C. Kim, R. Knight, and Y. Y. Liu. Challenges in benchmarking metagenomic profilers. *Nat Methods*, 18(6):618–626, June 2021.

22 Stefan Walzer. Load thresholds for cuckoo hashing with overlapping blocks. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPIcs*, pages 102:1–102:10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.ICALP.2018.102.

23 Jens Zentgraf and Sven Rahmann. Fast lightweight accurate xenograft sorting. In Carl Kingsford and Nadia Pisanti, editors, *20th International Workshop on Algorithms in Bioinformatics, WABI 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 172 of *LIPIcs*, pages 4:1–4:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.WABI.2020.4.

24 Jens Zentgraf and Sven Rahmann. Fast lightweight accurate xenograft sorting. *Algorithms Mol. Biol.*, 16(1):2, 2021. doi:10.1186/s13015-021-00181-w.

25 Jens Zentgraf, Henning Timm, and Sven Rahmann. Cost-optimal assignment of elements in genome-scale multi-way bucketed cuckoo hash tables. In Guy E. Blelloch and Irene Finocchi, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, pages 186–198. SIAM, 2020. doi:10.1137/1.9781611976007.15.

26 J. M. Zook, D. Catoe, J. McDaniel, L. Vang, N. Spies, A. Sidow, Z. Weng, Y. Liu, C. E. Mason, N. Alexander, E. Henaff, A. B. McIntyre, D. Chandramohan, F. Chen, E. Jaeger, A. Moshrefi, K. Pham, W. Stedman, T. Liang, M. Saghbini, Z. Dzakula, A. Hastie, H. Cao, G. Deikus, E. Schadt, R. Sebra, A. Bashir, R. M. Truty, C. C. Chang, N. Gulbahce, K. Zhao, S. Ghosh, F. Hyland, Y. Fu, M. Chaisson, C. Xiao, J. Trow, S. T. Sherry, A. W. Zaranek, M. Ball, J. Bobe, P. Estep, G. M. Church, P. Marks, S. Kyriazopoulou-Panagiotopoulou, G. X. Zheng, M. Schnall-Levin, H. S. Ordonez, P. A. Mudivarti, K. Giorda, Y. Sheng, K. B. Rypdal, and M. Salit. Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Sci Data*, 3:160025, June 2016.

A Dataset URLs

t2t human genome. https://s3-us-west-2.amazonaws.com/human-pangenomics/T2T/CHM13/assemblies/analysis_set/chm13v2.0.fa.gz

Minipigs. We use runs ERR2744277 through ERR2744286 from https://www.ebi.ac.uk/ena/browser/view/PRJEB2765

GIAB Ashkenazim Trio. We use the 6kb matepair data from https://github.com/genome-in-a-bottle/giab_data_indexes/blob/master/AshkenazimTrio/sequence.index.AJtrio_Illumina_6kb_matepair_wgs_08032015

B Executed Commands

In the command lines below, replace {input} with the name(s) of the (gzipped) FASTA / FASTQ / TXT input file. A text (TXT) file must contain one file name per line. Similarly, replace {output} by the output file name (a huge file containing a hash table or other data structure with k-mers and counts), and {report.json} (KMC only) by the name of a report file. Also, replace {mask} with the mask (strings in Table 1) (hackgap only) and {tmpdir/} with the name of a temporary directory (with up to 500 GB of working space).

B.1 t2t Human Genome Dataset

B.1.1 Time Benchmark

```
hackgap count  -v count 256 -n 3000000000 --nostats --fasta {input}\
    -p 6 --mask {mask} --subtables 5 -o {output}
```

```
kmc -v  -k25 -ci1 -m16 -sm -fm -t24 -j{report.json} @{input.txt}\
    {output} {tmpdir/}
```

```
gerbil -t 24 -i -k 25 -e 16G -l 1 {input.txt} {tmpdir/} {output}
```

B.1.2 Subtable Load Overhead

We ran the following command with 20 different combinations of hash functions; one(!) example is shown below. We varied the number of subtables between 3 and 15 (odd numbers only). Replace {ST} by the appropriate number.

```
hackgap count  -v count 3  -n 3000000000 --substatistics --fasta {input}\
    -p 6 --mask {mask} --subtables {ST} -o {output}\
    --hashfunctions linear73198857:linear11669733:linear79255563:linear56630389
```

B.1.3 Unique and Strongly Unique k-mers

The following command was executed for contiguous k-mers for $k \in \{23, 25, 27, 29\}$ and masks m2, m3, m4 from Table 1.

```
hackgap count -v  count 3 --strong  -n 3000000000 --fasta {input}\
    -p 6 --mask {mask} --subtables 15 -o {output}
```

B.2 Minipig Data

The following commands were executed for each of the 10 minipig samples separately. To compare running times, the times were summed over all individual pigs. We used 5 subtables (7 threads) and made up to 24 threads available to KMC3 and Gerbil. The memory limit was set to 16 GB which conveniently fits the k-mers and their counts.

```
hackgap count -v count 256 -n 3_500_000_000 --nostats --fastq {input}\
    -p 6 --mask {mask} --subtables 5 -o {result}

kmc -v -k25 -ci1 -m16 -sm -fq -t24 -j{report.json} @{input.txt}\
    {output} {tmpdir/}

gerbil -t 24 -i -k 25 -e 16G -l 1 {input.txt} {tmpdir/} {output}
```

B.3 GIAB Dataset

We executed hackgap with both 17 (for ungapped 25-mers, comparison among tools) and 9 subtables (comparison among gapped and ungapped masks, hackgap only). The reason is that our k-mer processor can serve 17 inserter threads sufficiently fast when it processes ungapped k-mers but only 9 when it processes gapped k-mers because encoding takes more time.

We gave KMC and Gerbil the same number of active threads as for our hackgap (19 threads for 17 subtables). Unfortunately, gerbil crashed in each case on this dataset (although it processed the smaller datasets perfectly).

The memory limit was set to 55 GB, which is sufficient to hold our k-mer table with counts.

Command lines for 17 subtables (19 threads), contiguous k-mers.

```
hackgap count -v count 256  -n 14297937824 --fill 0.88 --nostats\
    --fastq {input} -p 7 -k 25 --subtables 17 -o {output}

kmc -v -k25 -ci1 -m55 -sm -fq -t19 -j{report.json} @{input.txt}
{output} {tmpdir/}

gerbil -t 19 -i -k 25 -e 55G -l 1 {input} {tmpdir/} {output}
```

Command lines for 9 subtables (11 threads).

```
hackgap count -v count 256  -n 14297937824 --fill 0.94 --nostats\
    --fastq {input} -p 10 --mask {mask} --subtables 9 -o {output}
```

C CPU time and memory usage

With a comparable number of threads, or even fewer threads, our tool hackgap achieves faster wall-clock running times than KMC3 or Gerbil.

However, we noted that KMC3 and Gerbil only rarely use all threads given to them. This actually results in considerable lower CPU times than (number of threads) × (wall-clock time), wheras for hackgap all threads are busy (and possibly busy waiting with spin locks) for

Table 4 CPU time and actually used memory in our experiments.

Tool	mask	Dataset	CPU time [s]	Memory [MB]
hackgap	m2	t2t	838.09	13147.30
hackgap	m3	t2t	784.31	13139.45
hackgap	m4	t2t	836.24	13311.99
hackgap	k25	t2t	700.02	13138.26
kmc	k25	t2t	300.97	14916.35
gerbil	k25	t2t	276.89	953.88
hackgap	m2	Minipig	56168.94	15084.040
hackgap	m3	Minipig	56341.69	15083.906
hackgap	m4	Minipig	57177.59	15084.407
hackgap	k25	Minipig	51115.41	15083.565
kmc	k25	Minipig	15927.13	10156.220
gerbil	k25	Minipig	28691.98	2412.909
hackgap-s9	m2	GIAB	58216.93	56500.38
hackgap-s9	m3	GIAB	58482.28	56502.34
hackgap-s9	m4	GIAB	58439.08	56499.67
hackgap-s9	k25	GIAB	54730.04	56499.55
hackgap-s17	k25	GIAB	56847.42	56540.94
kmc	k25	GIAB	10913.06	52422.50
gerbil	k25	GIAB	error	error

most of the time. The measured numbers are given in Table 4. However, we must point out, although our CPU times seem comparatively high, the busy waiting uses the energy-efficient SSE2 `pause` instruction (x86_64, amd64) or spin lock `hint` instruction (arm64), so the energy consumption is less than that of an actually busy CPU.

We also noted that KMC3 tends to use all of the memory assigned to it, Gerbil uses much less memory (as reported by `/usr/bin/time -v`, maximum resident set size).