RapidGKC: GPU-accelerated K-mer Counting

Yiran Cheng[1], Xibo Sun[1], Qiong Luo[1,2]
[1]*The Hong Kong University of Science and Technology*
[2]*The Hong Kong University of Science and Technology (Guangzhou)*
{ychengbu, xsunax}@connect.ust.hk, luo@cse.ust.hk

***Abstract*—Many bioinformatics applications, e.g., genome assembly, genome profiling, and sequence alignment, break biological sequences into k-mers, or length-k substrings, for subsequent processing. In these applications, counting the number of occurrences of distinct k-mers is a common but expensive step due to the data and computation intensity. As such, prior work proposed to parallelize this task and utilize GPUs for further acceleration. However, these solutions under-utilize the GPU parallelism because the encoding format of intermediate data forces sequential decoding. To address this problem, we design a new encoding scheme for variable-length genomic data to support parallel encoding and decoding. Furthermore, we propose a novel rule to select common substrings among k-mers for partitioning, reducing the space cost as well as facilitating efficient parallel processing. Finally, we parallelize the entire workflow of partitioning and counting through pipelining, CPU-GPU co-processing, and work stealing. As a result, RapidGKC, our end-to-end GPU-accelerated k-mer counting system, outperforms state-of-the-art CPU-based and GPU-accelerated methods on real-world datasets.**

***Index Terms*—bioinformatics, GPU acceleration, k-mer counting, high-throughput sequencing, multi-GPU, parallel processing**

I. Introduction

K-mers refer to length-K substrings of a sequence, such as a *read*, a genome sequence fragment produced from a sequencing machine. Figure 1 shows an example read and its k-mers ($k = 9$ in this example). Counting the number of occurrences of each unique k-mer in biological sequences, or *k-mer counting*, is a fundamental step in bioinformatics applications [1] [2], including de novo assembly, genome profiling, sequence feature extraction, sequence alignment, and read error correction [3] [4] [5] [6].

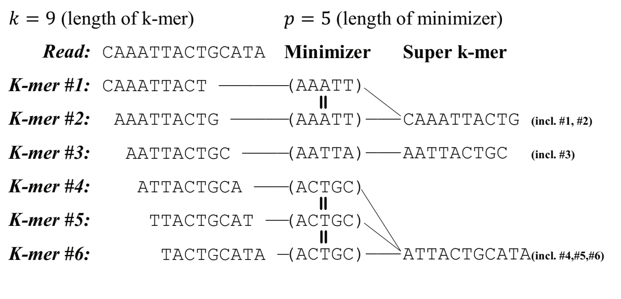


Fig. 1. A read, its k-mers, k-mers' minimizers, and generated super k-mers.

With the development of third-generation sequencing technology [7], both the *read* length and the sequencing system throughput are increasing [8]. These increases demand more computation power and space for k-mer counting tools. Currently, most k-mer counting methods run on central processing units (CPUs), which are limited in the number of cores and performance growth [9]. In comparison, graphics processing units (GPUs), as general-purpose manycore processors, possess massive thread parallelism and rapid technological advances. They have been adopted in many bioinformatics and computational genomics workflows [6] [10] [11]. A couple of existing k-mer counting tools [12] [13] have also employed the GPU; however, our experiments show that the accelerations are limited, possibly due to ineffective systematic and algorithmic design for the GPU. Therefore, in this paper, we identify the main issues in the k-mer counting workflow that hinder GPU acceleration, address these challenges with new designs, and implement RapidGKC, a prototype GPU-accelerated k-mer counting tool.

Current k-mer counting methods usually split all k-mers into disjoint partitions as intermediate results and then count unique k-mers in each partition. This *partitioning-and-counting* workflow is adopted because the entire data set may not fit into memory but each partition can be sized to. Furthermore, since all duplicates of a k-mer must be in the same partition, k-mer counting can be completed within each partition.

A commonly adopted partitioning scheme in k-mer counting is the minimum substring partitioning (MSP) method [14]. Rather than directly storing k-mers in partitions, MSP stores *super k-mers* by combining *consecutive k-mers* that satisfy certain conditions to reduce the space cost. As shown in Figure 1, a *super k-mer* is a substring of a read that contains multiple k-mers. These k-mers start at consecutive positions in the read and share a *minimizer* of length p ($p<k$). A length-p *minimizer* among a set of length-p strings is the alphabetically least one in the set.

State-of-the-art k-mer counting methods, such as CPU-based KMC2 [15] and KMC3 [1], and GPU-accelerated Gerbil [12], all adopt MSP. However, their encoding methods for variable-length super k-mers simply store the length and bit string of each super k-mer, requiring complex processing if extracting k-mers in parallel. To address this problem, we propose a super k-mer encoding method that uses the two leading bits of each byte to indicate the number of characters encoded in the byte. Our method supports high throughput encoding and decoding on the GPU, resolving a major issue in GPU-accelerated k-mer counting [13].

A drawback of the conventional MSP approach is that the super k-mer partitions may be of uneven sizes, so KMC2 [15] uses *signatures* instead of minimizers as the common substrings between consecutive k-mers to identify super-kmers. However, the function that enforces signature rules contains a lot of branch statements to transition between states, and the resulting branch divergence degrades the GPU performance [16]. To investigate this issue, we propose a model to estimate the effects of minimizers on the generated super k-mers. Based on the modeling result, we design a novel signature rule that runs on the GPU with lower time complexity and achieves comparable or even more space savings than the existing signature rule in KMC2. Moreover, our proposed signature rule is generic and can be applied to other CPU or GPU-based methods to enhance performance.

With novel designs on signature rules and encoding schemes addressing pivotal issues of MSP-based k-mer counting methods, we develop RapidGKC, an end-to-end GPU-accelerated k-mer counting system. Different from existing solutions, our system parallelizes both the partitioning and counting phases on the GPU with kernel programs optimized for the GPU architecture. In particular, rather than using GPU-based hash tables for counting, our system adopts GPU-based radix sort for better elapsed time performance. Furthermore, within each phase, our system distributes the workloads to both CPU and GPU for parallel processing, overlaps the IO and in-memory processing, and supports multi-GPU processing.

To evaluate the performance of RapidGKC, we select as baselines three state-of-the-art methods with the lowest time or space cost and conduct experiments on four datasets from different sequencing technologies. The results demonstrate that RapidGKC outperforms these CPU-based or GPU-accelerated methods in processing speed by up to four times. Our proposed signature selection method reduces the total size of super k-mers compared with the canonical minimizer in MSP and the signature proposed by KMC2, and runs efficiently on the GPU.

In summary, our work makes the following contributions:

- We identify and address significant performance issues in existing k-mer counting methods that affect GPU acceleration, including
  - A new encoding scheme for short variable-length genomic data fragments that supports fast parallel encoding and decoding.
  - An optimized signature rule to reduce the intermediate space cost of traditional MSP, running at constant time complexity on the GPU.
- We develop an end-to-end GPU-accelerated k-mer counting system with the following features.
  - Optimized GPU kernels for partitioning and counting phases respectively.
  - Pipelined parallelism between the IO and in-memory computing, CPU-GPU co-processing and multi-GPU support.

Additionally, our system can be integrated as a component into other genomic applications, such as de novo assembly and sequence alignment, to accelerate the overall performance.

II. Background and Related Work

A. Preliminaries

Definition 1 (Read): A *read* refers to a string S over an alphabet $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. S is 1-based indexing and $|S|$ denotes the length of S. $S[i, i']$ denotes the substring of S from position i to i' (both ends inclusive), where $1 \leq i \leq i' \leq |S|$.

Definition 2 (k-mer): A *k-mer* refers to a length-k substring of a read S, namely $S[i, i+k-1]$. The set of all k-mers from S is denoted as $M = \{S[i, i+k-1] \mid i \in [1, n-k+1]\}$. We say two k-mers $S[i, i+k-1]$ and $S[i', i'+k-1]$ are *consecutive* if $|i'-i| = 1$.

Definition 3 (Minimizer): A *minimizer* of a k-mer $S[i, i+k-1]$ is a length-p substring s_p that is lexicographically minimal among all length-p substrings of the k-mer.

Definition 4 (Signature): The set M_c of a k-mer $S[i, i+k-1]$ contains all length-p substrings s of the k-mer such that $f(s) = True$, where f is a function that maps a string to a boolean value. The signature of a k-mer $S[i, i+k-1]$ is the lexicographically minimal length-p substring $s_c \in M_c$. We say f is a *signature rule function*.

Definition 5 (Super k-mer): A *super k-mer* refers to a substring $S_{super} = S[i, i']$ of a read S of length n such that (1) All the k-mers of S_{super} have the same minimizer s_p (or signature s_c); and (2) The minimizer (or signature) of the k-mer $S[i-1, i+k-2]$ (if $i > 1$) and $S[i'-k+2, i'+1]$ (if $i' < n$) must be different from s_p (or s_c).

B. Sequencing Technologies and K-mer Counting Applications

Short reads from previous sequencing technologies are tens to hundreds of base pairs long [17]. In comparison, long reads, generated by the third-generation single-molecular sequencing technology (TGS), are thousands to tens of thousands of base pairs in length [18]. Long reads contain more consecutive information and are increasingly used in genomic analysis applications.

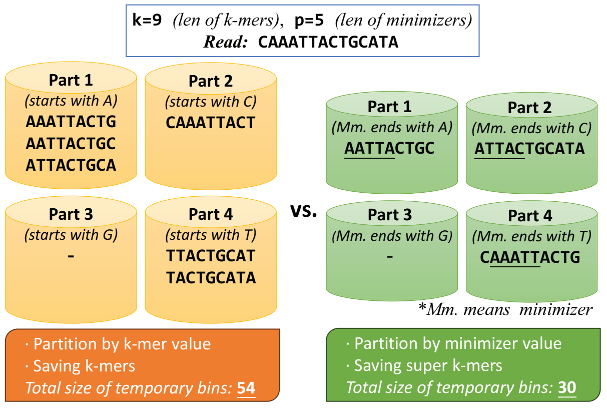
K-mer counting is widely employed in bioinformatics applications. For instance, in short read de novo assembly, unique k-mers are extracted as vertices in the assembly graph and duplicates counted in each vertex [19] whereas some long read de novo assemblers, e.g., Wtdbg2 [3], use k-mers and their counts to facilitate segment matching. In read aligners, such as DALIGN [20] and Minimap2 [21], k-mers are identified as anchors in the reads to limit the search space. Furthermore, some genome profiling tools use results from existing standalone k-mer counting tools, such as KMC3, to estimate complex genomic characteristics. Such genome profiling facilitates researchers studying genetic variations across diverse species [4].

No matter long or short reads, a genome dataset can reach up to hundreds of gigabases, and may consume over terabytes of memory to store all k-mers. Thus, k-mer counting tools, including our RapidGKC, take the partitioning-and-counting approach to reduce peak memory usage.

C. GPU Architecture and Branch Divergence

A GPU is a manycore processor that runs a massive number of parallel threads in the SIMT (Single Instruction, Multiple Thread) style. In a GPU, threads are organized in groups, or *thread blocks*, and the minimum scheduling unit is called a *warp*, or 32 threads. Because threads execute the same instruction on different data items, they may divert in subsequent instruction paths if the current instruction is a conditional test. If such *branch divergence* happens to threads within a warp, the execution will go branch by branch sequentially because a warp can only run the same instruction at a time, due to the GPU architectural design. Therefore, GPU kernels should minimize branch divergence, in particular, *warp divergence* for performance improvement. [16]

D. Minimum Substring Partitioning



k=9 (len of k-mers), p=5 (len of minimizers)
Read: CAAATTACTGCATA
Part 1 (starts with A) AAATTACTG AATTACTGC ATTACTGCA
Part 2 (starts with C) CAAATTACT
Part 3 (starts with G) -
Part 4 (starts with T) TTACTGCAT TACTGCATA
vs.
Part 1 (Mm. ends with A) AATTACTGC
Part 2 (Mm. ends with C) ATTACTGCATA
Part 3 (Mm. ends with G) -
Part 4 (Mm. ends with T) CAAATTACTG
*Mm. means minimizer
· Partition by k-mer value · Saving k-mers Total size of temporary bins: 54
· Partition by minimizer value · Saving super k-mers Total size of temporary bins: 30

Fig. 2. Minimum substring partitioning stores super k-mers, requiring less space than storing k-mers.

The minimum substring partitioning (MSP) [14] is a representative technique supporting memory-efficient partitioning-and-counting methods for bioinformatics data. Unlike previous methods of splitting and storing k-mers, MSP generates and stores super k-mers. MSP groups *consecutive k-mers* with the same *minimizer* in a read into a super k-mer to reduce the partition data size (Definition 2, 3). An example is shown in Figure 1. The given read contains six k-mers ($k = 9$) in total. The first and second k-mers have the same length-p ($p = 5$) *minimizer* (Definition 3) of "AAATT", so they can be grouped as a *super k-mer* (Definition 5). The third k-mer will be regarded as an independent super k-mer since no k-mer close to it has the same minimizer. The fourth to the sixth k-mers all have the same minimizer ("ACTGC") to be grouped as a super k-mer.

Figure 2 shows a comparison of partitioning by k-mer and with MSP using the example in Figure 1. It shows that partitioning and saving k-mers require 54-base storage whereas partitioning by minimizer and saving super k-mers require only 30-base storage. MSP is also the major technique in many other k-mer counting tools [1] [15] [12] [13] [22] and our RapidGKC.

E. From Minimizer To Signature

However, according to a previous study [15] , with the canonical minimizer (Definition 3), the generated partitions are of skewed sizes, and the largest partition size is much higher than the average. The main reason is that traditional MSP methods define the minimizer as the alphabetically smallest among a set of fixed-length strings [14] [23], so the string "AAA. . . A" is often selected as the minimizer for genomic strings of "A", "T", "C", and "G" characters. As a result, the super k-mer partition corresponding to the all-"A" minimizer is often the largest. In general, strings commencing with multiple "A"s will likely be selected as minimizers due to their lexicographical order. Our evaluation will show that such minimizers result in shorter super k-mers and larger total sizes. To balance partition sizes and minimize the total size of super k-mers, KMC2 [15] first used *signatures* (Definition 4) with specified rules instead of minimizers to generate super k-mers. Specifically, the signature rule function discards candidate length-p substrings that start with "AAA" or "ACA" or/and contain "AA" at any position except the beginning. Our experiments find applying signatures can reduce the total size of super k-mers by approximately 11% on MSP. Signature is also used in some other k-mer counters to reduce the temporary file size or the communication cost [12] [24]. However, this rule is quite complex to implement on the GPU efficiently. It is because determining whether there are "AA" in the length-p substring either relies on the result of the previous substring or a for-loop is needed on the current substring. The former solution limits the large-scale parallelism, while the latter brings higher complexity and more branches to the GPU. Therefore, our study proposes to adopt the partitioning-and-counting pattern with a faster signature rule for the GPU, aiming to achieve higher efficiency on the GPU than the existing solutions with comparable memory saving.

F. Super K-mer Encoding

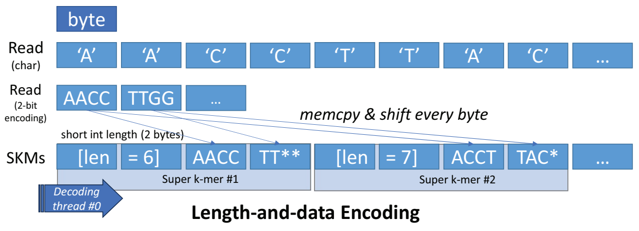


Fig. 3. Length-and-data super k-mer encoding (SKM: super k-mer).

Current tools adopt a "length-and-data" format to encode super k-mers along with their lengths in a consecutive area, as illustrated in Figure 3. It is a variant of the variable length encoding (VLE) method. Specifically, each base (A, C, T, or G) is encoded as two bits and the super k-mer length is encoded as an integer (e.g., a 2-byte short integer in the figure). Since the 2-bit encoded super k-mers are not byte-aligned, the control bytes store the super k-mer length in the units of bases (2 bits) rather than in the unit of bytes. This encoding method effectively saves storage space and is also suitable for sequentially decoding to extract k-mers. However, it is complex to parallelize such encoding and decoding on variable-length super k-mers because all super k-mers are stored consecutively, and the starting offset of each super k-mer must be computed through a scan from the beginning. Current GPU-accelerated methods, such as GPU-KMC2 [13] and Gerbil [12], continue to rely on the CPU to carry out super k-mer encoding and decoding.

Modern SIMD VLE methods have been developed to enable parallel encoding and decoding, such as Stream VByte [25]. Similar to the Compressed Sparse Row (CSR) format, Stream VByte stores the elements and their lengths in two separate arrays, and each element and its length are stored at the same position index in its corresponding array. Using extra locking or mutex to synchronize the length and data arrays is complex and costly on the GPU. Another line of work related to the compression on GPUs combines FLE (Fixed-length encoding) with VLE, but it introduces extra storage costs [26]. Therefore, we propose an alternative encoding method that facilitates the parallel super k-mer generation and k-mer extraction on the GPU.

G. Existing K-mer Counting Methods

A benchmark study [2] categorized k-mer counting methods based on their primary data structures and algorithms. In Table I, we select five representative existing methods that excel in time performance and/or space cost. Among the six methods in the table, four (MSPKC [23], Gerbil [12], KMC3 [1], and our RapidGKC) take the partitioning-and-counting approach. Both MSPKC and Gerbil use hash tables for counting and store intermediate results on disk. Gerbil [12] further supports GPU-based counting and estimates the hash table size using a simple linear model. In comparison, KMC3 [1] supports both in-memory and on-disk storage options for intermediate results and uses radix sort for counting. Our RapidGKC supports both disk and memory storage options for intermediate results and uses radix sort for counting. We further utilize the GPU in both the partitioning and counting phases.

Among the two non-partitioning methods, CHTKC [27] utilizes a lock-free hash table and employs an efficient memory layout to handle high sequencing depth data effectively. Squeakr [28] uses a counting quotient filter (CQF) and inserts each k-mer directly into the table. The CQF is stored in the memory, so if the table size exceeds the main memory capacity, the performance will significantly slow down due to swapping.

Gerbil uses GPU-based hash tables to accelerate the counting phase but does not accelerate the partitioning. It does not provide a suitable super k-mer encoding method for GPU parallel access. Our experiment results show that the performance improvement of Gerbil is insignificant with GPU acceleration.

The challenges in GPU-based hash tables include accurate estimation of the dynamic table size, costly dynamic memory allocation on the GPU, and control divergence in parallel hash table lookups. Furthermore, some post-counting applications require not only the counts but also additional information, such as the positions of each k-mer. However, GPU-based hash tables usually only support fixed entry sizes. Such hash tables are acceptable in some short read applications such as de Bruijn graph construction for short read assembly [29]. However, in long read analysis applications, the hash table entry size may vary and requires extra reallocation cost.

Some recent work has investigated further optimization methods on GPU-based lock-free hash tables with advanced modular design for domain-specific problems, parallel probing, and dynamic reallocation strategies [30] [31]. Therefore, we also compare the k-mer counting performance with our GPU radix sort strategy and with a representative GPU-based counting hash table, as shown in Figure 12.

RapidGKC uses a GPU-based radix sort with a few other parallel primitives (such as parallel run-length encoding) to accelerate the counting step. In addition, we propose a new memory layout for accessing super k-mer data on the GPU, refine the signature rules for partitioning, and design the workflow to fully utilize the GPU. Specifically, rather than only accelerating the counting step only, our RapidGKC can execute partitioning, super k-mer encoding, decoding, and counting tasks all on the GPU. Furthermore, our work can be extended and adapted to accelerate post-counting applications such as genome assembly and sequence alignment.

III. Method

A. New Signature Selection Rule

Existing work applies signatures (Definition 4) to MSP for reducing the space cost and size imbalance of intermediate super k-mer partitions. Specifically, among a set of fixed-length substrings of a k-mer, they first remove all substrings that have two consecutive “A”s except if “AA” are the first two characters of the string, and they also discard all substrings that start with “AAA” or “ACA”. However, checking if there are any occurrences of “AA” in all length-p strings is complex. The conditional statements may lead to branch and warp divergence on GPUs, impeding the time efficiency. Moreover, on the CPUs, it can check the current substring using the result from the previous one within $O(1)$ time complexity. However, this approach relies on the result of the previous items, and it cannot be done in parallel, so it is potentially expensive on the GPU since it runs with a for loop of $O(p)$ on the GPU. Hence, we design a more efficient signature rule for MSP on the GPU to get comparable partition sizes reductions of the existing signature rule while reducing the time complexity to $O(1)$ running on the GPU kernel.

The previous research proposes the signature rule from simple intuitions and heuristics [15]. To illustrate the rationale of our signature rule, we first establish a criterion for evaluating the contributions of different minimizes to reducing the total size of generated super k-mers. When two different minimizers from two consecutive k-mers are close to each other’s positions within a read, the resulting super k-mers will likely be shorter,

TABLE I
COMPARISON OF EXISTING K-MER COUNTING METHODS WITH OUR RAPIDGKC.

Name	Partitioning Method	Counting Method	GPU Acceleration	To-disk / In-memory
Squeakr	N/A	Quotient filter	CPU only	In-memory
CHTKC	N/A	Hash table	CPU only	To-disk when memory insufficient
MSPKC	MSP	Hash table	CPU only	Disk-based
KMC3	MSP (signature)	Radix sort	CPU only	Both supported
Gerbil	MSP (signature)	Hash table	Counting phase only	Disk-based
RapidGKC	MSP (improved signature)	Radix sort	All (partitioning and counting)	Both supported

and the number of k-mers included in a super k-mer will be small. Therefore, such minimizers should be discarded by the signature rule.

To implement this criterion, we propose an approximate minimizer evaluation metric, denoted as S_m, which we refer to as the S-score. Given a minimizer value m, $d_{m,n}$ denotes the distance between a minimizer of value m and the next super k-mer's minimizer value n ($m \neq n$), and $\overline{d}$ denotes the average distance between all such neighboring minimizers in the dataset. Then, S_m is defined as:

$$S_m = \sum_{n \in Neighbor(m)} (d_{m,n} - \overline{d}) \tag{1}$$

When the S-score of a minimizer is positive and large, it indicates that this minimizer is evenly distributed among the reads and contributes to long super k-mers. Conversely, when the metric value is small and negative, it suggests that the minimizer is more likely to result in shorter super k-mers. By employing this metric, we can determine what minimizers are worth selecting and what should be discarded, thus improving the efficiency of our new signature rule.

Figure 4 shows some minimizers with the largest and the smallest metric values in a small dataset as examples. These examples serve to illustrate the rationale underlying the rules of signature. The guiding principle is to discard minimizers with small S-scores and retain those with large S-scores. For example, as Figure 5 shows, when there are multiple consecutive "A"s in a read, it is more likely that the minimizers of two consecutive k-mers overlap in the read, which causes the generation of short super k-mers. In this example, there are four "A"s at the beginning of the read, and the minimizer of the

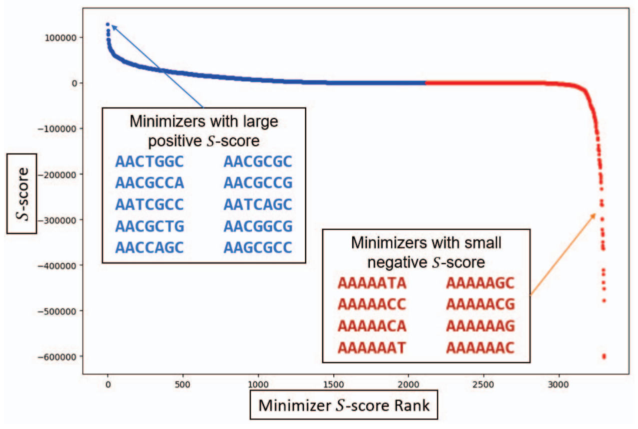


Fig. 4. S-score distribution on a dataset with example minimizers.

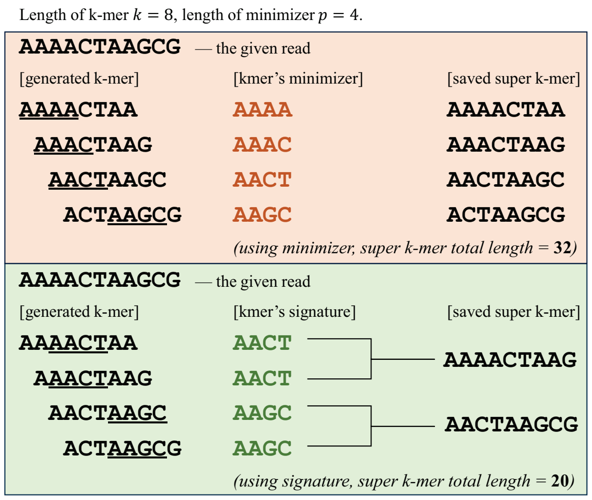


Fig. 5. Comparison of conventional minimizer and signature with their generated super k-mers.

first k-mer is "AAAA" since it is the substring with the smallest lexicographical order. In comparison, the second k-mer starts with three "A"s, so it will select "AAAC" as its minimizer. Such a scenario continues when there are multiple "A"s in the read, causing consecutive k-mers to have different minimizers and resulting in more super k-mers shorter in length, therefore costing more intermediate data space. To address this issue, the original signature and our proposed signature rules disqualify such minimizers. Note that no matter what signature rules are used, a set of super k-mers will generate the same set of k-mers, because signature rules only change the cutting positions of the reads but do not change the data in the reads.

Since there is no theoretical method to quantify the efficacy of different signature rules, we empirically adjust the existing signature rule inspired by the result of the S-score test and the criteria described in [15]. After trying and evaluating a number of candidate rules, we propose our refined signature rule as follows in C style:

$(m >>(p\text{-}3)*2 \ \& \ 0b101011) \ != 0 \ \&\& \ (m \ \& \ 0b111111) \ != 0$

Where p is the length of the minimizer in number of bases, m is the 2-bit encoded minimizer that for the bases, $g(\mathrm{A}) = 00_{(2)}$, $g(\mathrm{C}) = 01_{(2)}$, $g(\mathrm{G}) = 10_{(2)}$, $g(\mathrm{T}) = 11_{(2)}$, and C_i is the i-th base of the minimizer. $m = \sum_{i=1}^{p} g(C_i) * 4^{(p-i)}$, $C_i \in \{\mathrm{A, C, G, T}\}$.

The proposed rule means no "AAA", "ACA", "CAA", or "CCA" at the beginning, and no "AAA" at the end is allowed for a signature. Therefore, the complexity of checking if a length-p substring is a signature is $O(1)$. We also experimentally compared the time and space cost of the conventional minimizer (Definition 3), KMC2 signature (Definition 4), and our proposed signature, as shown in Table III. The result shows that our design can achieve a lower space cost than the conventional minimizer and runs faster than the KMC2 signature on the GPU. Our profiling results, reported in section IV-C, also indicate that our signature rule eliminates branch divergence on the GPU whereas the KMC2 signature incurs significant branch divergence.

B. Super K-mer Encoding for Parallel Access

As introduced in the background section, the existing length-and-data encoding for super k-mer only supports sequential access and decoding. When accessing partitioned super k-mers with GPU, each thread cannot determine the split positions between two super k-mers. This issue is a significant negative factor that affects the time efficiency when introducing GPU acceleration [13]. According to our in-memory experiment with GPU acceleration, if only using the CPU to decode the super k-mer, the decoding time will contribute to over 30% of the total time. It will also increase the data copy time due to transferring a lot of decoded data (k-mers) from CPU to GPU for subsequent processing on the GPU. Some modern VLE methods and the traditional CSR format can support parallel access. However, these encoding methods store the lengths and data into separate arrays, thus requiring costly locking or synchronization on the GPU during encoding to ensure matching the data order between the two arrays.

Therefore, there is a pressing need for a new time-efficient super k-mer encoding method that can support GPU parallel access to alleviate the decoding time bottleneck. With our design, the GPU can access the variable-length super k-mers in parallel. It can reach an over 40x speed-up than single CPU thread decoding, thus lowering the decoding time below 5% of the total time. It can also help reduce the overhead of data copy from host to device by transferring super k-mers rather than k-mers.

In RapidGKC, we propose a super k-mer encoding method for efficient parallel encoding and decoding on the GPU. As illustrated in Figure 6, each encoded byte in our method contains two control bits and six data bits. The two control

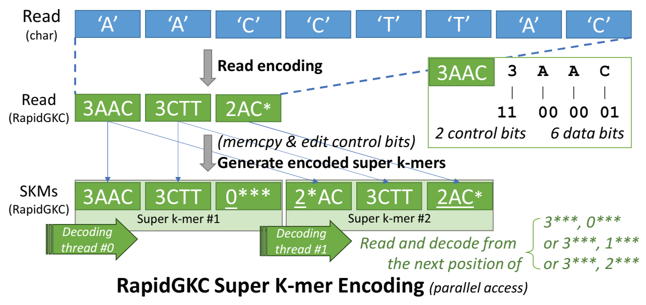


Fig. 6. The super k-mer encoding method supports parallel access, encoding, and decoding.

bits indicate the number of effective bases in the byte (from 0 to 3), and the six data bits represent three bases, with every two bits representing one base (0, 1, 2, 3 for A, C, G, and T, respectively). A byte with control bits equal to 3 ($11_{(2)}$) is referred to as a *full byte*, whereas a byte with control bits of other values (0, 1, 2) is a *non-full byte*. First, original reads are encoded and then further converted into encoded super k-mers (SKMs). In generating an encoded super k-mer, all bytes of the super k-mer are copied from the encoded reads, and the control bits of the first and last bytes are modified to exclude the bases from the preceding and following super k-mers, respectively. An *empty byte* (0) is added at the end if the super k-mer ends with a full byte. As a super k-mer's length is at least five bases in practice, a non-full byte following a full byte indicates the last byte of a super k-mer. Therefore, during decoding, each thread can identify a non-full byte after a full byte and commence processing the super k-mer starting from the following byte of the non-full byte. The reason for using a byte as a unit rather than a longer word (e.g., 32-bit integer) is that the longer word cannot support shorter k-mers and super k-mers, and they will be likely to waste more data bits preceding and following the super k-mers. Also, our proposed encoding has the advantage of supporting a non-full byte as the beginning of a super k-mer, thereby eliminating the need to shift every byte of the super k-mer from the 2-bit encoded reads.

In summary, this method facilitates parallel encoding, decoding, and k-mer extraction, reduces encoding time by avoiding shifting, and supports consecutive storage of encoded super k-mers. It also reduces the memory copy time from the host to the device by transferring super k-mers rather than k-mers. This encoding design is general and can be easily extended for other short variable-length data types for parallel access. Also, it is simple to adjust this method to support base-n data by increasing the number of bits for each base. Decoding in the GPU also enables the support of modern storage technologies such as GPUDirect Storage [32], which enables direct access from GPU via the PCIe interface to the external storage to achieve a higher data loading bandwidth than conventional I/O.

C. GPU Accelerated System Design

Currently, there is a lack of a comprehensive and highly efficient end-to-end GPU-accelerated k-mer counting tool. This section presents our RapidGKC, a GPU-accelerated k-mer counting system, with a specific focus on GPU-related design aspects. Compared to the existing k-mer counting tools, we apply GPU acceleration to the entire workflow, including the partitioning and counting phases, parallel data encoding, and decoding. To achieve optimal performance, we integrate the novel designs of memory layout and improved MSP for GPU acceleration on k-mer counting into the system. Also, respecting the GPU architecture, we design and implement the system with pipelining, CPU-GPU coprocessing, and other optimization techniques.

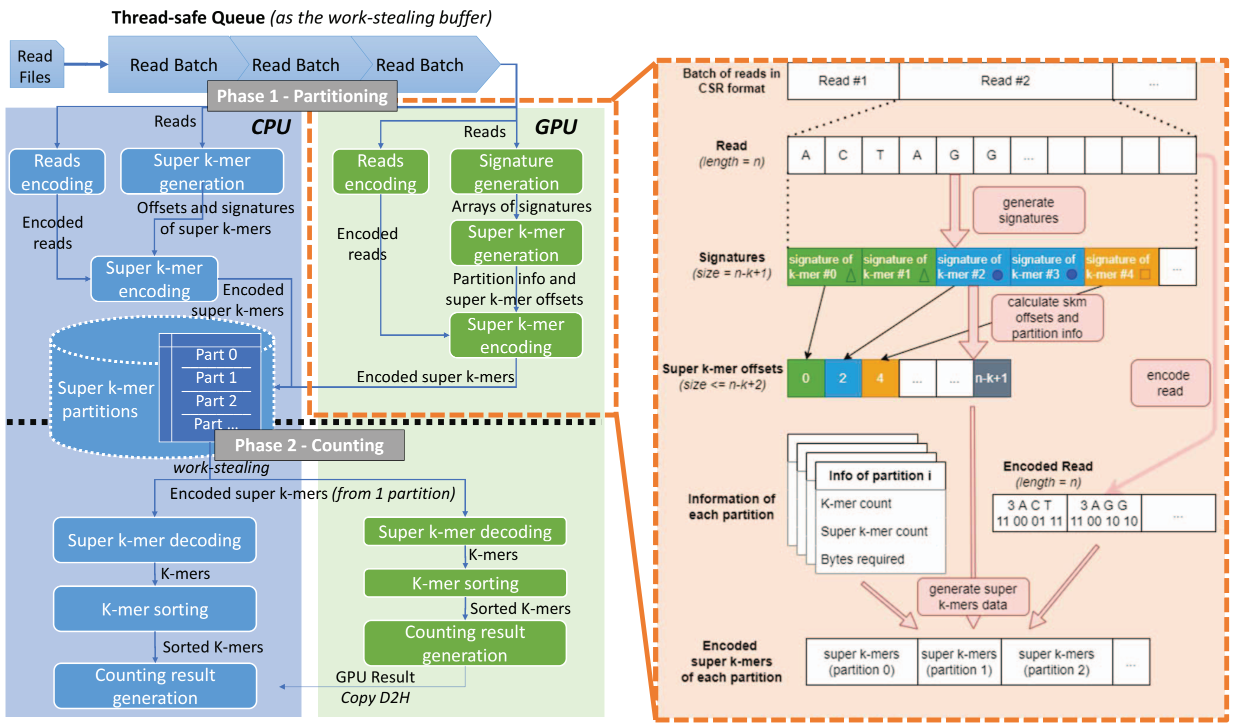


Fig. 7. The workflow of RapidGKC and GPU-based partitioning phase design.

1) System Overview: The workflow of RapidGKC is depicted in Figure 7. The workflow is pipelined with multi-threading and GPU streams for better concurrency, thereby overlapping the execution for higher hardware utilization. Initially, the asynchronous file loader extracts and loads the reads from the file. Thousands of loaded reads are grouped into a batch and pushed to a thread-safe work-stealing queue to balance the workloads. The CPU- and GPU- based splitters dequeue batches of reads from the buffer, generate super k-mers, and save super k-mer data to corresponding partitions in memory or on disk, depending on which storage option is taken. For in-memory counting, each super k-mer partition has a concurrent lock-free buffer that allows multiple threads to write data in parallel. If in the disk-based mode, the super k-mer data is written to multiple temporary files. Upon all reads are loaded and all super k-mers are saved, phase 1 finishes. In phase 2, each CPU/GPU worker loads all super k-mers from one partition at a time. Then the super k-mers are decoded to extract the k-mers. Subsequently, the workers uses radix sort to deduplicate the k-mers and obtain the counting result for each distinct k-mer. After retrieving the counting results, RapidGKC optionally filter out the k-mers based on the user-provided threshold of k-mer counts as the final results.

2) GPU-accelerated Partitioning: The current k-mer counting tools cannot offer comprehensive and well-performed GPU acceleration in the partitioning phase since they don't have a proper super k-mer encoding scheme, a low-time-complexity MSP signature rule design, and a proper CPU-GPU processing workflow design [12] [13]. Our RapidGKC addresses these issues with new encoding methods, the refined signature rule, GPU kernel designs for partitioning respecting the GPU architecture, and pipelining workflow design with CPU-GPU coprocessing.

The right side on Figure 7 shows the detailed works of GPU in the partitioning phase. The inputs are the reads, and the outputs are the intermediate super k-mers in different partitions. Initially, reads are loaded from files. Since each read is an independent variable-length data chunk, our design groups thousands of the reads as a batch and concatenate them in the CSR format in consecutive memory space. The batch size is set based on the number of GPU threads and the read length to fully use the GPU resource.

After loading the reads into GPUs, RapidGKC has four major GPU kernel functions for the partitioning. First, it generates the signatures of each k-mer. In this step, multiple GPU threads collectively process one read at a time so that their consecutive accesses to the bases in each read in the global memory can be coalesced to utilize the GPU memory bandwidth effectively. Every GPU thread processes one k-mer in the read and generates its signature. Then, the second GPU kernel function calculates the super k-mer offsets based on whether the signature is of the same value as the previous k-mer's. In this kernel, each thread processes one read to calculate the starting offsets of all super k-mers in the read. It also updates the k-mer counts, super k-mer counts, and numbers of bytes required of target super k-mer partitions. The information about the super k-mer partitions is in the global memory and can be updated with atomic operations by multiple threads. This information is kept for the current read batch until all super k-mer data of this batch are saved.

After that, the third GPU kernel encodes the character-based reads in our proposed encoding format. With the super k-mer offsets, the last GPU kernel processes the super k-mer data from the encoded reads and saves them in a temporary CSR array on the GPU. The offsets of the rows in the CSR array are calculated based on the size of each partition so that all super k-mers in one partition are saved consecutively. Finally, the temporary super k-mer array on the GPU is transferred to the CPU memory and saved.

3) GPU-accelerated Counting: Radix sort on CPU is proved to be effective in k-mer counting by KMC3 [1]. Recent research shows that GPU-based radix sort can bring remarkable speed-up compared to CPU sorting [33]. Our experiment in Figure 12 also suggests that the GPU radix sort solution is much efficient in k-mer counting. In this case, RapidGKC proposes to use GPU-based radix sort. In addition, we apply some other GPU parallel primitives and kernel functions working with GPU radix sort for extracting k-mers from super k-mers, preprocessing the k-mers, deduplication on the sorting result, and optionally filtering out k-mers with low occurances. In the experiment, We also compared our GPU radix sort strategy and a representative GPU-based counting hash table in counting speed to validate the efficiency of our GPU sort strategy.

In more detail, once all k-mers in the super k-mer partition are extracted, we can use GPU parallel primitives to sort and deduplicate the k-mers and count the number of occurrences of each distinct k-mer. As depicted in Algorithm 1, given the array of all k-mers extracted from a super k-mer partition (denoted as A with size n), we first use the parallel transform (map) to convert each k-mer to its canonical form (the alphabetically smaller one of a k-mer and its reverse complement) (Line 6). The transform operation will apply the given function to every element in the array in parallel. Then, we use the GPU radix sort to sort the canonical k-mers at Line 7. After that, we call the GPU-based parallel run-length encoding (RLE) on the sorted k-mers (Line 8), which will return two arrays, one of the items A' and the other of the run lengths S. Since we have sorted the k-mers before RLE, all the k-mers in A' after RLE will be unique, and the run-lengths of each k-mer indicate its counts in S. After these steps, A' and S can be used together to fetch each distinct k-mer and its count to the output.

4) GPU Memory Management: For GPU-accelerated computing, the memory allocation, deletion, and data transferring between the CPU (host) and the GPU (device) are significant factors affecting performance.

In the partitioning phase, millions of variable-length reads are copied from host to device. The effective bandwidth of transferring reads to the GPU significantly affects the upper bound of the partitioning phase time efficiency and throughput. *pinned memory*, or page-lock memory, is a kind of host memory that the GPU can access with high efficiency. RapidGKC groups the reads into batches and allocates the pinned memory for the entire batch. Then, organize the reads into CSR format in the pinned memory. Finally, copy the entire batch from the pinned memory to the GPU. To validate this design,

Algorithm 1: K-mer Counting with Parallel Primitives

```
   Input: An array $A$ of $n$ k-mers
   Output: $A'$: distinct k-mer list, $S$: k-mer count list
1  Function Canonical($x$):
       // $x$: 2-bit-encoded k-mer
2      $x' \leftarrow$ ReverseComplement($x$)
3      return $\min(x, x')$
       // type: 2-bit encoded k-mer
4  End Function
5  Function Counting($A$, $n$):
6      $A \leftarrow$ Transform ($A$, Canonical)
7      $A \leftarrow$ Sort ($A$)
8      $A'$, $S \leftarrow$ RunLenghEncoding(A)
       // It returns (deduplicated item list,
           length of each item)
9      return $A', S$
10 End Function
11 $A'$, $S \leftarrow$ Counting($A$, $n$)
   // For k-mer $A'[i]$, its count is $S[i]$
```

we measure the effective bandwidth of three approaches for copying data to the device. The first approach is copying each read with one calling. The second approach is allocating pinned memory for a batch of reads, then concatenating the reads in the pinned memory for copying to the GPU at a time. The third approach bypasses the allocation of pinned memory by reusing the allocated pinned memory from the last batch. As shown in Figure 8, organizing reads in the pinned memory and then copying the entire chunk can obtain the best performance. And it can reduce over 50% time cost than the first approach for one GPU-based partitioning task.

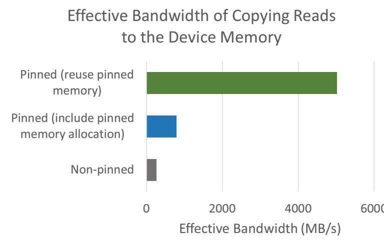


Fig. 8. The effective bandwidth of three approaches for copying reads to the GPU.

Aside from memory copying between host and device, device memory allocation and free are also time-consuming operations that affect 15% of the total time even with pipelining. Therefore, RapidGKC reuses the allocated device memory between multiple data batches and reallocates only when insufficient.

D. Additional Features

1) Asynchronous File Loading: In our experiments with file cache, RapidGKC can process the partitioning phase in memory with a throughput of over 2000 MB/s. This speed is already over the maximum I/O speed of 6Gbps in SATA 3.0, which means I/O will be the bottleneck of phase 1. To accelerate the file loading, we use an asynchronous file loader that pipelines the I/O, file format parsing, and reads copying. Figure 9 illustrates the time breakdown of the sequential read loader and the elapsed time of our pipelined read loader on a data file. FASTA and FASTQ are two formats of sequencing data while the latter contains not only the bases but also the quality score of each base, which is redundant in k-mer counting. Compared with sequential loading and processing, the pipelined mode can fully overlap the loading, parsing, and saving, reaching a higher throughput.

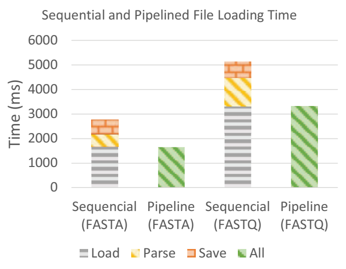


Fig. 9. The time breakdown of sequential and pipelined file loading.

2) *GPU Streams and Multithreading:* A significant target of GPU-based system design is to make better use of GPU. A *GPU stream* can be regarded as a sequence of GPU kernel functions and operations. The instructions from one GPU stream will be executed sequentially. In RapidGKC, multiple GPU streams are enabled to obtain high efficiency by overlapping the executions from different streams. The CPU is also involved in some tasks that are not computing-intensive before and after the GPU executions, such as data organizing, memory copying, and indicator statistics. The first row of Figure 10 illustrates a typical workflow of what a phase 1 GPU worker performs in super k-mer generation. In this example, we compare three simple strategies of applying CPU multi-threading on GPU streams. The optimal approach to fully utilize the GPU is to enable multiple CPU threads for multiple GPU streams. We also conduct an experiment to illustrate the effectiveness of different numbers of CPU threads assisting the GPU as Figure 11.

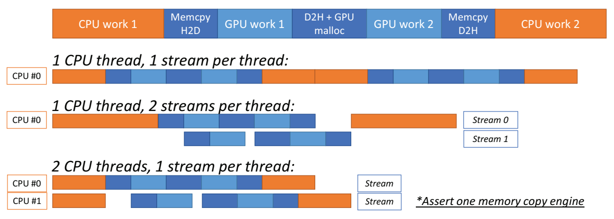

CPU work 1
Memcpy H2D
GPU work 1
D2H + GPU malloc
GPU work 2
Memcpy D2H
CPU work 2
1 CPU thread, 1 stream per thread:
CPU #0
1 CPU thread, 2 streams per thread:
CPU #0
Stream 0
Stream 1
2 CPU threads, 1 stream per thread:
CPU #0
CPU #1
Stream
Stream
*Assert one memory copy engine

Fig. 10. Workflow with GPU streams and CPU multi-threading.

3) *CPU Coprocessing:* When the CPU is close to the GPU in performance or the read lengths are too short for the GPU thread parallelism, CPU co-processing becomes beneficial in RapidGKC. As the leftmost column of Figure 7 shows, CPUs can perform the same task as the GPU independently. In our experiments with high-end GPUs and long-read datasets, we only enable the CPU counter as a backup to handle those large partitions that exceed the GPU memory capacity. The reason is that the GPU alone can achieve higher speedups than adding

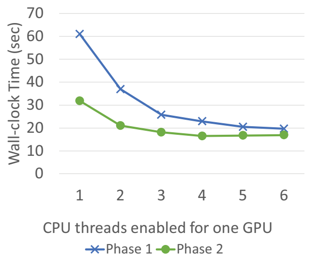


Fig. 11. Performance with a single GPU with the number of CPU threads varied.

the CPU workers, and CPU workers occupy memory space and bandwidth. Experiment results related to CPU-GPU co-processing are in Figure 14.

4) *Load Balancing and GPU Utilization:* To mitigate load imbalance or under-utilization, we adopt work-stealing between the CPU and the GPU as well as partition reads by their lengths [34] [35]. Specifically, RapidGKC employs a work-stealing queue in both phases to distribute the workload evenly among CPU threads and GPU streams. Furthermore, to balance the workload among counting workers in phase 2, we calculate a histogram of the signature with a small batch of reads. Subsequently, this histogram is leveraged to determine a suitable partitioning scheme of signatures. As lengths of reads exhibit variability within a range depending on the sequencing technology, in the partitioning phase, we employ a simple technique to handle variable-length reads. By loading multiple batches of reads at once and subsequently sorting them based on their lengths, each GPU kernel can process reads with similar lengths, thus reducing divergence when multiple threads generate super k-mer offsets of multiple reads. Furthermore, during signature generation, GPU threads effectively utilize coalesced access to bases. This approach enhances the effective bandwidth of global memory and promotes data locality, thereby improving overall performance [36].

IV. Evaluation

A. Experiment Setup

1) *Hardware:* We use two servers in our experiments. System 1 is a server with two Intel Xeon E5-2683 v4 CPUs (each of 16 cores 32 threads) and four NVIDIA GeForce RTX 2080 Ti GPUs (11GB VRAM). The main memory is 256GB. The disk for read files and temporary files is a RAID0 system with two SATA SSDs. System 2 is a newer server with two AMD EPYC 7302 CPUs (each of 16 cores 32 threads) and four NVIDIA GeForce RTX 3090 GPUs (24GB VRAM). The main memory is also 256GB. The disk for read files and temporary files is an NVMe SSD.

To mitigate the impact of IO fluctuations and accurately measure the throughput, on system 1, we executed each experiment multiple times to warm up the file cache and then obtained stable time measurements. Such a warm cache setting is common in practice.

TABLE II
DATASETS USED IN OUR EXPERIMENTS.

Dataset	**F. vesca**	**N. crassa**	**HG002**	**HG00733**
File Format	FASTQ	FASTA	FASTA	FASTA
# Reads	12.8 M	2.9 M	6.6 M	14.2 M
Avg Read Len.	353	7778	13478	19642
# Bases	4.5 G	22.9 G	89.1 G	279 G
File Size	8.9 GB	21.3 GB	83.3 GB	260.5 GB
28-mer Depth	6.54	1.05	15.10	1.13

2) Datasets: We selected four datasets from different sequencing technologies, as listed in Table II, for our experiments. F. vesca is a short-read dataset; the other three are third-generation sequencing long-read datasets. Moreover, N. crassa and HG00733 have low *k-mer depths* (the ratio of total k-mers over distinct k-mers). A lower k-mer depth means more distinct k-mers. Each of the three smaller datasets and its super k-mer data can fit into the 256GB main memory, so on system 1, we conduct the experiment with a warm start (with the read file cached from warm-up runs). The HG00733 dataset is too large to fit in the main memory, so on system 2, all experiments are started with no data cached.

3) Baselines: We selected three representative k-mer counting methods to compare with RapidGKC. Specifically, KMC3 is a sorting-based method supporting both in-memory and to-disk modes. Gerbil is a hash-table-based method supporting GPU acceleration in the counting phase. CHTKC is a hash-table-based multi-pass counting method. Previous studies have shown that these three outperformed other k-mer counting tools in time and/or space performance [2].

We set the number of CPU threads for all k-mer counting methods to 16, 32, 48, and 64, and recorded their best time performance. We kept the default settings for each algorithm unless specified otherwise. CHTKC requires a given memory space size, and we set it to the maximum peak memory usage of all other methods. The time of CHTKC on dataset HG00733 is over 3600s, so we discard its result from Figure 13.

B. Comparison of GPU radix sort and GPU-based hash table in counting k-mers

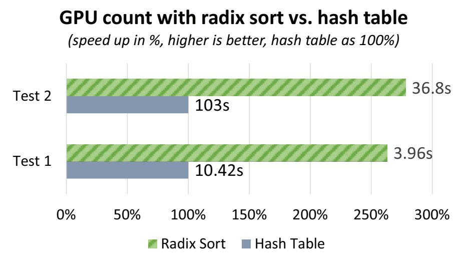


Fig. 12. Counting speedups of GPU radix sort vs. GPU hash table.

To demonstrate the efficiency of GPU radix sort based k-mer counting, we conducted a performance comparison between GPU radix sort and a recently proposed GPU-based counting hash table [30]. We integrated the hash table into our prototype and conducted two tests with datasets and systems varied to evaluate their counting speeds. Test 1 involved a small dataset with a k-mer depth of 1.5, utilizing one GPU. Test 2 was conducted on dataset HG002 with a k-mer depth of 15.1, employing two GPUs. The results clearly indicate that our proposed GPU radix sort solution outperforms the GPU-based hash table counting by a factor of 2.5 in counting speed.

C. Comparison of Conventional Minimizer, Signature, and Our Improved Signature

The experiment in this section assesses the effectiveness of our proposed signature selection rule in reducing the time cost, balancing the partition size, and reducing the total size of super k-mers. We compared our proposed signature rule in the partitioning phase with canonical minimizer [14] and existing signature rules [15]. In this experiment, only GPU partitioning functions were enabled. We evaluated both the total size and count of generated super k-mers, the maximum partition size, the partitioning phase time, and the signature rule kernel function time with corresponding rules. (Two GPUs run the kernel function with multiple streams, so the time cost in the table may be larger than the overall phase 1 time.) The conventional minimizer [14] (listed as *"Canonical"*) is the lexicographically smallest length-p substring of a k-mer without any other requirement as in Definition 3. The signature rule (Definition 4) listed as *"Signature"* is proposed by KMC2 [15] and also used by KMC3 [1] and Gerbil [12]. *"RapidGKC"*, listed in the table, denotes our newly proposed signature rule in this study.

Table III shows that our rule runs as fast as the canonical minimizer on the GPUs and achieves the smallest total size of super k-mers. Moreover, we measured the branch efficiency of the GPU kernels with Nvidia Nsight Compute profiling tool. The branch efficiency of KMC2 signature is only 61.41% while our proposed signature has a 99.99% branch efficiency, which means there is nearly no branch divergence in our GPU signature generation kernel. These findings suggest that our solution addressed the efficiency issue of the existing signature rule running on the GPU and the space cost issue of the traditional canonical minimizer. Our proposed signature rule can help reduce the disk space or total memory requirement of the minimum substring partitioning approach for genomic data, reduce the branch divergence and run efficiently on both the CPUs and GPUs. Therefore, our proposed signature selection rule is an advanced alternative that can potentially enhance the performance of k-mer counting methods for various applications.

D. K-mer Counting

Figure 15 illustrates the k-mer counting performance in time (in seconds) and space (in GB) cost of four methods in-memory or to-disk on three datasets on system 1. On system 1, all read files are cached in main memory to avoid the I/O speed fluctuation and evaluate the peak throughput of each method. Figure 13 illustrates the k-mer counting time on dataset HG00733 for three different methods on system 2. This dataset is bigger than the main memory. To ensure stable cold start time, the file cache is cleared before each run. Consequently,

TABLE III
TIME AND SPACE COST OF MINIMUM SUBSTRING PARTITIONING WITH MINIMIZER, SIGNATURE, AND OUR MODIFIED SIGNATURE RULE.

Rule	**P1 time (s)**	**Kernel time (s)**	**Size (GB)**	**SKM ct. (M)**	**Max part k-mers (M)**
HG002	p=9	k=28			
Canonical	68.12	**53.42**	118.85	9798	915.32
Signature	90.34	194.19	105.60	8375	349.72
RapidGKC	**66.83**	55.31	**103.65**	**8166**	**283.74**
HG002	p=7	k=16			
Canonical	72.43	**35.58**	129.30	17653	1240.69
Signature	75.76	93.02	123.69	16673	**623.83**
RapidGKC	**67.11**	37.40	**115.54**	**15732**	787.42
N. crassa	p=9	k=28			
Canonical	23.13	**15.07**	29.93	2454	360.20
Signature	29.86	55.42	26.80	2118	**62.77**
RapidGKC	**22.73**	15.83	**26.74**	**2111**	66.21
N. crassa	p=7	k=16			
Canonical	**22.43**	**10.15**	32.21	4496	429.64
Signature	24.22	27.02	30.83	4249	89.87
RapidGKC	22.77	10.76	**29.94**	**4091**	**83.47**

the results of this experiment involve disk I/O. We measure the execution time of each method till the counting finishes, i.e., the list of distinct k-mers and their number of occurrences is generated in the main memory, excluding the result output time. In accordance with established practices [1] [30] [12], we have configured all the baseline methods to collect only the k-mers with a minimum number of occurrences of 2. This treatment is because, in real-world applications, k-mers with a count of 1 typically do not provide valuable information and are often the result of sequencing errors.

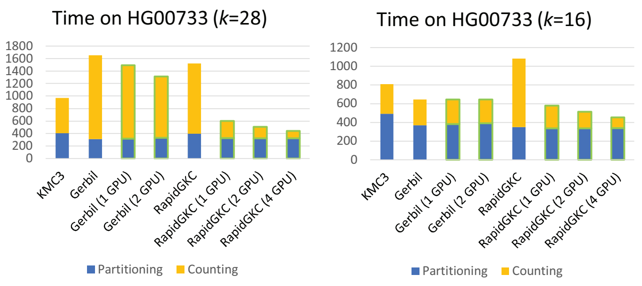


Fig. 13. K-mer counting on dataset HG00733 on system 2 (cold start without file cache, time: seconds).

The results show that with GPU acceleration, RapidGKC runs faster than the other methods, especially for the third-generation sequencing long-read datasets (a 5x speedup than the best of the others).

On system 1, in most cases, RapidGKC runs faster with in-memory mode with fewer file-writing system calls since all data can fit into the main memory. However, for F. vesca, k=28, as the short-read dataset with the minimum partition data size, it has more short memory fragments of super k-mers, thus, it will benefit most from the disk mode because the file writing call will concatenate the short fragments together for a faster sequential memory copy in phase 2.

On system 1, where the read files are cached, our work-stealing buffer monitor reveals that the bottleneck in phase 1 transitions from computation in CPU-only setups to parsing and loading reads in GPU-enabled setups. The experiment conducted on system 2 indicates that the phase 1 time bottleneck is primarily attributed to disk I/O without file cache, including reading the read files and writing the partition files. Moreover, in the counting phase, the scalability of multiple GPUs is limited due to the necessity of loading approximately 300GB of super k-mers from disk files.

The I/O cost of input reads and output super k-mers puts a limit on the phase 1 time. Therefore, the counting phase is usually more significant than phase 1. The experiment shows RapidGKC has significant time improvement in phase 2. The space cost by the intermediate data (super k-mers) of our method is comparable to KMC3 and Gerbil.

E. Scalability

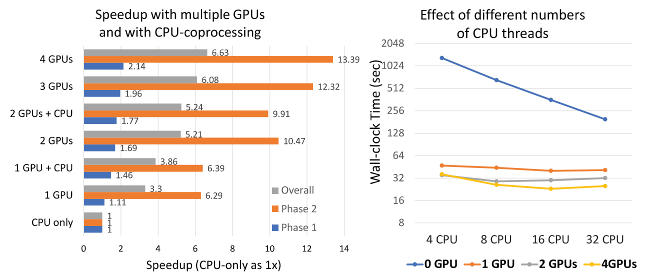


Fig. 14. Scalabilities on different numbers of CPU threads and GPUs.

RapidGKC can support multiple GPUs in both phases since the partitioning phase is parallelized among batches of reads and the counting phase among super k-mer partitions. We run RapidGKC on the same dataset with different settings to evaluate the scalability. The left sub-figure in Figure 14 shows the speedup of GPU acceleration and CPU-GPU coprocessing. The right sub-figure compares the overall time performance with CPU threads varied. The results show that our system has moderate scalability on one-to-two GPUs and a significant speedup to the CPU-based version. As mentioned in the previous section, the limited speedup of phase 1 is mainly caused by read loading and the disk I/O speed, which means the phase 1 speed of RapidGKC has already reached the inherent limitation of the partitioning-and-counting paradigm. When the GPU is comparable with the CPU performance, it is helpful to involve CPU coprocessing to further accelerate the counting. In contrast, when multiple GPUs are employed, the acceleration by the GPU alone is already substantial, and the inclusion of CPU co-processing does not speed up further but slows down the overall performance.

F. Effect of Different k Values

The k-value determines k-mer length. This section investigates the time cost of each phase with k-value varies. The result on Figure 16 shows the time in the two phases, the super k-mer size and the reciprocal of k-mer depth. In the partitioning phase, the time is highly related to the total size of super k-mers. In the counting phase, the time cost is highly related to the reciprocal of k-mer depth. In more detail, when

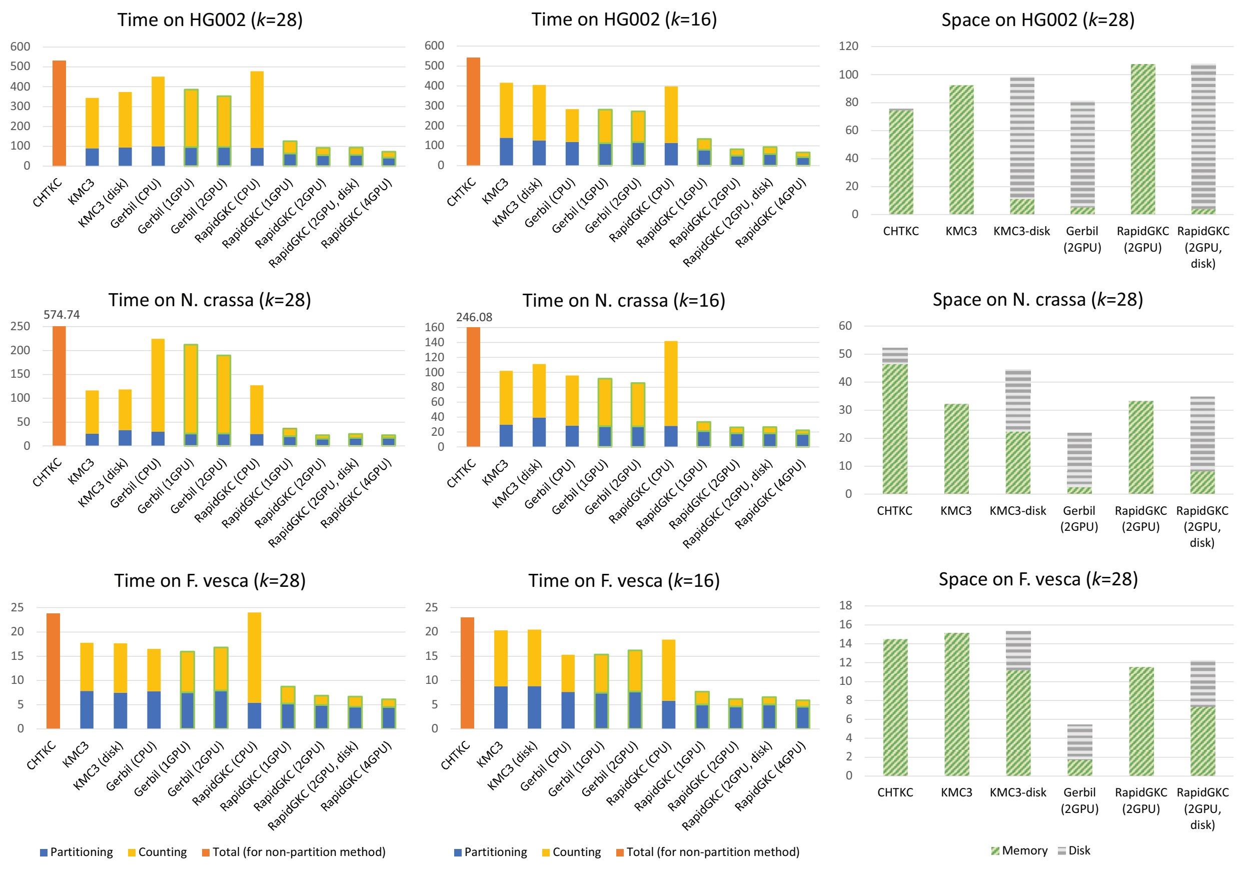


Fig. 15. K-mer counting experiments on three datasets on system 1 (warm start with file cache, time: seconds, space: GB).

the k-mer depth is smaller, there are more distinct k-mers, and it will take longer time to copy the counting result from the GPU back to the CPU main memory. Our profiling result shows that the copy time can take up more than 60% of the counting phase time when the k-mer depth is close to 1 and all k-mers are counted. In practice, users usually discard the k-mers whose number of occurances is 1. RapidGKC can efficiently filter out those k-mers on the GPU to avoid redundant transfer. For smaller k, we use short integer types to represent the k-mers to reduce the copying workload.

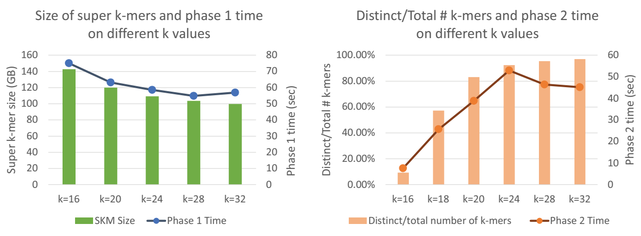


Fig. 16. Time cost of each phase with different k values.

V. Conclusion

K-mer counting is an elementary step of many bioinformatics applications. The major focus of k-mer counting methods is the time cost and memory usage. In this study, we proposed a GPU-accelerated k-mer counting method, RapidGKC, and evaluated the performance of our method in comparison with some state-of-the-art methods on datasets from different sequencing technologies.

We designed a new encoding scheme for variable-length genomic data to support parallel encoding and decoding. Furthermore, we proposed a novel rule to select common substrings among k-mers for partitioning, reducing the space cost as well as facilitating efficient parallel processing. Integrating these novel designs, we implemented a prototype GPU-accelerated k-mer counting system, RapidGKC. It parallelizes all the major tasks of k-mer counting across the CPU and the GPU, including partitioning, counting, data encoding, and decoding. RapidGKC achieves the best overall time performance among all methods under comparison. Our experiment setup consists of representative hardware and software environments as well as datasets. In our experiments, RapidGKC can achieve at most 5x speedup of the overall time over current SOTA k-mer counters. Our code and the artifact description are available at https://github.com/cyr20040123/RapidGKC.

References

[1] M. Kokot, M. Długosz, and S. Deorowicz, "Kmc 3: counting and manipulating k-mer statistics," *Bioinformatics*, vol. 33, no. 17, pp. 2759–2761, 2017.

[2] S. C. Manekar and S. R. Sathe, "A benchmark study of k-mer counting methods for high-throughput sequencing," *GigaScience*, vol. 7, no. 12, p. giy125, 2018.

[3] J. Ruan and H. Li, "Fast and accurate long-read assembly with wtdbg2," *Nature methods*, vol. 17, no. 2, pp. 155–158, 2020.

[4] T. R. Ranallo-Benavidez, K. S. Jaron, and M. C. Schatz, "Genomescope 2.0 and smudgeplot for reference-free profiling of polyploid genomes," *Nature communications*, vol. 11, no. 1, p. 1432, 2020.

[5] D. Kleftogiannis, P. Kalnis, and V. B. Bajic, "Progress and challenges in bioinformatics approaches for enhancer identification," *Briefings in bioinformatics*, vol. 17, no. 6, pp. 967–979, 2016.

[6] Z. Feng, S. Qiu, L. Wang, and Q. Luo, "Accelerating long read alignment on three processors," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[7] V. Marx, "Method of the year: long-read sequencing," *Nature Methods*, vol. 20, no. 1, pp. 6–11, 2023.

[8] M. Alser, J. Lindegger, C. Firtina, N. Almadhoun, H. Mao, G. Singh, J. Gomez-Luna, and O. Mutlu, "From molecules to genomic variations: Accelerating genome analysis via intelligent algorithms and architectures," *Computational and Structural Biotechnology Journal*, 2022.

[9] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the top: What will drive computer performance after moore's law?" *Science*, vol. 368, no. 6495, p. eaam9744, 2020.

[10] S. Aluru and N. Jammula, "A review of hardware acceleration for computational genomics," *IEEE Design & Test*, vol. 31, no. 1, pp. 19–30, 2013.

[11] K. A. O'Connell, Z. B. Yosufzai, R. A. Campbell, C. J. Lobb, H. T. Engelken, L. M. Gorrell, T. B. Carlson, J. J. Catana, D. Mikdadi, V. R. Bonazzi *et al.*, "Accelerating genomic workflows using nvidia parabricks," *BMC bioinformatics*, vol. 24, no. 1, pp. 1–15, 2023.

[12] M. Erbert, S. Rechner, and M. Müller-Hannemann, "Gerbil: a fast and memory-efficient k-mer counter with gpu-support," *Algorithms for Molecular Biology*, vol. 12, pp. 1–12, 2017.

[13] H. Li, A. Ramachandran, and D. Chen, "Gpu acceleration of advanced k-mer counting for computational genomics," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018, pp. 1–4.

[14] Y. Li, P. Kamousi, F. Han, S. Yang, X. Yan, and S. Suri, "Memory efficient minimum substring partitioning," *Proceedings of the VLDB Endowment*, vol. 6, no. 3, pp. 169–180, 2013.

[15] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, "Kmc 2: fast and resource-frugal k-mer counting," *Bioinformatics*, vol. 31, no. 10, pp. 1569–1576, 2015.

[16] J. Tan and X. Fu, "Addressing hardware reliability challenges in general-purpose gpus," *Adv. GPU Res. Pract*, pp. 649–705, 2017.

[17] T. Hu, N. Chitnis, D. Monos, and A. Dinh, "Next-generation sequencing technologies: An overview," *Human Immunology*, vol. 82, no. 11, pp. 801–811, 2021.

[18] A. Rhoads and K. F. Au, "Pacbio sequencing and its applications," *Genomics, proteomics & bioinformatics*, vol. 13, no. 5, pp. 278–289, 2015.

[19] P. E. Compeau, P. A. Pevzner, and G. Tesler, "How to apply de bruijn graphs to genome assembly," *Nature biotechnology*, vol. 29, no. 11, pp. 987–991, 2011.

[20] G. Myers, "Efficient local alignment discovery amongst noisy long reads," in *Algorithms in Bioinformatics: 14th International Workshop, WABI 2014, Wroclaw, Poland, September 8-10, 2014. Proceedings 14*. Springer, 2014, pp. 52–67.

[21] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[22] S. Saravanan and P. Athri, "Hmspkmercounter: Hadoop based parallel, scalable, distributed kmer counter for large datasets," in *2018 International Conference on Bioinformatics and Systems Biology (BSB)*, 2018, pp. 112–118.

[23] Y. Li *et al.*, "Mspkmercounter: a fast and memory efficient approach for k-mer counting," *arXiv preprint arXiv:1505.06550*, 2015.

[24] I. Nisa, P. Pandey, M. Ellis, L. Oliker, A. Buluç, and K. Yelick, "Distributed-memory k-mer counting on gpus," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 527–536.

[25] D. Lemire, N. Kurz, and C. Rupp, "Stream vbyte: Faster byte-oriented integer compression," *Information Processing Letters*, vol. 130, pp. 1–6, 2018.

[26] H. Yin, Y. Shao, X. Miao, Y. Li, and B. Cui, "Scalable graph sampling on gpus with compressed graph," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 2022, pp. 2383–2392.

[27] J. Wang, S. Chen, L. Dong, and G. Wang, "Chtkc: a robust and efficient k-mer counting algorithm based on a lock-free chaining hash table," *Briefings in Bioinformatics*, vol. 22, no. 3, p. bbaa063, 2021.

[28] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "Squeakr: an exact and approximate k-mer counting system," *Bioinformatics*, vol. 34, no. 4, pp. 568–575, 2018.

[29] S. Qiu and Q. Luo, "Parallelizing big de bruijn graph construction on heterogeneous processors," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 1431–1441.

[30] D. Jünger, R. Kobus, A. Müller, C. Hundt, K. Xu, W. Liu, and B. Schmidt, "General-purpose gpu hashing data structures and their application in accelerated genomics," *Journal of Parallel and Distributed Computing*, vol. 163, pp. 256–268, 2022.

[31] Y. Li, Q. Zhu, Z. Lyu, Z. Huang, and J. Sun, "Dycuckoo: dynamic hash tables on gpus," in *2021 IEEE 37th international conference on data engineering (ICDE)*. IEEE, 2021, pp. 744–755.

[32] N. Corporation. (2023, Apr.) Magnum io gpudirect storage — nvidia developer. [Online]. Available: https://developer.nvidia.com/gpudirect-storage

[33] E. Stehle and H.-A. Jacobsen, "A memory bandwidth-efficient hybrid radix sort on gpus," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 417–432.

[34] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," *ACM Sigplan Notices*, vol. 47, no. 8, pp. 117–128, 2012.

[35] M. Osama, S. D. Porumbescu, and J. D. Owens, "A programming model for gpu load balancing," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 79–91.

[36] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang, "Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on gpu," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 38–52.