QSketch: GPU-Aware Probabilistic Sketch Data Structures

Huanyi Qin*, Zongpai Zhang†, Bo Yan‡, Madhusudhan Govindaraju§ and Kenneth Chiu¶
Department of Computer Science, Binghamton University
Binghamton, New York, USA
Email: {*hqin4, †zzhan145, ‡byan2, §mgovinda, ¶kchiu}@binghamton.edu

***Abstract*—A fundamental problem in data analysis is determining the frequency distribution of items within a data set. A count-min sketch is a widely used data structure that can estimate such a distribution. General purpose computation with graphics processing units (GPUs) has become increasingly prevalent over the last decade or so. A GPU provides much higher instruction throughput and memory bandwidth than a typical CPU. However, achieving maximum throughput on GPUs for count-min sketch is challenging, especially for large, uniformly distributed data sets, because each operation needs several random memory accesses that are slow and inefficient on GPUs. In this paper, we propose a suite of novel count-min sketch structures called QSketch. QSketch structures need only one coalesced memory access on average. We also present methods and algorithms to improve the accuracy without decreasing the searching performance. We implemented the QSketch with CUDA 11 and evaluated its performance and accuracy on various GPU platforms.**

I. Introduction

A fundamental problem in data processing and analysis is determining the frequency distribution of items within some stream or over some set. Traditionally, this would be done with a lookup table to maintain the number of occurrences of each item. Hash tables and balanced binary trees are two typical ways of implementing such a lookup table. If the number of unique items is very large, however, these data structures may require a prohibitive amount of memory, in part because they need to store the keys. Furthermore, in many situations, approximate answers are sufficient, leading to what are known as *sketch* data structures [1], [2]. Sketch data structures provide an approximate summary of some aspect of the data set, using a greatly reduced amount of memory.

In the hardware realm, graphic processing units (GPUs) have become increasingly powerful over the past decade or so, and, with the development of CUDA, have led to general purpose computing on GPUs. With their massive parallelism and memory bandwidth, GPUs have great potential for speeding up algorithms and data structures, but can be challenging to exploit due to their architecture.

One such sketch data structure is known as count-min sketch (CM sketch) [3]. A CM sketch estimates the frequency count of items, but may overcount, and thus provides a upper-bound. While CM sketch works well on a host CPU, a direct implementation on a GPU performs poorly. In this work, we present a suite of novel data structures for CM sketch on the GPU, which are named as *QSketch*. We firstly focus on improving the performance and show two algorithms. The first algorithm has some improvement compared to the original CM sketch. Our second algorithm achieves one memory access for each operation, but has lower accuracy. In order to improve the accuracy without losing performance, we then integrate two additional methods. The first method attains higher accuracy than the previous sketches in this paper, but it is still limited by the GPU memory size. The second method can utilize the CPU memory to improve the accuracy. We implemented QSketch with CUDA 11 and evaluated its performance and accuracy on variable GPU platforms. Specifically, we make the following contributions:

- We investigated parallel count-min sketch algorithms and noted that its memory access pattern is uncoalesced. To improve the throughput, we developed QSketch, a suite of novel, heterogeneous CM sketch data structures for GPUs.
- We first developed *warp sketch* in Section III-A, which coalesces some memory transactions and has 60–200% higher throughput than the CM sketches for a variety of distributions.
- Then we developed *sub warp sketch* in Section III-B. It coalesces all memory transactions of an operation into one. Therefore, it has a consistent 200% higher throughput than the CM sketches on different GPUs.
- The *warp sketch* and *sub warp sketch* have worse accuracy than the CM sketch. To address this, We also develop methods to efficiently utilize the GPU memory and improve the accuracy without decreasing the throughput in Section IV-A.
- We integrate algorithms to break the GPU memory size limitation in Section IV-B. Under the help of host memory, *QSketch* can provide higher accuracy while keeping the querying throughput unchanged.

II. Background & Related Work

A. GPU

A graphics processing unit (GPU) provides much higher instruction throughput and memory bandwidth than a CPU within a similar price and power envelope [4], [5]. Below we describe Nvidia's GPUs using their terminology, though the concepts apply generally. Threads are grouped into blocks (1-, 2-, or 3-D) and blocks are organized into a grid (also

1-, 2-, or 3-D). A thread block may contain up to 1024 threads and the GPU schedules and executes threads of the same block in groups of 32 parallel threads, which are called warps. A warp of threads can only run in the same Streaming Multiprocessor (SM), and they must execute a common instruction in lockstep. Branches may cause warp divergence, which reduces performance [4].

Each SM has its own L1 cache, which is also used for shared memory. The shared memory can only be accessed by the threads of the same block and is managed manually. The GPU also has an L2 cache that is accessible to all SMs and it caches access to global memory [4]. The global memory is the largest device memory and current GPUs usually have global memories of 4–16 GB. The device memories are weakly-ordered [4], thus the written order of one thread is not guaranteed to be the same as the order observed by another thread. (Note that CUDA uses the term "shared memory" to refer to fast memory that can be shared within a block. Global memory is shared across the device, but is slower.)

B. *Throughput of Global Memory*

One global memory transaction instruction can access a tile of data, which consists of 32 bytes for devices of compute capability 6.0 or higher [6]. When an instruction accesses global memory, if other threads within the warp are accessing nearby addresses, the accesses can be coalesced into fewer transactions, greatly increasing memory throughput. For example, a warp can load a sequence of 32 4-byte integers in only 4 32-byte transactions. However, if the 32 integers are dispersed in a large memory area, they are unlikely to be close to each other and can not be coalesced, so the 32 accesses may need 32 memory transactions. Under this circumstance, each transaction still transfers 32 bytes of data but only 4 bytes out of the 32 bytes are useful, which means that the throughput is divided by 8 [6]. So, global memory accesses should be coalesced whenever possible [4], especially for memory bound algorithms [7]–[11].

C. *Count-Min Sketch*

Algorithm 1: Thread Sketch

```
1  $table[w \times d] \longleftarrow 0$
2  Function insert ($keys$)
3      parallel for each $key$ in $keys$ do
4          for $i \leftarrow 0$ to $d$ do
5              $id \leftarrow hash_i(key)\%w$
6              $atomic\_add(table[i \times w + id], 1)$
7  Function query ($keys$, $counts$)
8      parallel for each $key$ in $keys$, $count$ in $counts$ do
9          $count \leftarrow$ MAX_INTEGER
10         for $i \leftarrow 0$ to $d$ do
11             $id \leftarrow hash_i(key)\%w$
               $count \leftarrow min(count, table[i \times w + id])$
```

A *count-min sketch* (CM sketch, or just *sketch*, for brevity) [3] is a probabilistic data structure which estimates the frequency distribution of a set (or stream) of keys. Sketches do not bound the size of a data set, regardless of the number of different keys; but may overestimate the count of a particular key, thus providing an upper bound on the count. They support three operations: INSERT, REMOVE, and QUERY. INSERT and REMOVE operations insert and remove items from the CM sketch, respectively; while QUERY retrieves an estimated count of the frequency of the given item. CM sketches have some relationship to Bloom Filters [12], but provide a frequency count rather than only set membership information.

A CM sketch uses d count arrays of fixed length w, each with an independent hash function, A key is inserted or removed by locating the value within each count array, and incrementing or decrementing it, respectively. The count of a key is retrieved with QUERY by locating the value within each count array, and returning the minimum of all the values. If two different keys hash to the same value within a count array, the count will be an overestimate of the count for each of those keys.

In Algorithm 1, we show the original CM sketch algorithm, implemented for multiple threads, and can be directly implemented on a GPU. We assume that operations are called with an array of keys and that each key is assigned to a separate thread within the for-loop. We only show the INSERT operation because the REMOVE operation merely replaces `atomicAdd()` with `atomicSub()`.

D. *Related Work*

Some recent papers for sketch [13], [14] focus on improving accuracy rather than performance. The SF-sketch [13], [15] uses two CM sketches as its base. Although its accuracy is higher compared to CM sketch, performance is not improved. Another drawback of the GPU SF-sketch is that it places the whole data structure in device memory, so they can not store a large data set due to limited GPU memory. We overcome those drawbacks in Section IV-B by using QSketch as the base data structure and placing the data separately on host and device.

The Matryoshka-sketch [16] is focused on improving the performance of skewed or power-law data by utilizing fast caches and reducing the interference due to multiple atomic operations to the same variable. A similar strategy is used in another paper [17], which is fast for the Zipfian distribution. Neither of them is efficient for less skewed distributions, however, especially for uniform distribution, which is our focus. A uniform distribution is worst case for the L2 cache of the GPU, since in a skewed distribution some keys will have higher frequencies than others, and thus higher cache hit rate (see Section V).

Below, we will refer to any algorithms that have several count arrays and each key is processed by a single thread, such as Algorithm 1, a *thread sketch*. This category includes [13], [14], [16]–[18].

A number of related papers, such as DyCucko [19], SlabHash [20], and WarpCore [21] investigate GPU implementations for hash tables, and can achieve one atomic operation or one global memory access on average for each operation. Their throughput is similar to thread sketch with $d = 1$. Hash tables, however, are not sketch data structures, and have significantly different functionality and performance from count-min sketches. Hash tables need to store all the keys, however, and thus are not suitable when the space of keys is much greater than the actual number of items. We thus do not provide a detailed comparison of QSketch with hash tables.

III. QSketch: Improving GPU Performance

TABLE I
The notations used in this paper

Notation	Description
w, d	width and depth of thread sketch table
m	width of the bucket
b	bytes of each bucket
n	the number of buckets
H	the number of hash masks (e.g. 1024).
tid	thread index in a warp
sid	the index of the sub warp.
$stid$	the thread index in a sub warp
$smask$	the bit mask of each sub warp
lm, hm	width of the bucket in low and high frequency table
ln, hn	the number of buckets in low and high-frequency table
hi	high bucket index

In this section we develop our QSketch suite of sketches. Recall that a thread sketch has d count arrays and each array has w counter variables where w is much larger than d [3], [22]. An operation will access one random counter variable for each count array. The counter variables for a given key will likely be separated in memory, due to the structure of a thread sketch and the accesses of them are unlikely to coalesce into 32-byte global memory transactions [6]. This characteristic of sketch algorithms significantly reduces their performance, especially for the sketches on GPU.

Thus we focus on improving the throughput and show two variations. The first, termed *warp sketch*, places all the counter variables for a given key near to each other, thus coalescing some memory accesses. The second, the *sub warp sketch*, places the counter variables closer than warp sketch. It achieves one global memory access for each operation. The input to our algorithms for Insert or Remove operation is an array of keys, and each operation modifies a set of counter variables that estimate the occurrence of keys. For Query, the input only has the keys while the output is the estimated count of the given key.

A. Warp Sketch

Memory accesses in thread sketch are likely to be uncoalesced because the counter variables are likely to be far apart from each other. The key insight in warp sketch is to group all the counter variables for a key into a smaller memory region, thus increasing memory coalescing. To facilitate this, warp sketch uses one count array which is then split into n buckets of m counters each. A key is first hashed to a bucket within the array. Then, a subset of the counters within the bucket is selected by hashing into a table of bitmasks, each of size m. The masks are randomly generated, but exactly d bits set to 1. A counter is only active for a key if the corresponding bit in the selected bitmask is 1. The randomized mask is to ensure that keys that collide to the same bucket do not use the same counter variables.

Algorithm 2: Warp Sketch

```
$table[n \times m] \longleftarrow 0$
$hash\_mask\_table[H] \longleftarrow generate\_hash\_mask(H)$
**Function** insert ($keys$)
    **for each** $key$ **in** $keys$ **do**
        $hv \leftarrow hash(key)$
        $id \leftarrow hv\%n$
        $hash\_mask \leftarrow hash\_mask\_table[hv\%H]$
        **for** $i \leftarrow tid$ **to** $m$ **step** $warp\_size$ **do**
            **if** $hash\_mask[i]$ **then**
                $atomic\_add(table[id \times m + i], 1)$
**Function** query ($keys, counts$)
    **for each** $key$ **in** $keys$, $count$ **in** $counts$, **do**
        $result \leftarrow$ MAX_INTEGER
        $hv \leftarrow hash(key)$
        $id \leftarrow hv\%n$
        $hash\_mask \leftarrow hash\_mask\_table[hv\%H]$
        **for** $i \leftarrow tid$ **to** $m$ **step** $warp\_size$ **do**
            **if** $hash\_mask[i]$ **then**
                $result \leftarrow min(count, table[i \times m + id]$
        $result \leftarrow warp\_reduce\_min(result)$
        **if** $thread\_index == 0$ **then**
            $count \leftarrow result$
```

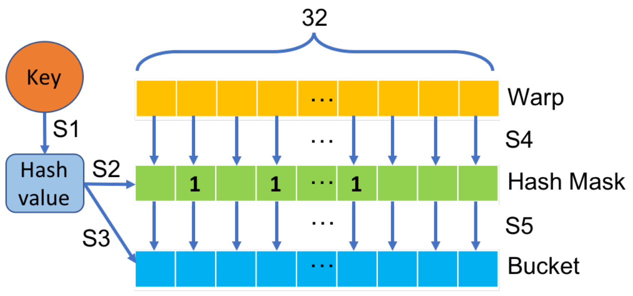


Fig. 1. The Insert procedures of warp sketch ($d = 3, m = 32$). S1–5 are steps. The yellow squares are 32 threads in a warp. The green squares are bits of the hash mask. 3 of them are 1 while the rests are 0. The blue squares are counter variables.

Our algorithm is shown in Algorithm 2, with a conceptual diagram of the steps in Figure 1. (Since Remove is very similar to Insert, only replacing an addition with a subtraction, we only show Insert.) The variable tid is the thread index relative to the warp (0–31). The variable $hash\_mask$ is the m-bit bitset, with exactly d bits randomly set to 1. Since the hash mask table is much smaller than the sketch, it is stored in shared memory (not global memory) for fast access by all threads in the same block [4], [23]. For example, a hash mask table storing 1024 128-bits hash masks

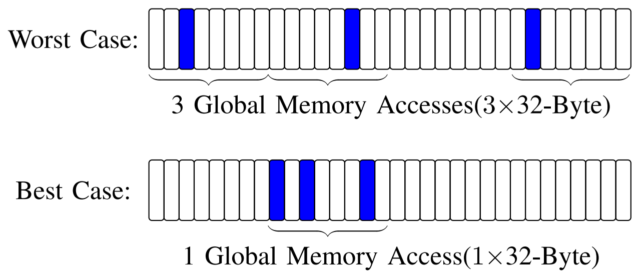

Worst Case:
3 Global Memory Accesses(3×32-Byte)
Best Case:
1 Global Memory Access(1×32-Byte)

Fig. 2. Worst And Best Cases For Warp Sketch ($d = 3$, $m = 32$). The blue rectangles are selected (its bit in hash mask is 1) counter variables (4-byte integers) in an INSERT or QUERY operation. The horizontal brackets are global memory accesses. Each access can read or write the 8 variables in the bracket.

will only need 16 KB memory. Different sketch objects may share the same hash mask table, and they can be pre-generated.

Each thread first calculates the hash value of the key (S1 in Figure 1). It will then use the hash value to select a bucket and load a random hash mask (S2 and S3 in Figure 1). (Note that due to SIMT model of CUDA, there is no advantage to having only one thread compute the hash value.) Though we use a single hash value, as long as H is relatively prime to n, keys that hash to the same bucket will use different masks.

Then all threads in the warp will increment or decrement the counter variables if the corresponding bits in the hash mask are 1 (S4 and S5 in Figure 1). The increment or decrement function must be atomic because different warps may access the same counter variable simultaneously, although they have different keys. Those memory accesses are guaranteed to access the variables in the same bucket, and the size of the bucket is relatively small so the memory addresses are close to each other. This causes more memory accesses to be coalesced, which will increase the insertion performance.

We assume that the operations are performed in batch mode. For an operation, each warp is allocated a list keys and processes each key sequentially, but the threads in the warp operate in a parallel for a single key.

The QUERY operation will invoke similar functions but will not change the counter variables. The counter variables will be loaded if the corresponding bits in the hash mask are 1. The minimum of these is then written to the result array. The minimum value is calculated by several warp shuffle functions [4]. CUDA 11 introduces some new warp reduce functions, such as `reduce_min_sync()` [4], which may improve search performance. Nevertheless, those functions are only supported by devices of compute capability 8.x or higher, so we do not study their performance in this paper.

The size of the bucket m will influence both the throughput and accuracy. The smaller m will lead to higher performance because the memory accesses are more likely to be coalesced. If m is greater than the warp size, each warp must step through the bucket in a loop, which reduces the throughput. If m is equal to the warp size and the counter variables are 4-byte integers, the warp will need to access 32 4-byte integers, which will be coalesced into at most 4 32-byte memory accesses. So the number of actual memory accesses for each operation is the minimum of d and 4. For example, there can be only one 32-byte memory access if all bits of value 1 are concentrated in a quarter of the 32-bit hash mask, which is the best case in Figure 2. However, there will be $\min(d, 4)$ memory accesses for most operations, because all bits of a hash mask have equal possibility to be set and they should be discrete and appear in multiple quarters of a 32-bit hash mask. Figure 2 shows examples of worst case and best case.

B. Sub-Warp Sketch

Algorithm 3: Sub-Warp (Quarter-Warp) Sketch

```
1  $table[n \times m] \longleftarrow 0$
2  $hash\_mask\_table[H] \longleftarrow generate\_hash\_mask(H)$
3  $sid \longleftarrow tid/sub\_warp\_size$
4  $stid \longleftarrow tid\%sub\_warp\_size$
5  $smask \longleftarrow$ 0XFF << $(sid \times sub\_warp\_size)$
6  Function insert (keys)
7      for each $key[4]$ in $keys$ do
8          if $stid == 0$ then
9              $hv[sid] \leftarrow hash(key[sid])$
10             $id \leftarrow hv[sid]$ % $n$
11         $hash\_mask[sid] \leftarrow hash\_mask\_table[hv[sid]$ $\%H]$
12         $id \leftarrow sub\_warp\_broadcast(smask, id)$
13         if hash_mask[sid][stid] then
14             $atomic\_add(table[id \times m + stid], 1)$
15 Function query (keys, counts)
16     for each $key[4]$ in $keys$, $count[4]$ in $counts$ do
17         $result \leftarrow$ MAX_INTEGER
18         if $stid == 0$ then
19             $hv[sid] \leftarrow hash(key[sid])$
20             $id \leftarrow hv[sid]$ % $n$
21         $hash\_mask[sid] \leftarrow hash\_mask\_table[hv[sid]$ $\%H]$
22         $id \leftarrow sub\_warp\_broadcast(smask, id)$
23         if hash_mask[sid][stid] then
24             $result \leftarrow min(result, table[id \times m + stid]])$
25         $result \leftarrow sub\_warp\_reduce\_min(result)$
26         if $stid == 0$ then
27             $count[sid] \leftarrow result$
```

As mentioned in the previous section, the smaller m will lead to higher throughput because the memory accesses are more likely to be coalesced, but it still needs several 32-byte global memory accesses for most operations even m equals the warp size. If m is smaller than warp size, measurements show degraded performance. In order to continue to improve the throughput based on warp sketch, we developed *sub warp sketch*. Sub warp sketch divides a warp (32 threads) into several sub warps, and the threads of a sub warp will work together to accomplish INSERT and QUERY operations. In other words, a single warp will execute multiple operations simultaneously.

In order to achieve the best performance and balance the workload among sub warps, the size of the sub warp should be a factor of the warp size (32 threads) [4]. The m should be equal to the sub warp size. If m is greater than sub warp size, a larger sub warp size should be more efficient because an extra iteration is then unnecessary. The sub warp can be a half-warp or a quarter-warp if the counter variables are 4-byte integers. They are named as *half warp sketch* and *quarter warp sketch*. In this paper, sub warp sketch by default refers to quarter warp sketch. For half warp sketch, the 16 threads of the half-warp will operate on up to 16 4-byte adjacent integers, and they can be coalesced into one or two 32-byte memory transactions. When $d = 3$, $\frac{1}{4}$ operations need one memory transaction, And $\frac{3}{4}$ operations need two memory transactions, so 1.75 memory transactions are needed for each operation on average. Fewer memory transactions are required compared to warp sketch. Similarly, for quarter warp sketch, the 8 threads of the quarter-warp will operate on up to 8 4-byte integers but they are guaranteed to be coalesced into one 32-byte memory transaction.

The sub warp can not be smaller than a quarter-warp if the counter variables are 4-byte integers, because CUDA's memory transactions for global memory are 32-byte, which can exactly load a bucket of 8 4-byte integers. And the two adjacent $\frac{1}{8}$ parts of one warp may not load the adjacent buckets since they have different keys. Both of them will need a 32-byte transaction to load their bucket and only half of the data is useful, which results in the overall performance halving.

Algorithm 3 shows the INSERT and QUERY operations for sub warp sketch, where each sub warp is a quarter warp and contains 8 threads. The sid is the sub warp index (0–3), and $stid$ is the thread index of each sub warp (0–8). The $smask$ is a special mask that the bits are 1 for all the threads in the same sub warp, the bits are zero for the threads in the same warp but in different sub warps [4]. They are used in `sub_warp_broadcast()` and `sub_warp_reduce_min()`. For example, 0X000000FF, 0X0000FF00, 0X00FF0000, and 0XFF000000 are the masks of 4 sub warps, and all threads within the sub warp have the same $smask$.

For the INSERT and QUERY, the first threads of each sub warp (4 threads in total), will calculate 4 hash values, then load the 4 selected hash masks. The 4 hash masks are stored in the shared memory, and each sub warp will operate in its area. The `sub_warp_broadcast()` broadcasts the bucket id to other threads in the same sub warp, the 4 broadcasting functions are processed in one warp shuffle function (`__shfl_sync()`), with the help of $smask$. After the loading of hash masks, each sub warp will read or modify the counter variables if the corresponding bits of the hash masks are 1. The QUERY also needs to calculate the minimum values within the sub warp using `sub_warp_reduce_min()`.

Those memory accesses are guaranteed to be coalesced into one 32-byte memory transaction because each bucket contains 8 4-byte integers (32 bytes in total) and all the buckets are at least 32-byte aligned.

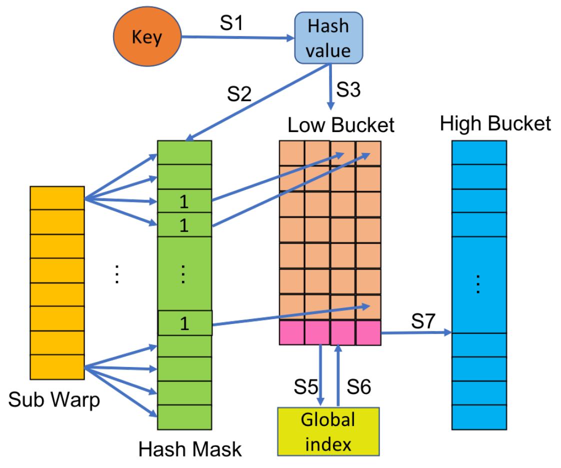


Fig. 3. INSERT of Multi-level Sub Warp Sketch. S1–7 are steps and the last 4 bits of hash mask are always 0. In the low bucket, each square represents a byte. The light brown square is a counter variable and the 4 pink squares form a high bucket index.

IV. QSKETCH: IMPROVING THE ACCURACY

The warp sketch and sub warp sketch improve the INSERT and QUERY throughput as the size of bucket (m) shrinks. However, the smaller m also leads to lower accuracy. We show the accuracy in Figure 7. In this section, we integrate other methods to increase the accuracy without decreasing INSERT or QUERY throughput.

The accuracy of a sketch is limited by the total size of the underlying count arrays. A larger count array usually leads to a more accurate sketch, because it can store more counter variables and each counter variable has a lower probability of accounting for multiple items (hash collision), which allows all counter variables of an item to be closer to the actual value [3], [15], [17]. Thus, the estimated frequency which is the minimum value of counter variables will be closer to the real frequency of the item and the sketch can achieve a high QUERY accuracy.

In this section, we integrate two different methods into QSketch. The two methods can work with both warp sketch and sub warp sketch. We only show the details with sub warp sketch since it has higher throughput than warp sketch. The first, which we term *multilevel warp sketch* (MWS) uses two sketches. One sketch is capable of storing a large number of low frequency items and another one can store a small number of high frequency items. The second, which we term *SF sub warp sketch* or SFSWS, also uses two sketches. One sketch is stored in device memory (GPU memory) and it can achieve high throughput for QUERY. Another sketch is stored in host memory (CPU memory), which can store more items without the limit of device memory.

A. Multi-level Sub Warp Sketch

Some sketches use an unsigned character (1-byte integer) as a counter variable instead of a 4-byte integer, allowing 4 times as many counter variables for the same memory size.

Algorithm 4: Multi-level Sub Warp Sketch (MSWS)

```
1  $low\_frequency\_table[ln \times lm] \longleftarrow 0$
2  $high\_frequency\_table[hn \times hm] \longleftarrow 0$
3  $mask\_table[H] \longleftarrow generate\_hash\_mask(H)$
4  $sid \longleftarrow tid/sub\_warp\_size$
5  $stid \longleftarrow tid\%sub\_warp\_size$
6  $smask \longleftarrow$ 0XFF $<< (sid \times sub\_warp\_size)$
7  $global\_index \longleftarrow global\_index\_start$
8  $limit \longleftarrow low\_bucket\_limit$
9  Function insert(keys)
10     for each $key[4]$ in $keys$ do
11         if $stid == 0$ then
12             $hv[sid] \leftarrow hash(key[sid])$
13             $id \leftarrow hv[sid]$ % $ln$
14         $hash\_mask[sid] \leftarrow$
             $hash\_mask\_table[hv[sid]$ $\%H]$
15         $id \leftarrow sub\_warp\_broadcast(smask, id)$
16         $low\_bucket \leftarrow load\_bucket\_lock(id)$
17         if $low\_bucket == NULL$ then
18             $degenerate()$
19             continue
20         $cv, hi \leftarrow low\_bucket$
21         if $any\_sync(hi > global\_index\_start)$ then
22             $warp\_insert\_high()$
23         $inc \leftarrow merge(cv)$
24         $max\_low \leftarrow max(split(cv))$
25         if $inc \neq 0$ and $max\_low < limit$ then
26             $sub\_warp\_insert\_low()$
27         if $max\_low >= limit$ then
28             $id \leftarrow atomic\_allocate(low\_bucket)$
29             $warp\_insert\_high()$
30 Function query(keys, counts)
31     for each $key[4]$ in $keys$, $count[4]$ in $counts$ do
32         if $stid == 0$ then
33             $hv[sid] \leftarrow hash(key[sid])$
34             $id \leftarrow hv[sid]$ % $ln$
35         $hash\_mask[sid] \leftarrow$
             $hash\_mask\_table[hv[sid]$ $\%H]$
36         $id \leftarrow sub\_warp\_broadcast(smask, id)$
37         $low\_bucket \leftarrow$
             $load\_bucket(low\_frequency\_table, id)$
38         $cv, hi \leftarrow low\_bucket$
39         $result\_low \leftarrow sub\_warp\_search\_low()$
40         if $any\_sync(hi > global\_index\_start)$ then
41             $result\_high \leftarrow warp\_search\_high()$
42         $min\_low \leftarrow warp\_reduce\_min(result\_low)$
43         $min\_high \leftarrow$
             $warp\_reduce\_min(result\_high)$
44         if $stid == 0$ then
45             $count[sid] \leftarrow min\_low + min\_high$
```

This method is simple but useful for data with low-frequencies, especially for more uniformly distributed data. However, the maximum value of unsigned character is 255 and thus may overflow.

The *multi-level sub warp sketch* (MSWS) uses two tables to handle the 1-byte integer overflow problem, which can improve the accuracy without decreasing the throughput, especially when the majority of data is not too skewed. The first table is the high-frequency table and is the same as the table in warp sketch ($m = 32$). It uses 4-byte or 8-byte unsigned integers as its counter variables and is stored in global memory. But it is much smaller than a normal warp sketch and thus the accesses to the high-frequency table are more likely to be cached in the L2 cache or other fast on-chip caches. A high-frequency table that stores 1024 128-byte buckets (termed *high buckets*) will only need 128 KB memory and is smaller in size than the L2 cache size of current GPUs [4].

The second table is the low-frequency table, which is based on the sub warp sketch. It can store a large number of low frequency items with high accuracy and it will use 1-byte unsigned integer as its counter variable. In Algorithm 4 and Figure 3, we show the implementation where the low-frequency table is based on quarter warp sketch instead of half warp sketch, since the quarter warp sketch has higher throughput than half warp sketch. The low-frequency table contains many buckets (termed *low buckets*), and Figure 3 shows the connection between the low bucket and the high bucket. The size of each low bucket is 32-byte which can fit in one global memory access. It stores 28 1-byte counter variables and the last 4-byte (high bucket index) of a low bucket can be used to store an index to the high-frequency table. In Algorithm 4, *hi* is the short name of high bucket index, 0–1023 of the high bucket index are reserved and 1024–(2^{32} − 1) are valid indices. The 4-byte unsigned integer is enough for the size of current device memory, it can support about 2^{32} 128-byte high buckets and the high-frequency table can use up to 512 GB global memory which is much bigger than the largest device memory. Although the low bucket needs to leave 4 bytes for preventing the overflow problem, it still can have 3.5 times counter variables than the buckets with 4-byte integers. Each hash mask is still 32-bit and the bits for the high bucket index are always zero, it will calculate a normal hash mask of 28-bit and pad 4-bit zero.

The INSERT of MSWS is similar to that of *sub warp sketch*. In Algorithm 4, the special processes start at Line 16. The sub warps will lock the low bucket and then load the contents of each low bucket by single global memory access. Then sub warps will check the values of 4 high bucket indices via a warp vote function (`any_sync()`) [4], [24]. If there is at least one of them pointing to a valid high bucket, the 4 sub warps will first merge into a warp, and it will insert the keys into the high bucket for the valid high bucket indices (S7 in Figure 3) . The warp then split into 4 sub warps, and the sub warps that have the valid high bucket indices are now idle.

The sub warps with invalid high bucket index will then insert the items to their low buckets simultaneously. The sub warp will first load the low bucket and calculate the maximum value of the counter variables within a sub warp. If the maximum value is greater than the limit (e.g. 128) and the low bucket is going to overflow, the warp will request a new high bucket by `atomicAllocate()` and insert the element to the newly allocated bucket. If the maximum value is not greater than the limit, the sub warps will merge the 1-byte integers into 4-byte integers and insert them to the low-frequency table.

Algorithm 5: Load Bucket Lock

```
1  Function load_bucket_lock(id)
2      $bucket \leftarrow low\_frequency\_table + id \times lm$
3      $match \leftarrow have\_duplicate\_address(bucket)$
4      if $match$ then
5          return NULL
6      $hi \leftarrow 1$
7      $lock \leftarrow bucket + lm - 1$
8      while $hi == 1$ do
9          $hi = atomicExch(lock, 1)$
10         $threadfence\_block()$
11     $barrier()$
12     return $load(bucket)$
```

The Algorithm 5 shows the details of `load_bucket_lock()`. If multiple sub warps of the same or different warps access the same block simultaneously, it is possible that they all read the same old contents from the bucket and then they all insert items into low bucket. An overflow may happen due to the race condition. A lock is needed to prevent this race condition. The lock is implemented var `atomicExch()`, and stored into the high bucket index. It will use the reserved indices, so no extra memory is needed for the lock. The threads of a warp are in lockstep and they will try to lock 4 buckets at the same time. After all sub warps get their locks and reach the barrier, they will load and return the contents of the bucket. However, if two or more sub warps of the same warp try to lock the same bucket, there is at most one sub warp that will get the lock and others will fail. Then the sub warps will never get the needed locks. In order to prevent this, for sub warps of the same warp, *thread sketch*we check if they try to lock the same bucket by comparing the addresses of buckets. If so, the INSERT of those 4 keys in the current warps will degenerate from *sub warp sketch* to *warp sketch*. They need 4 individual INSERT of *warp sketch*. The overhead of lock and `degenerate()` is non-trivial, and it will decrease the throughput by 5%–15%. If the majority of data are not highly skewed, the interference rarely happens in practice, and thus there is minimal performance loss. We tested MSWS more than 100,000 times and each test case has more than 1 million insertions. We do not observe any interference. Of course, some very skewed distributions may have different behaviors.

For QUERY, the sub warps will first load the counter variables from low bucket without need of lock and spilt them into 1-byte integers. If the low bucket is linked to any high bucket, it also needs to load the counter variables from high bucket. Then the sub warps will calculate two minimum values, the first is the estimated frequency for low-frequency table, and the second is for high-frequency table. The last step is to add them together to get the estimated frequency of the element.

Most operations do not need to access the high-frequency table, and the high-frequency table is usually small enough to be well cached. So, each INSERT or QUERY will still need one global memory access on average.

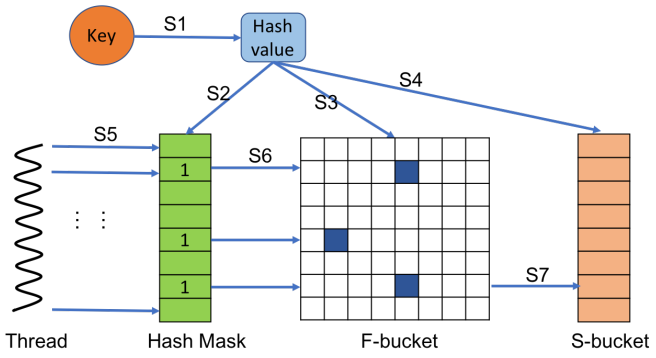


Fig. 4. INSERT of SF Sub Warp Sketch ($d = 3, m = 8, z = 8$). S1–7 are steps. The thread is a normal CPU thread and the F-bucket is a bucket of Fat Sketch in host memory. The height (m) of the F-bucket is always 8 for SFSWF. But the width (z) may be different. The blue squares are selected counter variables. The S-bucket is in global memory.

B. SF Sub Warp Sketch

Algorithm 6: SF Sub Warp Sketch

```
1  $device\_table[n \times m] \longleftarrow 0$
2  $host\_table[n \times m \times z] \longleftarrow 0$
3  $device\_mask\_table[H] \longleftarrow generate\_hash\_mask(H)$
4  $host\_mask\_table[H] \longleftarrow generate\_hash\_mask(H)$
5  Function insert(keys)
6      parallel for each $key$ in $keys$ do
7          $hv \leftarrow hash(key)$
8          $id \leftarrow hv\%n$
9          $sub\_bucket\_id[d] \leftarrow$
             $conver\_to\_index(host\_mask\_table[hv\%H])$
10         for $i \leftarrow 0$ to $d$, step 1 do
11             $sbi \leftarrow sub\_bucket\_id[i]$
12             $sub\_bucket \leftarrow host\_table[id][sbi]$
13             $j \leftarrow hash(key, i)\%z$
14             $atomic\_add(sub\_bucket[j], 1)$
15             $device\_update(sub\_bucket)$;
```

For some users, the QUERY throughput is much more important than the INSERT and REMOVE. And many clusters or servers also have a large unused host memory when running the sketches on GPU. We develop a new version of *QSketch*, which is named as *SF sub warp sketch* or (SFSWS). The SFSWS can use the host memory to improve the accuracy without losing the QUERY throughput. It keeps the QUERY throughput the same as *sub warp sketch*, although it has to suffer a much lower INSERT performance. In this section, we only use the *quarter warp sketch* because it has higher throughput.

This idea was inspired by *SF-Sketch* [13], [15], which improves the accuracy by using two sketches. The *SF-Sketch* has two sketches, *Slim Sketch* and *Fat Sketch*, both of them are *count-min sketch*. The *Slim Sketch* is very small and is stored in the fast cache, while the *Fat Sketch* is large and in the slow memory. They will make sure that each counter variable of *Slim Sketch* is always the maximum value of a corresponding group of *Fat Sketch* counter variables. The size of each group is z, and the *Fat Sketch* is z times larger than *Slim Sketch*. They also implemented it with CUDA and their performance is the same as *count-min sketch* and *thread sketch*, which is much slower than *QSketch*.

In Algorithm 6, the SFSWS also uses two sketches. The first sketch is the device table, and its data structure is a *sub warp sketch*. It has p buckets and each bucket has w counter variables. The device table serves as *Slim Sketch* and is stored in the global memory. Another sketch is the host table. Its data structure is a *warp sketch*. It has p buckets but each bucket has w sub bucket. Each sub bucket has z counter variables. The host table serves as *Fat Sketch* and is stored in the host memory. The host table is z times larger than the device table, and each counter variable in the device table is always the maximum value of z counter variables of a sub bucket in the host table.

The SFSWS executes INSERT on the host side. In Algorithm 6 and Figure 4, the host thread first calculates a hash value (S1 in Figure 4) then selects a bucket and the hash mask according to the hash value (S3 in Figure 4). It will update the random counter variables (under the help of another hash function) of the sub buckets if the bits in the host hash mask are 1 (S5 and S6 in Figure 4). After finishing the operation, the host thread also needs to update the device counter variable (S7 in Figure 4). The host thread copies z counter variables to the device memory. Then the device threads will calculate the maximum value of z variables and update its device counter variable.

The throughput of INSERT is limited by the computing power and memory bandwidth of CPU, and it is much lower than the *Sub warp sketch*. The QUERY is exactly the same as what the *sub warp sketch* does, so the QUERY performance is unchanged.

V. RESULTS

We evaluated *QSketch* on both NVIDIA Tesla P100 and Nvidia GeForce RTX 2080 Ti since they have different global memories. The P100 has 12GB HBM2 memory (with ECC enabled) while The 2080ti has 11GB GDDR6 memory. We compiled the codes with CUDA 11 and run the program with a large range of grid size (1024–2^{23}) to get the peak performance [6], [25]. We tested the *QSketch* for 10 batches and each batch ran the test code more than 50 times. We only recorded the best performance of each batch for P100 because the P100 was installed on the cluster and the node may be shared by other users. The final results are the average of all the best performances of each batch. We claim that the *quarter warp sketch* only needs a single coalesced atomic access on average for each INSERT or REMOVE. We only show the throughput of INSERT since REMOVE is similar to INSERT. And we also claim that *quarter warp sketch* requires only one global memory access for each QUERY on average.

We show results for uniform distributions, since it is the worst case for the L2 cache of the GPU. In a skewed distribution, some keys will have higher frequencies than others, and thus higher cache hit rate. To confirm this, we tested with a variety of skewed distributions, and all showed better throughput. We thus focus on the worst case, the uniform distribution.

We carefully choose the arguments of all the sketches so that they consume the same amount of device memory. The time usages of generating random keys and transferring data between host and device are not calculated in the performance. The random keys of uniform distribution are generated by `curand` [4] on the device. The input keys are grouped into batches so they can fit in a device buffer. We also test different buffer sizes (2–128 MB) and we find the 64 MB buffer is enough for most situations. The number of input data is highly related to the throughput and accuracy. We test the QSketch for different Work Load Factor (WLF).

$$WLF = \frac{inserted\ keys}{total\ counter\ variables}$$

.

We compared the throughput of thread sketch, warp sketch, sub warp sketch, and MSWS; keeping $d = 3$ constant for all sketches. For the warp sketch, we present its throughput at $m = 32$, which is its maximum performance. For the sub warp sketch, we also show its maximum throughput, which is the performance of quarter warp sketch. The throughput of MSWS is slightly different from sub warp sketch, and we only show the throughput of the lock-free version. The throughput of MSWS with lock is 5%–15% less than that of the lock-free version. We do not graph the performance of SFSWS; its QUERY throughput is exactly the same as sub warp sketch, but its INSERT throughput is much lower than other sketches, at about 20 Mops/s. This is expected because INSERT is executed on the host.

And we compared the accuracy of thread sketch, warp sketch, sub warp sketch, MSWS, and SFSWS. For SFSWS, we show two different results for $z = 3$ and $z = 8$. We use the relative error (RE) to quantify the accuracy of sketches. The relative error is $|e - a|/a$, where e is the estimated value of frequency and a is the actual value.

A. Throughput

Figure 5 shows the INSERT throughput of 4 sketches on 2080 Ti and P100. For small data, INSERT throughput improves as the input data size expands because if there are not enough insertion keys, some SMs will be idle and the

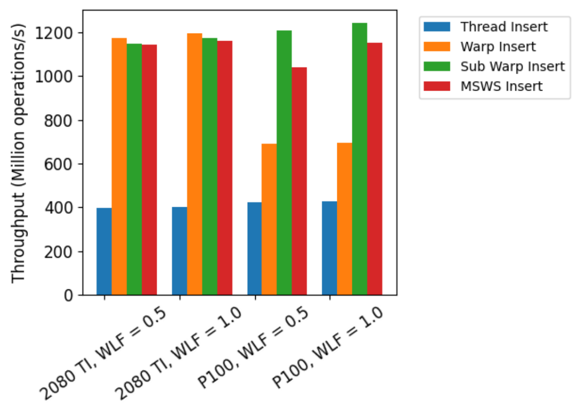


Fig. 5. Insert throughput under different workload factors: This figure shows the average throughput for 2^{27} insert operations. WLF is the Work Load Factor (see Section V). The *thread sketch* is the CM Sketch (see Section II-D). The throughput of *sub warp sketch* is 3x faster than *thread sketch* for large data set.

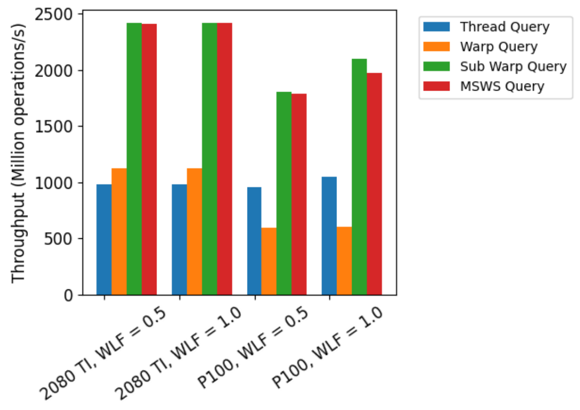


Fig. 6. Query throughput under different workload factors: This figure shows the average throughput for 2^{27} query operations. WLF is the Work Load Factor (see Section V). The *thread sketch* is the CM Sketch (see Section II-D). The throughput of *sub warp sketch* is 2x faster than *thread sketch* for large data set.

overall performance will be low. The count arrays in sketches are also small because the WLF is a constant number. Most of the counter variables are cached in the L2 cache, so we will have a higher L2 cache hit rate for the small input data set. The sketches are not limited to the memory throughput, and *thread sketch*, *warp sketch*, *sub warp sketch* and MSWS have an increasing computational workload, which leads to decreasing INSERT throughput.

The INSERT throughput of *thread sketch* (CM Sketch) greatly decreases as the data size increases because it runs out of L2 cache and the L2 cache hit rate is low. Most memory accesses are not cached and the *thread sketch* is limited by the random access performance of global memory [26]. The INSERT throughput of *sub warp sketch* is approximately three times that of *thread sketch*, which means that *sub warp sketch* successfully coalesced three global memory accesses into one. Figure 6 shows the QUERY throughput of 4 sketches. The behaviors of QUERY are similar to INSERT. The experiments shows that QSketch improves the throughput by 1.6x–3.0x.

The INSERT and QUERY throughput of *sub warp sketch* are close to the performance of example one random memory access algorithms, such as *thread sketch* ($d = 1$) and GPU hash tables [19], [20]. But the *sub warp sketch* is still slightly slower than them because those algorithms only need to access 4 or 8 bytes of global memory for each operation, while the *sub warp sketch* needs to access 32 or more bytes for each operation.

2080 Ti and P100 have different device memories, 2080 Ti has 11 GB GDDR6 memory while P100 has 12 GB HBM2. In addition, the memory bandwidth of P100 is 540GB/s which is 87.7% of the bandwidth of 2080 Ti, and the searching performance of P100 is also about 87.7% of 2080 Ti. These indicate that the searching performance is highly related to the global memory bandwidth.

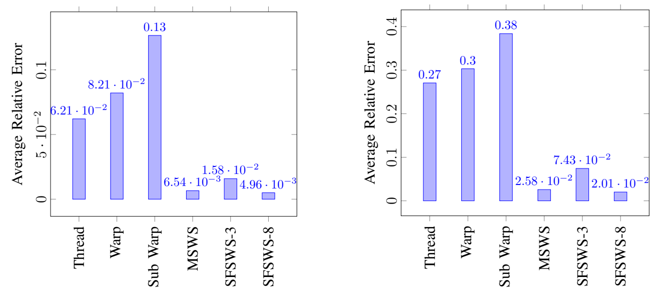
B. Accuracy

In Figure 7, the *thread sketch*, *warp sketch*, and *sub warp sketch* have a decreasing accuracy when the size of the bucket (m) is going smaller and the counter variables of an element in *warp sketch*, and *sub warp sketch* will become much more concentrated than that of *thread sketch*.

The *warp sketch* and *sub warp sketch* can only select a random hash mask based on the hash mask table. The total number of hash masks is much smaller than the size of a normal hash table if the sketches want to have a high L2 cache hit rate and high performance. So the hash masks have a relatively high probability of collision. Even if different hash masks are selected, the counter variables within a bucket still have a high collision probability for the small size of the bucket. MSWS outperforms thread sketch, warp sketch, and sub warp sketch because it can have more counter variables while using the same size of device memory. The SFSWS also has a smaller RE than thread sketch, warp sketch, and sub warp sketch. The accuracy of SFSWS increases with the increase of z. The larger z needs more host memory while the usage of device memory is unchanged.

VI. CONCLUSION

With the advent of streaming sources of large data, sketch algorithms are increasingly relevant. Often, only a sketch is needed. Written for CPUs, however, traditional sketch data structures such as CM sketch do not perform well on GPUs. In this paper, we propose a suite of four novel sketch data structures on GPU, with varying performance characteristics. We show that, despite the challenge of the GPU memory architecture, we can achieve one memory access for each operation and obtain a much higher throughput than the CM sketch. We also develop and present methods to improve its accuracy, and we also utilize the host memory to improve accuracy while still having a high QUERY throughput.



(a) WLF = 0.5 (b) WLF = 1.0

Fig. 7. Accuracy for different work load factors (WLF): This figure shows the relative error (lower is better) for different sketches Note that the accuracy is the same on different GPUs if they use the same hash functions. The accuracy of warp sketch and sub warp sketch are worse than thread sketch (CM Sketch). The accuracy of MSWS, SFSWS-3, and SFSWS-8 are 9.48x, 3.91x, 12.5x higher than the thread sketch for WLF = 0.5. The accuracy also has similar improvement (10.50x, 3.64x, 13.43x) compared to the thread sketch for WLF = 1.0.

References

[1] V. Wei, N. Ivkin, V. Braverman, and A. S. Szalay, "Sketch and scale geo-distributed tsne and umap," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 996–1003.

[2] A. Mandal, H. Jiang, A. Shrivastava, and V. Sarkar, "Topkapi: parallel and fast sketches for finding top-k frequent elements," *Advances in Neural Information Processing Systems*, vol. 31, pp. 10 898–10 908, 2018.

[3] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196677403001913

[4] NVIDIA Corporation, "CUDA C++ Programming Guide," 2021, version 11.4.

[5] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler, "Benchmarking opencl, openacc, openmp, and cuda: Programming productivity, performance, and energy consumption," in *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, ser. ARMS-CC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–6. [Online]. Available: https://doi.org/10.1145/3110355.3110356

[6] NVIDIA Corporation, "CUDA C++ Best Practices Guide," 2021, version 11.4.

[7] P. Chen, M. Wahib, S. Takizawa, R. Takano, and S. Matsuoka, "A versatile software systolic execution model for gpu memory-bound kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356162

[8] Y. Hu, P. Kumar, G. Swope, and H. H. Huang, "Trix: Triangle counting at extreme scale," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[9] Y. Hu, H. Liu, and H. H. Huang, "Tricore: Parallel triangle counting on gpus," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 171–182.

[10] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang, "Sep-graph: Finding shortest execution paths for graph processing under a hybrid framework on gpu," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 38–52. [Online]. Available: https://doi.org/10.1145/3293883.3295733

[11] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–10.

[12] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, Jul. 1970. [Online]. Available: https://doi.org/10.1145/362686.362692

[13] L. Liu, Y. Shen, Y. Yan, T. Yang, M. Shahzad, B. Cui, and G. Xie, "Sf-sketch: A two-stage sketch for data streams," *IEEE Transactions on Parallel & Distributed Systems*, vol. 31, no. 10, pp. 2263–2276, oct 2020.

[14] C.-Y. Hsu, P. Indyk, D. Katabi, and A. Vakilian, "Learning-based frequency estimation algorithms." in *International Conference on Learning Representations*, 2019.

[15] L. Liu, Y. Shen, Y. Yan, T. Yang, M. Shahzad, B. Cui, and G. Xie, "Sf-sketch: A two-stage sketch for data streams," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 10, pp. 2263–2276, 2020.

[16] A. Mandal, "Enabling parallelism and optimizations in data mining algorithms for power-law data," Ph.D. dissertation, Georgia Institute of Technology, 2020.

[17] F. Taşyaran, K. Yıldırır, M. K. Taş, and K. Kaya, "One table to count them all: Parallel frequency estimation on single-board computers," in *European Conference on Parallel Processing*. Springer, 2019, pp. 405–418.

[18] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Adaptive measurements using one elastic sketch," *IEEE/ACM Transactions on Networking*, vol. 27, no. 6, pp. 2236–2251, 2019.

[19] Y. Li, Q. Zhu, Z. Lyu, Z. Huang, and J. Sun, "Dycuckoo: Dynamic hash tables on gpus," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2021, pp. 744–755. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICDE51399.2021.00070

[20] S. Ashkiani, M. Farach-Colton, and J. D. Owens, "A dynamic hash table for the gpu," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2018, pp. 419–429. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPS.2018.00052

[21] 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 *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2020, pp. 11–20.

[22] O. Alipourfard, M. Moshref, Y. Zhou, T. Yang, and M. Yu, "A comparison of performance and accuracy of measurement algorithms in software," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3185467.3185475

[23] J. Chen, N. Xiong, X. Liang, D. Tao, S. Li, K. Ouyang, K. Zhao, N. DeBardeleben, Q. Guan, and Z. Chen, "Tsm2: Optimizing tall-and-skinny matrix-matrix multiplication on gpus," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 106–116. [Online]. Available: https://doi.org/10.1145/3330345.3330355

[24] A. Li, W. Liu, L. Wang, K. Barker, and S. L. Song, "Warp-consolidation: A novel execution model for gpus," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 53–64. [Online]. Available: https://doi.org/10.1145/3205289.3205294

[25] V. Volkov, "Understanding latency hiding on gpus," Ph.D. dissertation, University of California, Berkeley, 2016. [Online]. Available: http://proxy.binghamton.edu/login?url=https://www.proquest.com/dissertations-theses/understanding-latency-hiding-on-gpus/docview/1873487227/se-2?accountid=14168

[26] H. Zhou, S. Bateni, and C. Liu, "Gru: Exploring computation and data redundancy via partial gpu computing result reuse," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 43–52. [Online]. Available: https://doi.org/10.1145/3205289.3205318