Singleton Sieving: Overcoming the Memory/Speed Trade-Off in Exascale k-mer Analysis

Hunter McCoy* Steven Hofmeyr† Katherine Yelick‡ Prashant Pandey§

Abstract

Traditional filter data structures, such as Bloom filters, do not offer necessary features that modern high-performance data analytics applications need in order to efficiently perform complex data analysis tasks. For example, MetaHipMer, a de novo metagenome assembler, can use filters to weed out singleton k-mers and reduce memory usage by 30%-70%. However, the filter needs the ability to associate values with k-mers in order to perform the analysis in a single communication pass. Bloom filters do not support value associations and cause the application to perform an extra communication pass, thereby increasing the run time. Therefore, MetaHipMer faces a trade off between memory and speed due to the limited capabilities of traditional filters.

In this paper, we overcome the memory and speed trade off in MetaHipMer by integrating a GPU-based feature-rich filter, the Two-Choice filter (TCF), in the MetaHipMer pipeline. The TCF uses key-value association to approximately store k-mers with extensions. This allows MetaHipMer to perform k-mer analysis on the GPUs in a single communication pass. Our empirical analysis shows a 50% reduction in memory usage in k-mer analysis on each node in MetaHipMer without any effect on the overall run time or assembly quality. The memory reduction in turn results in a 43% reduction in the number of nodes required to assemble datasets and enables MetaHipMer to scale to much larger datasets.

1 Introduction

Many modern-day high performance data analytics (HPDA) applications are designed to scale to thousands of nodes and process petabyte scale data sets, stressing the memory and computing capacity of the largest machines available today. These systems often use GPUs to achive massive parallelism and scale up computations. GPUs offer both an opportunity for performance improvement and a challenge for data analytics due to the limited GPU memory that is available compared to CPUs.

An example of HPDA application is metagenomic assembly in computational biology. Metagenome assembly is the process of transforming a set of short, overlapping, and potentially erroneous DNA segments from environmental samples into the accurate representation of the underlying microbiomes's genomes. MetaHipMer [14, 15] is an exa-scale *de novo* metagenome assembler that leverages GPUs to speed up raw data processing and is designed to scale out to thousands of nodes to handle petabyte scale data. However, raw data processing is the most-memory intensive phase in the MetaHipMer pipeline and using the GPUs results in an overall increase in the number of nodes required because of the limited memory available on individual GPU nodes.

Filters, such as Bloom [5], quotient [22, 10, 11, 3, 25, 27] and cuckoo filters [12, 6], maintain an approximate representation of a set or a multiset[1]. The approximate representation saves space by allowing queries to occasionally return a false-positive. For a given false-positive rate ε: a membership query to a filter for set S returns present for any $x \in S$, and returns absent with probability at least $1-\varepsilon$ for any $x \notin S$. A filter for a set of size n uses space that depends on ε and n but is much smaller than explicitly storing all items of S.

HPDA applications can greatly benefit from using filters on the GPU. For example, *k-mer analysis* is the very first data processing step in MetaHipMer and numerous other pipelines in computational biology [4, 21]. In k-mer analysis, we start by parsing the raw sequencing data into length-k subsequences (called k-mers) and counting the occurrences of each k-mer using a GPU-based hash table. K-mer counting is used to weed out erroneous data (singleton k-mers) caused by sequencing errors, estimate sequencing depth, prepare sequencing data for assembly, and many other downstream tasks [21]. A filter is often used in k-mer counting to separate out singleton k-mers [18, 20] before inserting them in the hash table.

Weeding out singletons saves a considerable amount

*University of Utah
†Lawrence Berkeley National Lab
‡University of California, Berkeley
§University of Utah

[1] Counting filters maintain count estimates of items in a multiset. A counting filter may have an error rate δ. Queries return true counts with probability at least $1-\delta$. Whenever a query returns an incorrect count, it must always be greater than the true count.

of space in the hash table during counting as singleton k-mers form a majority of unique k-mers. Each k-mer is first queried in the filter and if it returns true then only the k-mer is counted otherwise the k-mer is simply inserted in the filter and not counted.

However, modern data analysis pipelines have evolved and they require additional features in filters in order to be truly useful. Specifically, filters are often required to support associating small values with items, counting the frequency of fingerprints, deletions, and efficient mergeability.

For example, in k-mer analysis in MetaHipMer, along with the k-mers, the prefix and suffix sequence extensions of the k-mers are recorded to traverse the k-mer graph in later phases [15]. These extensions are also preserved with the first instance of the k-mer in the filter. The k-mer extensions are critical for allowing the filter construction to occur in the same phase as k-mer hash table insertion. Without it, the user must either pass over the data twice or lose the extensions of the k-mers stored in the table. While these extensions are not necessary for singletons, they are critical for k-mers inserted into the hash table as they are used in later phases of the pipeline.

Apart from associating small values filters are also often required to support counting and deletions in order to remove unwanted items in the later phases of the pipeline and avoid rebuilding them from scratch. For example, filters are used to represent an approximate representation of the de Bruijn graph by storing k-mers and their count estimates [24, 23].

Traditional filters do not support the features required by modern HPDA applications. For example, Bloom filters [5] do not support associating small values with keys, counting, or efficient mergeability. Furthermore, existing work on GPU filters is limited in terms of performance and features [8, 16, 13].

Due to the lack of feature-rich filters, modern GPU-based applications face a trade off between memory and speed. The applications often work around the limitations of filters which in turn result in sub-optimal use of resources and further hinder their scalability to larger datasets. For example, MetaHipMer would require an additional communication phase while using the Bloom filter to save memory during the k-mer analysis phase as Bloom filters do not support associating extensions with k-mers. This would result in lower memory usage, but at the cost of increased running time, thus negating the computational benefits of the GPUs.

The lack of high-performance and feature-rich GPU-based filters can constrain how efficiently exascale applications utilize GPU resources. These issues occur in single-node applications, but they are exacerbated in distributed environments by the additional movement of data, as an additional pass can move terabytes of data.

Recently, there have been major advancements in building filters that can support the necessary features required by modern HPDA applications. For example, the cuckoo filter [12] supports deletions at no additional space cost compared to the Bloom filter. The counting quotient filter [25] supports counting, associating values, deletions, and mergeability at no additional space cost.

There are also GPU implementations of these filters which can be directly integrated in the GPU-accelerated modern data analytics applications [19]. The Two-choice filter (TCF) [19] is a recently introduced GPU filter that offers massive parallelism and high performance on GPUs and important features such as value association and deletion required by HPDA applications. The ability to associate small values with keys is an especially important property that can be used to reduce the memory usage in k-mer analysis pipelines. This property has also been shown to be useful in database applications to speed up I/O in a log-structure merge tree [9].

Our contribution. In this paper, we show how to overcome the inherent trade off between memory and speed in a GPU-accelerated data analysis application using advanced GPU filters. We analyze the raw data processing pipeline in MetaHipMer and understand the various trade offs in terms of memory usage of the hash tables and communication passes. We then integrate the two-choice filter (TCF) [19], a GPU-enabled feature-rich filter, into MetaHipMer to reduce the memory usage of the most memory intensive phase without introducing any slow down. Specifically, we show that when using the TCF, MetaHipMer's memory usage goes down by around 2× during the k-mer analysis phase, which was the main bottleneck in terms of memory usage among all phases. We further show that using the TCF in MetaHipMer has no slowdown in the overall run time of the application and has no impact on the final quality of the results. The TCF enables MetaHipMer to tackle larger datasets that were not possible to assemble before due the lack of available memory resources, even on large supercomputers.

Our empirical evaluation (Section 6) shows that using the TCF reduces the memory usage of the k-mer analysis phase by up to 5.4×. This in turn results in the reduction of the number of nodes required to perform an assembly by 43% (from 64 nodes to 36 nodes for WA dataset) as k-mer analysis is the most memory intensive phase in the MetaHipMer pipeline.

2 Advanced filters

In this section, we describe the major GPU filter types and discuss whether we can use them in modern data analytics applications based on the features they support. We will consider dynamic filters as they do not require the set of items to be known in advance which is often the case in data analytics applications. The dynamic filters are broadly classified into four major types: Bloom filters [5],

Filter	**Insert**	**Query**	**Delete**	**Values**	**Concurrency**
GQF [19]	✓	✓	✓	✓	
TCF [19]	✓	✓	✓	✓	✓
BF [17]	✓	✓			✓
CF [12]	✓	✓	✓	✓	

Table 1: Various filters and supported features. The GPU quotient filter (GQF) is the only filter that supports a range of operations. However, it does not offer massive parallelism on GPUs. The two choice filter (TCF) can support value associations and massive parallelism but does not support counting and efficient mergeability. The cuckoo filter (CF) can also offer value association but does not achieve massive parallelism on GPUs due to random hoping during inserts. The Bloom filters (BF) does not offer features but achieves massive parallelism due to bitwise operations and simplicity.

quotient filters [3, 26, 10, 22, 11], cuckoo filters [12, 6], and two-choice filters [27].

We compiled the list of features needed by MetaHipMer and show the presence and absence of those features in the four major filter types in Table 1.

Bloom filters do not support deletions and associating small values with items that many data analytics applications require. However, Bloom filters are easily parallelizable and can exploit massive parallelism on GPUs. Cuckoo hashing involves multiple random writes during inserts which make it challenging to achieve high speed operations in a GPU implementation. The quotient filter [25, 19] supports deletion and associating small values with items by explicitly storing small values with the remainders in the table [28]. However, due to Robin Hood hashing, the operations in a quotient filter require large move operations which hurts concurrency on GPUs. It is hard for quotient filters to exploit massive parallelism on GPUs, especially for point operations. They can be parallelized for bulk operations [13].

Based on Table 1, we see that only the two choice filter (TCF) supports the basic operations (inserts/deletes/queries), features like associating values with items, and high GPU concurrency. The TCF [19] is the most appropriate filter among all the dynamic filters to be integrated in the data analytics applications like MetaHipMer and reduce their memory consumption on GPUs.

3 Two Choice Filter

In this section, we give the implementation details of the two choice filter [19] (TCF) on the GPU. In the TCF, the table is organized into blocks. Each block can store B f-bit tags, where a tag is the lower order f-bits of a fingerprint. The blocks are sized to fit inside a GPU cache line. The TCF uses the power-of-two-choice (POTC) hashing scheme [27] to perform operations. In a POTC scheme, every item is hashed to two blocks via a pair of unique hashes. The secondary hash is computed using the XOR operation between the primary bucket and the stored tag in order to delete items without introducing a false negative [12, 27]. For inserts, the fill of each block is queried, and the item is inserted into the less full block. Queries return true if the queried item is found in either block. POTC hashing helps to reduce the load variance across blocks, reducing the size of the largest block to $O(\text{loglog}n)$, where n is the number of items, as shown by Azar et al. [1]. This allows the table to achieve a high load factor without any combination of two blocks where both the blocks are full.

To avoid insertion failures (no empty slot in both blocks) before reaching a 90% load factor the filter comes with a small backing table. This table is sized to be 1/100 of the total size of the filter. Since $<<1\%$ items fail to be inserted, the extra cost required to insert and query from this table is negligible, and it has no measured effect on the speed of inserts or positive queries. However, it does have an effect on the performance of negative queries, as at least one extra block will have to be searched. The TCF can achieve 90% load factor using the backing table.

The backing table is based on linear probing. In linear hashing, we first use a hash function to determine a location for the item. If that location is occupied then we linearly step forward until we find an empty location or until a cutoff depth has been reached.

The false-positive rate for the TCF is given by $\frac{2B}{2^f}$, where B is the size of the blocks and f is the tag size. A larger tag size decreases the false-positive rate but increases the space. In CUDA, the minimum size for an atomicCAS (Compare-and-Swap) transaction is 2 bytes. With keys set to the minimum CAS size and a block size of 16, the error rate is .04%. Smaller keys are also permissible as long as the total size of the key and value fit within a CUDA atomic CAS operations (2,4, or 8 bytes).

3.1 GPU Implementation

The original TCF implementation [19] uses cooperative groups to collectively scan and insert tags into buckets. This is efficient as the threads in a warp share a memory pipeline, so collating them to work on the same operation reduces both warp and memory contention in the streaming multiprocessors (SM). This also allows for relatively large buckets to be used, which helps in achieving high load factors by reducing the variance across the buckets in the table.

Using cooperative groups is currently not possible in MetaHipMer, as the GPU data structures already present assign one GPU thread per k-mer. In order to account for this, we make some small modifications to the filter to allow it to run at maximum throughput. Using the bucket size of 16 for the primary table, we found that the computational cost of 1 thread scanning 16 tags per bucket dominated performance. By reducing the bucket size to 8, we were able to reclaim this lost performance and double the

throughput of the filter. Reducing the bucket size increases the variance across bucket and causes more items to be sent to the backing table. We account for this by increasing the size of the backing table to be 10% of the total size of the filter rather than 1%. The final result of these changes was a per-thread GPU filter with 800 million inserts and 2.4 billion queries per second on an NVIDIA V100.

4 MetaHipMer

Metagenome assembly involves reconstructing long contiguous sequences (*contigs*) of genetic material from short input *reads*. These reads are strings of bases (the DNA alphabet A,C,G,T) of length 150 to 250 that are produced by gene sequencing machines. For metagenomes, these reads are extracted from environmental samples (e.g. gut bacteria, or a soil sample) that contain the genes of potentially thousands of microbes, existing at varying abundances. The reads are error prone (typically about 0.24% error per base) and sequencing is done multiple times to ensure every region of genetic material is covered with some error free sequences.

In the approach used by MetaHipMer, the reads are first divided into overlapping substrings of fixed length k, called k-mers, which are then used to form a de Bruijn graph [7]. In a de Bruijn graph, the vertices are k-mers and edges connect any two k-mers that have an overlap of $k-1$ bases. These vertices are stored in a hash table that is distributed across all the compute processes. The size of the hash table is dependent on the number of unique k-mers. Traversal of the de Bruijn graph enables the construction of the contigs (longer sequences). This approach is more efficient than an all-to-all alignment of the reads, which would be prohibitive for the size of typical metagenome datasets (up to billions of reads).

Forward and backward extensions of the k-mer and the counts of those extensions are also maintained in the hash table along with the k-mer. Information regarding the extensions and their counts is critical to identifying correct paths in the de Bruijn graph and requires 28 to 52 bytes (depending on k) to store each k-mer.

To ensure accurate contigs, the k-mers that occur only once (singletons) are treated as errors and dropped. In a typical set of metagenome reads, 70 to 80% of unique k-mers are singletons, but they still need to be stored and counted in the distributed hash table. In the default MetaHipMer implementation, storing the unique k-mers is the most memory intensive part of the computation and can be roughly an order of magnitude larger than the input data. The space required to store the k-mers can be much larger than the size of the original raw dataset (up to 10× larger) as k-mers contain a lot of redundant information due to their overlaps.

Figure 1 shows different parts of the k-mer analysis phase in MetaHipMer. In the standard pipeline, all k-mers are counted in the hash table and then a separate phase is required to purge all the singleton k-mers. Filters help avoid adding singletons in the hash table and therefore a separate phase is not required to purge singletons. However, in practice some sinegletons can still pass through the filter due to false positives which are removed in MetaHipMer in a separate compress and off load to CPU phase.

Dataset	Percentage singleton k-mers				
	$k=21$	$k=33$	$k=55$	$k=77$	$k=99$
WA	66	73	76	78	78
Rhizo	67	75	80	83	85
Tymeflies	63	62	67	69	71

Table 2: Distribution of singleton k-mers in metagenomic datasets with different values of k.

The distributed hash table in MetaHipMer is implemented as a collection of local hash tables, one per process, with communication happening via UPC++ remote procedure calls (RPCs). In the hash table insertion phase, the k-mers are hashed to a remote process, aggregated, dispatched over the network, and inserted in bulk into the local hash tables, which are running on GPUs. This hashing helps achieve good load balance across processes. A special minimizer-based scheme is used to reduce the communication volume [21]. Using GPUs boosts performance, but further constrains memory (e.g. the Summit supercomputer [30] has an aggregate of 96GB GPU memory and 512GB CPU memory per node).

4.1 k-mer distribution Table 2 shows the distribution of singleton k-mers in three different metagenomic datasets. Singleton k-mers form a majority fraction of the total number of distinct k-mers. The distribution often depends on the sequencing depth and the size of k. A larger value k results in larger fraction of singleton k-mers. This is due to the higher probability of seeing an erroneous base in the k-mer given the higher value of k. These erroneous bases result in singleton k-mers.

Weeding out singleton k-mers before inserting them in the hash table to count is critical in any k-mer analysis phase to reduce the memory usage of the counting phase. These singleton k-mers can also be pruned from the hash table after the counting phase. However, that results in high peak memory usage and a much slower running time.

Using a space-efficient filter to weed out singleton k-mers helps to reduce the memory pressure on the counting hash table, thereby reducing the peak memory usage and increased run time. See Figure 1.

5 TCF Integration in MetaHipMer

We integrated the TCF into MetaHipMer [14, 15] and achieved a reduction of about 2× in the peak memory usage

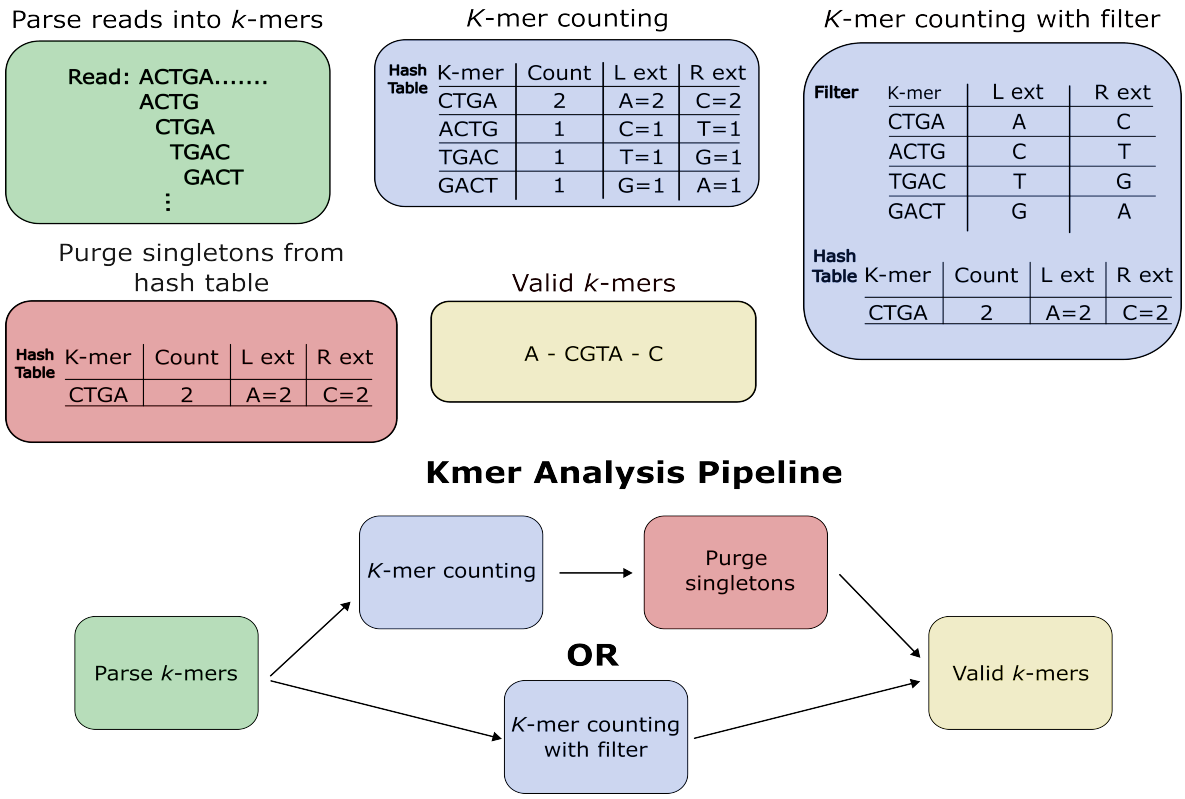


Figure 1: The k-mer analysis pipeline in MetaHipMer. A filter can help weed out singleton k-mers from being inserted into the hash table.

of the k-mer counting phase which is the most memory intensive phase in the MetaHipMer pipeline. MetaHipMer is written in UPC++ [2], and is the only metagenome assembler that scales to thousands of distributed memory nodes and tens of terabytes of input data.

5.1 GPU-based TCF in MetaHipMer Most of the memory required to store unique k-mers in the hash table is wasted as the singletons are not needed in the later phases of the assembly. We can use a filter to keep track of the singletons, only storing k-mers in the hash table that appear more than once. This reduces the memory considerably, since the storage cost per k-mer and the count of its extensions is 28 to 52 bytes (depending on the value of k), whereas a filter only requires $\approx$1 byte per item plus 1 byte to store an encoding of the extensions.

When using the filter to track the singletons, the first instance of each k-mer is inserted in the filter and only upon seeing subsequent instances of the k-mer is it inserted into the hash table. However, recall that we also need to record both extensions of each instance of the k-mer. Therefore, we need a filter where we can also record the extensions of the first instance of the k-mer. The TCF supports associating a small value with each item unlike other filters which can only store items.

This small value is critical for allowing the filter construction to occur in the same phase as k-mer hash table insertion. Without it, you must either pass over the data twice or lose the extensions of the k-mers stored in the table. While these extensions are not necessary for singletons, they are critical for k-mers inserted into the hash table.

Each k-mer is parsed, hashed, and communicated to the process that owns the corresponding hash table, where it is checked to see if the k-mer exists. If it does, the existing counts for the relevant extensions are increased. If it is not in the hash table, then we query for the k-mer in the TCF. If it is found then we insert the k-mer in the hash table along with the extensions found in the TCF. If it is not in the TCF, then we insert the k-mer along with its two extensions in the TCF.

This implementation adds one call to the TCF for each unique k-mer, and an additional call to the hash table for each non-unique k-mer. With the typical frequency of a non-unique k-mer being at least 10, this translates to a relatively small number of extra operations. See Figure 2.

Insert if not exists. When adding TCFs to MetaHipMer,

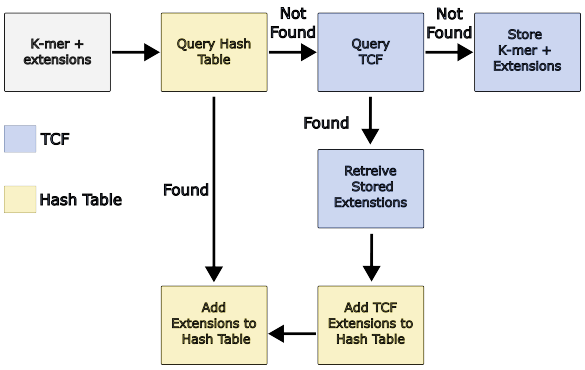


Figure 2: TCF integration in the k-mer counting phase in MetaHipMer.

we modify the internal point functions for inserts and queries to better support MetaHipMer's use case. Given the use case in MetaHipMer, we do not need to count the k-mers as each k-mer will only be stored once with its extensions and queried thereafter. That means every time a new k-mer arrives we have to first check if that k-mer is already present in the TCF and insert it only if it does not exist. This requires first invoking a query and then performing an insert if necessary.

MetaHipMer does not require counting, but does reveal an important usage pattern of a lookup followed by an insert. To avoid multiple calls for every k-mer, we expose a grouped function, $insert\_if\_not\_exists$, that retrieves an element x from the table, returning the stored extensions if found or encoding and storing the extensions if this is the first time the k-mer is seen. This new API call first looks to find the k-mer in the TCF by probing the remainders stored in the run corresponding to the k-mer. However, if it does not find the expected remainder in the run then it performs an insert operation by creating space for the new remainder. This avoids the redundant work needed to identify the run corresponding to the k-mer in the TCF and the cache misses that would have happened with separate query and insert function calls.

Deleting items from the TCF. As shown in Figure 2, you can stage the TCF after the hash table, so that only k-mers that have been seen less than two times are queried in the TCF. In this scenario, a non-singleton k-mer will never be queried in the TCF more than twice. Therefore, the k-mer can be safely deleted from the filter once it is queried twice as it is now present in the hash table and will never be queried in the filter again.To delete an item in the filter, we replace it with a tombstone which can then be overwritten by a new insertion. This saves space in the filter as we do not have to store non-singleton k-mers in the TCF.

Storing k-mer extensions. k-mer extensions are encoded using 3 bits corresponding to one of the five possible extensions:{A, C, T, G, Low Confidence}. Each k-mer has both a forward and backward extension that are packed together into a 6-bit value to be stored with the remainder. Therefore, each k-mer has a corresponding 6-bit value stored along with a 10-bit tag in the TCF.

The TCF is a single-GPU data structure, which fits well within the MetaHipMer architecture, where each process is written as though it has exclusive access to a single GPU, and inter-process communication happens through UPC++ operations. As described earlier, all processes share all available GPUs through the Nvidia MPS, and hence MetaHipMer fully exploits all GPUs on all nodes in a distributed system.

5.2 Choosing the right filter for MetaHipMer

In Section 2, we discuss the major filter designs. For the GPU case, the TCF supports all the necessary features required by MetaHipMer.

Apart from the TCF, GPU quotient filters (GQF) are also an adequate option for inclusion in MetaHipMer. We include the results of integrating the GQF in Section 6.4. However, the performance of quotient filters is only guaranteed up to load factors of 95%. Due to the variance in the number of k-mers processed by nodes during k-mer analysis, the maximum load factor in the filter is not bounded, so the quotient filter often stalls due to higher load factors, thereby slowing down the entire system.

Vector Quotient Filters or other CPU filters with value association and high performance at high load factors could be suitable. However, as discussed in Section 6.5, the effect of memory savings on the pipeline is less impact full on the CPUs due to their increased RAM.

6 Evaluation

In this section, we evaluate the performance of MetaHipMer with and without the GPU-based two choice filter (TCF) [19]. Our goal is to understand the impact of the TCF integration on the performance of MetaHipMer. We first perform a series of microbenchmarks to test different TCF configurations and determine the best configuration. We then use that configuration to evaluate MetaHipMer's performance for run time, peak memory usage, number of nodes required to finish an assembly, and assembly quality for three large metagenomic datasets.

We address the following questions about the MetaHipMer performance:

1. What is the impact of the TCF on peak memory usage?
2. What is the impact of the TCF on run time?
3. What is the impact of the TCF on assembly quality?

We assemble three different datasets during the evaluation: the WA dataset, which is a collection of marine microbial communities from the Western Arctic Ocean and consists of 813GB of 2.4 billion reads of length 150 [14, 15]; the Rhizo dataset, which is the rhizosphere of three biofuel crops and consists of 129GB of 393 million reads of length

150 [29]; and Tymeflies, a sample of freshwater microbial communities from Lake Mendota composed of 1,000GB of 3 billion reads of length 150.

We run all microbenchmarks on a 40 GB subsection of the WA dataset, which was used to profile and select the best filter configuration for the assemblies.

Machine specification. Our microbenchmarks and large-scale experiments were run Summit's [30] GPU nodes. Each summit node consists of 2 IBM Power9 nodes with a total of 42 available cores and 512GB DDR4 memory, and 6 NVIDIA Volta V100s, each with 16GB HBM2 memory. MetaHipMer runs with one process per core, and all of the processes share all 6 GPUs.

Configuring nodes in MetaHipMer. For the WA dataset, we experimented to find the minimum number of nodes needed to assemble the dataset without loss of quality. The quality of the final assembly drops when the hash table reaches the maximum capacity during the k-mer analysis phase and starts failing to insert new k-mers. The point just before k-mers start to be dropped represents the minimum memory required for the k-mer analysis stage, since that is the most memory intensive of all the stages in the default implementation of MetaHipMer. The larger WA dataset required 64 nodes without the TCF and 37 nodes with the TCF and 36 nodes when items were deleted from the TCF. When using the TCF, the k-mer analysis phase is not the most memory intensive phase. Instead, the assembler is now memory-bound by the local assembly phase and requires at least 56% of the nodes needed when the TCF is not used.

MetaHipMer setup. The default setup for MetaHipMer is hereafter referred to as *NO_TCF*. This is the current state-of-the-art for GPU k-mer analysis. It does not use filters to weed out singleton k-mers. The Bloom filter (BF) was used in an earlier CPU version of MetaHipMer, however, the BF was removed in subsequent versions as it resulted in an overall slowdown. The BF does not support associating small values (prefix-suffix extension in MetaHipMer's case). Without value association the extensions associated with the first occurrence of the k-mer will be lost, resulting in poor assembly quality. To avoid this, MetaHipMer ran two communication passes, one to identify singletons and another to count true k-mers. This resulted in more than $2\times$ slowdown.

Host-Device communication is automatically handled by MetaHipMer, as k-mers are batched into compressed variants called supermers [21]. These supermers allow for efficient communication of k-mers across the network, and one kernel call is applied per supermer to unpack and insert the stored k-mers into the device hash table. The block size and thread size are determined by the size of an incoming supermer, and are not configurable. All GPU operations in MetaHipMer are thread-independent, so we use cooperative groups of size 1.

6.1 Performance microbenchmarks The impact of the TCF on run time can be seen in Table 3. Using the TCF in MetaHipMer does not increase the GPU computation time used to insert k-mers into the hash table, but lowers the GPU computation time for large datasets, as it lowers the fill ratio of the hash table which leads to better performance. The microbenchmarks were chosen to run on 8 nodes. With less than 8 nodes, the limited space affects assembly quality even with filters enabled, and the assembler can crash in local assembly phase due to the low memory available. We include a reference assembly of 8 nodes without the filters enabled, but this assembly achieves poor assembly quality due to memory constraints.

6.1.1 TCF sizing comparison A minimum of 6 bits required to store the extensions in the filter. We can configure the TCF to maintain either 2 byte or 4 byte containers composed of a tag and a value. Therefore, there are two options for the TCF storage configuration: 10 bits for the tag with tag-value pairs packed into two bytes; or 26 bit tags with tag-value pairs packed into four bytes. The two configurations offer a different trade off in terms of space usage and false-positive rate. With larger tags the false positive rate goes down but the space usage is higher. Even with 10-bit tags and 8 tags in each block, the false-positive rate is 1.5% which is fairly small for k-mer analysis phase. As table 3 shows, we did not find there to be a substantial difference in run time, hash table load, or final assembly quality between using the two byte or four byte TCF configuration. As the two byte filter has half the space usage, we use that as the default configuration.

6.1.2 Order swap microbenchmark There are two places where a filter can be inserted into the k-mer analysis pipeline, and the performance of these placements is based on the relative number of singletons in the dataset. The first option is to pass all k-mers through the filter: every k-mer is queried in the filter first, with its extensions stored if the k-mer has not been seen before. Successful filter queries return the set of stored extensions, and if the k-mer has not been stored in the dataset, then both it and the stored extensions are placed in the hash table. Otherwise, only the new extensions are placed inside the hash table. This setup has the advantage that novel k-mers are stored in the filter, and do not have to query the hash table. The disadvantage of this design is that every k-mer passes through the filter, which can make the filter a bottleneck for performance. In addition, deleting items is not possible in this scheme, as items that should be stored in the hash table must pass through the filter.

The other insertion place is after an initial hash table update. If the item is found in the hash table, the k-mer extensions are updated without ever touching the filter. Novel

Dataset	Method	Nodes	TCF mem	TCF load	HT mem	HT load	HT effective size	GPU insert time	k-mer analysis time	Assembly Size (quality)
WA_0	No TCF	8	0	0.0	958.48	0.96	920.14	3.98	108.73	1291371078
	TCF 2 Byte	8	82.16	0.52	750.99	0.32	240.32	6.05	111.81	1739419860
	TCF 2 Byte Delete	8	82.23	0.38	750.99	0.32	240.32	6.72	111.82	1739052630
	TCF 2 Byte Reversed	8	92.34	0.52	750.99	0.32	240.32	3.61	112.59	1739101182
	TCF 4 Byte	8	164.44	0.52	750.99	0.31	232.81	5.65	109.05	1740182469
	TCF 4 Byte Delete	8	164.58	0.39	750.99	0.32	240.32	6.12	111.35	1740023061

Table 3: TCF and hash table memory usage and running time during the MetaHipMer microbenchmark. Effective HT size is the hash table size times the load factor, and is a measure of the memory being used by the table. **Memory is in MB per-process and time is in seconds.**

k-mers are sent to the filter as a secondary query, and if the k-mer is found in the filter both sets of extensions are passed back to the hash table to be inserted. The advantage of this design is that every unique k-mer interacts with the TCF at most twice. Depending on the ratio of unique k-mers in the dataset, this can vastly reduce the number of memory operations required to process the dataset, although it comes at the expense that every singleton must first query the hash table, which can cause performance degradation if the dataset is mostly composed of unique k-mers.

Table 3 shows the two schemes, with the filter before the hash table marked as "TCF Reversed". The two schemes show no difference in load factor, memory saved, or assembly quality. The reversed version is faster because we have replaced all queries to the hash table with queries to the filter, which has faster accesses than the hash table. However, we cannot delete k-mers from the filter which results in sub-optimal space usage. Therefore, we use the filter after the hash table as the default configuration.

6.1.3 Delete microbenchmark

The final design consideration of using the TCF is whether or not to delete items once they are found in the filter and are inserted in the hash table. Given the default configuration above, only unique k-mers are sent to the filter, so we can delete the stored tag of a non-unique k-mer without affecting the insertion of any future copies. This can theoretically save space and reduce the number of erroneous extensions passed to the hash table, as a false positive can only occur before two copies of the correct k-mer pass through the filter. However, these gains come at the cost of reduced performance, as the filters must perform extra work on insertions to account for the tombstone keys used to delete items.

Table 3 shows that deleting k-mers from the filter once they are seen twice and inserted into the hash table saves roughly 20% of the filter memory. However, it does not save memory in the hash table, as only non-singleton k-mers in the filter can be removed by deletion. While this does not save overall memory as the filters are sized statically in MetaHipMer, we can size the filters more aggressively with deletions.

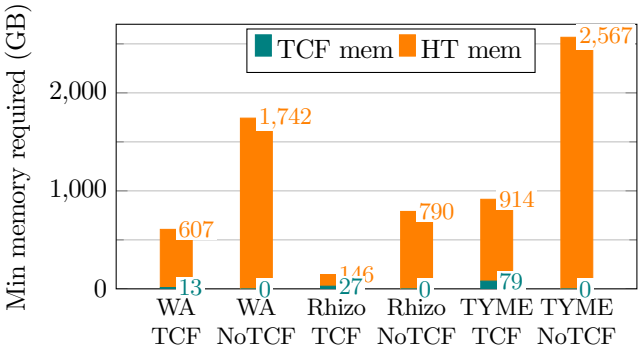


Figure 3: Aggregate Memory usage of the TCF and hash table in MHM for WA, Rhizo, and TYMEflies datasets.

6.2 Incremental assembly benchmark

The incremental benchmark holds the number of nodes constant, and increases the number of reads assembled by passing in successively more lanes from the Western Artic dataset. Each lane contains roughly 40GB of raw reads and is pulled from a subset of 5 space-time samples in the Western Artic. These samples may be skewed over the local ecosystem but are all pulled from the same underlying distribution. With 32 nodes, we can assemble ~325GB of data without the TCF before the limited space begins to affect assembly quality. With the TCF, we can assemble over 670GB before the assembly quality begins to drop. Figure fig. 5a shows the load factor of the filter and hash table as we increase the amount of data being processed, with assembly quality suffering once the load of the hash table exceeds ~90%.

6.3 Performance in MetaHipMer

MetaHipMer is designed to allocate as much GPU memory as possible for the hash table to reduce the overhead due to high load factors. Therefore, the memory used increases with available GPU memory. However, the minimum required memory can be computed by taking the number of items stored in the hash table and TCF, and multiplying those values by the amount of space required per item. The results of this computation with $k=99$ are shown in Figure 3. The

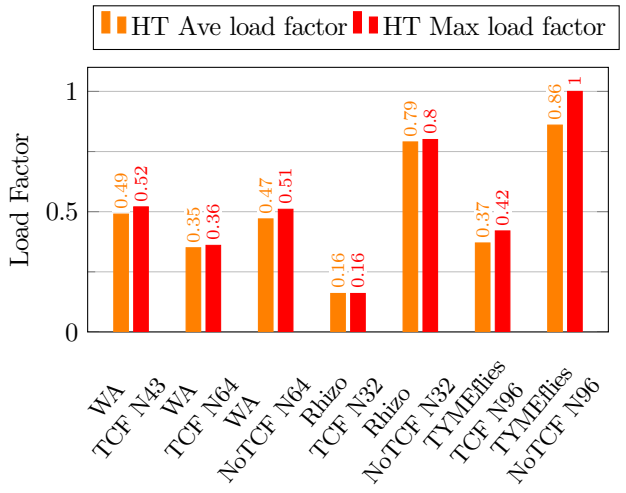


Figure 4: Load factor of the hash table in MHM for three different datasets. WA dataset was run on 43 and 64 nodes, the Rhizo dataset was run on 32 nodes, and TYMEflies on 96 nodes.

TCF uses 2 bytes per k-mer, whereas the hash table uses 52 bytes per k-mer. Without the TCF, the hash table has one entry per unique k-mer, but with the TCF, it does not store the erroneous singletons. In WA, 75% of k-mers are singletons, and in Rhizo there are 83%. Hence, the minimum memory required is significantly reduced, by a factor of 2.86× for WA and 5.4× for Rhizo dataset.

Memory usage. The effect of the TCF on the minimum memory required does not translate directly into a reduction in the number of nodes because there are other data structures that are not reduced in size by the TCF. Consequently, for practical purposes, the use of the TCF enables an approximately 43% reduction in the minimum number of nodes needed to assemble these datasets. This is supported by the load factor for the hash table, as shown in Figure 4. We can see that at the same node count (64 for WA and 24 for Rhizo), the load factor when using the TCF is a third to half of that without the TCF. Also, the variation in load of the hash table across nodes is more pronounced without the TCF, e.g., on WA, the average load factor for a process is 0.6 but the maximum is 0.9, which also means a less efficient use of memory.

Run time performance. Using the TCF in MetaHipMer does not substantially increase the GPU computation time used to insert k-mers into the hash table. In the 64 node runs, the dataset was assembled in 934 seconds with the filter and 940 seconds without, which is less than a 1% variation in the total run time and can be attributed to random variation from the Summit nodes. When the system operates at a high load factor, the time taken for k-mer analysis increases significantly. This is because the hash tables have a quadratic probing scheme, so if the hash table is filled to more than 77% full, future inserts become significantly slower. As the TCF reduces the load factor of the hash tables, fewer k-mers are inserted into the hash table, preventing this from being an issue until the memory pressure is so high that other parts of the assembler begin to run out of memory.

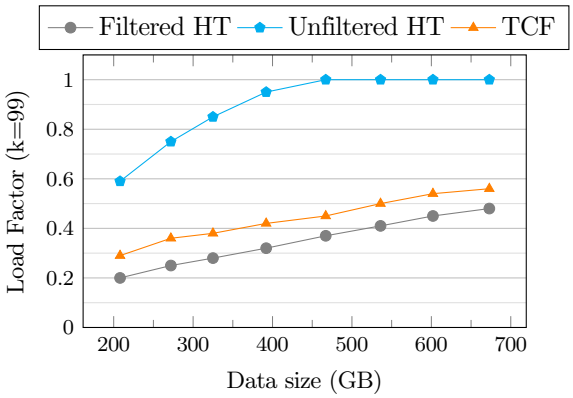
Assembly quality. One aspect of the TCF that could cause problems for MetaHipMer is the approximate nature of the data structure. We have configured two versions of the TCF, a two byte version and a 4 byte version. The two byte configuration uses 10 bits for the keys, giving it an error rate of $\frac{16}{1024}$ or 1.56%. Since all errors are false positives, this means that there may be some k-mers which are singletons but are not detected as such. However, this will only result in overestimation of the k-mer counts by at most 2, which should reduce their impact on the assembly quality. And indeed, we find that the assembly quality is not impaired when using the two byte version of the TCF. Total length assembled (a measure of quality) is within .1%. The 4 byte version uses 26 bit tags. This is accurate enough that false positives have no noticeable effect on assembly quality, even on small datasets. For this reason, the filter uses 10 bit keys over 26 bit keys as operations are half-word aligned which leads to better performance.

6.4 Using other GPU filters in MetaHipMer

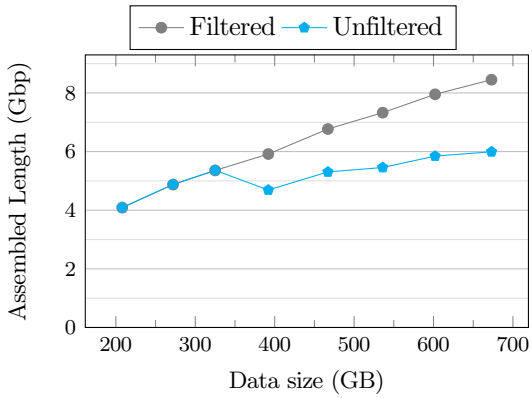
In addition to testing the TCF, we also benchmarked MetaHipMer with the GQF. We found that using the GQF resulted in an increase of over 2× the run time of the GPU phase of k-mer analysis, although it should be noted that this is a small section of the overall k-mer analysis run time. In addition, we found that on some runs the GQF would increase the overall run time for k-mer analysis by over 500 seconds. This is due to the load imbalance resulting in one node being overloaded and the GQF being due to high load factor. While the GQF does provide the same memory savings as the TCF, the high performance cost reduces its viability for GPU memory savings.

6.5 GPU Acceleration of MetaHipMer

The GPU version of MetaHipMer is up to an order of magnitude faster than the CPU implementation, and in the worst-case is still over 2× faster in all phases. The performance boost in the different phases of assembly are shown in Figure 6. This speedup comes at the cost of reduced memory, as most GPUs are memory-constrained compared to their CPU counterparts. CPU filters like the VQF [27] could perform a similar function to the TCF, but saving memory on GPU operations is more critical than saving memory on the CPUs.



(a) Load factor of data structures in k-mer analysis.



(b) Total number of base pairs assembled in Giga bp.

Figure 5: Incremental assembly benchmarks. These tests hold the number of nodes constant and scale the input data size by adding more lanes from the WA dataset. Maximum amount of input data that can be assembled on 32 nodes. Using the filter reduces the load factor of the k-mer counting hash table without any impact on the assembly quality (Total number of base pairs assembled).

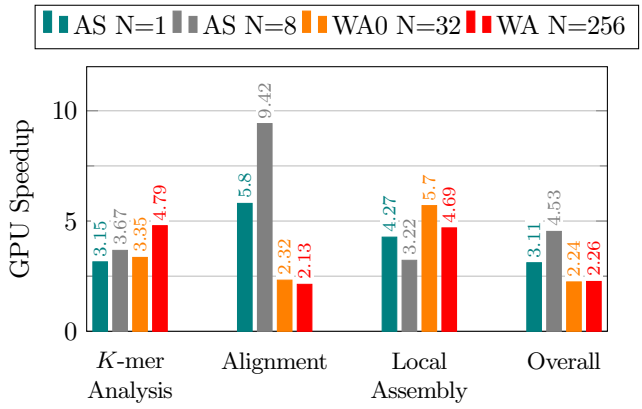


Figure 6: Performance comparison between the CPU and GPU implementations of MetaHipMer. The y-axis represents the relative speedup of the GPU version compared to the CPU version. Results were gathered on Summit. AS is ArticSynth, WA_0 is the first lane of Western Artic, and WA is the full Western Artic dataset. N is the #nodes.

7 Discussion

In this paper, we show how the GPU-based TCF can help overcome the memory and speed trade off in an exascale data analytics application MetaHipMer. Using the TCF, MetaHipMer can perform the k-mer analysis in a single communication pass thereby reducing the peak memory usage without affecting the running time. This further allows MetaHipMer to make use of accelerators like GPUs which help speed up data processing but are constrained in terms of available memory.

GPU filters can alleviate the memory problems associated with working on GPUs while maintaining all the performance gains of these accelerators. New filter designs such as the TCF offer advanced features that can accelerate exascale applications with no downsides. This makes their inclusion an ideal choice when it comes to building applications at scale.

High-performance and feature-rich GPU filters such as the TCF can be further be utilized in other high-performance data analytics applications to help scale to large scale datasets. For example, the TCF can be used to accelerate de Buijn graph construction and traversal similar to the approaches used by Pandey et. al. [24, 26] on CPUs.

Acknowledgments

This research is funded in part by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the Department of Enery (DOE) under contract number DE-AC02-05CH11231, the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We used resources of the NERSC supported by the Office of Science of the DOE under Contract No. DEAC02-05CH11231.

References

[1] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999.

[2] John Bachan, Scott B Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H Hargrove, and Hadia Ahmed. Upc++: A high-performance communication framework for asynchronous computation.

In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 963–973. IEEE, 2019.

[3] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kaner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11), 2012.

[4] Maciej Besta, Raghavendra Kanakagiri, Harun Mustafa, Mikhail Karasikov, Gunnar Rätsch, Torsten Hoefler, and Edgar Solomonik. Communication-efficient jaccard similarity for high-performance distributed genome comparisons. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1122–1132, 2020.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] Alex D Breslow and Nuwan S Jayasena. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment*, 11(9):1041–1055, 2018.

[7] Phillip EC Compeau, Pavel A Pevzner, and Glenn Tesler. How to apply de Bruijn graphs to genome assembly. *Nature biotechnology*, 29(11):987–991, 2011.

[8] Lauro B Costa, Samer Al-Kiswany, and Matei Ripeanu. Gpu support for batch oriented workloads. In *2009 IEEE 28th International Performance Computing and Communications Conference*, pages 231–238. IEEE, 2009.

[9] Niv Dayan and Moshe Twitto. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*, pages 365–378, 2021.

[10] Peter C. Dillinger and Panagiotis (Pete) Manolios. Fast, all-purpose state storage. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 12–31, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] Gil Einziger and Roy Friedman. Counting with tinytable: Every bit counts! In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, ICDCN '16, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[13] Afton Geil, Martin Farach-Colton, and John D Owens. Quotient filters: Approximate membership queries on the gpu. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–462. IEEE, 2018.

[14] Evangelos Georganas, Rob Egan, Steven Hofmeyr, Eugene Goltsman, Bill Arndt, Andrew Tritt, Aydin Buluç, Leonid Oliker, and Katherine Yelick. Extreme scale de novo metagenome assembly. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 122–134. IEEE, 2018.

[15] Steven Hofmeyr, Rob Egan, Evangelos Georganas, Alex C Copeland, Robert Riley, Alicia Clum, Emiley Eloe-Fadrosh, Simon Roux, Eugene Goltsman, Aydın Buluç, et al. Terabase-scale metagenome coassembly with metahipmer. *Scientific reports*, 10(1):1–11, 2020.

[16] Alexandru Iacob, Lucian Itu, Lucian Sasu, Florin Moldoveanu, and Constantin Suciu. Gpu accelerated information retrieval using bloom filters. In *2015 19th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 872–876. IEEE, 2015.

[17] Daniel Jünger, Robin Kobus, André Müller, Christian Hundt, Kai Xu, Weiguo Liu, and Bertil Schmidt. Warpcore: A library for fast hash tables on gpus. In *27th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2020, Pune, India, December 16-19, 2020*, pages 11–20. IEEE, 2020.

[18] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[19] Hunter McCoy, Steven A. Hofmeyr, Katherine A. Yelick, and Prashant Pandey. High-performance filters for gpus. In Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy, editors, *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, pages 160–173. ACM, 2023.

[20] Pall Melsted and Jonathan K Pritchard. Efficient counting of k-mers in DNA sequences using a Bloom filter. *BMC Bioinformatics*, 12(1):1, 2011.

[21] Israt Nisa, Prashant Pandey, Marquita Ellis, Leonid Oliker, Aydın Buluç, and Katherine Yelick. Distributed-memory k-mer counting on gpus. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 527–536, 2021.

[22] Anna Pagh, Rasmus Pagh, and S Srinivasa Rao. An optimal Bloom filter replacement. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 823–829. Society for Industrial and Applied Mathematics, 2005.

[23] Prashant Pandey, Fatemeh Almodaresi, Michael A Bender, Michael Ferdman, Rob Johnson, and Rob Patro. Mantis: A fast, small, and exact large-scale sequence-search index. *Cell systems*, 7(2):201–207, 2018.

[24] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics*, 33(14):i133–i141, 2017.

[25] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 775–787, 2017.

[26] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*, 34(4):568–575, 2017.

[27] Prashant Pandey, Alex Conway, Joe Durie, Michael A Bender, Martin Farach-Colton, and Rob Johnson. Vector quotient filters: Overcoming the time/space trade-off in filter design. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1386–1399, 2021.

[28] Prashant Pandey, Shikha Singh, Michael A Bender, Jonathan W Berry, Martín Farach-Colton, Rob Johnson, Thomas M Kroeger, and Cynthia A Phillips. Timely reporting of heavy hitters using external memory. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1431–1446, 2020.

[29] James Tiedje. Rhizosphere metagenomic dataset. `https://genome.jgi.doe.gov/portal/C55metaG_FD/C55metaG_FD.info.html`, 2013. [Online; accessed 19-July-2021].

[30] Sudharshan S Vazhkudai, Bronis R De Supinski, Arthur S Bland, Al Geist, James Sexton, Jim Kahle, Christopher J Zimmer, Scott Atchley, Sarp Oral, Don E Maxwell, et al. The design, deployment, and evaluation of the coral pre-exascale systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 661–672. IEEE, 2018.