High-Performance Filters for GPUs

Anonymous Author(s)

Abstract

High performance data analytics problems often use filters to approximately store or count a set of items while trading off accuracy for space-efficiency. Filters can also address the limited memory on accelerators, such as GPUs. However, there is a lack of high-performance and feature-rich GPU filters as most advancements in filter research has focused on CPUs.

In this paper, we explore the design space of filters with a goal to develop massively parallel, high performance, and feature rich filters for GPUs. We evaluate various filter designs in terms of performance, usability, and supported features and identify two filter designs that offer the right trade off in terms of performance, features, and usability.

We present two new GPU-based filters, the TCF and GQF, that can be employed in various high performance data analytics applications. The TCF is a set membership filter and supports faster inserts and queries, whereas the GQF supports counting which comes at an additional performance cost. Both GQF and TCF provide point and bulk insertion API and are designed to exploit the massive parallelism in the GPU without sacrificing usability and necessary features. The TCF and GQF are up to 4.4× and 1.4× faster than the previous GPU filters in our benchmarks and at the same time overcome the fundamental constraints in performance and usability in current GPU filters.

1 Introduction

Filters, such as Bloom [8], quotient [4, 6, 20, 21, 39, 44, 46] and cuckoo filters [10, 22], 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.

As scientific and commercial data sets explode in volume and data rates, some of the high performance data analytics pipelines take advantage of the massive parallelism and advanced computing architecture of the GPUs. GPUs have proven to be effective accelerators for machine learning and simulation problems [1, 41], database engines [11, 12, 34, 50, 55, 56], and large-scale genomics pipelines [7, 26, 28, 31, 54].

In this paper, we consider the use of GPUs in filtering, one of the key operations in many data processing and analytics pipelines. GPUs offer both an opportunity for performance improvement and a challenge for data analytics due to the limited GPU memory that is available.

Given the popularity and wide-scale impact of filters there have been many papers in the last decade that advance the theory and practice of filters [2, 4, 6, 9, 10, 13, 18–23, 29, 35, 39, 44, 46, 51, 53]. Most of these papers have focused on improving the state of the art in terms of space usage and performance. A few papers have also explored adding new features in the filter such as deletion, associating small values with hashes, and counting, which are critical for many applications [4, 6, 22, 44, 46].

However, there is very little work on building fast, space-efficient, and feature-rich filters for the GPU. Costa et al. [18] and Iacob et al. [29] showed how to build and query Bloom filters on the GPU. Geil et al. [25] first showed how to build and query a quotient filter on the GPU using the bulk build API. These filter implementations do not offer choices in terms of space usage and false-positive rate trade-off, offer sub-optimal performance, and do not have adequate APIs to be integrated in many data analytics applications. Furthermore, these implementations do not support critical features such as deletion, counting, or associating values with the items, which are required by modern-day applications. Due to the lack of available options modern GPU-accelerated applications often work around the limitations of filters which in turn results in sub-optimal use of resources and further hinders their scalability to larger datasets.

For example, MetaHipMer [26, 28] is an extreme-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 terabyte scale data. MetaHipMer requires a filter that can map fingerprints to small values to weed out singletons during raw data processing and use the output in later stages of the pipeline. It cannot use Bloom filters since Bloom filters do not support associating small values with the items. Similar to the Bloom filter, Geil et al.'s [25] quotient

[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. Counting filters offer no guarantee on the overestimate unlike count sketches. We refer the readers to Goswami et al's [27] paper for a detailed comparison.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

filter (SQF) cannot associate values. In addition, the SQF can only scale up to a few million items and does not offer the right trade off in terms of space usage and false-positive rate. Similarly, many database engines [32, 33, 55] that leverage GPUs to speed up merge and join operations cannot use existing filters as they do not support counting and enumeration of items which are required for those operations.

Designing filters for GPUs comes with a host of unique challenges. The architecture of GPUs, originally designed to accelerate rendering operations, provides massive parallelism at the expense of limited memory, simpler instructions, and synchronization tools. The differences in the architecture between CPUs and GPUs cause filters designed for CPUs to often have sub-optimal performance when ported to GPUs. Thread contention and thrashing are often issues for GPU data structures, even those with large sizes. In addition, the grouping of threads into warps is another constraint on the design, with grouped access patterns providing massive speed boosts over naive implementations. The random access pattern of many existing filters amplifies this problem as threads in a warp are likely to diverge while accessing memory locations randomly throughout the filter.

Our contribution. In this paper, we explore the design space of filters and identify the filters that can exploit the massive parallelism on the GPU without introducing fundamental feature limitations and giving up performance and usability. We identify two filters that offer the appropriate trade-offs in terms of the performance and necessary features. We further develop and evaluate the two new GPU filters, the two-choice filter (TCF) and the GPU-based counting quotient filter (GQF). The TCF does not support counting which enables faster inserts and queries, whereas the GQF supports counting at an additional performance cost. Both the filters support deletions and associating small values with fingerprints.

The TCF is designed to organize fingerprints in blocks sized to fit inside a GPU cache line. It uses cooperative groups to perform insert, query, and delete operations inside these blocks to achieve massive parallelism without any contention. The TCF further uses power-of-two-choice hashing [3] to minimize the load imbalance across fixed-size blocks and achieve a high load factor. The TCF strips out the ability to count in favor of faster inserts and query operations. The TCF offers both concurrent inserts and queries and bulk insertion API. The TCF can represent a set of items approximately and supports deletions, enumeration, and associating small values with items.

The GQF is a GPU-optimized implementation of the counting quotient filter [43]. The GQF is designed to overcome the fundamental limitations of the earlier implementation of the quotient filter [25] on the GPU, such as only supporting a fixed false-positive rate and scaling only to a few million items. The GQF offers all the features that modern data analytics applications demand, e.g., better space-accuracy trade-off, counting, deletions, associating values with items, and resizability. In addition, it offers both concurrent inserts and queries and a bulk insert API, unlike the earlier GPU-based filter implementations.

In the GQF, we exploit the high cache locality of quotient filters to design a novel coordinated lock-free implementation for batch insertions. The lock-free implementation partitions the filter into exclusive-access even-odd regions and assigns threads inside a warp to fixed memory regions to achieve low thread divergence and avoid thrashing. We believe that our even-odd scheme for bulk insertions can also be applied to other linear-probing-based hash tables to accelerate insertions.

Our results. The TCF and GQF offer far better (up to three orders of magnitude in some cases) performance and use less or similar space than other filters on the GPU offering a smaller set of features.

1. The point TCF is up to 4.45× faster for inserts and queries than all filter that support deletions.
2. The GQF is up to 1.93× and 2.4× faster than the GPU-based Bloom filter for inserts and queries respectively.
3. The Bulk TCF achieves an insertion throughput of 3.4 Billion items per second on NVIDIA A100 GPUs.
4. The bulk TCF achieves an insertion throughput of 70% of the Blocked Bloom filter with half the false positive rate.
5. The TCF is over an order of magnitude faster than all other filters for deletions.
6. The GQF supports high throughput counting (800+ Million/sec) on both simulated and real-world datasets.

2 A Brief History of Filters

In this paper, we consider dynamic filters as they have widespread applications in data analytics. Dynamic filters approximately represent a set of items that does not need to be known before the construction. Dynamic filters have seen much more advancement in the last few decades as applications often do not know the set of items in advance. Examples of dynamic filters are Bloom filters [8], quotient filters [4, 20, 21, 39, 45], and cuckoo filters [10, 22].

Bloom filters consume $\log(e)n\log(1/\varepsilon)$ space, which is roughly $\log(e) \approx 1.44$ times more than the lower bound of $n\log(1/\varepsilon)+\Omega(n)$ bits [14]. In contrast, for a set S taken from a universe U, where $|U|=u$, an error-free dictionary requires $\Omega(\log\binom{u}{n}) \approx \Omega(n\log u)$ bits. Bloom filters also incur $\log(1/\varepsilon)$ cache-line misses on inserts and positive queries, giving them poor insertion and query performance.

Blocked Bloom filters [52] overcome the poor cache locality of Bloom filters by constructing a series of smaller Bloom filters each of which is small enough to fit inside a small number of cache lines. The first hash function is used to select a block and rest of the hash functions are used to set/test bits inside the block. However, the cache efficiency comes at the cost of higher false-positive rate. Blocked Bloom

filters have theoretically and empirically higher (up to 5×) false positive rates compared to Bloom filters. See Table 2 for the empirical calculations of FP rate.

Quotient filters [4, 6, 16, 20, 40, 44, 46] represent a set approximately by compactly storing small fingerprints of the items in the set via Robin Hood hashing [15]. The quotient filter uses $1.053(2.125+\log_2 1/\varepsilon)$ bits per element, which is less than the Bloom filter whenever $\varepsilon \leq 1/64$, which is the case in almost all applications. It supports insertion, deletion, lookups, resizing, and merging. The counting quotient filter (CQF) [44], improves upon the performance of the quotient filter and adds variable-sized counters to count items using asymptotically optimal space, even in large and skewed datasets. In the counting quotient filter, we can also associate small values with items either by re-purposing the variable-sized counters [42] to store values or by explicitly storing small values with the remainders in the table [48].

Cuckoo filters [10, 22] also store small fingerprints compactly in a table. However, unlike the quotient filter that uses Robin Hood hashing, the cuckoo filter uses cuckoo hashing to resolve collisions among fingerprints. Cuckoo hashing uses kicking (or cuckooing) to find an empty slot for the new item when all the slots in a bucket are occupied. This results in a cascading sequence of kicks until the filter converges on a new stable state. Inserts become slower as the structure becomes full, and in fact inserts may fail if the number of kicks during a single insert exceeds a specified threshold (500 in the author's reference implementation).

Two-Choice filters [46] organize fingerprints compactly in blocks similar to the cuckoo filter. However, unlike the cuckoo filter, there is no kicking. The blocks in the two-choice filter are larger in size ($\approx \log n$, where n is the number of items which is usually the size of the cache line on most machines) than the cuckoo filter and power-of-two-choice hashing is used to reduce the variance across the blocks and achieve a high load factor. During insertions if both blocks corresponding to a fingerprint are full then the data structure is declared full. The power-of-two-choice hashing enables the filter to probe exactly two cache lines during inserts and queries and write to a single cache line during inserts. Given the larger block sizes the vector quotient filter [46] uses quotienting (similar to the quotient filter) to organize fingerprints inside blocks. It divides the fingerprints into a quotient and remainder part and only stores the remainder in the slot given by the quotient. It uses two additional metadata bit to resolve collisions among quotients.

3 Designing a GPU filter

Here we discuss the design principles needed to build a fast and space efficient filter on the GPU and use them to analyze various filter designs.

3.1 GPU design principles

There are four major design principles to consider when implementing data structures on GPUs:

1. **Low thread divergence:** threads inside a warp should execute the same instruction. This enables writing simple kernels that can exploit massive parallelism in the GPU.
2. **High memory coherence:** threads inside a warp should access the same memory from a local region. Random memory accesses are expensive and cause threads to stall.
3. **High degree of parallelism:** a high number of threads saturate memory bandwidth and hide memory latency.
4. **Atomic operations:** atomic operations help efficient thread scheduling inside a warp. Non-atomic writes and data movements cause slow downs and require locking large memory regions. Locking results in high overheads and affects the overall throughput.

3.2 Analysis of filter designs

We now look at the dynamic filters discussed in Section 2 and evaluate them based on the GPU design principles. Our goal is to identify the filters that offer necessary features such as deletions, counting, and value associations and at the same time satisfy most of the design principles.

Bloom filters are easy to implement on the GPU as they only require test and set operations. These operations can be implemented using atomic operations and achieve low thread divergence. However, each operation results in multiple cache misses and therefore Bloom filters have low memory coherence. They also have sub-optimal space usage. Moreover, Bloom filters do not support deletions, counting[2], and associating small values with items that many data analytics applications require.

Blocked Bloom filters on the other hand are better suited to GPUs. Each operation requires probing inside a single block. They achieve low thread divergence, high memory coherence, a high degree of parallelism, and atomic operations. Thus blocked Bloom filters can satisfy all the GPU design principles. However, blocked Bloom filters have a high false-positive rate compared to Bloom filters and also do not support necessary features like deletions and counting.

Operations in the quotient filter have high cache locality which makes it an appropriate choice to achieve high memory coherence. However, insert operations in the quotient filter requires shifting fingerprints which makes it harder to use atomic operations and also results in high thread divergence. However, the quotient filter can support all the necessary features like deletions, counting, and associating small values with items which makes the quotient filter a highly usable data structure that multiple applications can benefit from.

[2]The counting Bloom filter [23], a variant of the Bloom filter, supports counting but it comes at a high space-overhead which makes it highly inefficient in practice.

It is quite challenging to achieve high speed operations while maintaining all of the features in a GPU implementation of the quotient filter. Geil et al [25] implemented a preliminary version of the GPU quotient filter. However, that implementation was adapted from Bender et al.'s quotient filter [4], which did not have all the features, like counting and value association, and also had higher space overhead. Furthermore, Geil et al's GPU-based quotient filter has implementation-specific limitations (e.g., it supports a fixed false-positive rate and can only be sized to store less than 2^26 items) that makes it more difficult to use in applications.

The cuckoo filter stores fingerprints in fixed size blocks. This design is amenable to high memory coherence and low thread divergence. Atomic operations can also be used to read and write fingerprints. However, the cascading sequence of reads and writes to random memory locations makes the cuckoo filter hard to implement efficiently on the GPU. In particular, at high load factors when the number of kicked items becomes high, each insertion will result in very low memory coherence. Moreover, each kicking operation results in multiple cache-line writes. This makes it challenging to achieve high speed operations in a GPU cuckoo filter. Moreover, cuckoo filters do not support counting and associating small values with items that many data analytics applications require.

The two-choice filter has the advantages of the cuckoo filter design. It has fixed size blocks. Each operation requires probing into exactly two blocks, and inserts and deletes only write into a single block. This results in low thread divergence, high memory coherence, and a high degree of parallelism. However, due to large block sizes a more sophisticated structure is required to maintain fingerprints inside each block. Therefore, it is not straightforward to use atomic operations to read or write fingerprints inside blocks. It is a challenging task to implement a two choice filter on the GPU using atomic operations to achieve high throughput.

3.3 Most efficient GPU filter designs

We now identify the filters that offer necessary features and can achieve high speed operations on the GPU. First, we pick the two-choice filter (TCF). The TCF achieves three out of four design principles. It achieves low thread divergence, high memory coherence, and high degree of parallelism. It also supports deletions unlike the Bloom filter variants. We redesign the TCF to use atomic operations and cooperative groups to exploit massive GPU parallelism. Second, we pick the counting quotient filter (CQF). The CQF offers all the necessary features that modern applications demand. In particular, it supports counting and value associations which are critical features for many applications. However, it is hard to achieve low thread divergence and high parallelism in the CQF. We will redesign the CQF to use a coordinated lock-free approach and achieve massive parallelism and scalability.

4 TCF Implementation

In this section, we give the implementation details of the two choice filter (TCF) on the GPU. We first explain the version that supports concurrent inserts and queries via the use of atomics. We then explain the bulk lock-free version that utilizes sorting to precondition items for faster operations.

In the TCF, we organize the table into blocks. Each block can store B f-bit fingerprints. The blocks are sized to fit inside a GPU cache line. The TCF uses the power-of-two-choice (POTC) hashing scheme to perform operations. In a POTC scheme, every item is assigned two blocks via a pair of unique hashes. 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. The 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. [3].

Inserts and queries inside a block are performed using cooperative groups. A group cooperatively loads the block into shared memory before striding over the block to check for empty slots or the presence of an item. Once an empty slot has been found, the cooperative group ballots for a leader who will attempt an atomicCAS operation to write the item to global memory. On success, the cooperative group returns, while on failure the group will look for a new empty slot within the block and re-ballot to determine the new leader.

4.1 TCF design optimization

There are three factors that dominate the TCF performance: size of the blocks, the bits per item, and size of the cooperative groups.

The size of the blocks determines the number of cache line access during operations. Therefore, we enforce that the size of a block ≤ 128 bytes (a cache line on GPU) which limits the number of accesses to two for the majority of operations.

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 fingerprint size. A larger fingerprint size decreases the false-positive rate but increases the space. The minimum size for an atomicCAS transaction is 2 bytes. With keys set to the minimum CAS size and a block size of 16, the error rate is .04%. However, most practical applications require the error rate to be around 0.1%. To achieve that error rate, we can either increase the block size or decrease the fingerprint size. Increasing the block size has a negative effect on performance as each thread needs to look at more data. Storing 12-bit fingerprints brings down the space usage but 50% of inserts now require two atomic operations, and fingerprints can no longer fully occupy an atomic transaction, meaning that an atomicCAS could fail due to a change in bits outside of the slot being operated on.

The size of the cooperative groups is particularly important to the performance of this filter design, as it provides a trade off between computational and memory efficiency inside of a

warp. Increasing the number of cooperative groups in a warp increases the number of cache lines that can be scheduled for loading, but decreases the number of workers available per block. A more detailed analysis of this phenomenon, along with experimental results of varying the cooperative group size, are found in Section 6.4.

Backing table. To avoid insertion failures (no empty slot in both blocks) before reaching a 90% load factor we use a backing table. We use a small double-hashing backing table sized to 1/100th of the size of the main table for storing any items that fail to be inserted. 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 false-positive queries, as at least one extra block will have to be searched. The TCF can achieve 90% load factor using the backing table.

Shortcut optimization. As shown in Pandey et al. [47], in the case where the primary block has a very low fill ratio, we can safely insert into the primary block without querying the alternate block. This reduces the number of cache loads required to insert by one, improving speed. After empirical testing, we found a 0.75 fill ratio to be the ideal cutoff for this shortcut optimization, as it provided the best performance without affecting the variance between blocks.

4.2 Bulk TCF

The bulk version of the TCF utilizes sorting to increase the efficiency of read/write operations in the GPU. Like reads, writes on a GPU can be coalesced, with up to 128 bytes of contiguous memory being written in one operation. The SM that a warp is staged on has a memory pipeline that is shared between all threads in a warp and operates on cache blocks of 128 bytes. Any thread can freely read from a cache line that has been loaded by the SM, and any adjacent writes inside of a cache line can occur simultaneously.

If the time saved on insertion is less than cost of aggregating items, we can improve the throughput via an aggregation phase. Items are sorted and passed to the bulk TCF as a pointer to a sorted list of items to be inserted into a block. Blocks of the TCF are loaded into shared memory before items are inserted and all reads and writes are performed using shared memory atomics. At the end of the kernel writes occur as coalesced writes to global. This minimizes the data written to global as all writes to global occur as cooperative cache-wide coalesced writes.

Unlike the point TCF, blocks in the bulk version maintain a sorted list of items inside the block. This allows the blocks to be queried in logarithmic time via a binary search, or in linear time for a batch of queries. To efficiently insert while maintaining a sorted order, each cooperative group maintains three lists during insertion: the list of items currently stored in the block, the sorted list of items that can be shortcutted into the block, and the list of items assigned to the block via POTC hashing. The three lists are merged together using a parallel zip strategy, and the resulting block is cooperatively written to global memory.

The bulk filter has an error rate of 0.3% with a block size of 128 and a 16 bits per item. While this is appropriate for most applications, the bulk implementation requires 33% more space per item to achieve the same error rate.

5 GQF Implementation

In this section, we give an overview of Pandey et al.'s [43] counting quotient filter (CQF). We also describe the locking mechanism in the counting quotient filter for thread-safe operation because it acts as the building block in the GPU-based quotient filter (GQF). We finally explain how we design the counting quotient filter for the GPU.

5.1 CQF overview

The counting quotient filter (CQF) stores an approximation of a multiset $S \subseteq \mathcal{U}$ by storing a compact, lossless representation of the multiset $h(S)$, where $h : \mathcal{U} \rightarrow \{0,\ldots,2^p - 1\}$ is a hash function that maps items from the universe $\mathcal{U}$ to a p-bit fingerprint. To handle a multiset of up to n distinct items while maintaining a false-positive rate of at most ε, the CQF sets $p = \log_2 \frac{n}{\varepsilon}$ (see the original quotient filter paper for the analysis [4]).

The counting quotient filter divides $h(x)$ into its first q bits, ***quotient*** $h_0(x)$, and its remaining r bits, ***remainder*** $h_1(x)$. It maintains an array Q of 2^q r-bit slots, each of which can hold a single remainder. When an element x is inserted, the counting quotient filter attempts to store the remainder $h_1(x)$ at index $h_0(x)$ in Q (which we call x's ***canonical slot***). If that slot is already in use, then the counting quotient filter uses uses Robin hood hashing to find the next available empty slot to store $h_1(x)$. All the items that share the same canonical slot are stored together in a ***run*** and a sequence of runs stored contiguously with no empty space is called a ***cluster***. During an insert operation, the next available empty slot is found at the end of the cluster. If an item lands at the start of the cluster then all the items in cluster must be shifted to create an empty space.

5.2 Point insertion API

In the point implementation, each thread acquires exclusive access to a section of memory for writing. Internal remainder shifts are processed using a custom *memmove* function, as the driver API only provides support for *memcpy*, which does not guarantee write safety when the source and destination regions overlap.

To perform an insert operation, the thread needs to lock a big enough region so that shifting items will not corrupt the subsequent region where another thread might be operating. Therefore, the slots are divided into locking regions that are big enough to handle the shifting of remainders during insertions without causing an overflow to the next locking

region. Given that the filter is only filled to 95% load factor, we can safely say with that the maximum cluster size will be less than 8192 slots [43]. In order to guarantee that each insert has at least this many slots to work with, we divide the filter into sections of 8192 slots. An insert thread grabs two locks corresponding to the canonical slot of the item and the lock immediately after it. Locking two consecutive regions in the counting quotient filter ensures that memory corruption bugs are avoided, even if we overflow into the next region during an insert operation. The insert thread holds these locks until all changes are flushed to memory.

The length of the longest cluster is bounded by $O(\frac{\ln 2^q}{\alpha - \ln\alpha - 1})$ with high probability [5, 44], where q is the number of quotient bits, 2^q is the number slots in the QF, and α is the load factor. For example, if $q = 40$ (i.e., 2^{40} slots) and $\alpha = 3/4$, the largest cluster in the filter has 736 slots. On average, clusters are $O(1)$ in size. The theorem gives a high-confidence estimation of the size of the largest cluster when the QF is almost full.

The smallest possible lock implemented in CUDA uses one bit with *AtomicOr* and *AtomicAnd* intrinsics to set and release the locks. However, this implementation has poor performance in CUDA due to the memory contention issues when using atomics in global memory. To perform an operation, atomics require exclusive access to a cache line's worth of memory, e.g., 128 bytes on the Tesla V100s. With one bit per lock, there would be 1024 locks in a cache line, each with dozens or even hundreds of simultaneous locking attempts in the worst case. This would lead to heavy thread contention among threads acquiring locks and cause the vast majority of threads to thrash. To ameliorate this, we used cache-aligned locks, as the number of locks relative to the total size of the data structure is small enough that they only contribute a small percentage to the overall space usage.

The GQF implementation that uses locking has the high overhead of acquiring and releasing locks to perform operations. Furthermore, each thread locks two locking regions (a locking region comprises 8192 slots) and that creates contention among threads that are trying to operate in the same region. However, the locking implementation is necessary to support the point insertion and query API in the GQF.

5.3 Bulk insertion API

In the bulk API, we group items that hash to the same region and a single thread is assigned to each region for inserting all the grouped items. This guarantees that threads will have exclusive access to regions. However, there can still be memory corruption if two threads are simultaneously performing insertion in consecutive regions and there is an overflow from one region to the other during the insertion. To avoid the memory corruption, the threads would still need to acquire locks on two consecutive region.

To avoid the overhead of locking, we perform the insert operation in two phases. In the first phase, items belonging to even regions are inserted, with each thread assigned a specific region. Since there are no threads operating in the odd regions we can safely perform insertions without any memory corruption issues. In the second phase, the items belonging to the odd regions are inserted.

This "Even-odd region" insert scheme maximizes the number of inserts that can safely occur simultaneously. Although it only allows insertion into half of the regions during a given phase, for large filter sizes the number of regions far exceeds the number of threads, allowing for full saturation of the GPU. Each region is sized to 8192 slots and phased insertion guarantees that threads are $\approx$ 16K slots apart and will always find empty slots before overflowing into the next region.

Our implementation of this insert scheme uses temporary ***buffers*** to hold items corresponding to each region. To efficiently distribute items into regions we use atomic operations to set the buffer sizes and assign each item an index in the buffer. In practice, we do not allocate temporary buffers. Instead, we use pointers into the input array to mark the boundaries for the buffers. This saves memory and the time required to allocate memory at run time.

Sorting hashes to minimize shifting. Internally, the GQF stores items akin to a linear hash table, with the remainders in a run or cluster in sorted order. New items inserted into a run must therefore shift any remainders greater than them in order to maintain the sorted structure. These shifts are the dominating factor in time spent in insertion. However, as these shifts only occur when the new remainder is smaller than remainders in the run, we could avoid these memory shifts by inserting remainders (or hashes) in a sorted order. If the entire dataset were sorted before insertion, no shifts would be required, as each new item inserted would be the largest item and could therefore safely occupy the next empty slot. A variant of this holds true when the input dataset is batched: while it is impossible to avoid shifting items already in memory, sorting the input batch removes any extraneous memory shifts of items in the current batch.

Our implementation uses the Thrust library [38] to perform an in-place sort on the input data. After sorting, the starts of buffers are set using successor search which finds the index of the smallest item greater than or equal to the minimum hash of the current buffer. This eliminates the need to use atomics to set the buffers which in turn saves time during multiple phases of insertion.

5.4 Optimization for skewed distributions

Datasets with skewed distributions (where counts of the items are derived from a power-law or a Zipfian distribution [17]) cause high contention among threads in the point insert API and load imbalance in the bulk insert API. This results in much slower insertion throughput and limited scaling with increasing filter sizes.

The datasets with skewed distribution contain a lot of repeated items. During insertions, multiple threads try to insert copies of the same item in the dataset creating high contention on the locks and load imbalance in a few memory regions.

For the bulk insert API, we take the map-reduce approach to avoid the high contention in the GQF. We first sort the batch of input items and then perform a reduction to compress the duplicate items into ⟨item, count⟩ pairs. This reduction allows us to perform a single insertion with the aggregate count for every item in the batch instead of multiple insertions corresponding to each instance of repeated item. This amortizes the cost of acquiring locks and performing insertions. It further enables us to reduce the load imbalance across regions in the bulk insert API resulting in high insertion throughput. In our implementation, mapping and reduction are handled by the Thrust library [38].

6 Evaluation

In this section, we evaluate the performance of various GPU filter implementations. We include our implementations of the two-choice filter (TCF) and GPU-based counting quotient filter (GQF). We compare our filter implementations against Geil et al.'s [25] standard quotient filter (SQF) and rank-select quotient filter (RSQF). The SQF is a GPU implementation of the quotient filter and supports insertions, queries, and deletions. The RSQF does not supports deletions. Both SQF and RSQF do not support counting. We configure the SQF and RSQF to achieve the best performance based on author's recommendations.

As a baseline for the performance of a filter that does not deletions, we also include the Bloom filter (BF) and blocked Bloom filter (BBF) in our evaluation. The BF and BBF are not directly comparable to other filters used in the evaluation as they do not support similar features. The BBF is taken from Junger et al. [30] and is configured according to the author's recommendation to achieve best performance. We modified a C++ BF implementation [49] to a 1-bit encoded GPU implementation using CUDA atomic bitwise operations.

We evaluate each filter on two fundamental operations: insertions and lookups. Lookups are evaluated both for items that are present and for items that are not present in the filter. Our evaluation of filters is split on the status of the filter as either *bulk* or *point* API. Point filters have device-side APIs and can be called to insert or query a single item while bulk filters must be called from a host function. The TCF and GQF support both bulk and point APIs. We compare our bulk implementation of the TCF and GQF with the SQF and RSQF as they both are designed for bulk API. We compare our point implementations of the TCF and GQF with the Bloom filter and blocked Bloom filter. Both the Bloom and blocked Bloom implementations only support point API.

Please refer to Table 1 for a complete list of API supported by various filters. Only the GQF and TCF support both bulk and point modes for insert, query, delete, and count operations. We compare the GQF and TCF against other filters for insert and query operations, only against the SQF for delete operations. The GQF is compared against no other filter for counting as no other filter supports counting.

Filter	Insert		Query		Delete		Count	
	Point	Bulk	Point	Bulk	Point	Bulk	Point	Bulk
GQF	✓	✓	✓	✓	✓	✓	✓	✓
TCF	✓	✓	✓	✓	✓	✓		
BF	✓	✓	✓	✓				
SQF		✓		✓		✓		
RSQF		✓		✓				

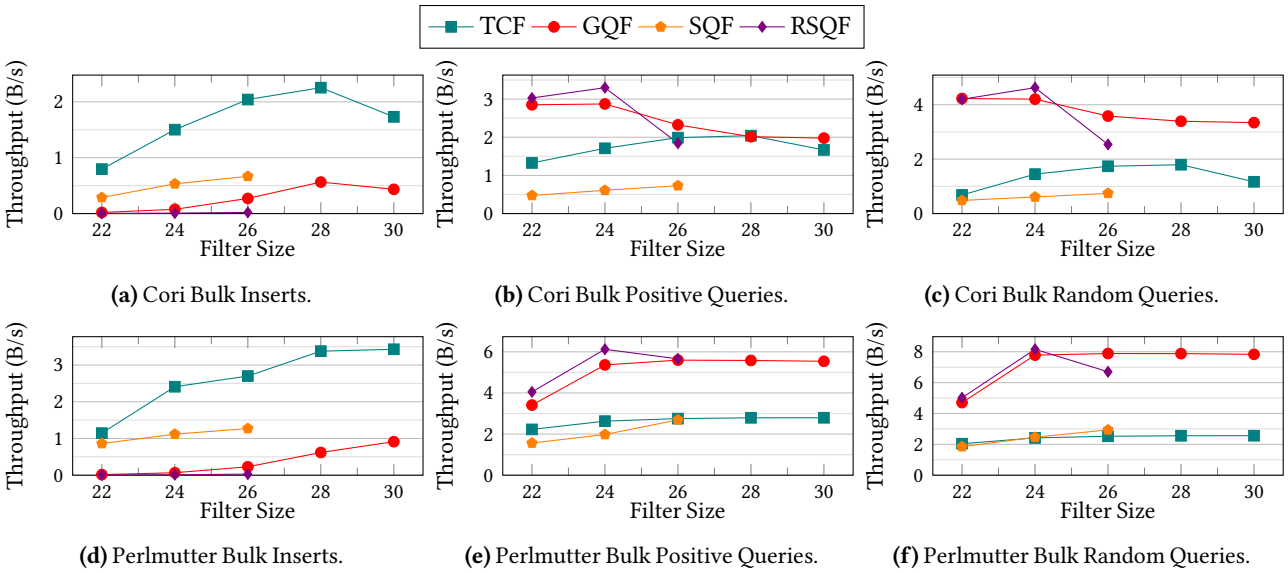
Table 1. API supported by various filters. The GQF is the only filter that supports a range of operations. RSQF can support deletes but it is not implemented by the authors.

Microbenchmarks setup. Our evaluation setup includes all the micro benchmarks employed by filter data structure papers [4, 6, 10, 21, 22, 24, 25, 44, 46] in the past.

We measure performance on raw inserts and lookups as follows. We generate 64-bit input items from the hashed output of a cuRand XORWOW generator. Items are inserted into an empty filter until it reaches its maximum recommended load factor (e.g., 90%). The workload is divided into slices, each of which is 5% of the load factor. These slices are generated on-the-fly to maximize the memory available for the filters. For successful lookups, we query items that are already inserted. For random lookups, we generate a different set of 64-bit hashes than the set used for insertion. This is done by using the hashed outputs of an XORWOW generator set with a different seed. We report aggregate throughput of the operations to insert a set of items.

One challenge that we face in designing our experiments is that the filters do not all support the same false-positive rate. For example, the GQF supports 8, 16, 32, and 64 bit remainders in order to keep the slots in the table machine-word aligned. This helps simplify the GPU implementation by avoiding memory conflicts when multiple threads are modifying different slots. However, SQF and RSQF filters only support remainder sizes of 5 and 13 as they pack the 3 metadata bits along with the remainder in 8 and 16 bit machine words. They further require the sum of the quotient and remainder bits to be less then 32. Therefore, they can only support up to 2^{26} items with 5-bit remainders and 2^{18} items with 13-bit remainders.

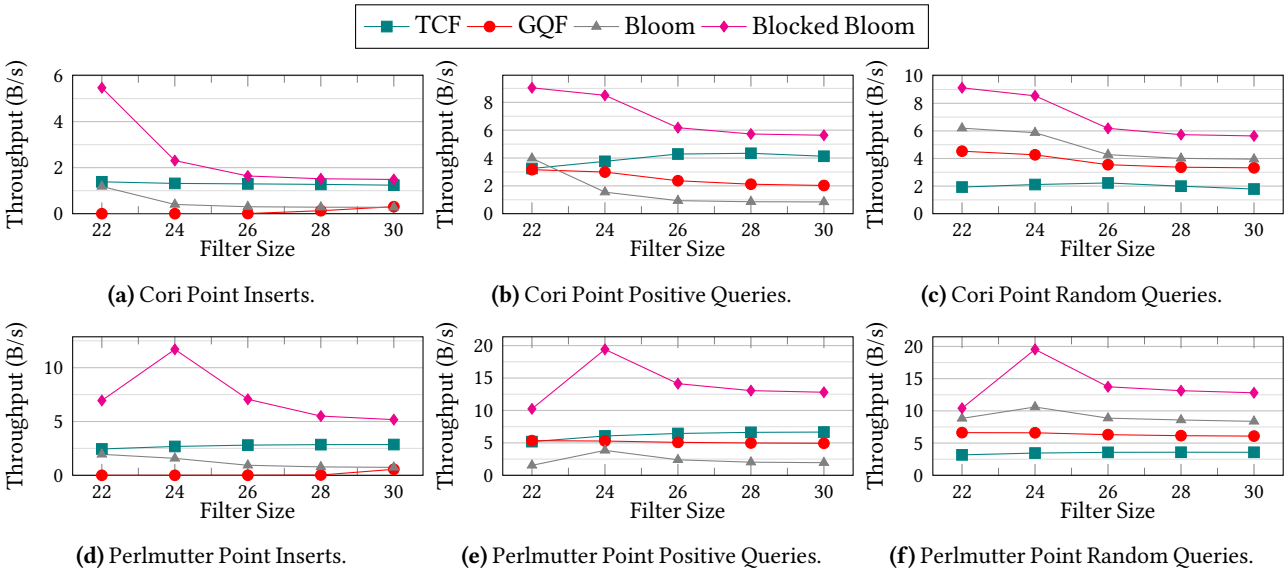
We pick a target false-positive rate of .1% and configure each filter to get as close to this false positive rate as possible. We use 8-bit remainders in the GQF. We use 7 hashes and 10.1 bits per item in the Bloom and blocked Bloom filter. We use 5-bit remainders for the SQF and RSQF and although this results in almost an order-of-magnitude higher false-positive rates, it supports the largest number of items (2^{26}) for these implementations. The smallest TCF word alignment under this error rate is 16 bits, so we report the results from this



(a) Cori Bulk Inserts. **(b)** Cori Bulk Positive Queries. **(c)** Cori Bulk Random Queries.

(d) Perlmutter Bulk Inserts. **(e)** Perlmutter Bulk Positive Queries. **(f)** Perlmutter Bulk Random Queries.

Figure 1. Bulk Aggregate comparison between filter types with 1 batch.



(a) Cori Point Inserts. **(b)** Cori Point Positive Queries. **(c)** Cori Point Random Queries.

(d) Perlmutter Point Inserts. **(e)** Perlmutter Point Positive Queries. **(f)** Perlmutter Point Random Queries.

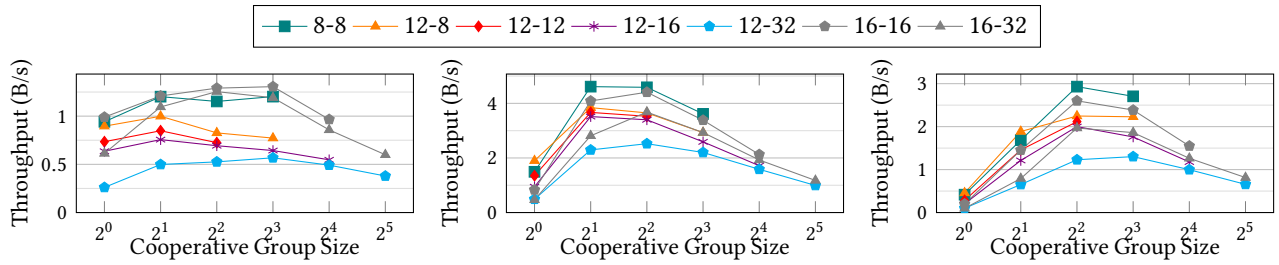
Figure 2. Aggregate comparison between point filter types. Filter operations occurred in batches of 5% of the dataset size.

variation of the filter. Table 2 shows the empirical space usage, false-positive rate, and bits-per-item (BPI) of different filters in these experiments. We measure the space-usage and false-positive rates empirically.

We evaluate the performance of these filters in the GPU memory and hence we size the filters in our experiments so that they can always reside in the GPU memory.

Counting benchmark setup. The counting benchmarks include three datasets with different count distributions. The uniform-random dataset contains items drawn from a uniform-random distribution with almost no duplicates. The uniform-random count dataset contains items where the counts of items are drawn from a uniform-random distribution between 1 and 100. The zipfian count dataset contains items where the counts of items are drawn from a Zipfian distribution (the coefficient is 1.5 and items are chosen from a universe of the same size as the dataset). All the items in the dataset are inserted in one big batch in the GQF.

We also include a real-world genomic dataset for the counting benchmark. We took a raw sequencing file, *M. balbisiana*, from the Squeakr [45] benchmark dataset and extracted k-mers for counting. Squeakr [45] is a k-mer counter which is built using the CQF. With the GQF, we can also port Squeakr to GPUs and accelerate the k-mer counting process.



(a) Inserts. **(b)** Positive queries. **(c)** Random Queries.

Figure 3. Comparison between cooperative group sizes. All tests were run on filters sized to 2^{28}. The left number in a label is fingerprint size and the right is the block size.

Machine specification. Our microbenchmarks and counting benchmarks were run on Cori's [36] and Perlmutter's [37] GPU nodes. Cori nodes consists of NVIDIA Tesla V100 with 5120@1445MHz microprocessors, 16 GB 4096-bit HBM2 memory, and an active thread limit of 82,000 simultaneous threads. Perlmutter nodes consists of NVIDIA A100 Tensor Core GPU with 6912@1410MHz 40 GB 5120-bit HBM2 memory and an active thread limit of 110,000 threads.

6.1 Point API Performance

The TCF has the highest insert and query performance among the filters that support insertion, queries, and deletions. It requires at most two cache line probes and one write for insertions and queries which is much smaller than all other filters.

The overhead of the backing table is negligible as less than 0.07% of items go in the backing table. However, for negative queries (i.e., the items not present in the filter), the backing table adds to the worst-case performance: the query must check at least one bucket in the backing table, and can probe up to 20 buckets in the worst case. In practice, the average performance of insert and query operations is much better due to the shortcut optimization mentioned in Section 4.

The TCF has a higher ($\approx 2\times$) false-positive rate compared to the GQF and BF in this evaluation. However, the TCF supports multiple configurations to offer a multiple trade offs in terms of the space usage and false-positive rate. We have evaluated the performance of various TCF configurations in Section 6.4.

The GQF performance is slower compared to the TCF due to the overhead of locking to perform point insertions. The locking implementation requires us to maintain separate locks for each chunk in the GQF and this causes lock thrashing. Based on the positive query performance, the GQF can reach a slot for insertion faster than the BF can operate on all 7 bits, as each bit requires a different cache load in the BF. However, the cost of locking is so prohibitive on GPUs that the BF is faster for insertions as all operations occur without thrashing.

6.2 Bloom and Blocked Bloom Filter

The BBF is the faster of the two filters. It requires a single cache line operation and used atomicOR which faster than atomicCAS required by other filters. However, the BBF has

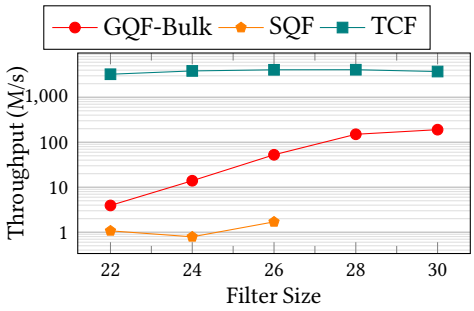


Figure 4. Deletion performance of GQF bulk, SQF, and TCQF on Cori GPU nodes. The x-axis shows $\log n$, where n is the number of slots in the filter. SQF only support up to 2^{26} slots.

$\approx 5.5\times$ higher false positive rate when compared to a Bloom filter with the same bits per item.

The Bloom filter has relatively low throughput on inserts and random queries, as it needs to check multiple random slots within the filter, each of which requires a different cache line load. The BF shows relatively high throughput for random lookups, as it has a high probability of finding a zero and terminating the search early.

The Bloom and blocked Bloom filters have outlier performance at 2^{22} for Cori and 2^{24} for Perlmutter. This is due to these filters being small enough to fit within the L2 cache, allowing for faster memory operations and saturating all the GPU threads efficiently.

6.3 Bulk API performance

The bulk TCF is the fastest filter for inserts, with a maximum throughput of over 3.4 Billion per second on Perlmutter. However, as this filter relies on binary search to find items within a bucket, it has lower throughput on queries, topping out at ≈ 2 Billion per second for both positive and random queries. The bulk SQF has the next highest insert throughput, though the sorted bulk lookup strategy used in the SQF has lower throughput than the other filters.

The throughput of the bulk GQF is based on the size of the filter, so we see an increase in performance as the filter grows, stopping at 2^{28} when the parallelism in the GPU is saturated. The queries of the bulk GQF scale directly with the

GQF		BF		SQF		RSQF		Bulk TCQF		TCQF		Blocked Bloom	
FP	BPI	FP	BPI	FP	BPI	FP	BPI	FP	BPI	FP	BPI	FP	BPI
0.19%	10.68	0.15%	10.10	1.17%	9.7	1.55%	7.87	0.36%	16	.024%	16	.71%	9.73

Table 2. False-positive rate (FP) and bits per item (BPI) of various filters for experiments in Figure 1 and Figure 2.

number of items, so the positive and random queries show high performance even on small filter sizes.

The RSQF has very high throughput on both types of queries. The performance drops as the filter grows to 2^{26}, due to the filter exceeding the 8 MB size of the V100 L2 cache. The filter has very poor performance on inserts, topping out at 8 Million per second, roughly three orders of magnitude lower than the other filters. As the RSQF and GQF have very similar internals, there is no reason the inserts of the filter could not be accelerated. However, an optimized function for inserts is the provided by the authors.

For inserts, all of the bulk filters show increasing throughput as the size of the problem is increased. The insert schemes used in these filters map CUDA threads or warps to sections of memory. This results in far less active threads than the point filters, which map warps to individual items and can quickly reach saturation.

6.4 TCF variations

Figure 3 shows the performance effects of modulating the cooperative group size for a variety of TCF filter variations. These results show that there is an optimal cooperative group sizing for each filter variation. For the majority of the configurations, this size is 4. These optimal sizes are an effect of the trade off between compute and memory latency due to how warps, and by extension cooperative groups, are scheduled on streaming multiprocessor.

Shrinking the cooperative groups increases the saturation of the memory pipeline while lowering the amount of compute available per cooperative group. Increasing the size of the cooperative group gives less divergence and better compute throughput at the expense of less memory operations being scheduled. When memory and compute are balanced, the filter can entirely overlap computation and communication, leading to the most efficient performance. For most designs, this optimal point occurs at a cooperative group of size 4, though some of the larger bucket designs also perform well at 8 due to the extra work to traverse a bucket.

The 8 and 16 bit versions of the filter have the fastest performance, as inserts and queries can be performed in one transaction. As 50% of operations require two memory transactions, the 12 bit filters are slower than their counterparts.

6.5 Deletion performance

Figure 4 shows the performance for deletions for filters that support the operation. The TCF is an order of magnitude faster for deletes than the GQF, as the filter deletes items by replacing them with a dedicated tombstone key. This means that deletions can be done with one atomicCAS operation.

Size	UR	UR count	Zipfian count	Zipfian Count (MR)	k-mer count
22	25.318	30.763	3.676	34.888	23.625
24	101.804	110.833	4.777	169.637	90.722
26	321.150	350.824	4.995	508.156	296.130
28	566.038	798.353	4.520	806.766	507.373

Table 3. Aggregate insertion throughput of GQF (Million operations/sec) for inserting (counting) items from datasets with three different distributions. Uniform-random (UR) datasets, Uniform-random (UR) count, Zipfian count (MR): count of items are drawn from a Zipfian distribution using the Map-reduce implementation from section 5.4.

The GQF is up to two orders of magnitude faster for deletion than the SQF. This is due to the even-odd phased approach that minimizes the amount of left shifting that is required during a delete operation. Left shifting is further reduced due the sorting of items before the operation and deleting larger items first. Overall deletes are slower compared to the inserts in the GQF as deletes are more compute intensive.

6.6 Counting performance in the GQF

Table 3 shows the aggregate insertion throughput for inserting (counting) items from datasets with three different distributions. Counting items from a Zipfian distribution using the map-reduce strategy explained in Section 5.4 achieves the highest throughput.

When counting items, especially when the counts are smaller than the maximum value in a GQF slot (which is 256 for a 8-bit slot), the insertions mostly involve incrementing the count of an existing item, which can be done fairly efficiently without the need to shift remainders. However, when the distribution is skewed, as in the case of a Zipfian distribution, many threads contend to insert the same item, causing long stalls which reduce throughput. This shows that the GQF is an efficient counting filter for datasets with small counts.

For the k-mer counting dataset, the GQF supports throughput of more than 500M k-mers per second which is orders of magnitude faster than the throughput of Squeakr [45], a CPU k-mer counter built using the CQF. With the GQF, we can easily port Squeakr to GPUs and accelerate k-mer counting.

6.7 Discussion

For most data analytics applications, the TCF is the choice for a GPU filter. It offers the right trade off between space efficiency and false positive rate, maintains high throughput for all operations, scales to larger datasets, and can be configured for a wide range of filtering use cases. For applications that require no associativity and are not bound by space usage or false positive rate, the blocked Bloom filter (BBF) is a good choice. The rich features of the GQF are critical to many analytics applications like MetaHipMer, database merges, etc. However, this comes at an additional performance cost. The GQF is often the only available filter option for many applications that need GPUs to accelerate complex data processing.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable Bloom filters. *Journal of Information Processing Letters*, 101(6):255–261, 2007.

[3] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations (extended abstract). In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '94, page 593–602, New York, NY, USA, 1994. Association for Computing Machinery.

[4] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kaner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11), 2012.

[5] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: how to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, 2012.

[6] M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. In *Proc. 3rd USENIX Workshop on Hot Topics in Storage (HotStorage)*, June 2011.

[7] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik. Communication-efficient jaccard similarity for high-performance distributed genome comparisons. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1122–1132. IEEE, 2020.

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

[9] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting Bloom filters. In *European Symposium on Algorithms (ESA)*, pages 684–695. Springer, 2006.

[10] A. D. Breslow and N. 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.

[11] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[12] S. Breß and G. Saake. Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms. *Proceedings of the VLDB Endowment*, 6(12):1398–1403, 2013.

[13] M. Canim, G. A. Mihaila, B. Bhattacharjee, C. A. Lang, and K. A. Ross. Buffered Bloom filters on solid state storage. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 1–8, 2010.

[14] L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 59–65, 1978.

[15] P. Celis, P.-A. Larson, and J. I. Munro. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 281–288, 1985.

[16] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE T. Comput.*, 33(9):828–834, 1984.

[17] B. Corominas-Murtra and R. V. Solé . Universality of zipf's law. *Physical Review E*, 82(1), jul 2010.

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

[19] B. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H. Du. BloomFlash: Bloom filter on flash-based storage. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS)*, pages 635–644, 2011.

[20] P. C. Dillinger and P. P. 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.

[21] G. Einziger and R. 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.

[22] B. Fan, D. G. Andersen, M. Kaminsky, and M. 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.

[23] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[24] A. Geil. Quotient filters: Approximate membership queries on the GPU. http://on-demand.gputechconf.com/gtc/2016/presentation/s6464-afton-geil-quoetient-filters.pdf, 2016.

[25] A. Geil, M. Farach-Colton, and J. 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.

[26] E. Georganas, R. Egan, S. Hofmeyr, E. Goltsman, B. Arndt, A. Tritt, A. Buluç, L. Oliker, and K. Yelick. Extreme scale de novo metagenome assembly. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 122–134. IEEE, 2018.

[27] M. Goswami, D. Medjedovic, E. Mekic, and P. Pandey. Buffered count-min sketch on ssd: Theory and experiments. *arXiv preprint arXiv:1804.10673*, 2018.

[28] S. Hofmeyr, R. Egan, E. Georganas, A. C. Copeland, R. Riley, A. Clum, E. Eloe-Fadrosh, S. Roux, E. Goltsman, A. Buluç, et al. Terabase-scale metagenome coassembly with metahipmer. *Scientific reports*, 10(1):1–11, 2020.

[29] A. Iacob, L. Itu, L. Sasu, F. Moldoveanu, and C. 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.

[30] D. Jünger, R. Kobus, A. Müller, C. Hundt, K. Xu, W. Liu, and B. 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.

[31] R. Kobus, A. Müller, D. Jünger, C. Hundt, and B. Schmidt. Metacache-gpu: ultra-fast metagenomic classification. In *50th International Conference on Parallel Processing*, pages 1–11, 2021.

[32] Y. Kozawa, T. Amagasa, and H. Kitagawa. Gpu acceleration of probabilistic frequent itemset mining from uncertain databases. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 892–901, 2012.

[33] J. Krueger, M. Grund, I. Jaeckel, A. Zeier, and H. Plattner. Applicability of gpu computing for efficient merge in in-memory databases. In *ADMS@ VLDB*, pages 19–26, 2011.

[34] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proceedings of the VLDB Endowment*, 9(14):1647–1658, 2016.

[35] G. Lu, B. Debnath, and D. H. Du. A forest-structured Bloom filter with flash memory. In *Proceedings of the 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, 2011.

[36] NERSC. Cori. https://docs-dev.nersc.gov/cgpu/.

[37] NERSC. Perlmutter. https://www.nersc.gov/systems/perlmutter/.

[38] NVIDIA. Thrust. https://docs.nvidia.com/cuda/thrust/index.html.

[39] A. Pagh, R. Pagh, and S. S. 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.

[40] A. Pagh, R. Pagh, and S. S. Rao. An optimal Bloom filter replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 823–829, 2005.

[41] Z. Pan, F. Zhang, H. Li, C. Zhang, X. Du, and D. Deng. G-slide: A gpu-based sub-linear deep learning engine via lsh sparsification. *IEEE Transactions on Parallel and Distributed Systems*, PP:1–1, 12 2021.

[42] P. Pandey, F. Almodaresi, M. A. Bender, M. Ferdman, R. Johnson, and R. Patro. Mantis: A fast, small, and exact large-scale sequence-search index. *Cell systems*, 7(2):201–207, 2018.

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

[44] P. Pandey, M. A. Bender, R. Johnson, and R. 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.

[45] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*, 34(4):568–575, 2017.

[46] P. Pandey, A. Conway, J. Durie, M. A. Bender, M. Farach-Colton, and R. 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.

[47] P. Pandey, A. Conway, J. Durie, M. A. Bender, M. Farach-Colton, and R. Johnson. Vector quotient filters: Overcoming the time/space trade-off in filter design. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1386–1399, New York, NY, USA, 2021. Association for Computing Machinery.

[48] P. Pandey, S. Singh, M. A. Bender, J. W. Berry, M. Farach-Colton, R. Johnson, T. M. Kroeger, and C. 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.

[49] A. Partow. Bloom filter library. http://www.partow.net/programming/bloomfilter/index.html, 2021. [Online; accessed 19-July-2021].

[50] R. A. Patta, A. R. Kurup, and S. M. Walunj. Enhancing speed of sql database operations using gpu. In *2015 International Conference on Pervasive Computing (ICPC)*, pages 1–4. IEEE, 2015.

[51] F. Putze, P. Sanders, and J. Singler. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121, 2007.

[52] F. Putze, P. Sanders, and J. Singler. Cache-, hash-and space-efficient bloom filters. In *Experimental Algorithms*, pages 108–121. Springer, 2007.

[53] Y. Qiao, T. Li, and S. Chen. Fast Bloom filters and their generalization. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 25(1):93–103, 2014.

[54] E. A. Sitaridi and K. A. Ross. Gpu-accelerated string matching for database applications. *The VLDB Journal*, 25(5):719–740, 2016.

[55] P. T. Strohm, S. Wittmer, A. Haberstroh, and T. Lauer. Gpu-accelerated quantification filters for analytical queries in multidimensional databases. In *New Trends in Database and Information Systems II*, pages 229–242. Springer, 2015.

[56] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with gpus. *Proceedings of the VLDB Endowment*, 7(11):1011–1022, 2014.