A Parallel Bloom Filter String Searching Algorithm on a Many-core Processor

WenMei Ong, Vishnu Monn Baskaran
Faculty of Engineering,
Multimedia University,
Cyberjaya campus, Selangor, Malaysia.
wenmei15@gmail.com,
vishnu.monn@mmu.edu.my

Poh Kit Chong
Faculty of Engineering and Science,
University Tunku Abdul Rahman,
Petaling Jaya, Selangor, Malaysia.
pkchong@utar.edu.my

Ettikan K. K, Keh Kok Yong
Accelerative Technology Lab,
MIMOS Berhad,
Kuala Lumpur, Malaysia.
ettikan.karuppiah@mimos.my
kk.yong@mimos.my

Abstract - **This paper analyzes the underlying architecture of a serial Bloom filter string searching algorithm to identify the performance impact of this algorithm for large datasets. Then, a parallel multi-core driven Bloom filter algorithm using software application threads is studied as benchmark. Experimental results suggest that for a set of 10 million strings, this algorithm exhibits speedups of up to 3.3× against a serial Bloom filter algorithm, when using an 8-logical processor multi-core architecture. To further improve the speedup, a many-core driven parallel Bloom filter algorithm is proposed using the Compute Unified Device Architecture (CUDA) parallel computing platform. The proposed algorithm segments the string list into blocks of words and threads in generating the bit table for the string searching process, which maximizes computational performance and sustains consistent string searching results. Experimental results suggest that the proposed algorithm extends the speedup to 5.5× against a serial Bloom filter algorithm, when using a 256-core CUDA graphics processing unit architecture.**

***Keywords* — Bloom filter, string searching, parallel computing, CUDA, GPGPU**

I. INTRODUCTION

String searching algorithms represent a fundamental method of finding an element or a set of elements with specific properties among a collection of elements. Since the advent of the computing and digital storage technology, the capacity of storing substantial amount of data, from which individual records are to be retrieved, identified or located have paved the way in the design and development of various string searching algorithms.

Rudimentary linear based string searching algorithms were initially applied due to its simplicity in design at the expense of poor searching performance for large datasets. Binary string searching algorithms substantially improved searching performance, albeit requiring the dataset to be sorted, which impacts its overall efficiency [1]. Hash tables were later introduced using associative arrays to map as a structure to maps keys to data set values, which prove to be a good choice when the keys are sparsely distributed [2]. It is also chosen when the primary concern is the speed of access to the data. However, sufficient memory is needed to store the hash table. This lead to limitations in the datasets as a larger hash table is required to cater for a larger dataset.

One approach to address the limitations of the hash table would be the utilization of the Bloom filter string searching algorithm as a memory-efficient data structure, which rapidly indicates whether an element is present in a dataset [3]. In spite of improved speed and efficiency, a serial Bloom filter design, which runs on a single logical processor, would still incur longer processing time for substantial increases in dataset size.

The advancement of computing technology has seen significant developments of the multi-core processing technology in line with the progression of parallel computing solutions. This in turn has also fueled the development of the many-core processing technology, such as the incorporation of generic stream processing units into a Graphics Processing Unit (GPU), allowing generalized computing devices with large numbers of processing cores and hence the term, General purpose computing on GPU (GPGPU). The deployment of the Compute Unified Device Architecture (CUDA) parallel computing platform by NVIDIA for GPGPUs has further increased adaptation of parallel computing solutions using a many-core architecture. A GPU offers high performance throughput with little overhead between the threads. By performing uniform operations on independent data, massive data parallelism in an application can be achieved.

However, a serial Bloom filter algorithm would not be able to leverage on the advancements of both the multi-core and many-core architectures for improved searching performance for large datasets. Hence, this paper investigates the impact of a serial Bloom filter algorithm for large datasets, in justifying the need for a parallel architecture in accelerating the string searching process. In achieving an effective parallel string searching design, a parallel Bloom filter algorithm using software application threads for a multi-core architecture is first investigated as a benchmark, which exhibits improved performance speedups against a serial Bloom filter. In further improving the speedup, a parallel Bloom filter algorithm using the CUDA parallel computing platform is proposed, to support batch-oriented string search operation in a data-intensive high performance system. The proposed algorithm segments the string list into blocks of words and threads in generating the bit table, which is used during the string searching process. This method maximizes the computational performance and sustains consistent string searching results.

978-1-4799-0285-9/13/$31.00 ©2013 IEEE

An existing CUDA implementation of the Bloom filter string searching algorithm by Costa et al. [4] limits the false positive probability (fpp) to 10^{-4} and the number of strings tested for both the insertion and searching processes to 1 million. In this paper however, the proposed parallel Bloom filter algorithm using CUDA removes the limit on fpp, and instead, computes the fpp based on the number of strings involved in the insertion process. In addition, the number of strings tested is increased to 10 million for a more accurate representation in performance speedup assessment.

The paper is organized as follows. Section II investigates the performance impact of a serial Bloom filter string searching algorithm. Section III describes the parallel Bloom filter algorithm on a multi-core architecture. Section IV proposes the parallel Bloom filter algorithm using the CUDA parallel computing platform on many-core architecture. Section V concludes this paper.

II. PERFORMANCE IMPACT OF A SERIAL BLOOM FILTER SEARCHING ALGORITHM

A. The Bloom Filter String Searching Algorithm

The Bloom filter data structure is defined as a set of $\mathbb{M}$ bit table array, with $m = |\mathbb{M}|$ and all array elements initially set to 0. Note that k different hash functions are defined; each will map or hash the set elements to one of the $\mathbb{M}$ array positions [5]. The hash functions must give uniform random distributed results, such that the array can be filled up uniformly and randomly [6].

To add or insert an element into the filter for a set of strings, $\mathbb{N}$ with $n = |\mathbf{N}|$, the element is first passed to every k hash function to generate k indices of a Bloom filter array. The corresponding bits in the array are then set to 1. To check for the presence of an element in the set, or query for it, the element is passed to every hash function to get indices of the array. In the event any of the bits at these array positions is found to be 0, then the element is confirmed as not in the set. If all of the bits at these positions are set to 1, then either the element is indeed present in the set, or the bits have been set to 1 by chance when other elements are inserted, hence causing a false positive [7].

Assuming that the hash functions used are independent and uniformly distributed, the array position will be selected with equal probability. The probability of a bit not set to 1 after inserting n string elements into a filter array of size m bits using k hash functions is [4][8]:

$$p_{not\ set} = \left(1 - 1/m\right)^{kn} \tag{1}$$

When a false positive occurs, all the corresponding bits in the array are found to be 1; the algorithm erroneously claims that the element is in the set. Hence, the false positive probability (*fpp*) of all corresponding bits of an element being set to 1 after inserting n elements into a filter array of size m bits using k hash functions is given as [4]:

$$fpp = \left[1 - \left(1 - 1/m\right)^{kn}\right]^k \approx \left(1 - e^{-kn/m}\right)^k \tag{2}$$

From Eq. (2), as m decreases:

$$fpp \approx \lim_{m \to 0} \left(1 - 1/e^{kn/m}\right)^k \cong 1 \tag{3}$$

and as n increases:

$$fpp \approx \lim_{n \to \infty} \left(1 - 1/e^{kn/m}\right)^k \cong 1 \tag{4}$$

Equations (3) and (4) show that the *fpp* increases when m decreases or when n increases, respectively. A large dataset indicates a large n value. To minimize the *fpp*, m would need to be set to a large value (i.e., using a large Bloom filter), which in turn increases the requirements of memory capacity. Trade-off between *fpp* and memory space is thus required.

B. Implementation and Performance Assessment

A serial implementation of the Bloom filter string searching algorithm was developed using Visual C++ as a complete software application. The computing hardware specifications were configured by utilizing an Intel Core i7-2700K processor at 3.50 GHz operating frequency, with 8-logical processors and 8GBytes of system memory. The *fpp* is initially set to $1/n$ with k = 500 and m is computed such that:

$$m = -kn / log(1 - fpp^{(1/k)}) \tag{5}$$

The implementation of the serial Bloom filter consist of the insertion and searching processes as illustrated in Fig. 1(a) and Fig. 1(b) respectively in line with a custom hash algorithm (i.e., *APHash*) developed by Partow [9]. The *APHash* algorithm is based on a hybrid rotative and additive hash function implementation, with an aim to produce the least amount of collisions for a set of data. To obtain several hash functions from the same hash algorithm, an array termed **`salt`** is utilized, whereby each element represents an initial hash value. As illustrated in Fig. 1 (a), to insert a string into the filter, hash functions are performed on the string and the corresponding bits are set. All strings in the input word file are inserted sequentially. Fig. 1(b) shows the algorithm for searching the filter. To check the existence of each query, the same hash functions are performed on the query and the corresponding bits of the Bloom filter are checked. If any of the bits is not set, the query does not exist in the filter.

An experiment was carried out to analyze the performance of the serial Bloom filter algorithm by varying the value of n, between 1000 and 10,000,000 strings. The length of each string in the word-list and query-list is fixed not to exceed 20 characters. A set of 500 hashes (i.e., k = 500) are performed on each string. Both the filter insertion and searching processes are executed and repeated for five consecutive runs, in which an average operation time is then recorded.

Fig. 2 illustrates the average total process time of both the serial filter insertion and searching processes for an increasing value of n. Results from Fig. 2 suggest that as n increases, an increase in the processing time for the serial Bloom filter algorithm is observed. The required processing time reached a high of 360 seconds for n = 10,000,000. In addition, CPU

utilization registered a 12.5% usage, constituting a meager $1/8^{th}$ usage of the available logical processors, which is in line with serial behavior of this algorithm.

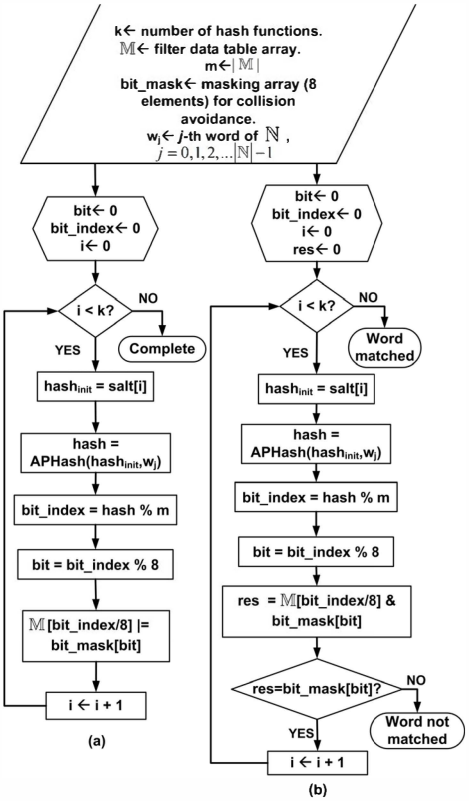

Figure 1. Bloom filter insertion (a) and searching (b) process for one word [9].

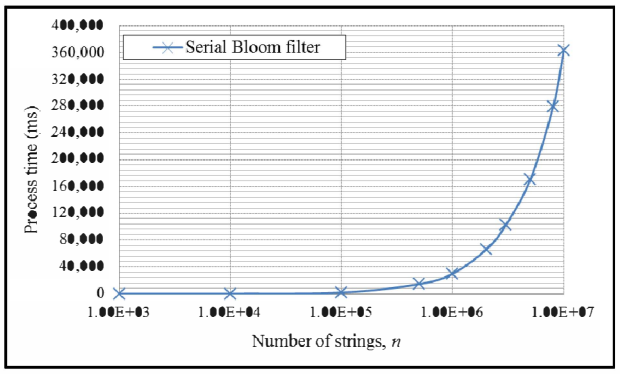

Figure 2. Performance assessment of a serial implementation of the Bloom filter algorithm (inserts and search) for an increasing *n*.

As *n* increases by 1, the total number of hash functions to be performed increases by a factor of *k*, which in turn increases the computational load. In order to reduce the execution time of the algorithm, a more time-efficient programming paradigm is needed. The repetitive process of performing hash functions on different strings suggests that parallelization may be applied. Bernstein's Conditions were examined and have proven that the data has no dependency on each other, confirming that the hashing algorithm for different strings can be executed concurrently. The following section discusses the implementation of a parallel Bloom filter algorithm on a multicore processor.

III. PARALLEL BLOOM FILTER ON A MULTICORE ARCHITECTURE

To address the performance impact of the serial Bloom filter algorithm for a large *n*, a parallel multi-core driven Bloom filter algorithm is implemented here using the OpenMP application programming interface. The program flow for this parallel algorithm is generally similar to the serial Bloom filter algorithm, with the exception on the workload distribution for the insertion and searching processes. Both these processes involve performing similar set of tasks on different datasets, providing an opportunity for data parallelism.

Fig. 3 illustrates the execution model of the parallel Bloom filter insertion process using OpenMP. Here, a set of software application threa0ds, $\mathbb{T}$ are created to concurrently execute the insertion process based on a $n/|\mathbb{T}|$ workload distribution. $|\mathbb{T}|$ equals the number of available logical processors on a multicore architecture. An atomic operation is applied when updating the bit-table array, $\mathbb{M}$ to minimize simultaneous writes on the same $\mathbb{M}$ array location pointed by different application threads.

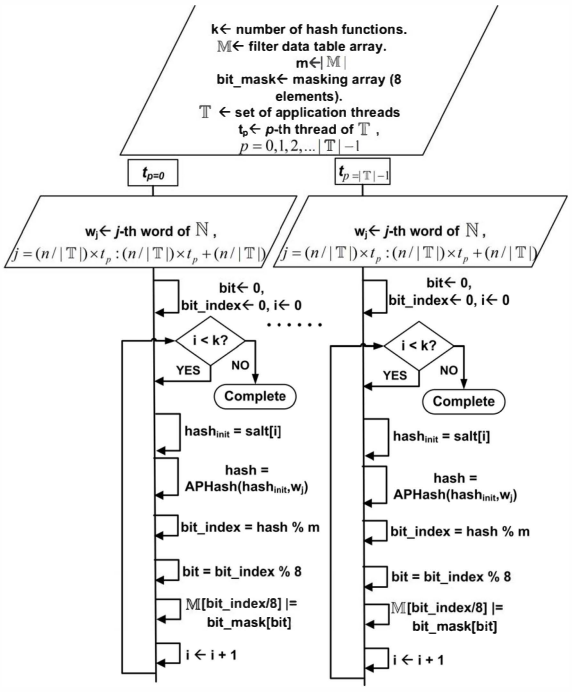

Figure 3. Parallel Bloom filter insertion on a multicore architecture.

Fig. 4 illustrates the execution model of the parallel Bloom filter searching process using OpenMP. The algorithm for the parallel Bloom filter searching algorithm in Figure 5 adopts a similar implementation to that of the serial Bloom filter searching as discussed in the aforementioned section. The atomic operation as applied in the insertion process is not required here as the application threads are only reading the bit table, $\mathbb{M}$ value from memory, thus leaving the $\mathbb{M}$ memory content unchanged. Similar to the parallel filter insertion process, $\mathbb{T}$ application threads concurrently execute the searching process based on a $n/|\mathbb{T}|$ workload distribution, with results of the search operation being stored in memory.

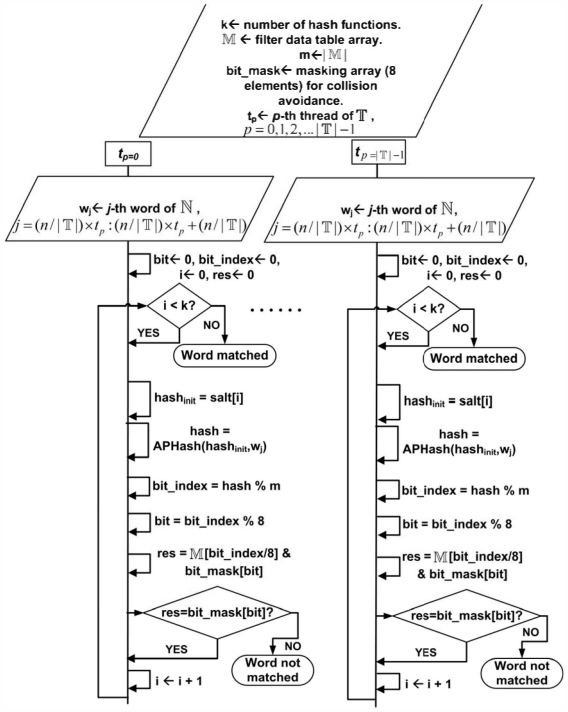

Figure 4. Parallel Bloom filter searching on a multicore architecture.

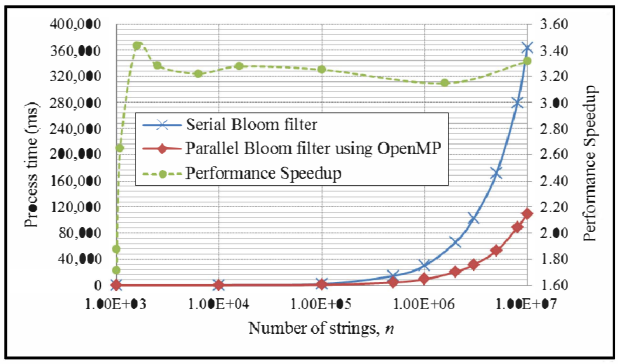

Figure 5. Performance analysis of a parallel Bloom filter algorithm on a multicore architecture against a serial Bloom filter for an increasing *n*.

Replicating the test setup in Section 2, a performance analysis was carried out to evaluate the effectiveness of the multi-core parallel Bloom filter algorithm, in comparison with the serial Bloom filter algorithm. Fig. 5 illustrates the average operation time (i.e., for both insertion and searching) of the parallel Bloom filter algorithm implemented using OpenMP with $|\mathbb{T}|=8$ in comparison with the serial Bloom filter algorithm. In addition, Fig. 5 also computes the performance speedup to analyze the scale of operation time improvement of the multi-core parallel Bloom filter algorithm against the serial Bloom filter algorithm (green circle). Results from Fig. 5 suggest marked speedups of the multi-core parallel Bloom filter algorithm against the serial Bloom filter algorithm. A notable improvement in computational performance is visible for n = 10,000,000 with the multi-core parallel algorithm registering a 109 second computational time against a 360 second computational time by the serial algorithm, which in turn yields a 3.3× in performance speedup. In addition, a steep increase in performance speedup is initially observed as n increases from 1000 to 100,000 strings, which then exhibits a consistent speedup as n > 500,000 strings. The consistent speedup as n > 500,000 is deduced as a result of logical processor (i.e., $|\mathbb{T}|=8$) saturation in processing the large values of n. Further speedups are attainable if $|\mathbb{T}|$ increases for an increase in the number of available logical processors. This puts forward the concept of leveraging on the many-core compute capabilities of a GPU processor for general purpose computing, which will explored in the following section.

IV. PARALLEL BLOOM FILTER USING CUDA ON A MANY CORE ARCHITECTURE

In an effort to further improve the performance speedup of the parallel Bloom filter string searching algorithm, this section proposes a parallel Bloom filter algorithm design by utilizing the CUDA parallel computing platform on a GPU processor. By using the CUDA parallel computing platform, application threads executed in a GPU are organized in blocks and a block is executed by a streaming multiprocessor unit.

By definition, a parallel application developed using CUDA contains a heterogeneous architecture, whereby serial portions of an algorithm are executed on the host (i.e., CPU) whereas the parallel portion of the application is executed on the device (i.e., GPU) as a kernel [10]. The kernel is executed as an array of threads, in parallel. The host and device memories are separate entities with no coherence [11]. As such, explicit data transfer from the CPU memory to the GPU memory is required. This is done by allocating memory on the device and then transferring data from the host memory to the allocated device memory. After device execution, resultant data is transferred back from the device to the host memory for results tabulation and storage. When the size of the data transferred is large, the blocks of threads are queued for execution; leading to a long execution time which might exceed the run time limit. As a result, data processing has to be performed in batches [4]. A fixed batch size of 50,000 strings is chosen to be read from host memory into device memory for both filter insertion and searching processes. Smaller batch size is feasible but not recommended as additional constant overhead is generated each time data transfer is invoked. Hence, fewer invocation of copy operation is preferred. With batch processing, values stored in the respective offset tables for word insertion and query searching will be invalid for the subsequent batches of data. A CUDA kernel is launched in order to calculate the offset position of each string in each data batch. The actual offset of each string (of the current batch) would be: the value in the original offset table minus the total length of previously processed strings.

In both filter insertion and searching, each block processes one string. Hence, the number of blocks per kernel invocation is set to 50,000. The string is copied from the GPU global memory into the shared memory to allow faster memory access by threads within the block. Note that k is maintained at 500. Therefore, the number of threads per block is set to 500 such that each thread within the same block performs a different hash function on the same string. The CUDA thread identifier is used to obtain the associated element in the **`salt`** array while the CUDA block identifier is used to obtain the associated string element in the batch.

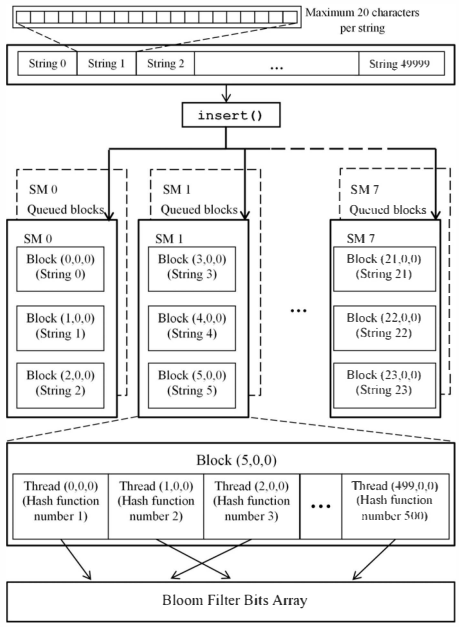

Figure 6. Execution Model for filter insertion (for one batch of data with batch size of 50,000) using CUDA.

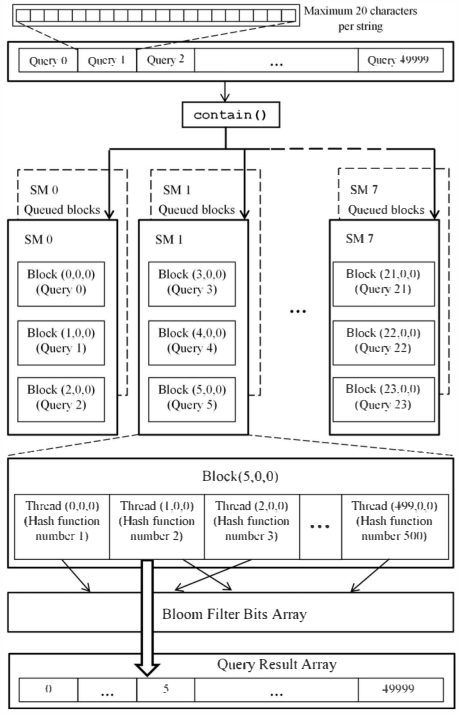

Figure 7. Execution Model for filter search (for one batch of data with batch size of 50,000) using CUDA.

Fig. 6 illustrates the execution model of the Bloom filter insertion process using CUDA. For filter insertion, each thread will set one of the elements in the m bit table array after calculating the associated index. Similar to the multi-core implementation of the Bloom filter algorithm in Section 3, an atomic operation is performed when writing into m. The use of the atomic feature in CUDA minimizes race condition as the writing operation is performed without interference from other threads when writing to different locations of m.

Fig. 7 illustrates the execution model of the Bloom filter searching process using CUDA. An array (in the GPU global memory) stores the result of the filter search function. Prior to invoking the kernel, the entire array is set with a logical **`true`** value. Similar to filter insertion, each block performs searching for one query and each thread within the block performs a hash function on the same query. Each thread also computes the bit-table index and checks the bit-table. If any of the threads within the block finds that the bit-table is not set, the result array at the index position indicated by the block identifier will be set to **`false`**. Atomic operation is not needed for this kernel function because only threads within the same block will write to the same array location and the logical value written is the same (i.e., which is **`false`**) regardless of which thread is writing to it. The results are then copied from the GPU memory to the CPU memory at the end of kernel computation.

Replicating the test setup in the aforementioned sections, a performance analysis was carried out to evaluate the effectiveness of the many-core parallel Bloom filter algorithm using CUDA, in comparison with the serial and multi-core parallel Bloom filter algorithms. The setup also includes an NVIDIA Quadro 4000 GPU processor with 256 cores (each at 950 MHz operating frequency), which are congregated between eight streaming multiprocessors and with 2GBytes of total GPU memory. Table I summarizes device property of the NVIDIA Quadro 4000 GPU processor.

TABLE I. NVIDIA QUADRO 4000 DEVICE PROPERTIES.

Number of streaming multiprocessors (SM)	8
Number of CUDA Cores per SM	32
Number of threads per warp	32
Maximum number of threads per SM	1536
Maximum number of warps per SM	48
Maximum number of blocks per SM	8

By referring to the GPU device property in Table I, the maximum number of resident blocks per SM is limited to eight. However, since each block is launched with 500 threads in line with k = 500, only 3 blocks can reside in an SM due to the maximum limit of resident threads per SM (i.e., 1536 threads), which are in line with design of Fig. 6 and Fig. 7. Since there are 8 SMs for this device, a maximum of 24 blocks are executed concurrently. The rest of the blocks will be queued for subsequent concurrent execution. Threads in each block are grouped into warps of size 32. Hence, there are 16 warps in a block, giving a total of 48 warps in an SM. The SM is thus considered to have achieved maximum occupancy under these conditions. The execution of a warp is scheduled by the warp schedulers in the SM. The execution time for both the filter insertion and search operations includes the time taken for data transfer between the host and the device. Fig. 8 illustrates the average operation time (i.e., for both insertion and searching) of the proposed parallel Bloom filter algorithm implemented using CUDA in comparison with the serial Bloom filter algorithm. In addition, Fig. 8 also highlights the performance speedup (green circle) to analyze the scale of operation time improvement of the parallel Bloom filter algorithm using CUDA against the serial Bloom filter algorithm.

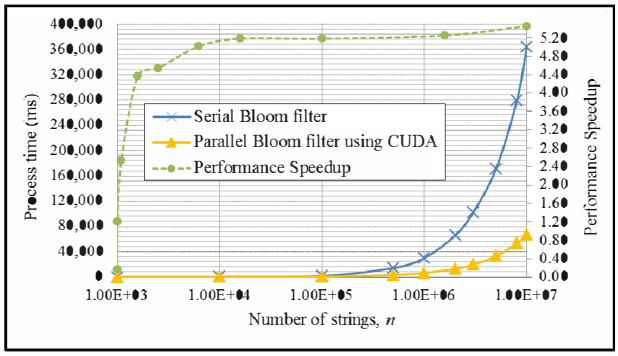

Figure 8. Performance analysis of a parallel Bloom filter algorithm using CUDA against a serial Bloom filter for an increasing *n*.

Results from Fig. 8 also suggest increased speedups of the proposed parallel Bloom filter algorithm using CUDA against the serial Bloom filter algorithm. For n = 10,000,000, the proposed parallel algorithm on CUDA registered a 69 second computational time, which translates into an improved performance speedup of 5.5× against the serial Bloom filter algorithm.

Although the proposed algorithm exploits the use of shared memory within blocks with the purpose of reducing the computational time, the $\mathbb{M}$ bit-table remained in the GPU global memory. This is because updates to the $\mathbb{M}$ bit-table must be visible to all threads. Consequently, the performance of the algorithm was affected due to the high latency on continuous GPU global memory access and in addition to the latency caused by data transfer between host and device memory. In spite of this limitation, as a whole, the batch size concept used in the proposed design did exhibit improved performance speedups against a serial Bloom filter algorithm.

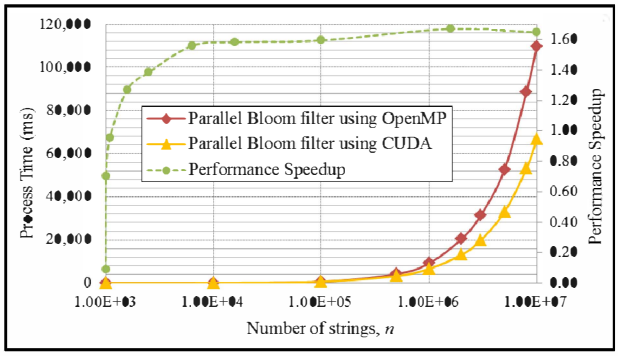

Figure 9. Performance analysis of a parallel Bloom filter algorithm using CUDA against a parallel Bloom filter on a multicore architecture.

To further evaluate the performance between the proposed parallel Bloom filter algorithm using CUDA and the benchmarked multi-core parallel Bloom filter algorithm, Fig. 9 compares the computational performance between these two algorithms. Fig. 9 also computes the performance speedup between the benchmarked and proposed parallel Bloom filter algorithms (green circle). Results from Fig. 9 undoubtedly exhibits noticeable performance improvements of the proposed parallel Bloom filter algorithm using CUDA to that of the multi-core version using OpenMP with a peak performance speedup of 1.65× for n = 10,000,000 strings.

V. CONCLUSION

In this paper, the underlying architecture of a serial Bloom filter was analyzed in identifying the performance impact for large datasets. A parallel multi-core Bloom filter algorithm using software application threads was implemented as benchmark. Despite exhibiting performance speedups of up 3.3× for n = 10,000,000 against a serial Bloom filter algorithm, the limited number of logical processors ($|\mathbb{T}|=8$) on a multi-core architecture constrained the speedup levels. As such, to improve the speedup, this paper proposed a parallel many-core Bloom filter algorithm using the CUDA parallel computing platform based on a batch string processing process. The proposed algorithm improved the performance speedup to 5.5× against a serial Bloom filter algorithm. However, latency in data transfer between host and device memory and the approach to classify $\mathbb{M}$ within the GPU global memory limited the performance speedup. Optimization in the design of data transfer between host and device memory should be given additional attention as future work to further improve the performance speedup.

VI. ACKNOWLEDGMENT

This research was done under Joint Lab of "NVIDIA - HP - MIMOS GPU R&D and Solution Center", which was established October 2012. Funding for the work came from MOSTI, Malaysia.

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms," 3rd ed., McGraw-Hill Education, 2001, pp. 253-255.

[2] I. Chai and J. D. White, "Structuring data and building algorithms," Updated edition, McGraw-Hill Education, 2009, pp. 263-275.

[3] C. Zhiwang, X. Junggang, and S. Jian, "A multi-layer bloom filter for duplicated URL detection," in *the 3rd Intl. Conf. on Advanced Computer Theory and Engineering 2010*, pp. 586-591, China, August 2010.

[4] L. B. Costa, S. Al-Kiswany, and M. Ripeanu, "GPU support for batch oriented workload," in *IEEE 28th Intl. Perf. Comp. and Comm. Conf.*, pp. 231-238, USA, December 2009.

[5] A. Natarajan, S. Subramanian, and K. Premalatha, "A comparative study of cuckoo search and bat algorithm for bloom filter optimisation in spam filtering," Intl. J. of Bio-Inspired Computation, vol. 4, no.2, pp. 89-99, 2012.

[6] B. Chazelle, J. Kilian, R. Rubinfield, and A. Tal, "The bloomier filter: an efficient data structure for static support lookup tables," in *Proc. of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 30-39, USA, January 2004.

[7] N. S. Artan, K. Sinkar, J. Patel, and H. J. Chao, "Aggregated bloom filters for intrusion detection and prevention hardware," in *the IEEE Global Telecommunications Conf.*, pp. 349-354, USA, November 2007.

[8] D. Eppstein and M.T. Goodrich, "Space-efficient straggler identification in round-trip data streams via newton's identities and invertible bloom filters," LNCS, vol. 4619, pp. 637-648, Aug. 2007.

[9] A. Partow (2012, April 3). *General-purpose hash function algorithms.* [Online]. Available: http://www.partow.net/programming/hashfunctions/index.html.

[10] Y. Liu, L. J. Guo, J. B. Li, M. R. Ren, and K. Q. Li, "Parallel algorithms for approximate string matching with k mismatches on CUDA," in *IEEE 26th Intl. Parallel and Distributed Processing Symposium Workshop & PHD Forum*, pp. 2414-2422, China, May 2012.

[11] T-Y. Liang, Y-W. Chang, and H-F. Li, "A CUDA programming toolkit on grids," Intl. J. of Grid and Utility Comp., vol. 3, no. 2-3, pp. 97-111, July 2012.