Efficient GPU Implementation of a Probabilistic Bloom Filter for Estimating Frequent Items

Abstract

To serve the purpose of utilizing and evaluating varieties of parallel computing techniques covered in the course's text, novel numerical algorithms which implement the Probabilistic Bloom Filter (PBF) have been selected to put in test on the platform of GPU computation (more specifically, in the framework of Nvidia's CUDA). PBF, as one of many variants of the classical Bloom Filter, has been recently constructed and applied in many different applications where the frequency estimations of certain items in a large stream data flow need to be acquired in a timely manner with threshold accuracy. Several model parameters in PBF such as the number of keys inserted, length of the filter, and number of hash functions will determine the structure of the CUDA kernel. By varying these parameters we would be able to probe accordingly many scenarios which require different parallel computing techniques in order to solve the problem. In this project we propose a GPU implementation of a PBF using randomly generated data to analyze its performance and accuracy.

Table of Contents

I. Problem Description 3

II. Serial Solution 5

III. Related Work 7

IV. Parallelization Strategy 7

A. Thread/Block Dimension 7

B. Algorithm Implementation 8

C. Data Placement 9

V. Parallel Optimizations 11

VI. Evaluation 12

A. Evaluation setup 12

B. Testing Data Generation 13

C. Evaluation results and analysis 15

VII. Conclusions 18

VI. References 18

I. Problem Description

One of the fundamental operations involving processing substantial amount of elements in a data set is to test the membership of a given element in this data set. If a certain level of false positive detection could be tolerated and, at the same time, the requirement of storage efficiency is urgent, one could use Bloom Filter (BF) as a reliable method to construct, maintain and query per request on a space-efficient data structure with dedicated algorithms. The first such example was shown by Bloom in 1970 [1], and since then, it has not received much attention until when the need arises of processing large data sets in a timely manner with storage-space constraint [2] [3]. Numerous applications are in need of efficient techniques to manage the data deluge experienced in today’s world. Mechanisms for storage and analysis become limited when handling vast amount of data, moreover, consider applications with timing constraints. A memory management unit tracking cached memory blocks, a router distributing packets via network ports, a DNA/RNA sequence alignment, a database management system, and an Internet monitoring system are examples of applications expected to quickly process large datasets. A naïve approach consists of using a counter for each type of item. This approach may incur in high latencies and may not be feasible on memory constrained computing devices, such as coprocessors and embedded systems. A possible solution is a bloom filter.

A basic BF is a data structure with associated algorithms (INSERTION and QUERY), which provides a compact probabilistic representation of the membership of elements in a data set. The data structure takes the form of an *m*-bit array and all of its bits are initially set to zero. Given a set of *n* elements, S = $\{x_1, x_2, \ldots, x_n\}$, for each element in the set, one can invoke INSERTION to calculate its probabilistic representation and record this result in the *m*-bit array. Specifically, after pre-selecting a set of hash functions of number *k*, $h_i(x)$, $0<i<k$, which could generate uniformly distributed hash values given element identifiers as input, the first step in INSERTION is to obtain the hash value (in the range from 1 to *m*) from each hash function for a chosen element x_j, i.e. $h_i(x_j)$, $0<i<k$. The following step in INSERTION is just marking the bit to one in the *m*-bit array at the position of $h_i(x_j)$, if it's not done already. When the whole set S has all been INSERTed into the *m*-bit array, the construction of the BF for S is called to an end. A simple example illustrates the construction of a BF with three elements (*n*=3) and three hash functions (*k*=3) in Figure 1.

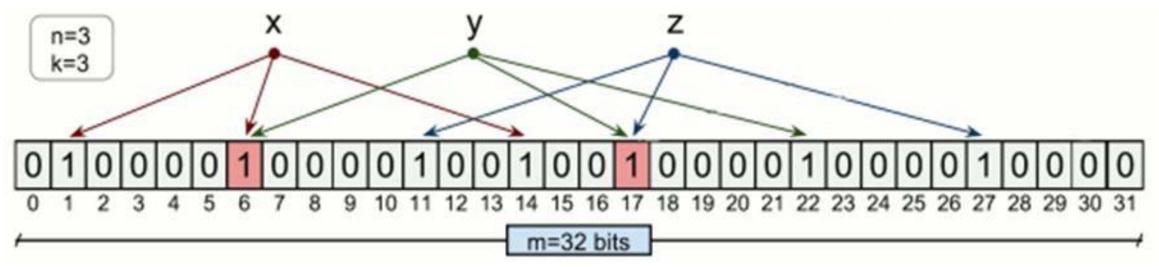

Figure 1. A simple layout of a BF. (Courtesy of Tarkoma et al. [3])

Upon receiving a request for testing the membership of an element, w, QUERY simply calculates the *k* hash values of w from $h_i(w)$, $0<i<k$, and check if the bit at $h_i(w)$ in the *m*bit array is set to zero. Once an instance is found (i.e. certain bit set to zero), QUERY returns false to the test of the membership. Otherwise, a true will be returned. In Figure 2, the QUERY operations are schematically illustrated. Here the same 32-bit wide array generated in the last example with the same hash functions is used, and a query of w's membership is carried out with a false return.

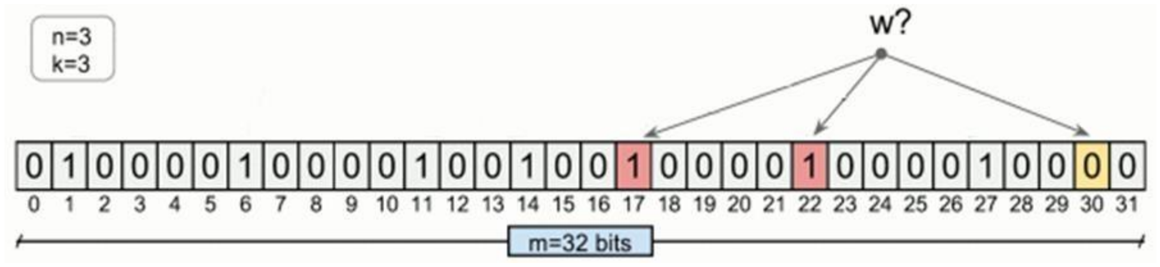

Figure 2. An example of a BF QUERY. (Courtesy of Tarkoma et al. [3])

Beyond the basic BF, one of the most interesting questions is how to not only test the membership of an element in a data set, but also record its frequency of duplication. The latter ability which enables the frequency estimation of an element in a large stream data flow would be of more use in many applications nowadays. Many approaches attempting to address this question have been proposed and corresponding implementations and applications have been developed in these years. For example, Counting Bloom Filter (CBF) has been constructed based on basic BF by replacing every single bit of the *m*-bit array with a fixed size counter (or bins as so called) [3] [4] [5]. Whenever the hash function of an inserted element, x, returns a position $h_i(x)$, the counter at the position will be increased by one. By sacrificing the storage efficiency of the basic BF, the duplication information of an element could be recorded. Many others follow this trend and a more completed reference list could be found in [3] [6].

To suit the needs for calculating the frequency estimations of certain items in a large stream data flow in a timely manner while under constraints of computation resources such as storage space, Yao et al. [6] proposed, implemented and tested an alternative of the BF to overcome the shortcoming of large storage requirement associated with those deterministic BF variants. In this approach, the standard counting operation has been replaced by a probabilistic counting operation which gives the new name Probabilistic Bloom Filter (PBF). Instead of actually increasing counts in the swollen counter at the hashed positions of the m-bit array when an element is to be inserted, the new INSERTION algorithm flips the bit at the hashed position from 0 to 1 according to a preset probability p, which needs no additional storage space to be carried out. Accordingly, for the frequency check of a given element, its appearance in the PBF (m-bit array) at positions determined by the set of hash functions is recorded in a counter and, at the end, being fed to a statistical inference function to obtain the approximate frequency of the element in the data set. A membership test based on bloom filters can result in false positives which means test true of an element in the set when it was not. On the other hand, there is never a false negative possible in the bloom filter.

Even though the nature of this new approach is fundamentally probabilistic and approximate, it has been proved that it would provide the ability of processing large volume of data flow with adjustable capacity and accuracy [6].

II. Serial Solution

We describe the serial solution by first specifying the input and output of the PBF. For the input, in order to construct the PBF, we need the number of hash functions, which partially decide the executing time and accuracy of the PBF. The larger k is, the more accurate the PBF is, but more time the PBF will spend. Next, for the input keys, we need two data set, all the keys and all the distinct keys, for insert and query purposes accordingly, and the number of total keys and number of distinct keys, hence the average frequency could be calculated. In addition, in order to calculate the accuracy of the PBF, we also keep record of the ground truth of frequency of all keys when generating testing data set. In terms of output, the PBF will yield the estimated frequency of each key, and we compare the estimated value with the ground truth, and compute the relative error for each key.

We present the general algorithms of INSERTION and QUERY operations of PBF here in the form of a serial solution. Corresponding parallel implementation of these algorithms will be discussed in details in the following sections. The pseudo-codes are in courtesy of Yao et al. [6].

Algorithm 1 The PBF Insert Algorithm

```
1: procedure INSERT($x$)                          ▷ Insert operation
2:     for $j = 1 \to k$ do
3:         $i \leftarrow h_j(x)$
4:         $random_i \leftarrow Uniform(0, 1)$
5:         if $random_i < p$ then
6:             $B_i \leftarrow 1$
7:         end if
8:     end for
9: end procedure
```

Algorithm 2 The PBF Frequency Query Algorithm

```
1: procedure FREQUENCY($x$)                       ▷ Frequency test operation
   $counter \leftarrow 0$
2:     for $j = 1 \to k$ do
3:         $i \leftarrow h_j(x)$
4:         if $B_i == 1$ then
5:             $counter++$
6:         end if
7:     end for
8:     $f \leftarrow estimation(counter)$
9:     return $f$
10: end procedure
```

The pseudo-code in Algorithm 1 gives a gist of the INSERTION operation of PBF. To initiate the algorithm, we need parameters (*m*, *k*, *p*) which are the size of PBF array, number of hash functions and pre-set probability of flip action, respectively. The flow of the calculation centers on the single for-loop which loops over the hash function index *j*. For each *j*, the hashed position, *i*, of element x and a random number (evenly distributed in [0, 1]) in accord with this position are obtained by $h_j(x)$ and function Unifrom(), respectively. If the random number (or the flip probability) is smaller than the pre-set value *p*, we set the bit B_i. If not, continue. A full scan of the whole data set will be carried out by the out-loop over each element in the data set. There will be *n* reads of data elements from the memory and a certain writes among the *m*-bit PBF array. The most intensive part of computation comes from hashing values of positions and random number generation.

The pseudo-code of the PBF QUERY operation, as shown in Algorithm 2, is structurally similar to that of the INSERTION operation. After initialization of a counter, the for-loop loops over the hash function index *j*, its purpose is simply to find the appearance count of the element x in the PBF array according to its position *i*. There will be *j* reads of bit information at various positions of the *m*-bit PBF array and a certain writes to update the counter. Like in INSERTION, the most intensive part of computation comes from hashing values of positions.

As a side note, we should mention that there is a third algorithm in the design of PBF which serves as an initializer to determine parameters (*m*, *k*, *p*) out of the total number of elements in the data set, *n*, and a threshold frequency, *f*, marking the heavy-hitter elements in a large stream data flow. Since they are more related to accuracy rather than time latency, so we don't adjust those parameters in all the implementations, we just calculate them based on number of hash functions, average frequency and the number of all keys according to same equations in all implementations.

III. Related Work

In general, there are parallelized algorithms designed to implement various BFs [7] [8] [9]. Costa et al. [7] explored the potential of using GPU to accelerate the traditional BF operations. Their implementation was, however, lack of optimizations generally seen in GPU calculation. In later works [8] [9], many more GPU-specific optimizations have been incorporated into the solution to release more power from GPU. For example, Ma et al. have been using both memory coalescing in off-chip memory access and privatization of BF array in the on-chip shared memory. These works are, however, not directly related to the PBF implementation under discussion. Even though PBF shares some common features in the algorithmic structure as other BF designs, its unique introduction of the probabilistic determination of the flip action distinguishes itself from the others. A direct consequence, for example, is that there is no need to access the *m*-bit PBF array prior to the decision of flip action because it is determined by a simple comparison of calculated probability to the pre-set value. We will be discussing more of these in the later sections.

IV. Parallelization Strategy

We describe how to implement the PBF on GPU with the following procedures. We first discuss how to organize the dimensions of the block and grid, followed by detailed implementation to fully utilize the parallel computation ability of GPU. In addition, we explain the rationale behind our design choice. Finally, we describe how data are stored and accessed in our preliminary version.

A. Thread/Block Dimension

When setting the dimensions of thread block and the grid, we consider two parameters, the number of hash function and the total number of keys inserted and/or queried. First, for the dimension of thread block, the x-dimension is set to be the number of hash functions rounded to multiples of 32, which is the warp size. For the y-dimension, the value is set to be the maximum number of threads in one thread block divided by the x-dimension. Also the y-dimension value is the number of keys processed in each thread block. Second, for the dimension of the grid, we calculate the number of blocks needed by dividing the total number of keys over the number of keys in each thread block.

However, when the number of hash function is greater than 1024, which is the maximum number of threads in one block, we adapt the thread coarsening technique, which means that each thread calculate multiple hash functions to make sure that one thread block could finish processing at least one key. This means that thread blocks perform data coarsening on 2-dimensions, horizontally and vertically, until covering the work structure, Figure 3.

B. Algorithm Implementation

There are two operations, including insert and query operation of the PBF. The two operations have a common phase, which is to calculate the k positions in the PBF using hash functions. We assign one key to one row of threads, where each thread is associate with one hash function. In the insert operation, after k positions have been determined, each thread will generate a random number, range from 0 to 1, and then decide whether to flip the corresponding bit from 0 to 1 by comparing the random number to the preset probability. In the query operation, on the other hand, we use an atomic operation to add the number of bits which have already been set to 1 out of k positions.

Note there is another way to design the way that how to assign each thread with keys and hash functions, which is shown in following figure, and we choose the first design option, whose advantages include that all threads in one warp access the same data, so the data get broadcast among all threads, and all threads access different positions in the PBF, hence there is no race condition.

Additionally, there are several disadvantages of the second option. First, although with very low probability (mainly depends on the performance of hash function and the size of the PBF), two threads in the same warp might access and update the same positions in the PBF, which causes race condition. Secondly, the consecutive threads (the threads in one warp) will have much higher probability of control divergence, which means different keys will have different frequency, hence the threads in one warp will higher probability to have different choices, either flip the bit or not. Finally, as mentioned before, the value of k is fairly large, hundred even thousands, hence the n is pretty small, it's more difficult to round n to multiples of 32, since we want to make sure $n*k \leq 1024$. Therefore, the second option will result in more thread padding overhead.

Thread Block
Hash 1
Hash 2
Hash k

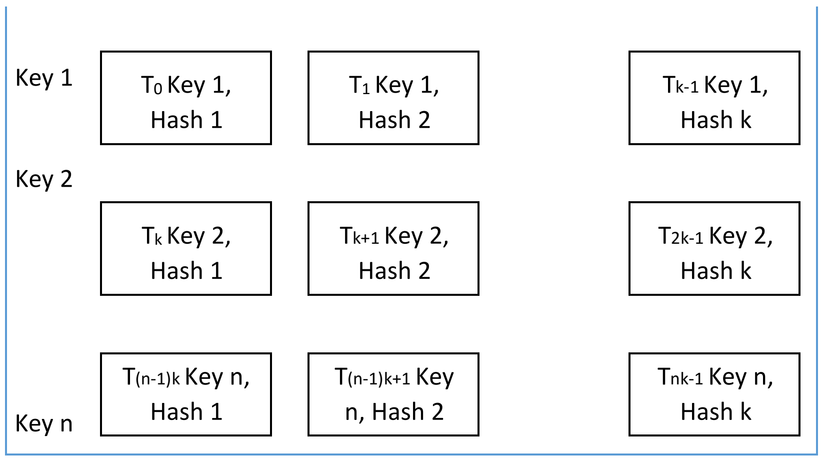

Figure 3. Algorithm design option 1

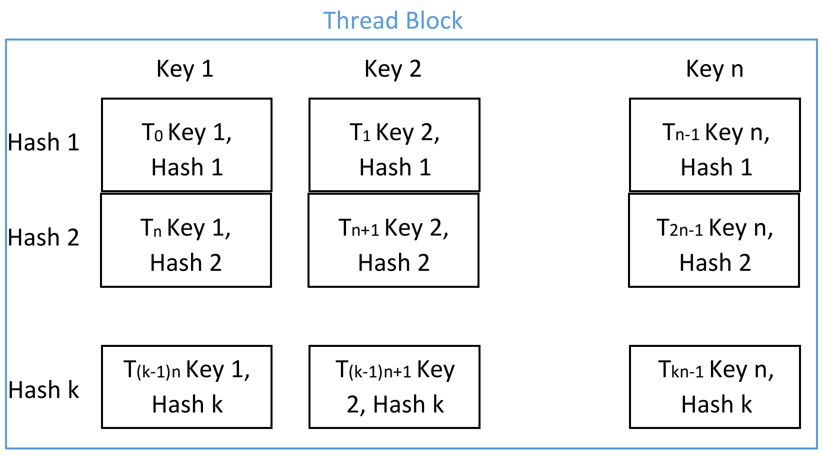

Figure 4. Algorithm design option 2

C. Data Placement

There are three types of data needed to store, the PBF bit vector, the keys and the results in the query operation. For the preliminary implementation, we put everything in global memory.

In terms of PBF vector, in the original design, we use one bit to represent one position, which uses least amount of memory. However, there are race conditions since multiple threads might access and update the same data point. One alternative way is to use one character to present one bit in the PBF, which will reduce the race condition but use eight times of memory of the original design. In addition, since the character is eightbit data type, there might be bank conflicts. Another options is to use integer to present one bit in the

PBF, which eliminates the bank conflicts, but use most of memory. In the evaluation section, we implement two different designs and compare their performance.

To store all the keys, first we concatenate all the keys into single key, and put all the starting points into a look up array. For the preliminary version, all threads access the keys from the global memory directly, and in our optimized version, we use shared memory for locality. For the query result array, since each thread block process at least for one key, we use the shared memory atomic operation to store the result first, and after traversing the whole PBF, we then result write the result into the global memory.

V. Parallel Optimizations

In terms of optimization, we mainly focus on using shared memory for locality and privatization, loading both the keys and the PBF vector from global memory to shared memory. Since the size of shared memory is much smaller than that of global memory, and the PBF is a relatively memory intense application, the design is more complex than what it seems like.

For the keys, the major concern is that whether shared memory is enough to store the keys. Due to the parameter setting and algorithm design, the number of keys processed by one block is equal to the maximum number of threads divided by the number of hash functions. In real applications, for accuracy purpose, the number of hash functions is large, hence the number of keys processed by one thread block is limited. For example, the minimum value of *k* is 200, hence *n* should be 4, due to that x-dimension need to be rounded to multiples of 32, which is 224, and suppose the maximum number of threads in each block is 1024. Another issue is that each key is an array of characters, and the length of each key may vary. In our testing data, the maximum length of all keys is 50. Hence the maximum size of all keys processed by one thread block is 200 bytes, and it could be fit into shared memory.

Table 1. Different parameter setting of the PBF

	f	N	k	p	m	Memory in bit (bytes)	Memory in char (bytes)	Memory in int (bytes)
Case 1	50	1000	200	0.0308	58466	7309	58466	233864
Case 2	100	10000	500	0.0153	726078	90760	726078	2904312

On the other hand, the size of the PBF vector is large, and some example cases are shown in the following table. According to the table, shared memory may not be large enough to store all the PBF vector even if we use the bitwise design. Our solution is three steps of processing. In the insert operation, we first calculate all the positions of all the keys in one thread block, and write the index of those positions where the corresponding bits are decided to be flipped into the position array. Second, we sort the positon array. Finally, we load and update the PBF vector by segment from global memory to shared memory. In the query operation, different from the insert operation, we maintain separate position array for each key in one thread block, and after sorting, we use atomic operation to add up the number of bits that have been set to 1.

VI. Evaluation

For evaluation, we compare the implementation of CPU, GPU preliminary version and GPU optimized version focusing on the time comparison and the accuracy of the PBF. By accuracy we refer to the how close the number of occurrences for each element in the set is from the actual count of occurrences. Intuitively, the higher the frequency of an item, the more 1s in the *k* bits corresponding to that item. The query operation of a particular item consists of counting the number of 1s in the *k* bits, denoted as *y*, and then calculating the frequency, *f*, as follows,

$$y \square \square f \square \frac{k \square n \square p \square m \square \ln \square \square 1 \square \overline{}}{\square k \square \quad p \square (k \square m)}$$

The frequency value provides a mechanism to select items based on a frequency threshold. The PBF provides an approximate solution since it uses probabilistic counting operations, nevertheless, its storage footprint compensates for obtaining less accuracy.

A. Evaluation setup

In terms of parameters to tune the evaluation, we choose the parameter that would affect the algorithm design and performance, which is the number of hash functions. In addition, we use different sets of input data set, which will be discussed in next section. For the comparison metric, we consider time latency as the most important one. Also, in order to demonstrate the implementations are correct, we check the cumulative distribution function (CDF) of the relative errors of all keys in the data set. Note the since the PBF is essentially a probabilistic approach, even two sets of results from the same input data set and the same parameters setting are different, hence we use the CDF of the relative errors.

Table 2. Hardware specifications of UTK Hydra (Dell OptiPlex 9010)

GeForce GT640	**Intel Core i7-3770**
CUDA cores: 384	CPU cores: 4 (HT x2)
Clock rate: 900 MHz	Clock rate: 3.4 GHz
Global Mem: 1 GB	L2 cache: 16 MB
Max BW: 28.5 GB/s	Max BW: 25.6 GB/s

B. Testing Data Generation

For the test data, the basic idea is that we first generate random strings, which are considered as keys, and then assign each key with a specific frequency value. To configure the way how we generate the keys, the first parameter is the type of the distribution of the frequency value. Four types of distributions were considered: uniform, random, Gaussian, and Poisson. The idea is to test the accuracy of the PBF for such data inputs which make a representable dataset of real-world scenarios. Each element of a sequence is generated using a random number generator with uniform distribution. Our random sequence generator can be parameterized based on the number of unique keys, size of the keys, seed values for RNG, scaling factor of probability distribution resulting values, and other that are dependent on the distribution chosen.

For uniform distribution, we considered the simplest case in which all the keys have exactly the same number of occurrences. Figure 5 shows a sample dataset consisting of 100,000 total sequences with uniform distribution. In the standard normal (Gaussian) distribution, the random sequence generator allows setting the mean, standard deviation, and the range values for producing the resulting sequence. Figure 6 shows a sample Gaussian dataset of approximately 1 million with sequences with mean = 0 and standard deviation = 1. The Poisson distribution is presented in Figure 7 and can also be configured with a specified mean. The last distribution supported is random, see Figure 8, which basically produces a non-uniform random number of occurrences for the keys being generated. This input dataset can be used to simulated applications where the data manipulated is randomized, such as web traffic.

Total sequences:
100,000

Distinct
sequences:
100

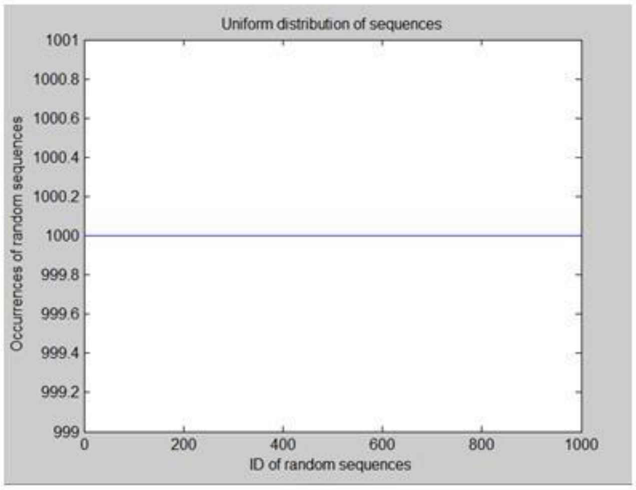

Figure 5. Generated random sequences with constant distribution

Total sequences:
999,836

Distinct
sequences:
20000

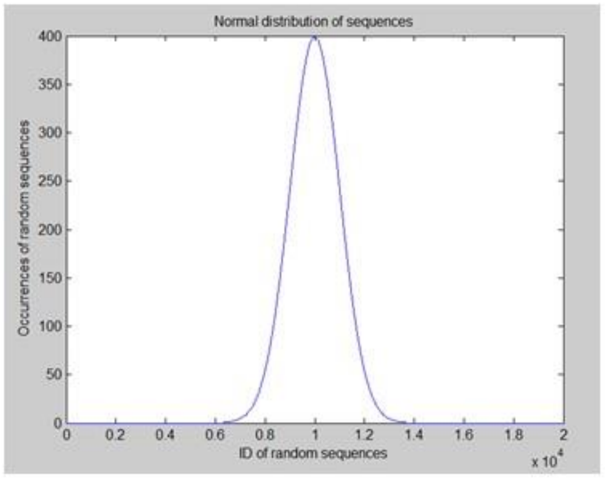

Figure 6. Generated random sequences with normal distribution

Total sequences:
7,547

Distinct
sequences:
200

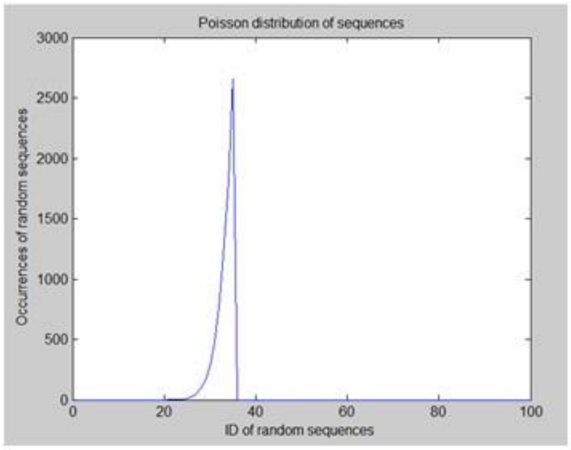

Figure 7. Generated random sequences with Poisson distribution

Total sequences:
49,748

Distinct
sequences:
1000

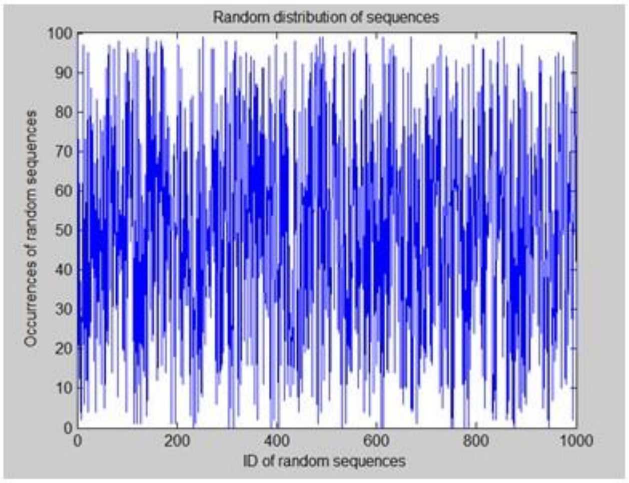

Figure 8. Generated random sequences with random distribution

C. Evaluation results and analysis

Evaluation of the PBF algorithm consisted of 4 implementations:

- CPU – serial version which served as the baseline solution.
- CPU OpenMP – includes parallel-for directives for parallelizing the hashing computations
- GPU preliminary version – baseline GPU version, uses global memory and is constrained by the number of hashes. The grid consists of a one dimensional array of thread blocks mapped to the key-hash matrix (Figure 3).

- GPU optimized version 1 – decomposes the thread blocks into 2-D to match the key-hash matrix. The thread blocks perform a blocking scheme along the hashes axis, then moves downward to a new set of keys. The grid

We want to make notice that during the implementation of the PBF on the GPU, several issues came up with running tests successfully, thus limiting our results. We provided several scripts to attempt to run the programs. We will continue working on these issues after the course terminates since we found that other methods should provide additional efficiency. One such aspect is to pad keys to multiples of 32 bits in order to minimize bank conflicts and control divergence, improved shared memory tiling for nonuniform key sizes, and performing a gather approach for small datasets and large bloom filters.

Figure 9 presents some preliminary results on the runtime required to perform the insert operation for the CPU and GPU implementations for each type of probability distribution. The CPU OpenMP version seemed to perform very poorly, even though we verified that directives were being used correctly, as well as specifying the shared and private variables. The plot shows that the CPU performed better that the GPU baseline version. We suspect this has to do with the fact that the PBF algorithm is data-intensive algorithm, which in these implementations make efficient use of CPU caches. Meanwhile, the GPU preliminary version exhibited additional contentions and delays. For the GPU optimized version, we implemented a blocking scheme in addition to privatization of the current key being hashed. Attempts to use shared memory correctly were unsuccessful but we have discussed new ideas that need to be tested.

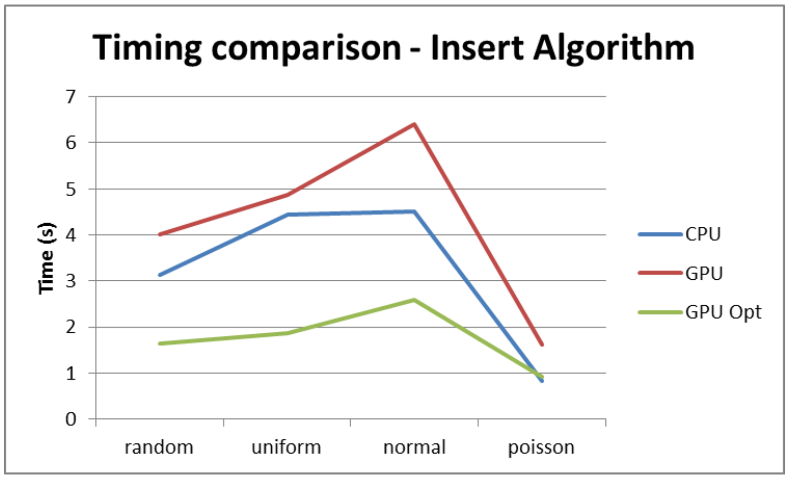

Figure 9. Performance comparison of CPU/GPU for different dataset distributions

The main goal of the PBF is to identify frequent sequences, also called “heavyhitters”. The idea is not to compute an exact frequency value but a fairly appropriate estimate. To validate the accuracy of the PBF algorithm we present CPU and GPU results for uniform and Gaussian distribution in Figures 10 and 11, respectively. In Figure 10, the plot on the left represents the estimated frequency for each input sequence and overdrawn on the plot is the constant frequency of each item (1000). At simple glance one can see that the estimates are valid and this is confirmed with the relative error of the estimated frequencies which is shown in the plot to the right. For this CPU run, the error is very close to zero for all random sequences.

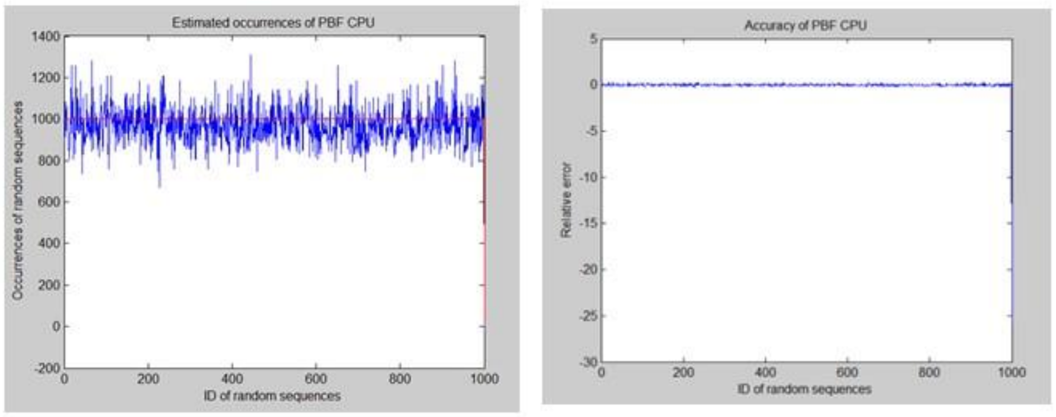

Figure 10. PBF accuracy in CPU version for uniform distribution

A similar experiment was performed for the GPU optimized version but using a dataset with Gaussian distribution, see Figure 11. The estimated frequency of each PBF item resembles closely the standard normal (mean = 0, stddev = 1) distribution. The plot to the right shows the relative error. Notice that for the sequences that are confined in a specific range, the relative error is close to none. Nevertheless, the sequences with very low occurrences result in a higher relative error. The results of such plots depend heavily on the PBF parameters (probability threshold) and the input dataset parameters.

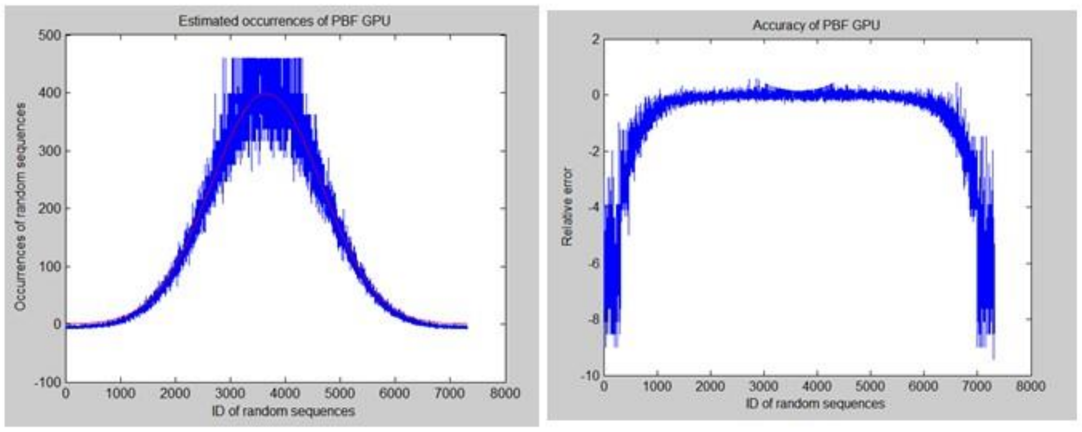

Figure 11. PBF accuracy of GPU version for Gaussian distribution

VII. Conclusions

Although a PBF is not a very complex data structure to use, its accuracy is application dependent and careful considerations must be taken when establishing design parameters. The GPU PBF project allows the authors to understand and compare tradeoffs of common techniques used to manipulate and manage large datasets. Learning to use variations of bloom filters is a handy tool that may serve as building block for novel ideas. Additionally, implementing a GPU version provides technical experience with recent many-core processors, as well as parallel programming skills. In turn, the GPU framework will be used to put into practice optimization techniques taught in the course, such as, coarsening, tiling, reduction, data layout/format, coalesced accesses, synchronization, and others.

VI. References

[1] B. H. Bloom. "Space/time trade-offs in hash coding with allowableerrors." Commun. ACM, vol. 13, no. 7, pp. 422 – 426, Jul. 1970.

[2] A. Z. Broder and M. Mitzenmacher, “Network Applications of Bloom Filters: A Survey,” Internet Mathematics, vol. 1, no. 4, 2003.

[3] S. Tarkoma, C.E. Rothenberg and E. Lagerspetz, “Theory and Practice of Bloom Filters for Distributed Systems”, IEEE Comm. Surveys and Tutorials, vol. 14, no. 1, pp. 131 – 155, first quarter 2012.

[4] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, “Summary cache: a scalable widearea web cache sharing protocol,” IEEE/ACM Trans. Netw., vol. 8, no. 3, pp. 281 – 293, Jun. 2000.

[5] A. Broder and M. Mitzenmacher, “Network Applications of Bloom Filters: A Survey,” Internet Mathematics, vol. 1, no. 4, pp. 485 – 509, 2003.

[6] Y. Yao, S. Xiong, J. Liao, M. Berry, H. Qi, Q. Cao. "IdentifyingFrequent Flows in Large Traffic Sets through Probabilistic Bloom Filters." Under review.

[7] L. B. Costa, S. Al-Kiswany, and M. Ripeanu, "GPU support for batch oriented workload," in IEEE 28th IntI. Perf Compo and Comm. Conj, pp. 231 – 238, USA, December 2009.

[8] Lin Ma; Chamberlain, R.D.; Buhler, J.D.; Franklin, M.A. "Bloom Filter Performance on Graphics Engines", Parallel Processing (ICPP), 2011 International Conference on, On page(s): 522 – 531.

[9] WenMei Ong; Baskaran, V.M.; Poh Kit Chong; Ettikan, K.K.; Keh Kok Yong "A parallel bloom filter string searching algorithm on a many-core processor", Open Systems (ICOS), 2013 IEEE Conference on, On page(s): 1 – 6.