UC Davis
Electrical & Computer Engineering

Title
Quotient Filters: Approximate Membership Queries on the GPU

Permalink
https://escholarship.org/uc/item/3v12f7dn

Authors
Geil, Afton
Farach-Colton, Martin
Owens, John D

Publication Date
2018-05-01

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Quotient Filters: Approximate Membership Queries on the GPU

Afton Geil
University of California, Davis
angeil@ucdavis.edu

Martin Farach-Colton
Rutgers University
farach@cs.rutgers.edu

John D. Owens
University of California, Davis
jowens@ece.ucdavis.edu

***Abstract*—In this paper, we present our GPU implementation of the quotient filter, a compact data structure designed to implement approximate membership queries. The quotient filter is similar to the more well-known Bloom filter; however, in addition to set insertion and membership queries, the quotient filter also supports deletions and merging filters without requiring rehashing of the data set. Furthermore, the quotient filter can be extended to include counters without increasing the memory footprint. This paper describes our GPU implementation of two types of quotient filters: the standard quotient filter and the rank-and-select-based quotient filter. We describe the parallelization of all filter operations, including a comparison of the four different methods we devised for parallelizing quotient filter construction. In solving this problem, we found that we needed an operation similar to a parallel scan, but for non-associative operators. One outcome of this work is a variety of methods for computing parallel scan-type operations on a non-associative operator.**

For membership queries, we achieve a throughput of up to 1.13 billion items/second for the rank-and-select-based quotient filter: a speedup of 3x over the BloomGPU filter. Our fastest filter build method achieves a speedup of 2.1–3.1x over BloomGPU, with a peak throughput of 621 million items/second, and a rate of 516 million items/second for a 70% full filter. However, we find that our filters do not perform incremental updates as fast as the BloomGPU filter. For a batch of 2 million items, we perform incremental inserts at a rate of 81 million items/second – a 2.5x slowdown compared to BloomGPU's throughput of 201 million items/second. The quotient filter's memory footprint is comparable to that of a Bloom filter.

***Keywords*-GPU computing; algorithms; data structures; approximate membership queries; Bloom filter**

I. Introduction

In this work, we focus on an *approximate membership query* (AMQ) data structure for the GPU. AMQs, such as Bloom filters [1], are probabilistic data structures that support lookup and update operations on a set S of keys. The chief advantage of AMQs lies in their space efficiency: they use much less space than traditional dictionaries like hash tables. This advantage is particularly important on GPUs, because even today's most powerful GPUs have a relatively small memory (e.g., NVIDIA's Tesla P100 has 16 GB of DRAM). Since many databases, networks, and file systems benefit from the quick filtering of negative queries (often to avoid costly disk or network accesses), AMQs have found wide use. Such applications are emerging research areas on GPUs [2], [3], [4].

This space advantage comes with a tradeoff: in an AMQ, membership queries are only approximate. For a key $k \in S$, $\textsc{lookup}(k)$ returns "present," but for $k \notin S$, $\textsc{lookup}(k)$ can also return "present," with probability at most ϵ, where ϵ is a tunable false-positive rate. An AMQ that has false positive rate ϵ requires at least $n \log(1/\epsilon)$ bits, and AMQs exist that achieve this bound, up to low order terms. So the introduction of a false positive rate allows the AMQ to use many fewer bits than an error-free data structure.

Bloom filters (BF) are the most well-known AMQ. A Bloom filter represents a set with a bit array. To insert a value, the filter hashes the value using k hash functions whose outputs each correspond to a location in the bit array and sets the bit at each of these locations. To perform a lookup on a value, the BF computes the k hashes and checks whether the bits at all of the corresponding locations are set.

The Bloom filter is straightforward to implement, but has three significant shortcomings: it achieves poor data locality, it does not support delete operations, and it is still a multiplicative factor bigger than the optimal bound noted above. We implement an alternative to the BF: the quotient filter (QF). The quotient filter [5] is designed to maintain locality of data, and beyond supporting all the functionality of the BF, it also supports deletions and the merging of filters. Additionally, the QF can be extended to include counters [6]. We implement two versions of the QF on the GPU, the standard quotient filter (SQF) and the rank-and-select-based quotient filter (RSQF), and compare their relative strengths and weaknesses on this massively parallel architecture. Prior to our work, full QF implementations have been limited to the CPU.

We describe new algorithms for parallel inserts into SQFs and RSQFs. We also investigate techniques for parallelizing a bulk build of the filter, when a significant portion of the full dataset is available at the outset. We find that this involves implementing a parallel scan with a non-associative operator, and we present implementations of three distinct approaches to this problem. We show that our GPU SQF achieves significantly faster lookups, has faster bulk build times, and uses significantly less memory than BloomGPU. In addition to enabling new applications with increased functionality, our GPU quotient filters can be used as a drop-in replacement for a Bloom filter in any of their existing GPU applications [7], [8], [9], [10], [11], [12].

II. Related Work

Prior work on AMQs for the GPU concentrates on Bloom filters. Much of this work has focused solely on using the GPU to accelerate lookup queries, using the CPU for filter construction and updates [7], [8], [9], [10], [11]; however, Costa et al. [13] and Iacob et al. [12] do implement both the filter build and queries on the GPU. Costa et al.'s implementation was open-sourced, so we chose to use their filter as our primary reference for comparison. Their BloomGPU filter parallelizes queries in a straightforward way, by assigning one insert or lookup operation to each thread.

There have been two previous parallel quotient filter implementations on CPUs. Dutta et al. [14] implement a parallel version of their streaming quotient filter, an AMQ designed for removing duplicates from streams of data. Pandey et al. [6] also implement a multithreaded version of their counting quotient filter, which uses the same structure as their rank-and-select-based quotient filter, described in Section III-B. Their implementation depends on per-thread locking that does not scale to the parallelism of a GPU.

III. The Quotient Filter

This section describes the standard quotient filter and rank-and-select-based quotient filter and algorithms for serial operations on these data structures.

A. Standard Quotient Filters

The *quotient filter* [5], which we refer to in this paper as the *standard quotient filter* (SQF), is an AMQ that represents a set S from a universe U by a set of fingerprints. Let $f : U \to [2^p]$ be a hash function that hashes elements of U into p-bit strings. Let $F = f(S) = \{f(x)|x \in S\}$ be the set of hash values of the elements of S. To perform an operation $\text{LOOKUP}(x)$, the filter checks whether $f(x) \in F$. The QF stores these fingerprints losslessly. Therefore, all false positives arise from collisions in the hash function, where $f(q) \in F$ for a query $q \notin S$.

To store the set F, divide each of the p-bit hash values into its upper and lower bits. The *quotient*, $f_q(x)$, is comprised of the q high order bits, and the *remainder*, $f_r(x)$, is comprised of the $r = p - q$ low order bits. A QF can be thought of as a hash table with chaining, where the quotients are the hash values and the remainders are the values stored in the table, as shown in the top of Figure 1. To insert a fingerprint $f(x)$ into the filter, store the remainder, $f_r(x)$ in the $f_q(x)$-th bucket. Although only r bits per item are stored, this scheme allows the complete fingerprint to be recovered by recombining the remainder value and the bucket number.

An SQF consists of an array A of length 2^q, as in the bottom of Figure 1, where each slot contains $r + 3$ bits: the remainder plus 3 metadata bits. To insert an item x into the filter, store $f_r(x)$ in slot $A[f_q(x)]$. If there is already an item in this slot, another item in the filter has the same quotient value. Quotient filters deal with these collisions using linear probing. Thus the remainder for a fingerprint may not always in the *canonical slot*, $A[f_q(x)]$, but it can be found nearby. The QF linear-probing algorithm maintains three invariants:

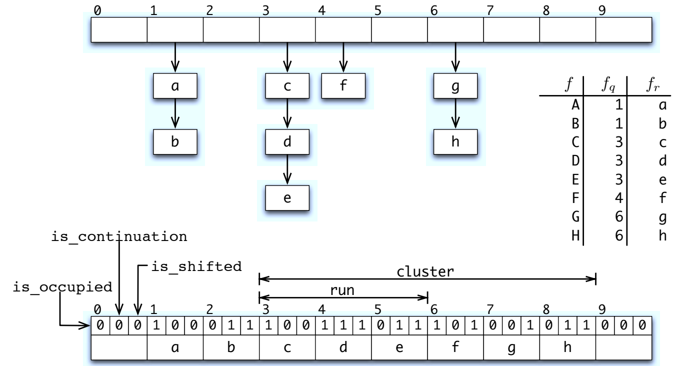


Fig. 1: An example quotient filter (bottom) with 10 slots, and its representation as a hash table with chaining (top). The filter stores the set of fingerprints $\{A - H\}$. The remainder, f_r, of a fingerprint f is stored in the bucket specified by its quotient, f_q. The quotient filter stores the contents of each bucket in contiguous slots, shifting elements as necessary and using three metadata bits to enable decoding.

(1) Remainders may only be shifted forward (to the right) of their canonical slot. (2) All remainders are stored in sorted order such that if $f(x) < f(y)$, $f_r(x)$ will be stored in a slot before $f_r(y)$. (3) There are no empty slots between an item and its canonical slot. These invariants guarantee that items with the same quotient will be stored in sorted order in contiguous slots, which we call a *run*. A *cluster* is a series of runs with no empty slots between them.

A lookup, insert, or delete operation requires a sequential search within a portion of the filter. Starting at the canonical slot, search to the left to find the beginning of the cluster, then search to the right to find the item's run. The SQF encodes the information needed to determine which run each remainder belongs to using three metadata bits per slot: `is_occupied`, `is_continuation`, and `is_shifted`. Operations maintain good locality, because all reads and writes are in the region around the canonical slot. The performance of all SQF operations is largely dependent on the time spent searching backwards and forwards through the clusters, which is determined by the cluster length. Bender et al. [5] prove that cluster lengths are bounded by a constant in expectation and logarithmically with high probability. Therefore, serial quotient filter operations finish in expected constant time.

As previously mentioned, a QF has a non-zero probability of false positives, meaning a membership query will occasionally return "present" for an item that is not in the set. False positives happen when two keys hash to the same fingerprint—a hard collision. However, as Bender et al. [5] demonstrate, the probability of a hard collision is 2^{-r}; therefore, increasing or decreasing the number of bits in the remainder gives a trade off between query accuracy and memory usage.

B. Rank-and-Select-Based Quotient Filters

Pandey et al. [6] designed the RSQF to improve upon the SQF by increasing lookup performance at high load factors

and reducing the number of metadata bits.[1] Their filter stores the remainders using the same slot locations and order as the SQF, but it uses a different metadata scheme for locating items within the filter. Figure 2 shows the basic structure of the RSQF. The RSQF stores two metadata bits for each remainder slot: `occupieds` and `runEnds`. These bits are stored in separate bit arrays, rather than within the remainder slots themselves, and are accessed via `rank()` and `select()` bit vector operations. To find a run, compute the rank of its `occupied` bit, then select the `runEnd` bit of the same rank. To maintain locality of these operations, the filter is divided into blocks of 64 slots, each with an 8-bit `offset` value to track any overflows from previous blocks. The work required to locate a run is independent of the fill fraction, which means lookup performance does not decrease much as the filter fills up. However, inserts do still require a search to locate the next empty slot and to move items around, so insert performance does decrease with fill fraction, just as in the SQF.

IV. GPU Standard Quotient Filter Operations

We now describe the GPU implementation of membership queries (lookups), insertions, deletions, and merges. We also devise three parallel methods to construct a quotient filter from a list of elements and consider the advantages of each.

The QF stores hashed keys. An important feature of hash values is that they are uniformly distributed, no matter the input, which has pros and cons. On the negative side, uniformity undermines memory locality. On the positive side, uniform distributions favor load balance. Finally, uniformity makes hashes easier to test, since all workloads yield the same behavior, as long as keys are not repeated.

A. Lookups

In order to maximize locality between neighboring threads, we first hash and sort the input values. We then assign one membership query per thread and perform a sequential lookup. Pseudocode is shown in Algorithm 1. Performing lookup operations in parallel does not require collision avoidance, because lookups do not modify the QF. Varying cluster lengths results in divergence between threads within warps. However, cluster lengths are small, and therefore each lookup operation will take constant time in expectation and logarithmic time with high probability.

B. Supercluster Inserts

Assigning one insert per thread can lead to race conditions if different threads try to modify the same slot at the same time. Therefore, we must determine a set of inserts that we can safely perform in parallel. To do this, we identify independent regions of the quotient filter, which we call *superclusters*; we only perform one insert per supercluster at a time. We define a supercluster as a region ending with an empty slot. This empty slot allows us to insert a single new element without shifting any elements into another supercluster's space. See Figure 3.

In parallel, we mark each slot whose preceding slot is empty with a 1. Then we use the CUB library `DeviceScan` primitive to perform a prefix sum of these bits and label each slot with its supercluster number. The items in the insert queue then bid for exclusive access to their supercluster. We then insert these items, remove them from the queue, and repeat the process until all items have been inserted. Pseudocode is shown in Algorithm 3.

The parallelism of this method is significantly constrained by the number of superclusters in the filter, and as the filter gets fuller, there are fewer superclusters. Additionally, like the lookup method, this insert implementation suffers from warp divergence and lack of memory reuse between threads. However, because the input values are hashed, the distribution of items between superclusters should be roughly uniform, resulting in good load balancing.

C. Bulk Build

Consider inserting a batch of items into an empty QF. We will do so by computing every item's final location in parallel and then scattering them to their locations.

We begin by computing the fingerprints, sorting them, and splitting them into quotient and remainder values. To compute the location of each item, recall from Section III-A that an element is located either in its canonical slot (shift = 0) or shifted to the right (shift > 0). Figure 4 illustrates how items from runs with lower quotient values can shift items in later runs. The shift amount for the first element in a run is the shift amount for the first element in the previous run plus the number of items in the previous run minus the distance between the canonical slots of the two runs. Essentially, we keep a running total of underflow/overflow.

Astute parallel programmers might immediately think "prefix-sum". But recall the resulting shift value must be non-negative, so we must saturate the sum at each step so that the resulting shift never goes below zero. Alternatively, when directly computing the location of elements (as opposed to computing their shift values), we could consider a prefix-sum operator of $\max(\textit{value}_{(i-1)} + 1, \textit{value}_i)$. Neither of these operations is associative, thus we cannot use any existing GPU methods that implement prefix-sum, all of which require associative operators. However, both of these operators have an important property that allows us to extract parallelism: certain inputs, including those we see in quotient filter construction, break the dependency chain between output items. At each point where the saturation to zero happens in the prefix sum formulation, or where the $\max(\textit{value}_{(i-1)} + 1, \textit{value}_i)$ operator outputs $\textit{value}_i$, the contribution from the scan of all preceding values to the items that follow is zero. In this way, the problem can be thought of as a segmented scan in which the segment divisions are initially unknown, and we can parallelize over segments.

We explored three methods for bulk QF builds on the GPU, each of which approaches the non-associative scan problem

[1]They also extend RSQF functionality by storing compact counters in the remainder slots. We chose not to include counters in our GPU implementations in order to focus on how the fundamental differences in the AMQs affect the parallelism we can extract from these data structures.

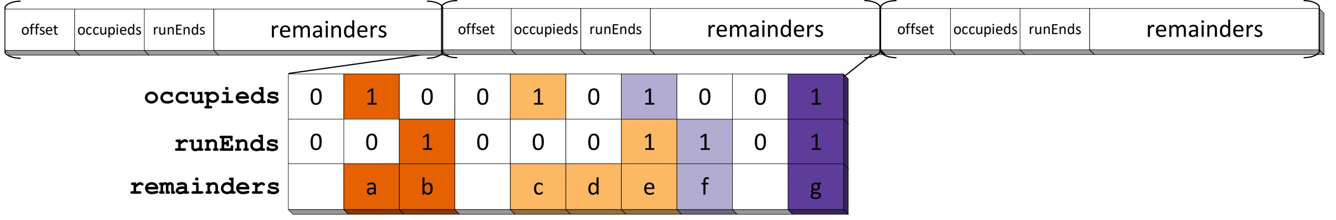


Fig. 2: Example rank-and-select-based quotient filter with three blocks of ten slots per block. The `occupied` and `runEnds` bit arrays are used to locate items in the filter, and each block has an `offset` value to account for any overflow from previous blocks. The example block (bottom) is shown with the metadata bit arrays oriented above the remainder values to illustrate how these bits are used to determine which run each remainder belongs to. This block has four non-empty runs (denoted by the different colored blocks) with one to three items in each run.

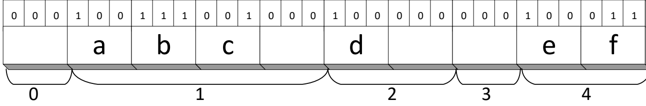


Fig. 3: Example quotient filter figure with corresponding supercluster labels. Superclusters represent independent regions of the filter, where we may perform inserts in parallel without incurring data races.

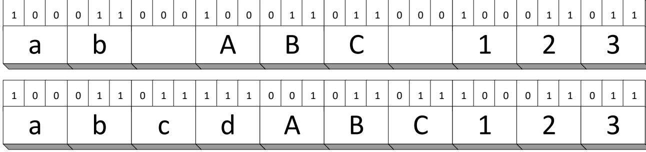


Fig. 4: Example quotient filter arrays showing the interdependence of item locations in different runs. ABC is a run of elements with $quotient = 3$. With the addition of c and d to slot 0, ABC must be shifted right one slot.

differently:

—*Parallel merging* (Section IV-C1) begins with one segment for each unique quotient in the dataset, then iteratively merges pairs of segments together, checking the saturation condition as each pair merges and only sending the output of the scan from the left segment as the input to the right if the saturation condition is not met.

—*Sequential shifting* (Section IV-C2) applies the operation to every pair of neighboring runs in each iteration, checking the saturation condition, and iterating until the scan has been carried through the end of the longest independent cluster.

—*Segmented layouts* (Section IV-C3) assumes that every segment of $\log(n)$ items is independent and computes the scan serially within these segments. In each iteration, each segment sends the partial scan for its last item to become the initial value for the segment to its right. Because the quotient values are the result of a hash function, this process converges after a small number of iterations.

1) Bulk Build Via Parallel Merging: This first implementation of bulk build uses an iterative merging process, which finishes after $\mathcal{O}(q) = \mathcal{O}(\log(n))$ iterations, or more precisely, $\log(\mathit{number_used_quotients})$ iterations. First, we compute the items' unshifted locations with a segmented scan. Next, we label the segments in parallel by checking $\mathit{quotient}[idx] \neq \mathit{quotient}[idx-1]$ then performing a prefix sum. Initially, items will only be grouped with the other items in their run, as shown in iteration 0 of Figure 5, but these segments will grow as we run our merging algorithm.

Each iteration of the merging algorithm, shown in Figure 5, consists of two steps. First, for all pairs of segments, we compare the last element in the left-hand segment and the first element in the right-hand segment and compute the overflow or underflow. Second, for all elements, we compute and apply the shift using the overflow/underflow for the segment. We account for any empty filter slots and prevent extraneous shifting by storing negative shift values in a credits array. The pseudocode for this build method is shown in Algorithm 4.

2) Bulk Build Via Sequential Shifting of Runs: In our second method, shown in Figure 6, we compute unshifted locations and label the segments, just as we did for the parallel merging bulk build. We then shift the filter elements iteratively; however, instead of combining the runs into larger segments, we launch one thread per run in each round to determine whether the run needs to be shifted to avoid overlap with the previous run. Threads set a global flag each time they perform a shift, which we check (then clear) after each iteration to determine whether or not to launch the kernel again. When a kernel finishes without shifting any elements, the algorithm is finished. Pseudocode is in Algorithm 5.

3) Segmented Layouts: Our third bulk build method computes shifts in segments of the filter itself, exploiting the fact that the input is hashed, and therefore, items are distributed approximately evenly throughout the filter. For this method, we partition the quotient filter into segments of length $\log(\mathit{numSlots}) = \log(2^q) = q$. Each segment has all items whose quotients fall within the segment and an initial shift value for the segment. We launch one thread per segment to lay out all of the items in its segment given the initial shift and output an overflow value. These overflow values are then passed as initial shifts for the next segment, as shown in Figure 7, and the process repeats until no new overflows are generated. Because the quotient values are the result of a hash function, this process converges after a small number of iterations. Pseudocode is in Algorithm 6.

a) Comparison of Build Methods: We have now devised four different ways (including supercluster inserts) to construct a QF from scratch. We evaluate these methods experimentally

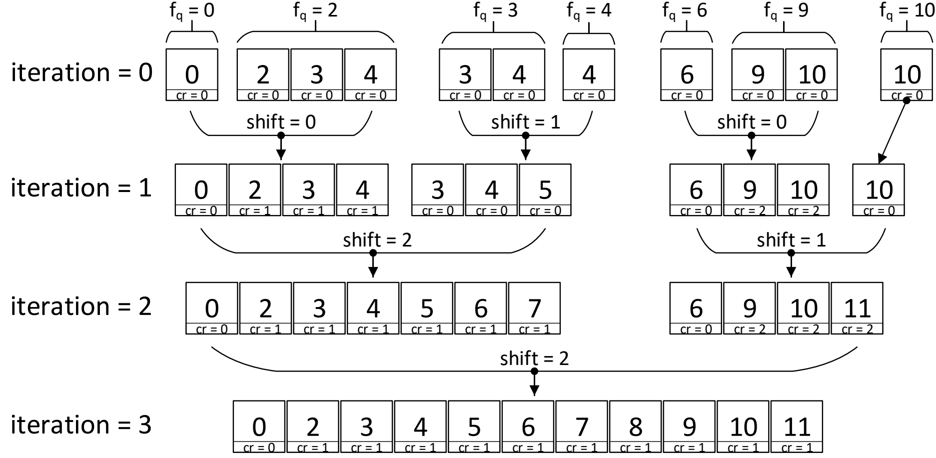


Fig. 5: Diagram of our parallel merging bulk build algorithm. At the start, items are grouped into segments according to their quotient value (canonical slot), f_q, then in each iteration, neighboring segments are pairwise-merged and any necessary shifts are applied. The large number in each box indicates the slot the associated remainder value will occupy in the final filter construction, and the number in the lower part of the box (cr) denotes any credits from empty slots preceding the current slot.

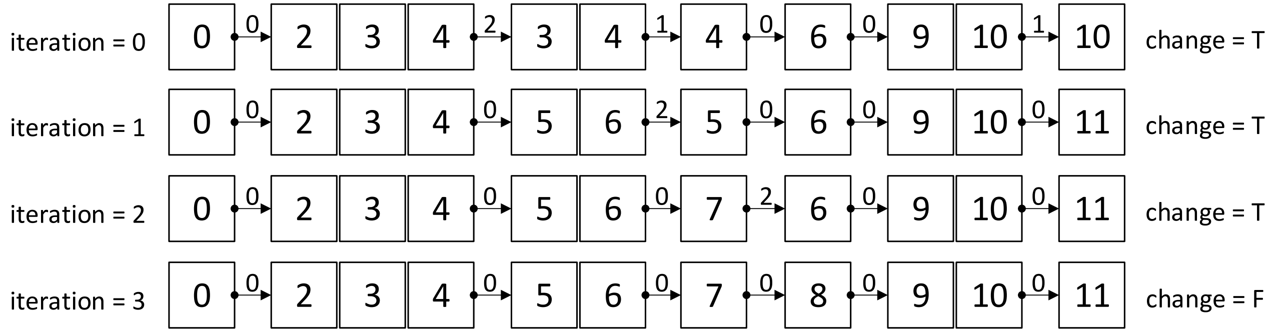


Fig. 6: Diagram of our bulk build method using sequential shifting of runs. In each iteration, we launch one thread per run to check for overlap between its run and the previous run and shift its run if necessary. As in Figure 5, the values in the boxes are the slots the items will occupy in the quotient filter. The values above the arrows indicate the amount the next run must be shifted to avoid overlap. When all of these shift values are 0, the process stops.

in Section VII-C, but intuitively, when is each one most appropriate? When a filter is empty, there are a lot of available superclusters, so supercluster inserts should work well. The sequential shifting method would also be likely to work better for filters that are less full, because clusters will be shorter, leading to fewer shifts, and therefore, fewer iterations. The parallel merging build requires a constant number of iterations, so it will be the most efficient for building very full filters. The segmented layouts build is likely to perform better for emptier filters, but overall we would expect its running time to increase only moderately as the fill fraction increases. Because the segment length is always q, the segmented layout method also requires the same amount of work in each round, independent of the filter fullness.

We now compare the parallel complexity of these bulk build algorithms. For the parallel merging and sequential shifting builds, preprocessing involves a sort and two prefix sums. For parallel merging, the merging process continues for $\mathcal{O}(\log(n))$ iterations, where each iteration uses a constant number of steps. So the makespan of the parallel merging build is $\mathcal{O}(sort(n) + \log(n))$.

For the sequential shifting build, the shifting process continues until the shifts have been carried through all clusters. This means the number of iterations is bounded by the number of runs in the longest cluster. As Bender et al. show [5], the largest cluster in a QF has size $k = (1+\epsilon)\frac{\ln(n)}{\alpha - \ln(\alpha) - 1}$ with high probability. For a reasonable QF fill fraction, we can approximate this as $\mathcal{O}(\log(n))$. Within each iteration, there are a constant number of steps. Therefore, the makespan of the sequential shifting build is $\mathcal{O}(sort(n) + \log(n))$ with high probability.

Preprocessing for the segmented layouts build only requires a sort. The layout operation itself requires a sequential iteration over the $q = \mathcal{O}(\log(n))$ slots in the segment. In the worst case, the number of iterations is bounded by the maximum shift for any one item in the filter. This shift is bounded by the maximum cluster length, which is $\mathcal{O}(\log(n))$ with high probability, so the complexity of this build method is $\mathcal{O}(sort(n) + \log^2(n))$ with high probability.

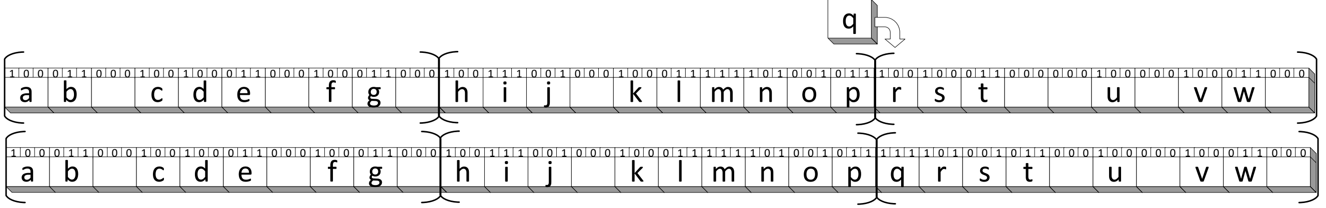


Fig. 7: Diagram of segmented layouts bulk build for a small example quotient filter. This filter contains three segments, and the layout of each segment is computed in parallel, then checked for overflows. In the first iteration, item q is an overflow item, and gets bumped into the next segment in the second iteration.

b) Generalization to Other Non-Associative Operators: The strategies we used to approach this problem could be used to compute a scan on other non-associative operators – in particular, operators that include a saturation condition. However, the performance of the sequential shifting and segmented layouts methods relies on the ability to break the chain of dependencies. Without independent segments, the performance of these algorithms will be the same as that of a serial algorithm. Fortunately, the QF provides us with ample independent segments, because hashing distributes items uniformly across the entire space of the filter. On the other hand, the parallel merging algorithm requires an operator that is associative between the breaks in the dependency chain. It does not extract parallelism from these discontinuities, but rather from the fact that these specific operators are associative for all items between the saturation locations.

D. Supercluster Deletes

Deletes use an algorithm similar to the supercluster inserts described in Section IV-B. We divide the filter into independent supercluster regions similar to those used in the insert method, but with a modification: we require that the first slot in a supercluster be occupied and unshifted. This means the slot is actually the head of a cluster. We know that the value in this slot, and any shifted slots to its immediate right, will not be affected by any deletes to the left because the item is in its canonical slot. This also prevents the supercluster from being comprised of only empty slots (with no items to delete). We perform a bidding process to choose which items to delete while avoiding collisions. We then delete one item at a time per supercluster, shifting items left and modifying metadata bits as needed. When all threads have finished, we remove successfully deleted items from the queue and repeat. The pseudocode for this operation is shown in Algorithm 7.

E. Merging Filters

Merging two QFs allows us to use one of our bulk build operations to add a new batch of items to the filter. This is a rebuild of the filter, but without the need to access or rehash the items already stored in the filter. Merging is helpful for some filter applications, e.g., combining datasets stored by different nodes in a distributed system. The first step in merging two QFs is to extract the original fingerprint values. We do this in parallel by assigning one thread to each slot, using the metadata bits to determine its quotient. We scatter these fingerprints to an array, and compact out all of the empty slots. We now have two sorted arrays: one for each of the original filters. We merge these arrays using the GPU merge path algorithm by Green et al. [15]. This leaves us with one sorted array of fingerprints, which we can now input to one of our three bulk build algorithms to construct the QF. Pseudocode is in Algorithm 8.

V. GPU Rank-and-Select QF Operations

In this section, we describe the algorithms we devised for querying and modifying the rank-and-select-based quotient filter (RSQF) on the GPU.

A. Lookups

For the RSQF, we parallelize lookups for the GPU in a similar fashion as we do for the SQF: hash and sort all inputs, then assign one query per thread and have each thread perform the same operation as in the serial case. The pseudocode for RSQF lookups is Algorithm 2. Again, because lookup operations do not modify the data structure, this simple parallelization works without the addition of any collision avoidance schemes. The sorting step increases memory locality between threads, as in our SQF lookups, but here the benefit is much greater, because metadata values are shared across all slots in an RSQF block, rather than scattered amongst the remainder values.

B. Inserts

For RSQF inserts, our general strategy was to parallelize over the blocks of the filter and perform inserts in batches. Because each block has an offset value to account for any spillover from previous blocks, the block holds all of the necessary information to insert an item in any of the 64 slots within the block, assuming it does not overflow to the next block. However, some items, particularly when the filter reaches higher fill fractions, will need to overflow to the next block. To deal with inserts that overflow into other blocks while still avoiding race conditions, we allow threads to modify more blocks as the filter gets fuller. To accomplish this, we partition the filter into insert regions, and assign one thread to each region. This is similar to the partitioning for multi-threaded inserts devised by Pandey [6], but because the GPU has many more threads than a CPU, our implementation uses smaller regions than theirs. Initially, regions are one

block, and as more items have been inserted, the regions increase in size (and, consequently, decrease in number). In our implementation, we increase the region size by one block every 16 iterations.

The entire insert operation proceeds as follows: Before inserting a batch of items, we hash the inputs, then sort the fingerprints, and divide them into queues based on their block number. We then launch one CUDA thread per region to perform the inserts, using the same basic algorithm as the CPU implementation described in III-B. If an insert operation would require a thread to modify a block that is not in its assigned region (in the case of overflow to the next block), the operation halts. When an insert is completed or halted, the thread sets a flag if it still has items in its queue, to indicate that the insert kernel should be launched again. The pseudocode for this algorithm is shown in Algorithm 9.

The maximum amount of parallelism we can extract using this method is constrained by the number of blocks in the filter. Work balancing is dependent on the distribution of the data: if many of the items hash to the same region of the filter, the threads for other regions will finish their inserts and have no more work to do while the busy thread is still working. Because the input data is hashed, assuming a good hash function, the most likely cause for an unbalanced workload is repeated items.

C. Bulk Build, Deletes, Merging Filters

We did not implement bulk build, delete, or merge operations for the RSQF, but these operations can be performed using a straightforward extension of the methods we used in the SQF algorithms. The final slot numbers for all items are the same for the SQF and RSQF, so we can use the algorithms described in Section IV-C to compute the locations of all items, with slight modifications in writing the values to the filter to account for the different memory layout and associated metadata values. Merges can also be implemented analogously to the operation in Section IV-E by assigning one thread per slot to extract fingerprints into an array, then merging the arrays and rebuilding the new filter. Finally, just as superclusters can be utilized in both SQF inserts and deletes, our RSQF insert strategy of breaking the filter up by into small regions and assigning one region to each thread can also be applied to RSQF deletes.

VI. Design Decisions and Trade-Offs

In this section we justify some of the many design decisions we made in adapting the SQF and RSQF to the GPU.

a) Remainders Divisible by 8 and `char` Containers: Quotient filters are designed to be flexible in the number of bits stored per item in order to allow the programmer to choose the optimal trade-off between memory and error for their application. However, this variability means that, because the elements are stored contiguously in memory, a single slot may be split between bytes (and even cache lines). For arbitrary values of r, this opens up the possibility of memory conflicts, even when we ensure threads are modifying different slots. To simplify this issue, we chose to only use SQFs where the number of bits per slot is divisible by 8. This gives us fewer options in the trade-off between size and false positive rate, but it simplifies our filter operations and increases the amount of parallelism we can extract from the problem. Similarly, we chose to store each SQF slot in one or more `char`s, rather than fitting one or more remainders into a single `int`, so that we are able to write to two neighboring slots without worrying about write hazards.

b) Duplicates: For datasets with many reoccurring values, deduplication may be essential to speed up membership queries and prevent the filter from over-filling. For the bulk build algorithms, because all items are being inserted at the same time, deduplication is not an inherent part of the algorithm, as it is for incremental inserts; therefore, we give an option for deduplication to be switched off or on, to allow flexibility for different applications.

c) One Query Per Thread: We chose to use only one item per thread for each QF lookup query. An alternative approach would be to launch one query per warp and have threads search cooperatively within clusters to locate the items. However, this would not be very efficient because: (1) the average cluster length is constant, as described in Section III-A, and (2) threads must also keep track of metadata values to compute the canonical slot for each value, which would be much more complicated to resolve cooperatively.

d) RSQF Insert Region Size: For inserts into an RSQF, we had a few different options for the granularity. We could have identified the smallest independent regions in the RSQF, similar to the superclusters in the QF. However, the blocked structure of the RSQF means that operations within the same block could lead to race conditions when modifying the block-wide `occupieds`, `runEnds`, and `offset` values. Alternatively, we could have based the size of the regions on the filter's fill fraction, to account for the increased likelihood of interblock overflows as the fill fraction increases. Both of these alternatives would also require a system of tracking which items-to-be-inserted belong to each of the variable-sized regions. We decided to take the simpler approach, where we could perform an initial sort and create stable queues of items to be inserted into each block.

e) Number of Iterations Between Each Extension of Insert Regions: We decided to increase the size of insert regions every 16 iterations based on empirical evidence from our experiments. This seemed to be the interval that worked best for building a filter from scratch. For incremental updates, the most efficient interval would likely vary based on the fill fraction, which would require additional tuning and work to track the fill fraction.

VII. Results

We evaluate our GPU SQF and RSQF implementations using synthetic datasets of 32-bit keys, generated using the Mersenne Twister pseudorandom number generator. Because the values are hashed before any QF operations are performed, the distribution of the input data should not affect performance,

and random data should be sufficient to estimate AMQ performance in real-world applications.

In all experiments, we used QFs with $q = 23 \to 2^{23}$ slots and $r = 5$, or an error rate of $\epsilon \approx 0.03125$. We compare our GPU SQF and RSQF with a variety of other AMQs: Costa et al.'s BloomGPU filter [13], Pandey et al.'s multithreaded counting quotient filter (CQF) [6], Arash Partow's Bloom filter [16], and our own CPU SQF implementation, which uses the serial operations described in Section III. We also modified the BloomGPU filter to create a version ("BloomGPU 1-bit") using only one bit per element of the filter bit array, rather than an entire byte. This required using atomic bitwise operations, which were not yet supported by CUDA at the time that Costa created BloomGPU. The invariant in our comparisons is false-positive rate. Because we also inserted the same number of items for all data structures and the BF error rate increases as more bits are set, we increase the BF size as the comparable QF fills up in order to maintain a similar false positive rate in both filters. To achieve a balance between error rate and accuracy, we chose to use 5 hash functions in all BFs.

All GPU experiments were run on a Linux workstation with 2×2.53 GHZ 4-core Intel E5630 Xeon CPUs, 12 GB of main memory, and two GPUs: an NVIDIA Tesla K40c with 12 GB on-board memory and an NVIDIA GeForce GTX 1080 with 8 GB of on-board memory. We ran all experiments on each GPU separately and have noted any significant differences in performance results between the two architectures in our discussion. All source files were compiled with CUDA 8.0. CPU experiments were run on a Linux workstation with one 3.7 GHz 4-core Intel E3-1280V5 CPU. We chose to use a different workstation for our CPU experiments in order to give a fair comparison to Pandey's multithreaded CQF, which utilizes recently-added x86 instructions to speed up the `rank()` and `select()` operations [6]. We used 4 threads for all multithreaded CQF experiments.

Summary of Results: Overall, we find that our SQF outperforms BloomGPU on lookups and initial filter construction, while the BF achieves higher throughput for incremental insert operations. The RSQF achieves a 2–3x speedup on lookups compared to BloomGPU, but has lower incremental insert throughput than the SQF. We find that the BloomGPU 1-bit modification has only a modest effect on the filter performance, so we use this version as our primary reference for comparison. We also discover that the segmented layouts method is generally our fastest QF construction method.

A. Lookups

Figures 8 and 9 show the performance difference for membership queries using BloomGPU and our GPU QFs on the Tesla K40c and GTX 1080, respectively. For a fill fraction $\alpha < 0.8$, our SQF achieves higher throughput than BloomGPU. Both the SQF and RSQF show an initial increase in throughput as the fill fraction increases. This is because at the lower fill fractions, the batch size is not yet big enough to fill the entire GPU. SQF query throughput is highly dependent on fill fraction, because each lookup requires reading an entire

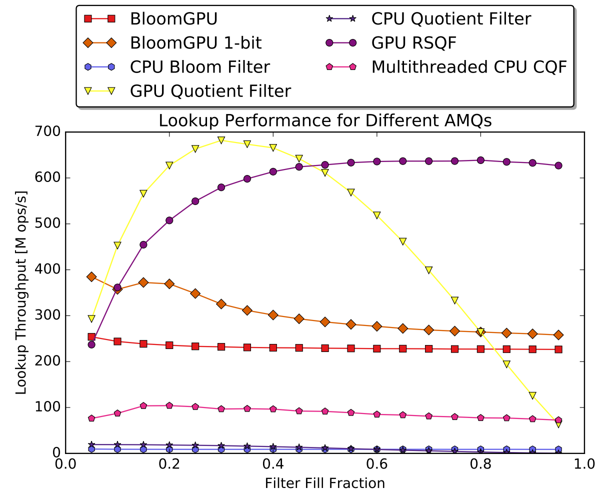


Fig. 8: Lookup performance on NVIDIA Tesla K40c for different AMQs with varying fill rates. The batch size is all items in the filter.

cluster of elements, and as the filter gets full, the average cluster length increases. Bender et al. recommend that a QF remain $\leq 75\%$ full for this reason. Our RSQF, on the other hand, maintains a similar throughput across all fill fractions, because the `rank` and `select` operations require the same amount of compute across all fill fractions. The only linear searching the RSQF performs in a lookup is within the item's run, which, at high fill fractions, is much smaller than a cluster. As a result of this property, RSQF lookup throughput is higher than the standard QF for $\alpha \geq 0.5$. BloomGPU filter throughput also remains constant because the filter performs the same number of reads per query for all fill fractions.

Comparing Figures 8 and 9, we see that all filters' performance improves with the new microarchitecture and increased memory bandwidth. One notable difference is that the BloomGPU 1-bit achieves very high throughputs at low fill fractions on the GTX 1080. This is likely because the smallest Bloom filters fit into the larger cache in the GTX 1080, and we resize the Bloom filters to maintain equivalent false positive rates.

All GPU filters show a large performance increase as the batch size grows (Figure 10a). This illustrates the importance of providing sufficient work to keep all GPU compute units busy. At around 10^6 items per batch, BloomGPU throughput levels out. At this point, the performance is memory-bound for the BF, but not the QF, due to the greater locality of the QF operations.

B. Inserts and Deletes

Figure 10b shows the change in insert and delete throughput for a constant batch size (100000 items) as a function of filter fullness. Performance for supercluster inserts and deletes decreases as the filter fills and the number of superclusters decreases. Also, the latency for each operation increases as clusters grow and the GPU must search through a longer section of the filter to locate the correct slot. This reinforces the rule of thumb of maintaining a filter fullness of $\leq 75\%$.

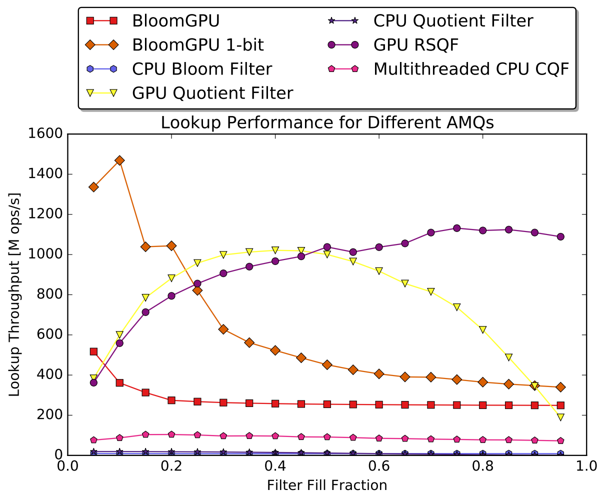


Fig. 9: Lookup performance on NVIDIA GeForce GTX 1080 for different AMQs with varying fill rates. The batch size is all items in the filter.

We find that our RSQF inserts achieve lower throughput than supercluster inserts, and that RSQF insert throughput decreases as the filter gets fuller. This is because RSQF inserts shift items to make room for new ones, so much of the compute time is spent searching for empty slots and rearranging items, and as the filter fills, these empty slots become more difficult to locate. Figure 10b also shows a performance comparison for supercluster inserts versus the merge-and-rebuild (merge inserts) approach. For filters below $\approx$ 80% full, supercluster inserts have a 2x speedup over rebuilding, and even at 95% full, supercluster inserts still achieve a slightly higher throughput.

All AMQs show a performance increase as the batch size grows (Figure 10c); however, both the overall performance and performance improvement are much lower for the QFs. This is likely because the available parallelism is restricted to one insert per supercluster for the SQF, and one insert per block region for the RSQF. We can also see that for smaller batch sizes, supercluster inserts are faster than merge inserts, but for batch sizes of ≥ 2 million items, it is actually faster to extract the quotients and rebuild the filter with the new values.

C. Comparing Filter Build Methods

Figures 11 and 12 show the build throughput for all AMQs on the Tesla K40c and GTX 1080. As with lookups, the BloomGPU 1-bit inserts achieve high throughputs at low fill fractions, likely because these smaller filters fit in the GTX 1080's larger L2 cache.

As with lookups, the original 1-byte BloomGPU insert performance does not vary with changing fill fraction; however, the 1-bit implementation does show a steady decrease in throughput as the fill fraction increases. For fill fractions $\alpha > 0.4$, the 1-byte version achieves higher throughput than the 1-bit version. This is likely due to the computational cost of atomic operations required for 1-bit Bloom filter inserts. All QF build methods have an initial increase in performance before throughput either decreases or levels off. This increase is probably because there is not enough work to fill the GPU for very low fill fractions. After this initial ramping up, we see different behavior for each of the build methods:

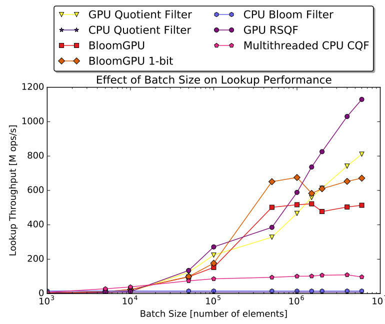
- Throughput for the parallel merging build increases monotonically with fill rate, because the number of iterations for this method is dependent only on the number of quotients used. This is largely independent of the fill fraction, so the computation required per item decreases as the total number of items increases.
- The performance of the sequential shifting build increases until the filter is about 50% full, then begins to fall off. This is because as the filter gets full, the number of shifts, and therefore, the number of iterations, will increase. Additionally, the shift operation performed by each thread is also serial, so as the quotients' runs get longer, the latency of each operation increases.
- The segmented layouts build method is the fastest for all $\alpha \leq 0.85$. This method achieves a peak throughput at around 40% full, then performance decreases steadily as the filter gets fuller and more iterations are required for convergence. Even at the ideal maximum QF capacity of $\alpha = 0.75$, this build method still achieves higher throughput than BloomGPU.
- Deduplication does generally cause a moderate decrease in throughput (Figure 13). Interestingly, in the sequential shifting method, deduplication is costly for low filter fullness, but becomes insignificant to overall throughput as the filter fills and compute time is dominated by the many iterations required to perform all of the shifts.

TABLE I: Data Structure Memory Use

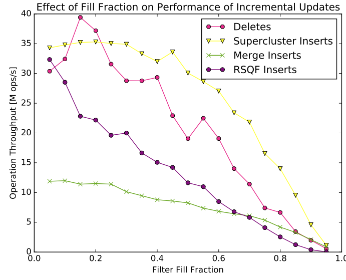
Error Rate	Bytes/Item			
	Standard QF	RSQF	BloomGPU	Bloom 1-bit
0.03	1.3	0.94	7.7	0.96
0.0001	2.7	2.0	20.4	2.6
5×10^{-7}	4.0	3.0	45.2	5.7

D. Memory Use

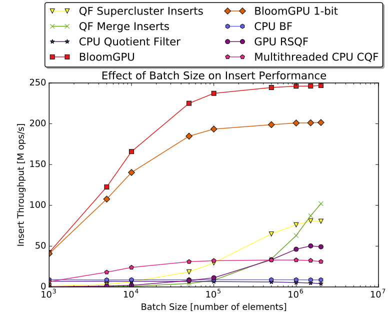
Table I shows memory usage for all GPU AMQs. The RSQF has a smaller memory footprint than the SQF because it can be filled up to 95% full without compromising lookup performance, while the SQF should be sized to be 75% full. We note two limitations of our QF implementation with respect to memory usage: (1) Our SQF does not support a more fine-grained selection of false positive rates because we require slot sizes to be divisible into complete bytes, as described in Section VI. (2) Our bulk build methods allocate additional (temporary) memory to calculate element positions within the filter. This means that we cannot bulk-build a filter on the GPU that will fill a majority of the GPU on-board memory. For these filters, we would need to perform incremental inserts in smaller batches to construct the filter without running out of memory. However, real-world use cases would require additional free memory in order to read in batches of items for lookups anyway.



(a) Lookup performance for different AMQs with varying batch sizes, with an initial fill fraction of $\alpha = 0.7$.



(b) Insert and delete throughputs for the GPU QF decrease as the filter fills.



(c) Insert performance for different AMQs with varying batch sizes.

Fig. 10: Lookup and insert performance on the NVIDIA GeForce GTX 1080.

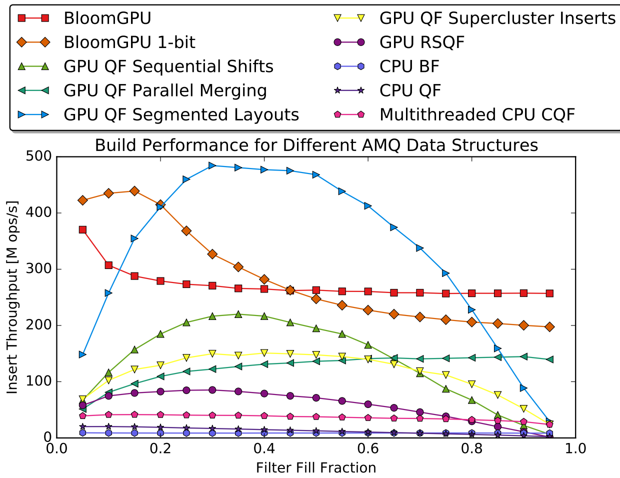


Fig. 11: Filter build performance on NVIDIA Tesla K40c for different AMQ data structures with varying fill rates.

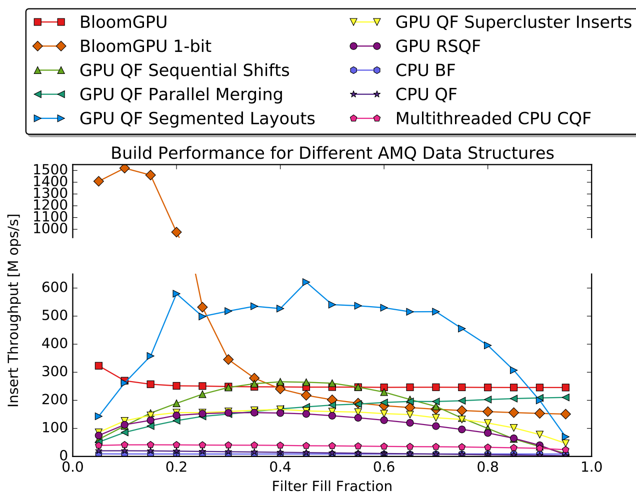


Fig. 12: Filter build performance on NVIDIA GeForce GTX 1080 for different AMQ data structures with varying fill rates

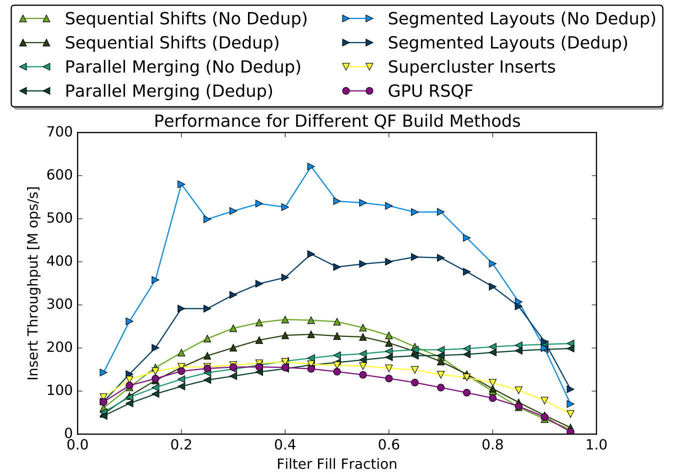


Fig. 13: Quotient filter build performance on NVIDIA GeForce GTX 1080 for all build methods at varying fill rates, with and without a deduplication step.

VIII. CONCLUSIONS

For inserts alone, the simplicity of modifications and resulting high level of parallelism available for BFs outweighs the locality benefits of the QF. In contrast, this locality does lead to better GPU performance for QF lookups, where memory conflicts are not an issue and parallelism is not constrained.

Recomputing dynamic independent regions between each round of updates (supercluster inserts) leads to higher throughput vs. parallelizing updates over fixed-sized regions (RSQF inserts). Although computing the supercluster locations requires additional work each round, it guarantees a priori that inserts in those regions will succeed. For RSQF inserts, the fixed-sized regions we use are not guaranteed to be conflict-free and therefore require a strategy for handling overflows. By contrast, in order to achieve a high level of parallelism while avoiding conflicts for the supercluster inserts, we only allow our SQF to use remainders that fit in full bytes, which limits the number of available remainder sizes available. In the RSQF, the blocking structure divides the filter into segments that align with word boundaries, so we need not restrict the RSQF size and corresponding false positive rate.

To parallelize bulk QF construction, we needed to perform a parallel scan operation on a non-associative operator. In our three bulk build implementations, we leverage the fact that this operator has a saturation condition, and extract parallelism from breaks in the dependency chain.

Finally, the GPU RSQF performance could be improved if NVIDIA added support for a bit-manipulation operation equivalent to the PDEP operation available on the Intel Haswell architecture. This gives a significant performance boost on the CPU and would likely have a similar benefit for the GPU.

ACKNOWLEDGMENTS

Leyuan Wang suggested the approach to bulk build that became the basis of the algorithm in Section IV-C1. We would like to thank Lauro Costa, et al. for giving us access to their code for BloomGPU. We would also like to thank Prashant Pandey for his help debugging our test code. We appreciate financial support from NSF awards CCF-1724745, CCF-1715777, CCF-1637458, CCF-1637442, and IIS-1541613, an NSF Graduate Research Fellowship, and gifts from EMC and NetApp.

REFERENCES

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[2] F. Daoud, A. Watad, and M. Silberstein, "GPUrdma: GPU-side library for high performance networking from GPU kernels," in *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '16, Jun. 2016, pp. 6:1–6:8.

[3] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a file system with GPUs," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, Mar. 2013, pp. 485–498.

[4] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey, "Streaming similarity search over one billion tweets using parallel locality-sensitive hashing," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1930–1941, Sep. 2013.

[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*, vol. 5, no. 11, pp. 1627–1637, Aug. 2012.

[6] 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*, ser. SIGMOD '17, 2017, pp. 775–787.

[7] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang, "IP routing processing with graphic processors," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10, Mar. 2010, pp. 93–98.

[8] A. B. Vavrenyuk, N. P. Vasilyev, V. V. Makarov, K. A. Matyukhin, M. M. Rovnyagin, and A. A. Skitev, "Modified Bloom filter for high performance hybrid NoSQL systems," *Life Science Journal*, vol. 11, no. 7s, pp. 457–461, 2014.

[9] Y. Liu, B. Schmidt, and D. L. Maskell, "DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI," *BMC Bioinformatics*, vol. 12, no. 1, p. 85, Mar. 2011.

[10] L. Ma, R. D. Chamberlain, J. D. Buhler, and M. A. Franklin, "Bloom filter performance on graphics engines," in *2011 International Conference on Parallel Processing*, ser. ICPP 2011, Sep. 2011, pp. 522–531.

[11] I. Moraru and D. G. Andersen, "Exact pattern matching with feed-forward Bloom filters," *J. Exp. Algorithmics*, vol. 17, pp. 3.4:3.1–3.4:3.18, Sep. 2012.

[12] 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*, ser. ICSTCC 2015, Oct. 2015, pp. 872–876.

[13] L. B. Costa, S. Al-Kiswany, and M. Ripeanu, "GPU support for batch oriented workloads," in *IEEE 28th International Performance Computing and Communications Conference*, ser. IPCCC 2009, Dec. 2009, pp. 231–238.

[14] S. Dutta, A. Narang, and S. K. Bera, "Streaming quotient filter: A near optimal approximate duplicate detection approach for data streams," *Proceedings of the VLDB Endowment*, vol. 6, no. 8, pp. 589–600, Jun. 2013.

[15] O. Green, R. McColl, and D. A. Bader, "GPU merge path: A GPU merging algorithm," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12, Jun. 2012, pp. 331–340.

[16] A. Partow, "C++ Bloom filter library." [Online]. Available: http://www.partow.net/programming/bloomfilter/index.html

Algorithm 1 SQF membership queries

```
 1: function FINDRUNSTART(S, f_q)
 2:   ▷ find beginning of cluster
 3:   b ← f_q
 4:   while is_shifted(S[b]) do
 5:     DECR(b)
 6:   ▷ walk forward to find the run for f_q
 7:   s ← b
 8:   while b ≠ f_q do
 9:     repeat
10:       INCR(s)                ▷ skip current run
11:     until !is_continuation(S[s])
12:     repeat
13:       INCR(b)                ▷ count number of runs
14:     until is_occupied(S[b])
15:   return s

16: function LOOKUP(S, inputs)
17:   ▷ preprocessing: hash and sort inputs
18:   for all inputs do
19:     f_q ← ⌊f[threadID]/2^r⌋
20:     f_r ← f[threadID]%2^r
21:     if !is_occupied(S[f_q]) then
22:       return FALSE
23:     s ← FINDRUNSTART(S, f_q)
24:     ▷ search slots in the run for f_r
25:     repeat
26:       if S[s] = f_r then
27:         return TRUE
28:       INCR(s)
29:     until !is_continuation(S[s])
30:     return FALSE
```

Algorithm 2 RSQF membership queries

```
 1: function LOOKUP(R, inputs)
 2:   ▷ preprocessing: hash and sort inputs
 3:   for all inputs do
 4:     f_q ← ⌊f[threadID]/2^r⌋
 5:     f_r ← f[threadID]%2^r
 6:     b ← f_q/SLOTS_PER_BLOCK
 7:     slot ← f_q%SLOTS_PER_BLOCK
 8:     if R[b][slot].occ = 0 then
 9:       return FALSE
10:     r ← RANK(R[b].occ, slot)
11:     end ← SELECT(R[b].run, r)
12:     while end = NULL do
13:       ▷ run end is in next block
14:       r ← r− POPCOUNT(R[b].run)
15:       INCR(b)
16:       end ← SELECT(R[b].run, r)
17:     s ← end
18:     repeat
19:       if R[b][s].rem = f_r then
20:         return TRUE
21:       if R[b][s].rem < f_r then
22:         return FALSE
23:       DECR(s)
24:     until s < f_q ∨ R[b][s].run = 1
25:   return FALSE
```

Algorithm 3 SQF inserts

```
 1: function LOCATESUPERCLUSTERS(S)
 2:   ▷ mark supercluster starts by checking for empty slots
 3:   if ISEMPTY(S[threadID − 1]) then
 4:     return 1
 5:   else
 6:     return 0

 7: function BIDDING(S, inputs, labels)
 8:   ▷ one thread per input bids for supercluster
 9:   (f_q, f_r) ← HASHANDQUOTIENT(inputs[threadID])
10:   sc ← labels[f_q]
11:   return winners[sc] ← threadID

12: function INSERTITEMS(S, inputs, winners)
13:   ▷ each thread performs its own sequential insert operation
14:   (f_q, f_r) ← HASHANDQUOTIENT(inputs[threadID])
15:   if ISEMPTY(S[f_q]) then
16:     SETELEMENT(S, f_q, f_r)
17:     return f_q
18:   s ← FINDRUNSTART(S, f_q)        ▷ from Algorithm 1.1
19:   if is_occupied(S[f_q]) then
20:     ▷ search through run for item
21:     repeat
22:       if S[s] = f_r then
23:         SETELEMENT(S, s, f_r)
24:         return s
25:       else if S[s] > f_r then
26:         break
27:       INCR(s)
28:     until !is_continuation(S[s])
29:   ▷ insert item at location s; move items right as needed
30:   INSERTHERE(S, s, f_r)
31:   return s

32: function INSERT(S, inputs)
33:   repeat
34:     for all SQF slots do
35:       flags ←LOCATESUPERCLUSTERS(S)
36:     for all SQF slots do
37:       labels ←CUBSCAN(flags)
38:     for all remaining inputs do
39:       winners ←BIDDING(S, inputs, labels)
40:     for all superclusters do
41:       slots ←INSERTITEMS(S, inputs, winners)
42:     for all remaining inputs do
43:       ▷ compact out inserted values
44:       inputs ←CUBSELECT(inputs, winners)
45:   until length(inputs) = 0
46:   return slots
```

Algorithm 4 Parallel Merging Bulk Build

```
 1: function CALCOFFSETS(slot, label, credit, offset, carry)
 2:     seg ← label[threadID]
 3:     ▷ compute offsets at odd-numbered segment heads:
 4:     if seg ≠ label[threadID − 1] ∧ seg%2 = 1 then
 5:         offset[seg] ← slot[threadID − 1] − slot[threadID] + 1
 6:         carry[seg] ← credit[threadID − 1]
 7:     return

 8: function SHIFTITEMS(offset, carry, slot, label, credit)
 9:     seg ← labels[threadID]
10:     overlap ← offset[seg] − credit[threadID]
11:     empties ← 0
12:     if overlap > 0 then                              ▷ shift item
13:         slot[threadID] ← slot[threadID] + overlap
14:         empties ← 0
15:     else                                             ▷ track any empty slots
16:         empties ← empties − overlap
17:     credit[threadID] ← empties + carry[seg]
18:     ▷ merge segments
19:     label[threadID] ← label[threadID]/2
20:     return

21: function PARALLELMERGEBUILD(S, inputs)
22:     ▷ preprocessing: hash, sort, compute unshifted locations, label segments
23:     for i ← 0, ⌈log2(segments)⌉ do
24:         for all inputs do
25:             CALCOFFSETS(slot, label, credit, offset, carry)
26:         for all inputs do
27:             SHIFTITEMS(offset, carry, slot, label, credit)
28:     ▷ post-processing: write remainders and metadata
29:     return
```

Algorithm 5 Sequential Shifting Bulk Build

```
 1: function SHIFTSEGMENTS(starts, slots, change)
 2:     index ← starts[threadID]
 3:     shift ← slots[index − 1] − slots[index] + 1
 4:     if shift > 0 then
 5:         length ← starts[threadID + 1] − index
 6:         for i ← 0, length do
 7:             slots[index + i] ← slots[index + i] + shift
 8:         change ← 1
 9:     return

10: function SEQUENTIALSHIFTBUILD(S, inputs)
11:     ▷ preprocessing: hash, sort, compute unshifted locations & segment starts
12:     change ← 1
13:     while change = 1 do
14:         change ← 0
15:         for all segments do
16:             SHIFTSEGMENTS(starts, locations, change)
17:     ▷ post-processing: write remainders and metadata
18:     return
```

Algorithm 6 Segmented Layouts Bulk Build

```
 1: function LAYOUT(f_q, start, shift, overflow, change)
 2:     first ← start[threadID]
 3:     last ← start[threadID + 1] − 1
 4:     n ← last − first + 1
 5:     if n ≤ 0 then
 6:         ▷ segment is empty
 7:         overflow[threadID] ← 0
 8:         return
 9:     ▷ track the furthest right element in the segment
10:     max ← threadID ∗ q + shift[threadID − 1]
11:     for i ← first, last do
12:         if f_q[i] > max then
13:             max ← f_q[i]
14:         INCR(max)
15:     ▷ check for overflow and changes from last iteration
16:     end ← ((threadID + 1) ∗ q) − 1
17:     extra ← (max − 1) − end
18:     if extra > 0 then
19:         overflow[threadID] ← extra
20:         if extra > shift[threadID] then
21:             change ← 1
22:     else
23:         overflow[threadID] ← 0
24:     return

25: function SEGMENTEDLAYOUTSBUILD(S, inputs)
26:     ▷ preprocessing: hash, sort, compute segment starts
27:     change ← 1
28:     while change = 1 do
29:         change ← 0
30:         shift ← overflow
31:         for all segments do
32:             LAYOUT(f_q, start, shift, overflow, change)
33:     ▷ post-processing: write remainders and metadata
34:     return
```

Algorithm 7 SQF deletes

```
 1: function LOCATEDELETESUPERCLUSTERS(S)
 2:     ▷ superclusters for deletes are regular clusters
 3:     if !ISEMPTY(S[threadID])∧!is_shifted(S[threadID]) then
 4:         return 1
 5:     else
 6:         return 0

 7: function DELETEITEMS(S, inputs, winners)
 8:     ▷ each thread performs sequential delete operation
 9:     (f_q, f_r) ← HASHANDQUOTIENT(inputs[threadID])
10:     if !is_occupied(S[f_q]) then
11:         return
12:     s ← FINDRUNSTART(S, f_q)                        ▷ from Algorithm 1.1
13:     repeat
14:         if S[s] = f_r then
15:             break
16:         else if S[s] > f_r then
17:             return
18:         INCR(s)
19:     until !is_continuation(S[s])
20:     if S[s]! = f_r then
21:         return
22:     ▷ s now points to item to be deleted
23:     ▷ delete item; move other items over as needed
24:     DELETEITEMHERE(S, s)
25:     return

26: function DELETE(S, inputs)
27:     repeat
28:         for all SQF slots do
29:             flags ←LOCATEDELETESUPERCLUSTERS(S)
30:         for all SQF slots do
31:             labels ←CUBSCAN(flags)
32:         for all remaining inputs do
33:             ▷ from Algorithm 3.7
34:             winners ←BIDDING(S, inputs, labels)
35:         for all superclusters do
36:             DELETEITEMS(S, inputs, winners)
37:         for all remaining inputs do
38:             ▷ compact out inserted values
39:             inputs ←CUBSELECT(inputs, winners)
40:     until length(inputs) = 0
41:     return
```

Algorithm 8 Merging Filters

```
 1: function EXTRACTFINGERPRINTS(Q, empty)
 2:     if ISEMPTY(Q[threadID]) then
 3:         empty[threadID] ← TRUE
 4:         return
 5:     if !is_shifted(Q[threadID]) then
 6:         ▷ item is beginning of cluster
 7:         return (threadID << r) ∨ Q[threadID]
 8:     ▷ for shifted items, find beginning of cluster
 9:     b ← threadID
10:     repeat
11:         INCR(b)
12:     until !is_shifted(Q[b])
13:     ▷ step through cluster, counting runs
14:     s ← b
15:     while s ≤ threadID do
16:         repeat
17:             INCR(s)
18:         until !is_continuation(Q[s])
19:         if s > threadID then
20:         repeat
21:             INCR(b)
22:         until !is_occupied(Q[b])
23:     return (b << r) ∨ Q[threadID]

24: function MERGEFILTERS(Q_1, Q_2)
25:     for all QF slots do
26:         f_1 ← EXTRACTFINGERPRINTS(Q_1, empty_1)
27:     for all QF slots do
28:         f_2 ← EXTRACTFINGERPRINTS(Q_2, empty_2)
29:     for all QF slots do
30:         THRUSTREMOVEIF(f_1, empty_1)
31:     for all QF slots do
32:         THRUSTREMOVEIF(f_2, empty_2)
33:     for all extracted values do
34:         f_combined ← MGPUMERGE(f_1, f_2)
35:     ▷ rebuild new filter
36:     SEGMENTEDLAYOUTSBUILD(Q_new, f_combined)
37:     return
```

Algorithm 9 RSQF inserts

```
 1: function INSERTINTOREGIONS(R, starts, nexts, f_q, f_r, size)
 2:     first ← threadID ∗ size
 3:     last ← first + size − 1
 4:     value ← nexts[first]
 5:     block ← first
 6:     while value = NULL ∧ block < last do
 7:         ▷ insert queue for block is empty - check next one
 8:         INCR(block)
 9:         value ← nexts[block]
10:     if value = NULL then
11:         return                                       ▷ no items in queue
12:     home ← f_q[value]%SLOTS_PER_BLOCK
13:     r ← RANK(R[block].occ, home)
14:     end ← SELECT(R[block].run, r)
15:     while end = NULL do
16:         ▷ run end is in next block
17:         r ← r− POPCOUNT(R[block].run)
18:         INCR(block)
19:         end ← SELECT(R[block].run, r)
20:     if block > last then
21:         return TRUE                                  ▷ item is in next region
22:     if end < home then
23:         ▷ slot is empty; insert item here
24:         INSERTHERE(R, end, f_r[value])
25:         INCR(nexts[block])
26:         return TRUE
27:     else
28:         ▷ search through filter for first empty slot
29:         INCR(end)
30:         s ← FINDFIRSTUNUSEDSLOT(R, block, end)
31:         if block > last then
32:             return TRUE                              ▷ out of region
33:         while s > end do
34:             ▷ move items over until we get back to item's run
35:             R[block][s].rem ← R[block][s − 1].rem
36:             R[block][s].rem ← R[block][s − 1].rem
37:             DECR(s)
38:         ▷ find correct slot in run
39:         repeat
40:             if R[block][s − 1].rem ≤ f_r[value] then
41:                 INSERTHERE(R, s, f_r[value])
42:                 INCR(nexts[block])
43:                 return TRUE
44:             DECR(s)
45:         until s < home ∨ R[block][s].run = 1
46:     INSERTHERE(R, s, f_r[value])
47:     INCR(nexts[block])
48:     return TRUE

49: function FINDFIRSTUNUSEDSLOT(R, block, slot)
50:     r ← RANK(R[block].occ, slot)
51:     s ← SELECT(R[block].run, r)
52:     while slot ≤ s do
53:         if s = NULL then
54:             INCR(block)
55:             s ← R[block].offset + 1
56:         slot ← s + 1
57:         r ← RANK(R[block].occ, slot)
58:         s ← SELECT(R[block].run, r)
59:     return slot

60: function FINDBLOCKSTARTINDICES(f_q, starts)
61:     block ← f_q[threadID]/SLOTS_PER_BLOCK
62:     previous ← f_q[threadID − 1]/SLOTS_PER_BLOCK
63:     if block ≠ previous then
64:         starts[block] ← threadID

65: function INSERT(R, inputs)
66:     ▷ preprocessing: hash, sort, quotienting
67:     for all inputs do
68:         FINDBLOCKSTARTINDICES(f_q, starts)
69:     iterations ← 1
70:     size ← 1
71:     while more do
72:         more ← FALSE
73:         nregions ← nblocks/size
74:         for all regions do
75:             more ← INSERTINTOREGIONS(R, starts, nexts, f_q, f_r, size)
76:         INCR(iterations)
77:         size ← iterations/16 + 1
78:     return
```