The SEPO Model of Computation to Enable Larger-than-Memory Hash Tables for GPU-accelerated Big Data Analytics

Reza Mokhtari and Michael Stumm
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
{*mokhtari, stumm*}@*ece.toronto.edu*

***Abstract*—The massive parallelism and high memory bandwidth of GPU's are particularly well matched with the exigencies of Big Data analytics applications, for which many independent computations and high data throughput are prevalent. These applications often produce (intermediary or final) results in the form of key-value (KV) pairs, and hash tables are particularly well-suited for storing these KV pairs in memory. How such hash tables are implemented on GPUs, however, has a large impact on performance. Unfortunately, all hash table solutions designed for GPUs to date have limitations that prevent acceleration for Big Data analytics applications.**

In this paper, we present the design and implementation of a GPU-based hash table for efficiently storing the KV pairs of Big Data analytics applications. The hash table is able to grow beyond the size of available GPU memory without excessive performance penalties. Central to our hash table design is the SEPO model of computation, where the processing of individual tasks is selectively postponed when processing is expected to be inefficient. A performance evaluation on seven GPU-based Big Data analytics applications, each processing several Gigabytes of input data, shows that our hash table allows the applications to achieve, on average, a speedup of 3.5 over their CPU-based multi-threaded implementations. This gain is realized despite having hash tables that grow up to four times larger than the size of available GPU memory.

I. Introduction

The goal of our work is to exploit GPUs to significantly accelerate the performance of Big Data analytics applications. Such applications comprise a wide class of software with the notable feature of consisting of relatively simple operations performed on a large number of independent input records. GPUs appear particularly well-suited for these trivially parallelizable applications given the GPUs' high degree of parallelism and high memory bandwidth. However, GPU memory sizes are limited and neither the "big" input data nor the intermediate/result data will fit entirely in GPU memory. This problem is compounded by the fact that the PCIe bus that connects CPU and GPU memories has limited bandwidth and high latency.

In this paper we present the design and implementation of a hash table for GPUs that is intended to be used as a key-value (KV) store for Big Data applications. The hash table is designed specifically so that (i) it can grow beyond the available size of GPU memory and when it does, its performance degrades gracefully; (ii) it supports variable-length keys and values; and (iii) it can perform on-the-fly grouping of KV pairs with the same key. Despite the added complexity introduced by these features, we show in Section VI that our hash table still performs on-par or faster than the existing GPU-based hash table designs that are limited to the size of available GPU memory.

A key novel aspect of our hash table is its use of what we call the *SEPO model of computation* that allows the hash table to grow beyond the size of available GPU memory without incurring excessive performance penalties. In the SEPO model of computation, a service requestee (e.g., server) may postpone the servicing of a request by declining the request and asking the requestor (e.g., client) to re-issue the request at a later time. The requestee might postpone the servicing of a request because required resources are not available or because it would be inefficient to provide the service at the time it is requested. This scheme is similar to the way some OS system calls return the `EAGAIN` error code, indicating that the required resources are temporarily unavailable and that the request must be reissued.

The SEPO model of computation imposes the following two requirements on an application. First, the application must be able to tolerate having its computations processed in any order. That is, if we break the computation of the application into independent tasks (e.g. one for each input record) then the order in which the tasks are processed should not affect the correctness of the application. The vast majority of Big Data analytics applications satisfy this requirement. Second, the application must be able to track requests that have been declined and then reissue these postponed requests at a later time. We show in Section IV that this is straightforward to implement for the applications we are targeting.

An approach like SEPO is needed to be able to deal with the non-trivial challenges of implementing a hash table that might grow larger than the available size of GPU memory. The hash table must be able to grow larger than available memory, because GPU memory is limited in size and because, for Big Data analytics applications, is not possible to predict a priori — before runtime — whether or not a given input dataset can be processed successfully using a hash table that fits within the available GPU memory. Unfortunately, hash tables, unlike other structures like arrays, cannot be broken down into smaller segments than can be operated on independently. Despite this limitation, hash tables remain attractive for Big Data analytics applications if they can be implemented effectively, because storing and retrieving KV pairs is very efficient, and because KV pairs with the same keys can be identified quickly.

Using SEPO, our hash table reorganizes the computation so as to minimize CPU-GPU data transfers. In consequence, the performance degradation that occurs when the hash table grows beyond the size of available GPU memory is "graceful". In fact, our experiments show that SEPO allows the hash table to grow up to more than four times larger than the size of available GPU memory before GPU acceleration is no longer effective. Our experimental results comparing seven GPU-based Big Data analytics applications to their CPU-based multi-threaded counterparts show that a speedup of 3.5 is achieved on average, despite having the hash table grow to more than four times larger than the available GPU memory. The effectiveness of our approach is further evaluated by having a MapReduce runtime we developed use our hash table as a KV-store. When compared to a state-of-the-art CPU-based MapReduce runtime, our runtime shows similar performance gains.

This paper makes the following specific contributions:

1) We present the first GPU-based hash table design that (*a*) can grow beyond the size of GPU memory without excessive performance penalties, (*b*) supports variable-sized KV pairs, and (*c*) supports on-the-fly grouping;
2) we introduce the SEPO model of computation; and
3) we introduce the first GPU-based MapReduce runtime that is capable of processing data larger than what GPU memory can hold.

The paper is organized as follows. Section II provides a short motivation to further explain the challenges we tackle. Next, an overview of the SEPO model of computation is presented in Section III. Section IV goes deep into the design and implementation of our hash table and how SEPO is used for it. Section VI presents our performance evaluation. We close with related work and concluding remarks.

II. Motivation

GPUs appear to be particularly well suited to accelerate Big Data analytics applications. A GPU, with its many cores, offers aggregate compute power an order of magnitude larger than what a CPU offers, and GPU memory has significantly higher theoretical bandwidth than CPU memory, yet GPUs are priced as commodity components.[1] This computational power and memory bandwidth nicely matches the high number of simple and independent operations that Big Data analytics applications are comprised of.

[1]E.g., Nvidia GTX 1080 GPU with 8.3 TFLOPS and 320 GB/s memory bandwidth vs. Intel Skylake CPU with 1.5 TFLOPS and 115 GB/s memory bandwidth. Yet, the price of GTX 1080 is about 1/8 the price of a Skylake.

High-performant KV stores on GPUs are necessary if GPUs are to be considered as a hardware base for Big Data analytics applications. Many of these applications store their (intermediate or final) results in the form of KV pairs; in part, this is because the Big Data ecosystem started off with MapReduce, which stores data in the KV format. Today, the high level of storage and interoperability support for the KV format has made it a *de facto* standard for Big Data analytics applications.

The hash table is an obvious data structure to consider for efficiently storing KV pairs in memory. Despite the irregularity of hash table memory accesses, which is a performance hazard for GPUs, several previous studies have shown clear performance advantages of hash tables over other potential data structures for KV storage [6], [7], [9]. Hash tables are not only able to store and lookup data efficiently, but they also offer on-the-fly grouping of pairs (where the values of KV pairs with the same key are "grouped" or "combined"). On-the-fly grouping of pairs eliminates two overheads that might otherwise occur when grouping is postponed to a later stage of execution: the overhead of storing multiple copies of the same key and the overhead of a separate grouping stage, that typically requires the data to first be sorted.

The use of hash tables on GPUs has one major challenge. Unlike simpler data structures like arrays, hash tables cannot be broken into smaller segments that can be operated on independently, because each key may index into any location of the hash table. As a result, it is non-trivial to algorithmically support a hash table that can grow larger than the available GPU memory yet still be efficient.

Two obvious system-level solutions to support larger-than-GPU-memory hash tables both incur high data transfer overheads. The first allocates the entire hash table as a pinned region in CPU memory and has the GPU threads directly access the structure in CPU memory remotely over the PCIe bus. The second solution uses a hardware demand paging mechanism for GPUs which uses CPU memory as the secondary storage: if the part of the hash table being accessed is not in GPU memory, then the corresponding page(s) is paged in before the access can complete. Both solutions incur a high number of data transfers over the PCIe bus, resulting in poor performance, as will be shown in Section VI-D. This overhead prevents GPU acceleration of applications that use a hash table if the table does not fit entirely in GPU memory. To make matters worse, due to the dynamic memory space requirement of hash tables, there is typically no way to predict whether a given dataset can be processed successfully within the available GPU memory or not. This makes GPUs an *unreliable* hardware base for real-world Big Data analytics applications.

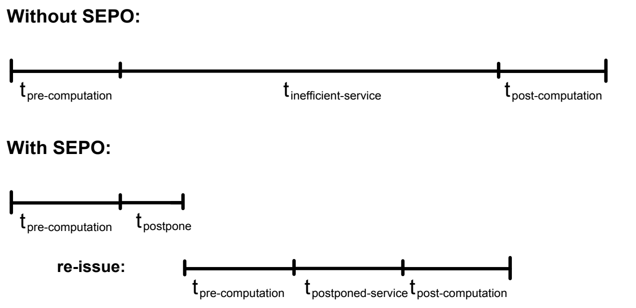


Figure 1: How the SEPO model of computation can improve performance.

III. SEPO Overview

We present SEPO as a general model of computation and then describe how it is used specifically to support larger-than-memory hash tables for GPUs.

A. *SEPO as a General Model of Computation*

In the SEPO[2] model of computation, a service requestee may postpone the servicing of a request by declining the request and asking the requestor to re-issue the request at a later time. The requestee might postpone the servicing of a request because it is inefficient to provide the service at the time it is requested.

Figure 1 shows how the SEPO model can improve the performance of an application. It considers two scenarios for processing an individual task. There is a basic trade-off between servicing the request inefficiently and the added overhead of some re-computation and servicing the request more efficiently. SEPO is effective when the following condition is true:

$$(t_{\text{pre-computation}} + t_{\text{postpone}}) + (t_{\text{pre-computation}} + t_{\text{postponed-service}} + t_{\text{post-computation}}) < (t_{\text{pre-computation}} + t_{\text{inefficient-service}} + t_{\text{post-computation}})$$

where $t_{\text{pre-computation}}$ is the expected time between the start of the task and the time the requestor issues the request including all of the direct or indirect overheads that starting a task might entail (e.g., data transfer overheads); t_{postpone} is the overhead of postponing the servicing of a task including keeping track of whether the task has been processed or not and reverting back/disposing any result that may have been produced during the corresponding $t_{\text{pre-computation}}$; $t_{\text{inefficient-service}}$ is the expected time to service the task when it is inefficient to do so, and $t_{\text{postponed-service}}$ is the expected time to service the task more efficiently after it was postponed; and finally, $t_{\text{post-computation}}$ is the time it takes to finalize the task after it has been successfully serviced (e.g., recording a log).

B. *Using SEPO for larger-than-memory hash tables*

We use *Page View Count* (PVC) to describe how SEPO might work on a hash table in practice. The application reads in a large log, where each line consists of an input record containing a URL. It extracts the URL and inserts the KV pair <`url,1`> into the hash table. On each insert, the hash table automatically *combines* KV pairs with the same key, so that the hash table would store <`url,n`> if the KV pair <`url,1`> had been inserted n times.

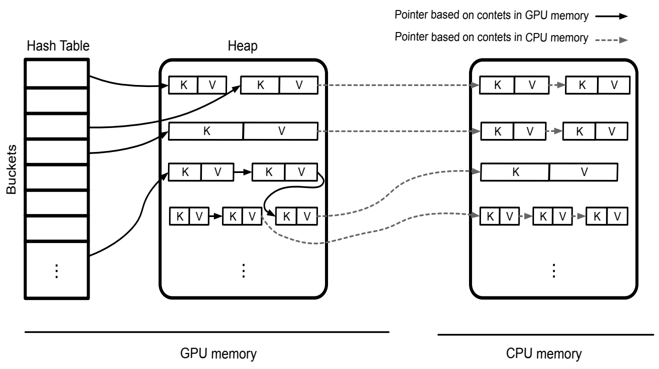
In this example, the application is the requestor and the hash table is the requestee. For each <`url,1`> insert request, if the key (i.e., the `url`) is already in the hash table, then its value is combined with the currently stored value of the key. If the key is not yet in the hash table, then space is allocated for the new KV pair (<`url,1`>) before it is inserted into the hash table. If the space allocation is unsuccessful, then the requestee responds with *POSTPONE*, and the input record is marked as not having been processed. We keep track of whether the input records have been successfully processed or not in a bitmap that has one bit per input record.

The application iterates over the entire set of input records multiple times in sequence until all input records have been successfully processed. In each iteration, it only considers input records that have not yet been processed, and it attempts to insert the KV pair <`url,1`> for the `url` extracted from those records in sequence. If the key being inserted has previously been inserted (from an earlier record), then an existing KV record with that key is guaranteed to be in GPU memory, and the value of the new KV pair (i.e., `1`) can be combined with the currently stored value. Otherwise, space for the new KV pair will have to be allocated (which, may or may not be successful).

Each time the application reaches the end of the set of input records, the hash table is signalled that it will no longer actively need any of the KV pairs currently located in GPU memory. The hash table then copies all of the pairs to CPU memory and frees up the heap in GPU memory to make it ready for the next iteration. Note that it will no longer need any of the KV pairs being copied back to CPU memory because all pairs (generated from the input) with the same keys will have already been successfully inserted/combined.

Figure 2 shows a snapshot of a hash table during a second

[2]SEPO is short for *Selective Postponement*, which implies that the requestee can selectively and temporarily postpone the provisioning of its service.



Pointer based on contets in GPU memory
Pointer based on contets in CPU memory
Hash Table
Heap
Buckets
K
V
GPU memory
CPU memory

Figure 2: Snapshot of a hash table at a subsequent iteration of computation. (Not all pointers are shown in this figure.)

iteration of computation. Note that new KV pairs are always inserted at the head of the *bucket linked list* (i.e. the linked list of KV entries that have been mapped to the same bucket) so that there is no need to traverse the linked list elements that might no longer be in GPU memory. Moreover, our implementation stores a set of two pointers in the hash table where ordinarily one would be used: one is based on the location of contents in GPU memory and another is based on the eventual location of contents in CPU memory – when copied back to CPU memory. This allows the hash table to be eventually accessible from both CPU and GPU sides.

IV. Hash Table Design and Implementation

Out hash table employs the closed addressing method and thus uses separate chaining with linked lists. Inserted KV pairs that map to the same bucket will be stored in a linked-list of *entries* rooted in the table. The entries are dynamically allocated using a custom dynamic memory allocator we designed for this purpose.

Using separate chaining with dynamic allocation of entries has a number of advantages. First, it allows the hash table to approach and surpass a load factor of 1 while having its performance degrade gracefully. This is an important attribute for our target applications considering that the number of KV generated by a Big Data analytics is often difficult to predict.[3] With a hash table that uses open-addressing (e.g., one that uses Cuckoo hashing [1]), insert operations are more costly, and even more expensive hash table re-organizations may be needed when the hash table approaches a load factor of 1.

Second, dynamically allocating bucket entries allows the hash table to start with nothing but a simple array of $null$ pointers, requiring little space. This allows the array to be allocated with a large number of elements – i.e. buckets – without allocating too much memory. Having a large number of array elements reduces lock contention among GPU threads when performing hash table operations.

Third, dynamic memory allocation allows bucket entries to be allocated exactly as large as they need to be. This not only preserves GPU memory, but also adds support for variable-sized KV pairs. A hash table that pre-allocates bucket entries has a difficult time supporting variable-sized KV pairs and often pre-allocates the entries conservatively large so that they can hold a wide range of KV pairs, hence consuming an unnecessary large amount of memory.[4]

A. Dynamic Memory Allocator

The dynamic memory allocator we designed for our hash table uses a heap that is pre-allocated in GPU memory. The heap is partitioned into pages, from which allocation requests are serviced. To determine the largest size the heap can be allocated as, we wait until all other data structures have been allocated, then query GPU memory for its remaining free space, and then allocate the heap with that size.

The primary objective of our dynamic memory allocator is to achieve high performance when used by 1,000's of concurrent threads. This is essential because the dynamic memory allocator is used in the critical path of GPU threads that populate the hash table. To make the allocator's service scalable, we distribute the allocation load onto multiple pages, instead of having all allocations serviced from one page. This way, instead of accessing one free-list pointer, the accesses are distributed over multiple free-list pointers (one per accessed page), reducing memory access contention. To do this, we partition the hash table buckets into *bucket groups*, each containing n contiguous buckets, and we allocate memory for each bucket group from a different page.

While having several pages to allocate memory from improves the performance of the memory allocator, it increases the potential for memory fragmentation, as some pages might not be fully used when the allocator fails to allocate memory for some allocation requests. This is a trade-off in which the right balance might be different for each application. Our hash table library, therefore, allows each application to balance this trade-off by adjusting the size of the bucket groups, which in turn changes the number of pages from which allocations are actively serviced from (e.g. a larger bucket group will have the hash table to be partitioned into fewer bucket groups and thus, distributes the allocation load onto fewer number of pages).

B. Bucket Organizations

Key-value pairs that are generated by Big Data analytics applications often have duplicate keys. However, different Big Data analytics applications handle such pairs differently. In our hash table design, we offer three different bucket organizations to cater to different kinds of KV pair handling: *basic* method, *multi-valued* method, and *combining* method:

Basic method:
Two KV pairs with the same key are stored as two separate entries in the linked list of bucket entries. This approach is appropriate for applications that do not require grouping.

Multi-valued bucket entries:
A separate list of values is associated with each key, resulting in a two dimensional linked structure with keys linked along one dimension and values linked to the their keys along a second dimension.

An example application that uses this method is *Inverted Index* which takes HTML pages as input and outputs a *1:N* mapping from the hyperlinks seen in the pages (keys) to the pages that have those hyperlinks in them (values). To do this,

[3]Sometimes even different input datasets result in significantly different number of KV pairs being generated by a single Big Data analytic application.

[4]For example, Inverted Index deals with URLs that are between 5 and thousands of characters and thus, requires the hash table to conservatively pre-allocate buckets of thousands of bytes.

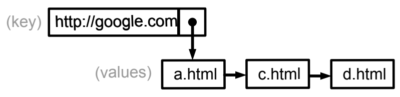


Figure 3: The final structure of an example bucket entry under the *multi-valued* method.

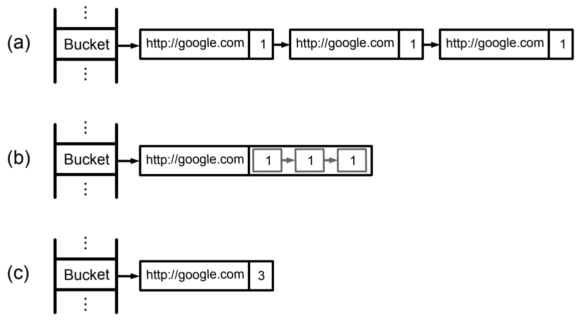


Figure 4: A snapshot of the hash table when filled by PVC data under three bucket organizations: (a) *basic*, (b) *multi-valued*, and (c) *combining*.

each time a hyperlink is found in a page, a pair in the form of <*hyperlink, pagePath*> is inserted into the hash table. At the end of the execution, each bucket entry will have one or more URLs (i.e. keys) and each will have a list of `pagePath`s associated with each URL. For example, if the hyperlink `http://google.com` is found in documents `a.html`, `c.html`, and `d.html`, the final bucket entry will look like the structure in Figure 3.

We store keys and values in separate pages when the *multi-valued* method is used. This allows the set of values grow independently of the set of keys and thus, offers more flexibility in handling the keys and values, which is essential for the SEPO model (see Section IV-C). However, separating keys from values when storing them in the hash table increases the chances for memory fragmentation.

Combining method:

A *combiner* is typically used to aggregate (i.e., *reduce* in MapReduce terminology) multiple values into a single value [12]. When inserting a KV pair with a key that already exists in the hash table, then it is only necessary to update the value of the existing bucket entry with the corresponding key. Memory space needs to be allocated only the first time a KV pair with a given key is inserted.

In our implementation, a callback is used to have the application handle the combining; it is called every time a new pair with a duplicate key is inserted.

Figure 4 shows a snapshot of the hash table when using each of the three different bucket organizations for PVC. As can be seen, providing the additional bucket organization methods can potentially save a substantial amount of memory, which is important when designing applications for GPUs. Moreover, on-the-fly grouping of entries with duplicate keys saves runtime by not requiring a separate grouping phase that is otherwise required to run after the hash table is fully produced.

C. Applying the SEPO model of computation

Here we describe how the Big Data analytics applications that use our hash table operate with the SEPO model of computation. We only focus on how the SEPO model handles hash table *inserts* while the hash table is being populated, because it is typically the only operation used when Big Data analytics applications process the input data during the first phase of the application, and because the first phase is typically the one with the highest computational demand (which is why we propose to use GPUs to accelerate it). The SEPO model can also be used for *lookup* operations on larger-than-memory hash tables when subsequent phases use/analyze the results but we leave that to the reader as a mental exercise.

The application starts by processing input data records, inserting the generated KV pairs into the hash table. Initially, all *inserts* will be successful, since all GPU-side pages have free space to store the inserted pairs. Every time a memory allocation request is made to a GPU-side page that is full, our dynamic memory allocator allocates a new page from the memory pool to satisfy the allocation request. After some time, however, the memory pool runs out of free pages, and then if more pages run out of free space, the hash table will be unable to store some of the pairs the application is trying to insert. The hash table *insert* method returns a boolean value to the requestor indicating whether it has successfully stored the pair or not (i.e., *SUCCESS* or *POSTPONE*). In our implementation of PVC, we use a bitmap array to record which tasks have been successfully processed. A *SUCCESS* return value causes the appropriate bit to be set.

With the SEPO model of computation, there may come a time when the computation will need to be halted so that the data and computation can be rearranged. For example, when no more KV pairs can be inserted into the hash table due to a full heap, the computation may need to be halted so that the heap can be copied to CPU memory and freed up in GPU memory before the computation can continue. Both decisions, when to halt the computation and how to rearrange the data and computation depends to a large extent on whether the *basic*, the *multi-valued*, or the *combining* bucket organization method is used. For this reason we describe them separately.

Basic method: for applications that use the *basic* method, Figure 5 (a) shows the points where the computation stops/restarts. The computation is allowed to continue until the requests from 50% of the bucket groups, a configurable parameter, are being postponed – i.e., when 50% of the bucket groups fail to allocate more memory for the inserted KV pairs.[5] Once the 50% threshold is reached, (i) the computation is halted, (ii) the entire heap on GPU memory is copied back to CPU memory, (iii) the heap on GPU is

[5]We observed acceptable performance with setting the threshold to 50%.

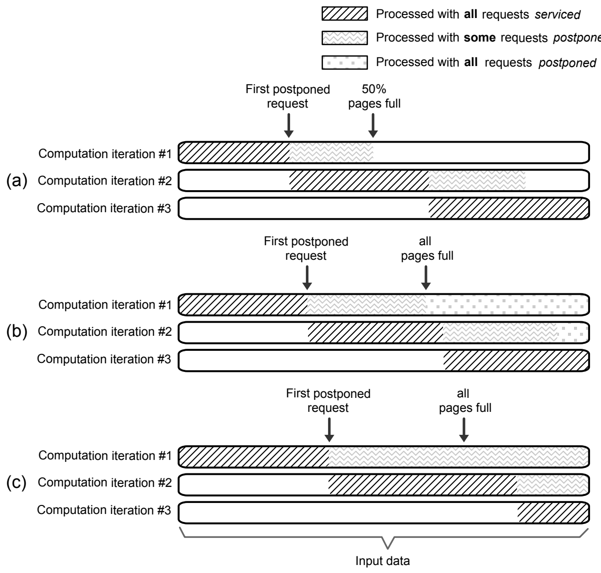


Figure 5: How input data is processed using each of the three bucket organization methods: (a) *basic*, (b) *multi-valued*, and (c) *combining*. Note that in (c), even after all pages get full, pairs with duplicate keys are still stored in the hash table.

freed up, adding the pages back to the memory pool and, (iv) the computation restarts to process input data records from the point where a request was postponed for the first time in the previous iteration.

An alternative approach is to not halt the entire computation, but only halt the threads that are unsuccessful in allocating more memory until a page is freed up in GPU memory. However, this approach is expected to be inefficient because efficient GPU hardware interrupt support does not exists and because the cost of extra synchronization that this method needs is high.[6]

Multi-valued method: for applications that use the *multi-valued* method, Figure 5 (b) shows the points where the computation stops/restarts. In each iteration, the computation processes all input records that have not successfully been inserted until the end of the input data has been reached (regardless of the percentage of the requests that are postponed). We need to do this to identify keys that have values that have not yet been inserted. At the end of each iteration (i) instead of copying the entire heap to CPU memory, only those pages are transferred that either are *value-pages* or are *key-pages* that do not contain any key that has values that have not yet been inserted, (ii) the GPU-side pages that were copied to CPU memory are freed up and added back to the memory pool, and (iii) the computation restarts to process input data records from the point where a request was postponed for the first time in the previous iteration.

[6]Note that, if we halt a GPU thread in software, it will have to spin-wait, which also causes all of its 31 neighbor threads in its warp to wait. Given the high latency of CPU-GPU communication, these threads will have to wait a long time before a page is freed up, which wastes a significant aggregate computing power. Even if a hardware interrupt support is provided, this alternative approach might still not work well because, as Zheng et al. envisioned, such hardware support might still halt a large number of GPU threads upon an interrupt [16].

Combining method: for applications that use the *combining* method, Figure 5 (c) shows the points where the computation stops/restarts. The *combining* method uses only one type of page to store both keys and values. Similar to the *multi-valued* method, in each iteration the computation continues to process input records until the end of the input data has been reached, because, even if the heap runs out of memory, pairs with duplicate keys can still be stored since they do not need additional memory space – they would only update the existing values. At the end of each iteration (i) the entire GPU heap is copied to CPU memory, (ii) the GPU heap is freed up, adding the freed pages back to the memory pool and, (iii) the computation restarts to process input data records from the point where a request was postponed for the first time in the previous iteration.

V. Use case: a simple MapReduce runtime

To test our hash table infrastructure, we developed a MapReduce runtime that uses BigKernel as the input memory manager [10], our hash table as the KV store, and a few more lines of code to schedule *map* and *reduce* phases. The runtime leaves the core logic of the application to be implemented by the application programmer inside the *map* and *reduce/combine* functions. Some MapReduce applications do not need a *reduce* phase, in which case the *reduce/combine* function is left empty. Finally, the application programmer is asked to provide an *input data partitioner* function which partitions the input data into smaller chunks, ready to be processed by the *map* functions.

Our MapReduce runtime can be configured by the programmer to work in the `MAP_REDUCE` or `MAP_GROUP` modes [6], [9], [12]. The `MAP_REDUCE` mode is used for MapReduce applications with a reduce phase that generate final <*key*, *value*> pairs and, the `MAP_GROUP` mode is used for application with no reduce phase that generate <*key*, *value***s**> pairs.

The flow of execution in our runtime is as follows. First, the *input data partitioner*, which runs on the CPU, is called; it splits the raw input data into smaller chunks. Next, BigKernel pipelines the chunks to the GPU cores where they are processed by a number of *map* function instances – one per input chunk created by the *input data partitioner*. Each instance is expected to generate zero or more KV pairs. The generated KV pairs are inserted into our hash table by the *map* function.

When the `MAP_REDCUE` mode is used, the hash table uses the *combining* method and the provided *reduce/combine* function as its callback function to aggregate/update the values associated with each distinct key. This means that the *reduce* phase is embedded into the *map* phase (as opposed to being run only after the *map* phase ends). This saves memory and improves performance [12]. When the `MAP_GROUP`

Application	Dataset #1	Dataset #2	Dataset #3	Dataset #4
Inverted Index	2 GB	3 GB	4 GB	5 GB
Page View Count	0.6 GB	2.2 GB	3.8 GB	5.8 GB
DNA Assembly	2 GB	4 GB	6 GB	8 GB
Netflix	1.6 GB	3.2 GB	4.8 GB	6.4 GB
Word Count (MapReduce)	0.2 GB	2 GB	3 GB	4 GB
Patent Citation (MapReduce)	0.2 GB	2.0 GB	3.4 GB	4.8 GB
Geo Location (MapReduce)	0.2 GB	1.8 GB	3.2GB	5 GB

Table I: Input dataset sizes used in our experiments.

mode is used, the hash table uses the *multi-valued* method to group (without reducing) all values associated to a key.

The SEPO model of computation is used so that the MapReduce runtime can handle large amounts of input and result data. In fact, we believe the SEPO model of computation makes our MapReduce runtime the first GPU-based MapReduce runtime that is capable of processing data for larger than what GPU memory can hold.

VI. Experimental Results

In this section, we present the results of our performance evaluation.

A. Experimental Setup

We performed our experiments on a PC with a 3.8GHz Intel Xeon Quad Core E5 with 8 hardware threads and 10MB of combined L2/L3 cache, connected to 16GB of quad-channel memory clocked at 1800MHz. All GPU kernels were executed on an Nvidia Geforce GTX 780ti GPU with 2,880 cores each running at 875MHz and 3GB of DRAM with a maximum bandwidth of 336 GB/s. The GPU is connected to the rest of the system via a PCIe Gen3 x16 bus interconnect. All GPU-based applications were implemented in CUDA, using CUDA toolkit and GPU driver release 6.0.1 installed on a 64-bit Ubuntu 12.04 Linux with kernel 3.5.

For our experiments, we implemented seven applications consisting of four stand-alone Big Data analytics applications (Netflix, DNA Assembly, Page View Count, and Inverted Index), and three MapReduce applications (Word Count, Geo Location, and Patent Citation). These applications were chosen primarily due to the amount of data they need to insert into the hash table. Each application is run with a variety of input dataset sizes, which in turn results in a variable number of KV pairs that have to be inserted into the hash table. Table I provides details on the application data sets used in our experiments. We briefly describe each application.

Netflix: calculates a similarity score between each pair of users based on their movie preferences [3]. Each KV pair inserted into the hash table is of the form <*userA&userB, similarity score between two users for a movie*>. The application uses the *combining* method.

DNA Assembly: merges fragments of a DNA sequence to reconstruct a larger sequence [2]. Each KV pair inserted into the hash table is of the form <*part of the DNA fragment, edges of the fragment*>. The application uses the *combining* method.

Page View Count: counts the number of occurrences of each URL in a web log. Each KV pair inserted into the hash table is of the form <*URL, 1*>. The application uses the *combining* method.

Inverted Index: builds a reverse index from a series of HTML files. Each KV pair inserted into the hash table is of the form <*link URL, HTML file path*>. The application uses the *multi-valued* method.

Word Count (MapReduce): counts the number of occurrences of each word in a document. Each KV pair inserted into the hash table is of the form <*word, 1*>. The application uses the `MAP_REDUCE` mode.

Geo Location (MapReduce): groups Wikipedia articles based on the geographic location from which they have been created. Each KV pair inserted into the hash table is of the form <*geographic location string, article ID*>. The applications uses the `MAP_GROUP` mode.

Patent Citation (MapReduce): produces a reverse patent citation directory – similar to what Google Scholar offers by the "cited by" functionality. Each KV pair inserted into the hash table is of the form <*the cited patent, the citing patent*>. The application uses the `MAP_GROUP` mode.

We modified our Big Data analytics applications to use BigKernel so as to minimize the overhead of transferring input data from CPU to GPU memory. Having more efficient input data transfer between CPU and GPU is especially important with the SEPO model of computation, because input data may be transferred to GPU memory multiple times.

B. Overall results

All of the execution times presented in this section include the input data transfer from CPU to GPU and transfer of the hash table from GPU to CPU. We believe this is the only fair way of comparing GPU implementations to CPU ones. Moreover, all CPU implementations that require dynamic memory allocation use `TCMalloc` [4] which is substantially faster than glibc's `malloc` in multi-threaded applications. Finally, all GPU-based implementations are configured to run with the number of GPU threads that result in the best execution time, as determined through experimentation.

We compared each of the four non-MapReduce GPU accelerated applications using our hash table with a CPU-based multi-threaded implementation. The CPU-based versions use a hash table design similar to our GPU-based hash table design except that they do not use the SEPO model of computation given that the entire hash table fits in CPU memory for all of our input datasets. The three MapReduce applications built with our MapReduce runtime are compared against the corresponding CPU-based applications developed using Phoenix++, a state-of-the-art MapReduce runtime for multi-core CPUs [12].

Figure 6 depicts the achieved speedups of the GPU-based applications over their CPU-based multi-threaded counter-

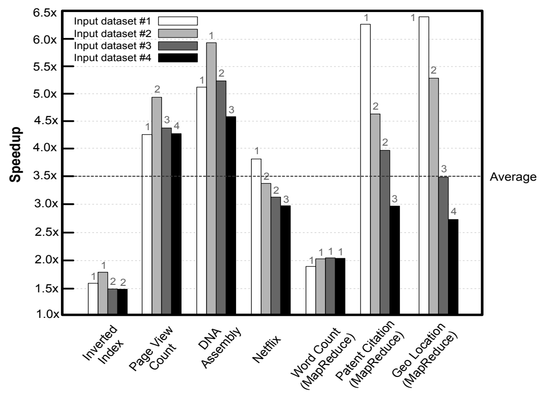


Figure 6: Application speedup over CPU multi-threaded implementation. For the last three, the baseline is Phoenix++.

parts for different dataset sizes. The numbers shown on top of the bars indicate the number of iterations that were necessary to successfully store all KV pairs into the hash table when using the SEPO model of computation. Focusing only on the datasets that are processed in a single iteration of computation – bars with `1` shown on their top – the applications exhibit a range of performance gains when accelerated with GPUs. All applications except Inverted Index and Word Count exhibit reasonable speedups.

Inverted Index and Word Count do not perform as well on GPUs for different reasons. Inverted Index has a long `switch-case` block in its core logic, which causes a high degree of thread divergence in GPUs, negatively affecting performance. Word Count suffers from lock contention when accessing buckets because of the small number of distinct keys and large number of duplicate keys.[7] A CPU implementation also suffers from lock contention, but not as much, given the significantly lower number of threads that run on the CPU. In fact, when we artificially increased the number of distinct keys in the input dataset of Word Count (by adding random, meaningless words to the input documents), performance quickly improved (not shown).

The number of iterations an application/dataset needs depends on how much memory is needed to store the KV pairs, which in turn depends on the number, size, and uniqueness of KV pairs and also on the bucket organization method used. For example, Word Count will rarely need multiple iterations even for large input datasets because (i) Word Count uses the *combining* method which saves memory by not allocating memory space for KV pairs with duplicate keys and (ii) the input dataset of Word Count typically consists of text documents which contain a limited number of distinct words no matter how large the documents is.

[7]For instance, the number of occurrences of the word 'that' in a document is high.

Application	Speedup
Word Count (MapReduce)	1.05X
Patent Citation (MapReduce)	2.42X
Geo Location (MapReduce)	2.55X

Table II: Speedups over MapCG.

C. *Comparing with MapCG*

We took MapCG as a state-of-the-art GPU MapReduce system and implemented our three MapReduce applications on it to compare it with the MapReduce system we built using our hash table [7]. MapCG also uses a hash table to store KV pairs generated by the *map* function instances. Similar to all existing GPU-based MapReduce runtimes that use a hash table, however, MapCG is unable to support a larger-than-memory hash table, and thus the execution fails when there is no more free memory to store newly inserted KV pairs. In fact, we were able to compare the performance of MapCG with our own MapReduce runtime only for the smallest input datasets (input datasets between 200MB-600MB despite having a GPU with 3GB of internal memory[8]).

Not being able to process large input datasets in these experiments means that our hash table was, effectively, not using the SEPO model of computation (i.e., no KV pair insertions were postponed). Consequently, the comparison with MapCG only evaluates the efficiency of the basic design of our hash table, including dynamic memory allocation and synchronization.

Table II lists the speedups of the three applications when run using our MapReduce runtime over MapCG on the same testbed. Our MapReduce runtime performs on par with MapCG for Word Count, primarily because the performance of both runtimes are limited by the heavy contention for locks during the KV pair insertions. For the other two applications, however, our MapReduce outperformed MapCG by over a factor of 2.

D. *Comparing with alternative approaches*

In this section, we compare the performance of our hash table to the two alternative system-level solutions we described in Section II that potentially could be used to allow a larger-than-memory hash table for GPUs, namely (i) pin the hash table in CPU memory, and (ii) using a hardware demand paging mechanism.

Hash table pinned to CPU memory

When a memory region is pinned in CPU memory, the operating system will not page it out to disk and it can be accessed directly by GPU threads over the PCIe bus. Given that typical CPU memories are much larger than GPU memories, a much larger hash table can be allocated and

[8]Even though our testbed GPU has 3GB of memory space, its memory is shared among different data structures and thus each data structure is given a smaller space.

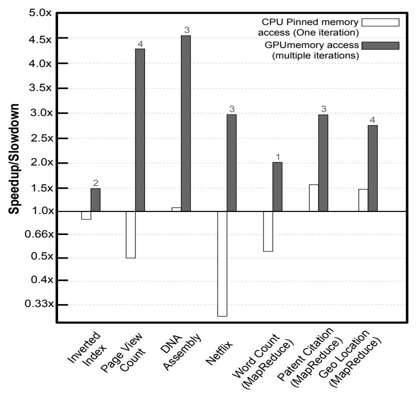


Figure 7: Speedups compared to the pinned version.

fully populated (by the GPU) without needing the SEPO model of computation.

As an experiment we modified our dynamic memory allocator to pre-allocate its heap as a pinned CPU memory region (thus storing the content of the hash table in CPU memory). Everything else is kept in GPU memory for higher memory performance (e.g. locks). The heap is allocated sufficiently large so that the hash table's entire content can fit in it. We ran all applications with the largest dataset (i.e. input dataset #4) on this new version of the hash table and compared it with our GPU-based hash table using SEPO.

Figure 7 shows the result of this experiment in which we show the speedups of our applications when using this modified version of the hash table as well as when using our version of the hash table. Speedup is measured relative to the CPU-based multi-threaded implementation of the applications. Even though the original hash table needs multiple iterations of computation to process all of the input data, it still significantly outperforms the version that allocates the heap in CPU pinned memory. Worse, in four out of seven applications, the CPU pinned memory version of the hash table performs worse than the CPU-based multi-threaded implementations. The reason for this poor performance is not only the high volume of data that has to be transferred over the PCIe bus, but the fact that the data is transferred over many small PCIe transactions, which is much costlier than a few bulky PCIe transactions.

CPU-side hash table with demand paging

Another system-level solution that supports larger-than-GPU-memory hash tables is to use a GPU hardware that has built-in demand-paging support [11]. Such GPU would allow the application to allocate more GPU memory than is physically available. It will copy pages between CPU and GPU memories as needed to ensure the accessed data is available in GPU memory prior to completing the access. Due to the irregularity of accesses to a hash table, a larger-than-memory hash table with demand paging is expected to exhibit frequent paging activity which degrades performance substantially.

Currently, GPU hardware demand paging support is still

Assumed physical GPU memory	Data transfer time (1MB page size)	Data transfer time (128KB page size)	Data transfer time (4KB page size)	Total execution time with our hash table
1200	0.00s	0.00s	0.00s	1.22s
1100	14.8s	2.04s	0.07s	1.29s
1000	101.6s	11.4s	0.60s	1.37s
900	261.5s	32.5s	1.21s	1.45s
800	496.5s	62.1s	2.14s	1.56s
700	801.4s	104.4s	4.47s	1.65s
600	1178.3s	157.7s	6.12s	1.76s
500	1626.5s	128.7s	7.87s	1.89s
400	2148.3s	292.2s	10.33s	2.02s

Table III: Calculated lower bound data transfer time if PVC was run on a demand paging-equipped hardware compared to the total execution time when PVC is run using our hash table.

very immature and inefficient [16]. We expect it to get more efficient in the near future. In the absence of an efficient demand paging hardware, we could have simulated the corresponding hardware with the demand paging support using a GPU simulator to measure the efficiency of this alternative solution, but instead we came up with a simple experiment that provides us with a lower bound on the overhead for this solution. In this experiment, we instrumented the code of PVC to record the access pattern to the hash table. We use this access pattern to simulate and then count the number of page replacements that demand paging hardware would have imposed during the runtime of the application. Multiplying this number by the page size yields the total amount of data that has to be transferred over the PCIe bus, which in turn gives us the lower bound runtime that PVC would have spent transferring data under a demand paging-equipped GPU.

Table III shows the results of this experiment. The input dataset we used for this experiment ends up populating a hash table that reaches 1.2 GB in size. In our simulations we initially set GPU memory to have 1.2 GB of free space (so that the entire hash table fits in GPU memory and no paging is required, considering that all pages are initially GPU resident). We then ran the experiment multiple times, each time reducing the available free space so as to increase the frequency of paging. For each run, we calculated the amount of the data that had to be transferred between CPU and GPU and, based on that, calculated the time it takes to transfer the data over the PCIe bus, and reported that number in the table as the data transfer time. Even though this data transfer time is only one of the overheads associated with demand paging (others including overhead to initiate PCIe transactions and overhead of page fault interrupt handling) it still, in many cases – including all cases where the hash table is about 1.5 times or more larger than the available GPU memory – exceeds the total execution time of running the application using our hash table.

VII. Related Work

Despite the challenges involved in using hash tables for GPUs (including thread and memory divergence and need for costly synchronization when accessing them), hash tables have been used extensively because they offer very fast insert and lookup operations [1], [5], [13], [15].

MapCG that we compared against in our performance evaluation also uses a hash table to store key-value pairs [7].

Despite the fact that a MapReduce application can generate many key-value pairs with duplicate keys and thus some contention in accessing the hash table, MapCG shows 1.6-2.5X performance speedup compared to earlier implementations that used arrays to store the key-value pairs.

Stadium hashing proposes a hash table design where the hash table itself is located in a pinned portion of CPU memory, where it is directly accessed by GPU threads [8]. To reduce the number of accesses to CPU memory, a compact indexing data structure located in GPU memory is used to store a fingerprint hash token for each item stored in the hash table; every operation on the hash table first consults this index before accessing the hash table. For instance, on an insert, the GPU thread first uses the index data structure to find an empty bucket, and only then will it access CPU memory to store the data item. Stadium hashing reports 2-3 times faster operations over earlier GPU-based hash table implementations.

Mega-KV is a CPU-based in-memory key-value store for distributed systems that uses a hash table to store key-value pairs locally on a node [14]. The hash table is accelerated by a GPU-based index table. Similar to the Stadium hashing's approach, Mega-KV uses the GPU only for the heavy-lifting part of the operations (e.g., scanning the hash table for an empty bucket during an insert, or finding a bucket item during a lookup). However, the actual data is kept on and accessed in CPU memory.

Unlike our solution, neither Stadium hashing nor Mega-KV handle key-value pairs with duplicate keys even though they are common in Big Data analytics applications. They both store pairs with duplicate keys as if they are pairs with different keys that happen to map to the same buckets.

VIII. Concluding Remarks

GPUs have, so far, rarely been used to accelerate real-world Big Data analytics applications despite their enormous computing power and high memory bandwidth. A main reason, we believe, has been the lack of an efficient key-value store that can efficiently store keys and values produced by Big Data analytics applications and not fail when the data grows beyond what GPU memory can hold.

The GPU-based hash table we developed for storing key-value pairs of Big Data analytics applications is capable of retaining reasonable efficiency even when its data grows beyond the size of GPU memory. This is made possible with the help of SEPO, a model of computation we developed to reduce the overhead of CPU-GPU data transfers when the hash table does not fully fit in GPU memory. Under our SEPO model of computation, a larger-than-memory hash table will postpone certain operations (i.e., insert or lookup) if they attempt to access non-resident portions of the hash table. Such operations are postponed until the requested portions become resident. Our experimental results comparing GPU-based Big Data analytics applications to their CPU-based multi-threaded counterparts, showed that an average speedup of 3.5 is achieved, despite having the hash table grow up to four times larger than the available GPU memory.

References

[1] D. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. Owens, and N. Amenta. Building an efficient hash table on the GPU. *GPU Computing Gems*, 2:39–53, 2011.

[2] J.A. Chapman, I. Ho, S. Sunkara, S. Luo, G.P. Schroth, and D.S. Rokhsar. Meraculous: De Novo Genome Assembly with Short Paired-End Reads. *PLoS ONE*, 6(8):1–13, 2011.

[3] S. Chen and S.W. Schlosser. Map-Reduce meets wider varieties of applications. *Intel Research Pittsburgh, Tech. Rep. IRP-TR-08-05*, 2008.

[4] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc, 2007. *Retrieved from: http://goog-perftools.sourceforge.net/doc/tcmalloc.html*.

[5] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proc. of the ACM SIGCOMM Conf.*, pages 195–206, 2010.

[6] B. He, W. Fang, Q. Luo, N.K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proc. of the 17th Intl. Confl. on Parallel Architectures and Compilation Techniques*, pages 260–269, 2008.

[7] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. MapCG: writing parallel program portable between CPU and GPU. In *Proc. of the 19th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 217–226, 2010.

[8] F. Khorasani, M. Belviranli, R. Gupta, and L. Bhuyan. Stadium Hashing: scalable and flexible hashing on GPUs. In *Proc. of the 2015 Intl. Conf. on Parallel Architecture and Compilation (PACT)*, pages 63–74, 2015.

[9] R. Mokhtari, A. Abbasi, F. Khunjush, and R. Azimi. Soren: Adaptive MapReduce for Programmable GPUs. In *Proc. of the 4th Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, pages 118–134, 2011.

[10] R. Mokhtari and M. Stumm. BigKernel – High Performance CPU-GPU Communication Pipelining for Big Data-Style Applications. In *Proc. of the IEEE 28th Intl. Parallel and Distributed Processing Symp. (IPDPS)*, pages 819–828, 2014.

[11] Nvidia. GP100 Pascal Whitepaper. Retrieved from: $https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf$, 2016.

[12] J. Talbot, R.M. Yoo, and C. Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. In *Proc. of the 2nd Intl. workshop on MapReduce and its Applications*, pages 9–16, 2011.

[13] S. Tzeng and L. Wei. Parallel white noise generation on a GPU via cryptographic hash. In *Proc. of the 2008 Symp. on Interactive 3D Graphics and Games*, pages 79–87, 2008.

[14] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-KV: a case for GPUs to maximize the throughput of in-memory key-value stores. In *Proc. of the VLDB Endowment*, pages 1226–1237, 2015.

[15] Y. Zhang, F. Mueller, X. Cui, and T. Potok. GPU-accelerated text mining. In *Proc. of the Workshop on Exploiting Parallelism Using GPUs and Other Hardware-assisted Methods*, pages 1–6, 2009.

[16] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. Keckler. Towards high performance paged memory for GPUs. In *Proc. of the 2016 Intl. Symp. on High Performance Computer Architecture (HPCA)*, pages 345–357, 2016.