GPU Support for Batch Oriented Workloads

Lauro B. Costa Samer Al-Kiswany Matei Ripeanu
NetSysLab
Electrical and Computer Engineering Department
The University of British Columbia
Vancouver, BC, Canada
{lauroc,samera,matei}@ece.ubc.ca

Abstract - **This paper explores the ability to use Graphics Processing Units (GPUs) as co-processors to harness the inherent parallelism of batch operations in systems that require high performance. To this end we have chosen Bloom filters (space-efficient data structures that support the probabilistic representation of set membership) as the queries these data structures support are often performed in batches. Bloom filters exhibit low computational cost per amount of data, providing a baseline for more complex batch operations. We implemented BloomGPU a library that supports offloading Bloom filter support to the GPU and evaluate this library under realistic usage scenarios. By completely offloading Bloom filter operations to the GPU, BloomGPU outperforms an optimized CPU implementation of the Bloom filter as the workload becomes larger.**

Keywords – gpu; bloom filter; batch workload; graphics processing unit

I. INTRODUCTION

High–performance systems that produce, process, and manage large amounts of data, use space-efficient data structures that provide fast response time when operating over these large datasets. A common characteristic of these systems is the use of batch operations to handle the data elements. That is, the data structures are built, queried, and processed for a set of elements at a time and not just for a single element. Since at the data-element level the operations in the batch are idempotent, batches have the attractive inherent potential for parallelization.

Recent technology trends make such potential for parallelization even more attractive indeed. After decades relying on single-core processor performance improvements (e.g., faster clock-rates, larger caches, instruction-level parallelism) for higher application performance, developers have now the choice of a wide array of novel, multi/many-core architectures from symmetric (massively) multi-core processors (e.g., existing quad-core architectures, Intel's 80-core Larabee prototype [18], GPUs), to asymmetric multicores (e.g., IBM's Cell Broadband Engine processor).

This paper explores the ability to exploit the parallelization opportunity triggered by batch workloads on a specific class of massively multicore processing units: namely Graphical Processing Units (GPUs). There are two reasons for this choice: GPUs' single- instruction-multiple-data (SIMD) architecture is a good match to batch operations, even though this architecture does not offer the same flexibility as chip-level multiprocessing. Second, today's GPUs have evolved to highly parallel devices that deliver an enormous computational power at relatively low-cost.

We explore the support for one, although simple, important and frequently used operation: *membership queries,* that is queries that simply ask "*Is element x in set S?".* To support set operations, we use *Bloom filters* [1] as their space- and time-efficiency characteristics make them a popular and practical implementation option for many existing systems [2].

Recently, Bloom filters have been used in scenarios that involve batch operations in high-performance data-intensive systems like Globus Replica Location Service (RLS) [5] and Google's BigTable [3]. The RLS is a mechanism to maintain information about file replica location in large distributed computing systems. RLS uses Bloom filters to represent the set of file replicas stored at a node thus avoiding to forward file requests to nodes that do not keep a replica of the file. Large RLS deployments use Bloom filters to represent sets of millions of files. BigTable is the primary storage system for several Google applications and handles tens of thousands of lookup operations per second. BigTable uses Bloom filters to represent the set of data items already loaded in the memory, thus helping to avoid touching the disk for lookups.

Additionally, Bloom filters rely on hash calculations over the sets' elements, rendering a relatively low ratio of computation over data. This characteristic makes Bloom filters a good baseline for comparison, useful in providing a lower bound for the speedups that can be obtained with more computationally intensive batch oriented workloads.

The contributions of this work are:

- This paper proposes BloomGPU, a library that enables the use of GPUs as a co-processor to offload Bloom filter operations, as the vehicle to evaluate the feasibility of using the novel multi-core architectures to support batch operations in current data-intensive high performance systems. We explore BloomGPU performance under different workloads and determine in which situations GPU's characteristics can be efficiently harnessed and lead to speedups compared to a CPU-based implementation.

- We analyze the overheads involved in GPU support for Bloom filter operations and present guidelines for possible performance improvements.
- We open-source the BloomGPU library and make it available to the community.

The rest of the paper is organized as follows. The next section presents Bloom filters and GPUs in detail. Section III describes the design of BloomGPU library. Section IV presents our experimental results. Section 5 summarizes related works. Section VI presents our conclusions and discussion for future directions.

II. BACKGROUND

This section describes the Bloom filter data structure, its benefits and drawbacks, as well as the GPU architecture and programming model.

A. Bloom Filters

Bloom filters are compact data structures (bit arrays) for probabilistic set representation. Set elements (or *keys*) are generally represented as strings. A Bloom filter works as follows: Initially, every bit in the bit-array is set to 0 To add a given key, the key is hashed by *n* hash functions, the values of the hash functions are used as an indexes to the bit-array and the corresponding bits are set to 1 (Figure 1). Query operations are similar: to find out whether a specific key has been added to the set, the query verifies whether all bits corresponding to the hash values used as indexes in the bit-array have been set to 1.

Apart from their compact size, Bloom filters have a number of other attractive features: add and query operations take constant time, set representations can be built incrementally as new keys are inserted in a set, and the union of two sets can be obtained by simply computing the bit-wise OR over the two corresponding bit-arrays (given that the bit-arrays have the same size and the same hash functions are used).

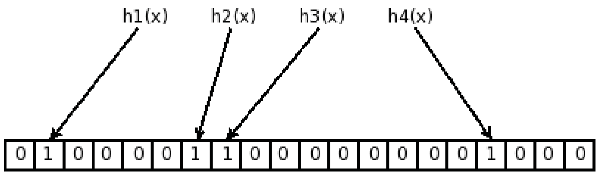

Figure 1. The ‘add’ operation for key x in Bloom filter represented as a 20-bit array and using four hash functions.

The payoff for the compact representation is the possibility of false positives: queries for non-member keys might return true. In practice, however, a numerous of applications can tolerate a small rate of false positives. Examples of Bloom filter use include: collaborative web caches [9], wide-area service discovery [2], databases [6][10] and backup systems [17].

The main factors that impact the probability of false positives are the ratio between size of the bit-array, the size of the set it represents, and the number of hash functions used. Larger bit arrays simply provide more bits to represent the set, thus reducing probability of a hash function collision and consequently reducing the likelihood of false positives. The number of hash functions impacts the false positive rate in a more complex way: initially, increasing the number of hash functions reduces the probability of a collision on all hash values, yet, after a threshold, the probability increases.

More concretely, to estimate the false positive rate we first estimate the probability of having a bit unset (zero) after inserting *n* keys into a filter of size *m* bits using *k* perfect hash functions (functions that spread the *returned values* evenly over the bit array {1..*m*}[9]:

$$p_0 = \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-\frac{kn}{m}} \qquad (1)$$

Thus, the probability of a false positive is the same probability of having all bits already set when inserting a new key:

$$p_{err} = (1 - p_0)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \qquad (2)$$

p_{err} is minimized by using $k = \frac{m}{n}\ln 2$ hash functions.

The hash functions that achieve good randomness in practice are computationally expensive [15]. (As a result, applications generally use fewer hash functions than optimally predicted by the Equation (2) to reduce the computational overhead). BloomGPU addresses this computational overhead by using the GPU as a co-processor to accelerate the hash functions computation. This enables decreasing the false positive rate while maintaining constant the execution time, or, alternatively, decreasing the execution time for the same false positive rate.

B. Graphic Processor Units

This section presents nVidia’s GPU architecture (Figure 2) and its programming environment - the Compute Unified Device Architecture (CUDA). We focus on nVidia GPU as a representative example of current GPU’s. Other vendors (e.g. AMD) offer comparable architectures and programming environments.

GPU architecture. NVIDIA GPUs have a Single Program Multiple-Data (SPMD) architecture. They offer a number of Single Instruction Multiple Data (SIMD) multiprocessors and four different memories each with their own performance characteristics: global, texture, constant, and shared. The global memory is the largest memory in the GPU (often hundreds of megabytes) however it has high latency taking several hundred cycles to fetch data. The texture and constant memories are much smaller and have restricted access policies. The shared memory is organized in small banks (kilobytes) dedicated to each multiprocessor, and it can be used as an application managed cache: access latency is two orders of magnitude lower than that for global memory and can be only accessed by the processors of the multiprocessor.

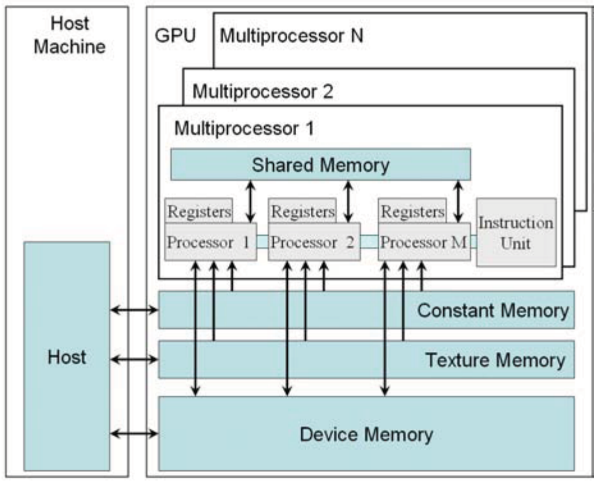

Figure 2. GPU Architecture

It is worth noting that the threads running on the GPU do not have direct access to the host's main memory. The programmer needs to explicitly transfer the data from the main memory to the GPU. The programmer is also in charge of explicitly managing the four types of memory available on the GPU.

Programming environment. The CUDA programming environment extends the C programming language, offering a set of abstractions to manage parallelism and memory resources. The libraries provided by CUDA allow programmers to control the memory type in which the application's variable reside, control the level of parallelism, obtain devices properties (such as the number of multiprocessors), and use per-multiprocessor thread synchronization.

GPU usage. The typical GPU usage cycle for a general purpose application is composed of five main stages: prepare the application's data (preprocessing), transfer the data from host memory to the device memory, launch threads to perform the computation (kernel function), copy back the results to the host's memory, and, if needed, postprocessing. While this workflow is straightforward, efficiency using GPU resources depends on several complex issues: task decomposition in order to exploit the GPU processing capabilities; fast memory allocation and data transfers between the host and the device; and, finally, efficient management of GPU's memory resources since each of the four types of memory offered by the GPU has its own latency, access policies, and capacity.

Moreover, the threads are grouped into thread blocks. All the threads in a thread block run on the same multiprocessors. The way applications divide threads into blocks affects the application's ability to use all the multiprocessors capabilities, and hence has a strong impact on the performance.

III. THE BLOOMGPU LIBRARY

The BloomGPU library supports high-performance Bloom filter operations for applications using Bloom filters intensively, such as Globus RLS and Google BigTable. These applications often issue add or query operations in batches of hundreds or thousands of keys. For each key (represented as an arbitrary length string) in the batch the Bloom filter needs to compute multiple hash functions, a considerable computational load. BloomGPU aims to transparently offload the CPU intensive Bloom filter operations to the GPU and as a result improve overall system performance.

The rest of this section presents the BloomGPU library API (A), the library design (B), and the GPU specific optimizations implemented (C).

A. BloomGPU API

We designed BloomGPU API to support the target applications' batch oriented usage pattern. After inspecting Globus RLS source code, we decided to implement the following API:

```
BloomFilter* bf_getInstance(int bfSize,
    int numberOfHashes, int hashFunction)
 void bf_free(BloomFilter* bf)
 void bf_addBatch(BloomFilter* bf,
    char** batch, unsigned int numOfElements)
 bool* bf_queryBatch(BloomFilter* bf,
    char** batch, unsigned int numOfElements)
```

The `bf_getInstance` function creates a new Bloom filter of `bfSize` bytes in the GPU global memory based on a `numberOfHashes` hash functions derived from a specific hash function. BloomGPU supports using MD5, SHA1 and a combination for general purpose hash functions maintained by Arash Partow [13]. The Bloom filter is stored in the GPU global memory. The `bf_free` function frees the GPU resources allocated to a Bloom filter.

The `bf_addBatch` and `bf_queryBatch` functions are batch oriented operations: process a batch of `numOfElements` elements. The batch is represented as an array of pointers, each referencing a key. This representation of the batch, although not GPU friendly (as we will see in the next section), corresponds to the current Bloom filter uses (such as the Globus RLS).

B. BloomGPU Design

A BloomGPU operation is carried in four main stages (Figure 3):

Preprocessing (Stage 1). Transferring the data to the GPU generates overheads proportional to the data size transferred and an additional constant overhead per transfer invocation. Consequently, to minimize these overheads, it is needed to transfer the input data in one or only a few copy operations. Since the elements of the input batch are not stored in a contiguous memory area at the host, preprocessing is required to lay out the data in a format that reduces the number of independent data transfer operations. Thus the preprocessing stage: (i) allocates a large enough contiguous space to hold the batch's keys, (ii) copies the keys to the new space from where they can be transferred to the GPU in only one operation, and (iii) prepares an index table indicating the start position and the size for each key of the batch. This layout makes the data transfer to the GPU significantly faster than transferring each element separately.

Copy in (Stage 2). The preprocessed input data and the index table are copied from the host memory to the GPU global memory.

Processing (Stage 3). This stage is the actual data processing stage: threads use the index table built in the preprocessing stage to locate a key. Each key is hashed by a separate GPU thread to produce its Bloom filter indexes (Stage 3.1). Then, the add operation sets the corresponding bits in the Bloom filter array at the global memory, while the query checks if the corresponding bits are set (Stage 3.2). Further, the query operation saves the query result in a result array (Stage 3.3). The processing stage leverages the GPU ability to spawn thousands of threads with minimum overhead, allowing BloomGPU to exploit the parallelism inherent in batch operations.

Copy out (Stage 4). This stage is required only for query operations to transfer the final result to the host.

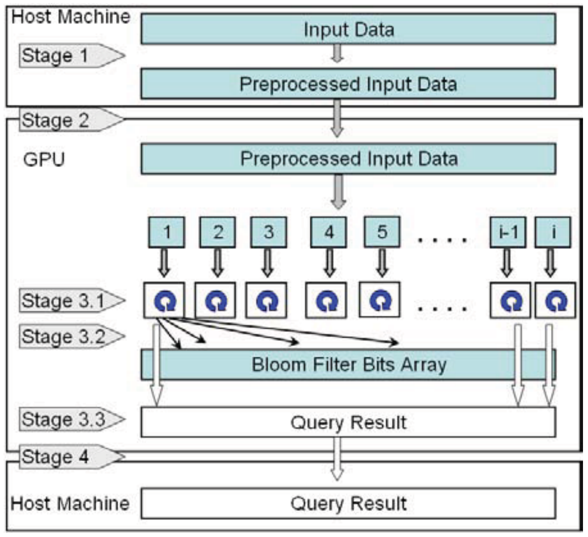

Figure 3. BloomGPU processing flow for a batch with i elements. The white steps and arrows are used in query operation only.

C. *Optimizations and Implementation Details*

BloomGPU implements two optimization features:

- *Automated tuning*: BloomGPU library automatically tunes its thread block size for every batch processing request. The auto-tuning aims to maximally use all the computing cores on the GPU card. The idea is to provide a number of threads per block large enough to take advantage of all cores in the multiprocessor, while creating enough blocks to hide memory access latency. Therefore, based on the number of multiprocessors available on the device and the number of threads a multiprocessor can execute in parallel (*warp-size*), BloomGPU tries to have more than one thread blocks per multiprocessor each with at least a *warp-size* of threads.
- *Pinned memory*. Transferring data between the host memory and the GPU global memory is performed by the DMA (Direct Memory Access). DMA requires that the data resides in a pinned (i.e., non-pageable) memory buffer. If the application provides the data in a pageable memory area, the GPU driver first allocates a pinned memory area, copies the data to the pinned memory, and then, calls the DMA to carry out the transfer. Allocating the memory initially as pinned avoids this overhead.

For the BloomGPU implementation, we had to use a full byte to represent a boolean entry in the Bloom filter instead of a bit. There are two reasons for this decision: the GPU card we use offers byte level access yet only limited support for thread synchronization. Since the GPU executes several threads simultaneously, more than one thread may modify bits of the same byte. Without synchronization the Bloom filter write access patterns could result in conflicts in bits of a same byte (i.e., different threads trying to set different bits of the same byte). Second, using a byte array avoids these problems without synchronizing since threads always write the same value (1- true) in a byte (and there is no situation in which another thread would try to write a different value).

IV. EVALUATION

This section evaluates the performance of the BloomGPU library. Section A describes the experimental environment and the methodology used, Section B presents the performance comparison, and explores the impact of the key size on performance and, finally, Section C analyses the relative overhead of the four processing stages of the processing flow.

A. *Experimental Environment and Methodology*

We use a host with 3GB of RAM and an Intel Core 2 Duo E6850 processor at 3.00 GHz with a 4MB L2 cache and 1.33 GHz front-side bus. The GPU installed on this host is a GeForce 8800GTX with 128@500MHz processors grouped over 16 multiprocessors. The GPU is connected to the host via a PCI Express x16 bus. The device global memory is 768MB.

Our experiments evaluate the execution time for *add* and *query* operations when using BloomGPU and compare them to a CPU optimized implementation. Timers start just before calling an add/query operation and stop when the operation finishes, thus we include all overheads that occur on the GPU: preprocessing, data transfers, and the actual data processing.

For the GPU, as we discussed in Section III.C, the filter is a byte-array stored in the global memory while for the CPU implementation it is a bit array kept in the host memory. Note that the comparison is fair since these choices provide the best performance for both the

CPU and GPU versions. For the CPU, an implementation using a byte-array would have lower performance due to a higher cache miss rate. Section B includes a brief evaluation of cache impact on a CPU.

The CPU version used for comparison is single-threaded. Our initial experiments also compared with two and four threads, but only one thread obtained the best performance. Bloom filters do not have good data locality due to the randomness of the hash functions used. Thus, multithreaded implementations, using two or four cores, impose high cache coherence overhead, degrading the overall performance. Further, if compact Bloom filter representation is used (every byte representing 8 bits in the filter) thread synchronization will add a significant overhead.

In our experiments, Bloom filters are designed to store up to one million elements with a false positive rate of up to 10^{-4}, a conservative assumption compared to most applications we have studied. We use eight hash functions. As a consequence the filters need to have 20 million boolean positions to achieve the desired rate of false positives.

Similar to most existing implementations, our implementation performs one hash operation (e.g., MD5) for each key, and derives the indexes used for Bloom filter operations using various permutations of the bytes returned by the actual hash value.

We focus the evaluation on a standard scenario where the keys have from 120 to 170 bytes. This size is the common size for a logical file name used in Globus RLS and the typical key size in the Google BigTable. Section B explores the impact of longer keys.

For all experiments, we report averages over multiple experiments. The number of experiments is adjusted to have a sample that guarantees 95% confidence intervals and accuracy of, at least, 5% according to the procedure described in [12]. Add and query operations have similar execution times; thus, we omit add operations results for simplicity.

B. Speedup

Figure 4 shows the average query times and Figure 5 shows the speedup while varying the batch size from 1 to 512,000 elements. For speedup, values larger than one indicate performance improvement, while values lower than one indicates slowdown. The results show that for small batches, BloomGPU does not offer a performance advantage. This situation changes as the batch size increases. For a batch sizes around 8,000 elements, the performances are similar; for larger batch sizes, BloomGPU offers up to 2.0x speedup.

BloomGPU cannot offer speedups for small batches as there is limited opportunity for parallelism and the overhead of managing the device and transferring the data is high compared to the amount of processing. As the batch size increases, the relative overhead decreases compared to the processing cost. Section C has a detailed analysis of these costs.

Impact of key size. Some applications use much larger although they are not as common as those used in Figure 4. For example, Google BigTable supports up to 64KB-long keys. Figure 5 also shows the speedups obtained for key sizes between 1.2KB and 1.7KB, all other parameters are the same as in the standard scenario. For larger keys BloomGPU starts to obtain speedups for batches as small as 2000 elements (four times smaller than for shorter keys). However, the peak of speedup reached is also smaller: 1.6x speedup.

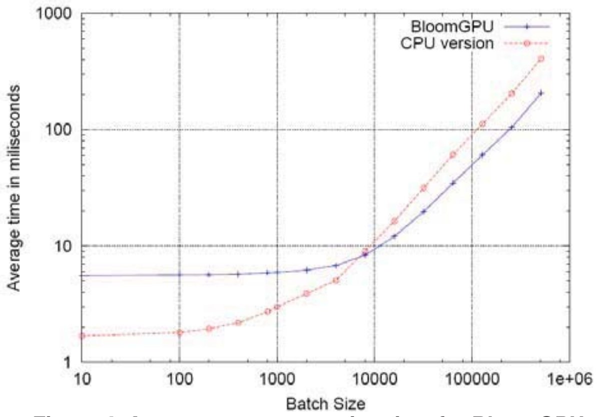

Figure 4. Average query execution time for BloomGPU and CPU version with small keys (120 to 170 bytes)

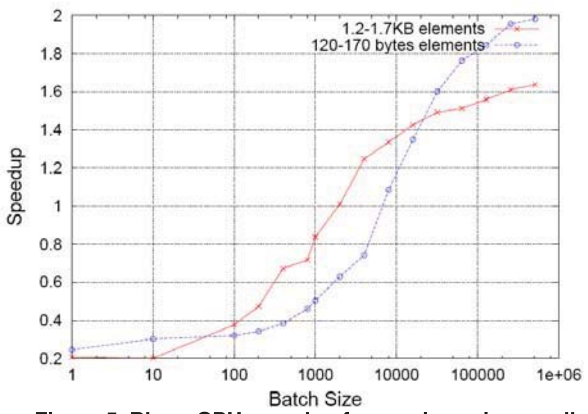

Figure 5. BloomGPU speedup for queries using small keys (120-170 bytes) and large keys (1.2-1.7 KB).

Figure 6 shows the GPU speedup obtained for a batch of 8000 elements varying the key size. For keys smaller than 100 bytes, the overhead of using GPU is high compared to the amount of data to process. The speedup increases until 1200 bytes. At this point, the overhead of transferring this data is too high and is not paid off by the GPU processing power.

Cache effects. A Bloom filter generates random memory accesses since the hash functions used spread randomly the bits that represent the elements. Thus, for maximum performance, a Bloom filter should be small enough to fit in the cache avoiding frequent accesses to main memory. To show the impact of the cache in the CPU version, we modified the implementation to use boolean variables of different sizes. Instead of using a bit for each boolean value (a bit-array), we use with one, two, and four bytes.

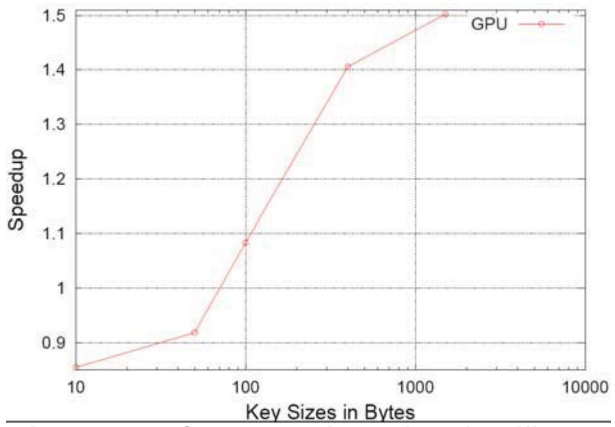

Figure 6. BloomGPU speedup for queries using different keys' sizes.

Figure 7 shows the speedup of these different implementations using the bit version as a baseline. GPU version obtained better performance than all CPU implementations that use a byte or more to store a Boolean. Note that, as the batch size increases, there is less room for the Bloom filter in the cache and all CPU implementations converge for the same result. (Putze et al observed the similar behavior [14] for CPU implementations).

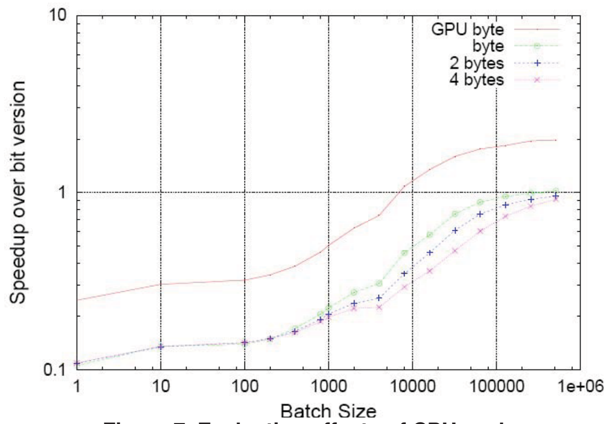

Figure 7. Evaluating effects of CPU cache

The main takeaway from the above observation is that once GPUs offer efficient synchronization functions that make possible efficient representation of booleans (as bits) for Bloom filters.

C. Decomposing the Overheads

This section analyses the relative overhead for each of the BloomGPU processing stages. We present only the breakdown for query operations when using small keys, results for add operations are similar.

Preprocessing. In this stage, BloomGPU allocates a contiguous pinned memory area on the host memory, copies each element to this area, and calculates where each element starts and ends. For batches up to about 1,000 keys the preprocessing times are almost constant as memory allocation overheads dominate (Figure 8). For larger batches, the preprocessing time grows linearly with the batch size.

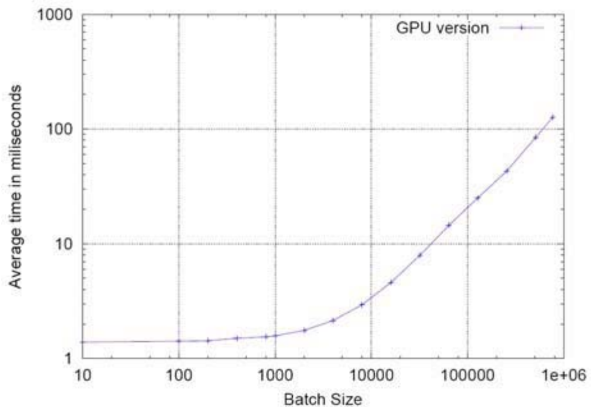

Figure 8. Average time of the preprocessing stage

Copy-in transfer stage. Figure 9 shows the input transfer times for different batch sizes. Similar to pre-processing times, for small batches, the transfer time does not depend on the amount of data transferred as CUDA library and device latency overheads dominate, keeping the time almost constant for batches up to 8000 elements.

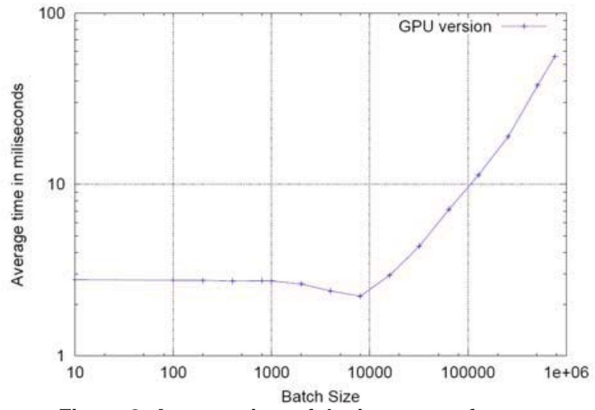

Figure 9. Average time of the input transfer stage.

Data processing stage. Figure 10 shows the data processing time. Small batches do not provide opportunity to launch enough threads and result in idle processors. The reason is that, with an insufficient number of threads, the GPU cannot hide the latency of memory fetches by scheduling other threads.

Output transfer. This stage exists only in query operations. Figure 11 shows the times to transfer data from the device to the host memory. Similar to input transfer, small transfers are dominated by device calls overhead and remains constant for up to 1000 keys. The main difference is the data size transferred is much smaller (one byte for each element in the batch) and the overall impact of this stage is small.

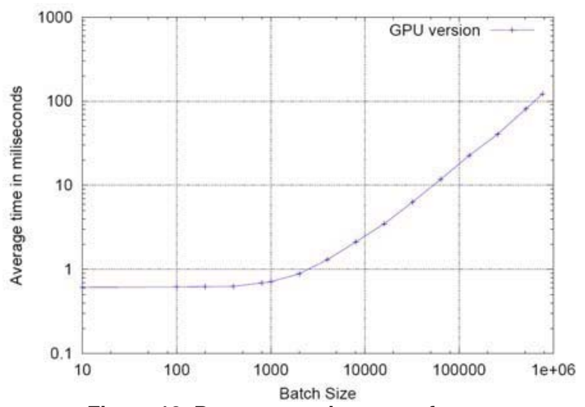

Figure 10. Data processing stage for query

Figure 12 shows the relative breakdown for these overheads. The results show the major impact of preprocessing in the total time; a result of making an additional copy of the entire input data. As the size of the batch increases, preprocessing and actual processing take equivalent times while, proportionally, the input transfer tends to remain the same.

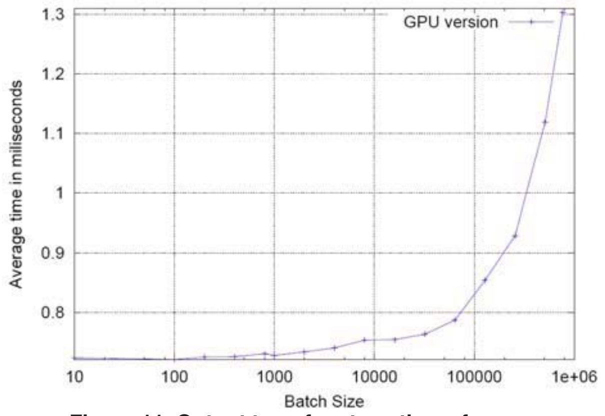

Figure 11. Output transfer stage times for query

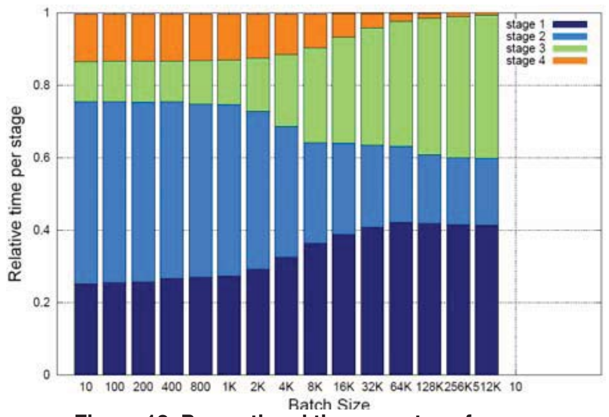

Figure 12. Proportional time per stage for query

It is worth pointing out the relatively large share of the preprocessing stage. In fact, our evaluation is conservative and the preprocessing can be avoided if the application provides the batch in a contiguous memory area. BloomGPU provides this alternative since an application can easily prepare the data and provide it in a format ready to be transferred and processed. For example, an application that reads the data from the disk with the only goal of creating a Bloom filter to represent the data stored in a local database - as in Globus RLS or during join queries carried out by databases. To estimate the impact of using a more GPU friendly data format, Figure 13 shows the speedup BloomGPU can offer when the application provides the batch in a contiguous memory area.

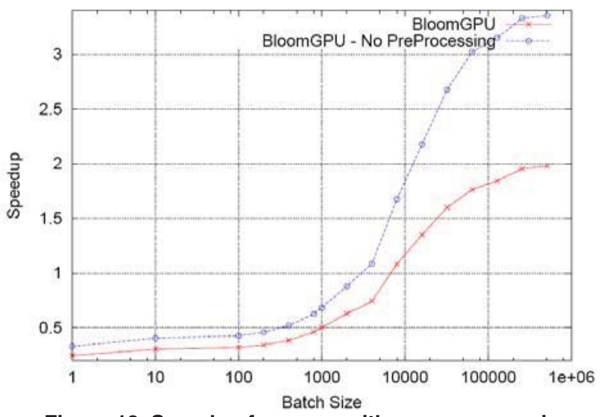

Figure 13. Speedup for query with no preprocessing

V. RELATED WORK

Exploiting GPUs for general purpose computing has recently gained popularity especially to accelerate scientific applications. Few projects, however, aim to explore the GPU computational power to accelerate computationally intensive operations at the middleware level. In this category, Curry *et. al* [7] explore the feasibility of using GPUs to enhance RAID system reliability. Their preliminary results show that GPUs can be used to accelerate Reed Solomon codes [16], used in RAID 6 systems to facilitate generating more than two parity blocks without affecting system overall performance.

Along the same lines Falcao et al [8] show that GPUs can be used to accelerate Low Density Parity Checks (LDPC) error correcting codes. Harrison and Waldron [11] study the feasibility of using the recent GPUs as a cryptographic accelerator.

These projects aim to accelerate data intensive applications, i.e. applications processing large inputs, such erasure coding a data block of few megabytes in size, or encrypting a megabyte-long file. BloomGPU aims to support a different class of data intensive application. BloomGPU accelerates the processing of a large batch of considerably smaller objects (few bytes compared to MB in previous work), which entails additional preprocessing overhead, and higher memory access overhead.

Other variations of Bloom filters exist. Counting Bloom filters additionally support delete operations yet pay the price of a significant size increase [9], Bloomier filters [4] extend the Bloom filters to support associating a value for each key in the filter, and Putze et al [14] propose a variation of Bloom filters that offer better cache locality while increasing the filter size.

VI. CONCLUSIONS

Massively multicore platforms such as GPUs have been successfully harnessed to accelerate scientific applications. This study provides evidence that GPUs can also be used to offload computationally demanding batch oriented operations. We developed BloomGPU, a library that supported batch-oriented Bloom filter operations. Our evaluation shows that BloomGPU effectively offloads the Bloom filter operations to the GPU and outperforms an optimized implementation running on the CPU by factors between 1.7x and 3.5x for large batch sizes (depending on the key size and the format of the input data).

In the future work we intend to extend this work in two directions. First, to further optimize the BloomGPU library we intend to explore the effectiveness of asynchronous operations introduced with the recent release of the CUDA 2.0 runtime library. The asynchronous operation mode holds the promise of overlapping the three time consuming stages: data preprocessing, input data transfer, and kernel computation on the GPU, thus effectively hiding the data preprocessing and transfer overhead. Second, we intend to integrate BloomGPU with Globus RLS service to evaluate the application perceived gains.

ACKNOWLEDGMENTS

We would like to thank John Dong and Greg Taylor for their work on the first version of BloomGPU.

REFERENCES

[1] B. H. Bloom, "Space/time Trade-offs in Hash Coding with Allowable Errors*," Communications of the ACM*, Vol. 13 (7). 1970. pp. 422-426.

[2] A. Broder, Michael Mitzenmacher. "Network Applications of Bloom Filters: A Survey". *Internet Mathematics*. 2002. pp. 636—646.

[3] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. "Bigtable: A Distributed Storage System for Structured Data." *ACM Transactions on Computer Systems.* Vol. 26(2) (Jun. 2008), pp. 1-26.

[4] B. Chazelle, J. Kilian, R. Rubinfeld, A. Tal, "The Bloomier filter: an efficient data structure for static support lookup tables", *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2004. pp. 30–39.

[5] A. L. Chervenak, R. Schuler, M. Ripeanu, M. Ali Amer, I. Foster, S. Bharati, A. Iamnitchi, C. Kesselman. "The Globus Replica Location Service: Design and Experience", *IEEE Transactions on Parallel and Distributed Systems* Vol. 20 (9) (September 2009), pp. 1260-1272

[6] S. Cohen, Y. Matias, "Spectral bloom filters", *ACM SIGMOD International Conference on Management of Data*, 2003.

[7] M. L. Curry, A. Skjellum, H. L. Ward, and R. Brightwell. "Accelerating Reed-Solomon Coding in RAID Systems with GPUs". *IEEE International Parallel and Distributed Processing Symposium.* 2008.

[8] G. Falcao, L. Sousa, V. Silva. "Massive Parallel LDPC Decoding on GPU". *ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP).* 2008.

[9] L. Fan, P. Cao, J. Almeida, A. Broder. "Summary cache: a scalable wide-area web cache sharing protocol", *IEEE/ACM Transactions on Networking*, Vol. 8 (2). 2000. pp. 281-293.

[10] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, J. D. Ullman. "Computing Iceberg Queries Efficiently", *International Conference on Very Large Data Bases*, 1998. pp.299-310

[11] O. Harrison, J. Waldron. "AES Encryption Implementation and Analysis on Commodity Graphics Processing Units". *Workshop on Cryptographic Hardware and Embedded Systems* (CHES). 2007.

[12] R. Jain, *The Art of Computer Systems Performance Analysis*. Wiley-Interscience, New York, NY, April. 1991

[13] A. Partrow http://www.partow.net/programming/hashfunctions/. Jan, 2009.

[14] F. Putze, P. Sanders, J. Singler, "Cache-, Hash- and Space-Efficient Bloom Filters", *6th International Workshop on Experimental Algorithms*, 2007

[15] A M. V. Ramakrishna, "Practical performance of Bloom filters and parallel free-text searching" *Communications of ACM* Vol. 32, 10. 1989. pp. 1237-1239.

[16] I. S. Reed, G. Solomon. "Polynomial Codes Over Certain Finite Fields". *Journal of the Society for Industrial and Applied Mathematics*, Vol. 8 (2). 1960. p. 300-304.

[17] S. Rhea, E. Cox, and A. Pesterev. "Fast, inexpensive content-addressed storage in foundation". In *USENIX Technical Conference*. 2008. pp. 143-156.

[18] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan. "Larrabee: a many-core x86 architecture for visual computing", *ACM Transactions on Graphics (TOG)*, Vol. 27 (3), 2008.