A heterogeneous CPU+GPU Working-set hash table

by

Ziaul Choudhury, Suresh Purini

in

MULTIPROG-2016

Report No: IIIT/TR/2016/-1

Centre for Software Engineering Research Lab
International Institute of Information Technology
Hyderabad - 500 032, INDIA
January 2016

Heterogeneous (CPU+GPU) Working-set Hash Tables

Ziaul Choudhury and Suresh Purini

International Institute of Information Technology,
Hyderabad,India
{ziaul.choudhury@research., suresh.purini@}iiit.ac.in

Abstract. In this paper, we propose heterogeneous (CPU+GPU) hash tables, that optimizes operations for frequently accessed keys. The idea is to maintain a dynamic set of most frequently accessed keys in the GPU memory and the rest of the keys in the CPU main memory. Further, queries are processed in batches of fixed size. We measured the query throughput of our hash tables using Millions of Queries Processed per Second (MQPS) as a metric, on different key access distributions. On distributions, where some keys are queried more frequently than others, we achieved on average 10x higher MQPS when compared to a highly tuned serial hash table in the C++ `Boost` library; and 5x higher MQPS against a state of the art concurrent lock free hash table. The maximum load factor on the hash tables was set to 0.9. On uniform random query distributions, as expected our hash tables do not out perform concurrent lock free hash tables, nevertheless matches their performance.

1 Introduction

A hash table is a key-value store which supports constant time insert, delete and search operations. Hash tables do not lay any special emphasis on the key access patterns over time. However, the key access sequences in real world applications tend to have some structure. For example, till a certain point in time a small subset of keys could be more searched more frequently than the rest. Data structures like Splay trees [17] are specifically designed to reduce the access times of the frequently accessed keys by keeping them close to the root using rotation operations on the tree. The working set property states that it requires at most $O(\log[\omega(x) + 2])$ time to search for a key x, where $\omega(x)$ is the number of distinct keys accessed since the last access of x. Splay trees satisfy this property in an amortized sense [17], while the working-set structure satisfies the same in the worst case sense [4]. The working-set structure is an array of balanced binary search trees where the most recent keys occupy the smaller trees at front. Through a sequence of insertions and deletions the older keys are propagated to the bigger trees towards the end.

Following the recent uprise of accelerator based computing, like the heterogeneous multi-core CPU+GPU based systems, many data structures, for example quad-trees [9] and B+ trees [5] have been successfully ported to these exotic platforms thus achieving greater performance. In this direction, inspired by the working-set structure, we propose a set of two level heterogeneous (CPU+GPU) hash tables in this paper. In all the designs the first level of the hash table is smaller in size and resides in the GPU memory. It essentially caches the most recently accessed keys or in other words hot data. The second level hash table resides in the CPU memory and contains the rest of the keys. We overlay an MRU list on the keys residing in the GPU hash table. The queries are batched and are processed first on the GPU followed by the CPU. Our overall hash tables can be viewed as a heterogeneous two level working-set structure. To the best of our knowledge, this is the first attempt towards designing heterogeneous (CPU+GPU) hash tables, wherein we use the GPU accelerator to improve the query throughput by exploiting the key access patterns of the hash table.

The rest of the paper is organized as follows. In section 2 we give a brief background necessary towards designing a heterogeneous (CPU+GPU) hash table. In section 3 we describe our hash table designs in detail followed by experimental results in section 4. We conclude with directions to some future work in section 5.

2 Background

This section gives an overview of the heterogeneous (CPU+GPU) architecture followed by a brief discussion on the multi-core CPU and GPU hash tables that inspired the work in this paper.

GPUs are composed of multiple streaming multiprocessors (SMs) each containing a number of light weight primitive cores. The SMs execute in parallel independently. The memory subsystem is composed of a global DRAM and a L2 cache shared by all the SMs. There is also a small software managed data cache whose access times is close to register speeds called shared memory. This is attached to each SM and shared by all the cores within a SM. A compute kernel on a GPU is organized as a collection of thread blocks which are in turn grouped into batches of 32 threads called warps. One instruction by a warp of threads is executed in a constant number of cycles within the SM. A warp is the basic unit of execution in a GPU kernel. The GPU is embedded in a system as an accelerator device connected to the CPU through a low bandwidth PCIe express bus. GPUs are programmed using popular frameworks like `CUDA` [1] and `OpenCL`. Heterogeneous computing using CPU and GPU traditionally involves the GPU handling the data parallel part of the computation by taking advantage of its massive number of light weight parallel threads, while the CPU handling the sequential code or data transfer management. Unfortunately a large fraction of time in a CPU+GPU code is spent in transferring data across the slow PCIe bus. This problem can be mitigated by carefully placing the data in the GPU so that fetching of new data from the CPU is as minimum as possible. The CPU after transferring the data and launching the kernel mostly sits idle during the computation. In this work the motivation is to keep both the devices[1] busy while executing operations on the respective hash tables.

[1] In this paper, by devices we mean the CPU and it's connected GPU.

Data structures that use both the CPU and GPU simultaneously have been reported in literature. Kelly and Breslow [9] proposed a heterogeneous approach to construct quad-trees by building the first few levels in the CPU and the rest of the levels in the GPU. The work load division strategy has also proven its worth in cases where the costly or frequent operations were accelerated on the GPU while the rest of the operations were handled by the CPU. Daga and Nutter [5] proposed a B+ tree implementation on an Accelerated Processing Unit (APU). They eliminated the need to copy the entire tree to the GPU memory, thus freeing the implementation from the limited GPU memory.

General hash table designs include linear hash table, chained hash table, cuckoo hash table and hopscotch hash table [7]. Among these the cuckoo hashing [15] technique can achieve good performance for lookup operations. Cuckoo hashing is an open address hashing scheme which uses a fixed number of hash tables with one hash function per table. On collision the key replaces the already present key in the slot. Now the "slotless" key is hashed into a different table by the hash function of that table and the process continues until all the keys have a slot. There has been efforts directed towards designing high performance hash tables for multi-core systems. *Lea's* hash table from the Java Concurrency Package [11] is a closed address lock based hash table based on chaining. Hopscotch hashing [7] guarantees constant lookup operations. It is a lock based open address technique which combines linear probing with the cuckoo hashing technique. Initial work on lock free hashing was done in [13] which used chaining. The lock free version of cuckoo hashing was designed in [14]. The algorithm allowed mutating operations to operate concurrently with query ones and required only single word compare-and-swap primitives. They used a two-round query protocol enhanced with a logical clock technique to ensure correctness. Pioneering work was done for parallel hashing in GPU by Alcantara et.al. [2]. They used cuckoo hashing on the GPU for faster lookup and update operations. Each thread handled a separate query and used GPU atomic operations to prevent race conditions while probing for hash table slots. This design is a part of the `CUDPP` [3] library, which is a data parallel library of common algorithms in the GPU. The work in [10] presented the Stadium Hashing (Stash) technique which is a cuckoo hash design and scalable to large hash tables. It removes the restriction of maintaining the hash table wholly on the limited GPU memory by storing container buckets in the host memory as well. It used a compact data structure named ticket-board separate from hash table buckets maintained in the GPU memory which guided all the operations on the hash tables.

3 Proposed Heterogeneous Hash Tables

In this section, we give an overview of the basic design of our hash tables and their memory layout across both the devices. The primary goal of our hash tables is to support faster operations on recently accessed keys similar to the working-set structure. Unlike previous works the size and scalability of our hash tables is not restricted by the limited GPU memory. The GPU is used to cache the most frequently accessed keys (hot data). These key-value pairs are processed in parallel by all the GPU threads. The *search* and *update* queries are bundled into batches of size B before processing. We intuitively expect that every key k with $\omega(k) \leq cM$ where M is the size of GPU global memory and $0 < c \leq 1$ is some constant, is available in the GPU. The value of c depends on the key-value pair record size.

All the key-value pairs in our heterogeneous hash tables are partitioned between a list and a CPU based hash table. The list is implemented using an array residing in the unified memory. Unified memory is an abstracted form of memory that can be accessed by both the devices without any explicit data transfers [1]. The support for unified memory is provided with `CUDA 6.0` onwards. This memory is allocated on the GPU initially and when the CPU accesses a chunk of this memory, that chunk is transferred to the CPU by the underlying `CUDA` framework implicitly. The key-value pairs in the list are arranged from the most recently to the least recently accessed pair in the left to right order respectively. The size of the list is $O(M)$ and has three sections: an active middle area which contains all the key-value pairs belonging to the list and empty left and right areas of size B each. The left area stores the most recent query vector after it is being processed by both the devices. This addition of the query vector is done before the MRU list is reorganized. This reorganization will be explained later in the paper. The right area stores overflowing keys from the list. These overflow keys are the oldest keys in the GPU memory and will be accommodated in the CPU memory during successive operations on the hash tables. The rest of the key-value pairs which are older enough and thus can not be accommodated in the MRU list, are maintained in the CPU hash table. The design of this hash table is different for all the cases and will be described in the later sections. Each element in the query vector, a query element, contains a key-value pair. The rightmost three bits in the key are reserved. The first two bits identify the operation being carried out with the key; i.e. a *search*, *insert* or *delete*. The last bit is set if the key-value pair is present in the GPU and vice-versa (Figure 1). The next three sections describe the hash table designs in detail.

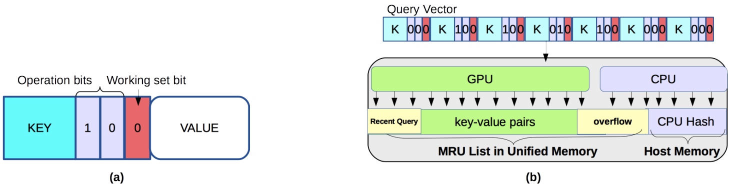

Figure 1: The left figure shows the structure of a query element. The figure on the right shows the overall structure of the hash tables. The value part in the query vector is omitted for simplification.

3.1 A Spliced Hash Table

A spliced hash table (S-hash) is the simplest of our designs where a standard GPU hash table from the `CUDPP` library is fused together with a serial CPU hash table from the `C++ Boost` [16] library, within the framework described in the previous section. The GPU hash table (`CUDPP` hash) is separate from the MRU list and is maintained as a separate data structure in the GPU memory. The `CUDPP` hash processes *search* and *update* queries in batches. The `CUDPP` hash and the `Boost` hash communicate through the MRU list in the unified memory. Recall that the MRU list contains $O(M)$ slots. To identify the position of a key in the MRU list $O(\log M)$ bits are stored along with the value part. These position bits link a key-value pair in the `CUDPP` hash to its location in a specific slot of the MRU list.

Operations: The operations are bundled in a query vector first and sent to the `CUDPP` hash for processing. The working set bit is set for each `insert(key)` operation in a query element. The `CUDPP` hash can not handle mixed operations. Hence the query elements with search operations are separated from the delete operations before processing. Each GPU thread handles an individual query element. For a search query, if the key is found in the `CUDPP` hash, the position bits located in the value field are read and the working set bit corresponding to the location of the key in the MRU list is set to 0 and the working set bit in the query element is set to 1. Delete queries are handled similarly without setting the working set bit in the query element after a successful removal. The search and delete queries that could not be serviced by the GPU are sent to the CPU. The CPU takes one query at a time and executes the corresponding operation on the `Boost` hash. The setting of the working set bit in the query element is done as before. The query vector is now copied to the leftmost section in the MRU list. This copying can be avoided if the query vector is placed in this section of the MRU list at the beginning only. To prevent duplicate keys in the hash tables, the query vector is scanned for repeated keys with the working set bit set. If duplicates are found, the working set bit is set for one key and unset for the rest. The keys in the query vector whose working set bit is set and is not available in the `CUDPP` hash are added to the hash table in a separate batch. Now a *Reorganize* operation is executed on the list which arranges the keys in the MRU order (Figure 3).

Reorganize: This operation shrinks the MRU list by removing all the keys in the list whose working set bits are set to 0. Figure 2 shows an instance of this operation. The MRU list with the associated left and right sections is shown along with an example query vector with insertion of the key 98 and search for the other keys. After the addition of the query elements to the leftmost section of the MRU list, an exclusive prefix scan operation is carried out on the list. This prefix scan is carried out using the working set bit values. The overflow area where the working set bits are set to X, is not included in the prefix scan. The keys are then packed using the indices returned by the scan operation. The index starts at B, where B is the size of the query vector. If a key overflows to the overflow section due to addition of a new key to the MRU list (key 98 in

Figure 2), then it gets added by the CPU exactly when the CUDPP hash starts processing the next batch of queries. Since this overflow section belongs to the MRU list in unified memory, the CPU can read these keys without any explicit data transfers. The prefix scan is carried out in-place using a scan operation in the Thrust high performance GPU library [8].

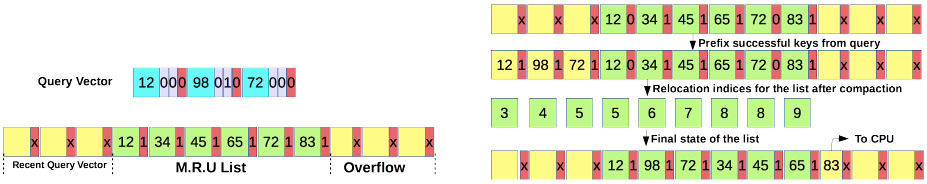
Query Vector
12 0 0 0 98 0 1 0 72 0 0 0
x x x 12 1 34 1 45 1 65 1 72 1 83 1 x x x
Recent Query Vector
M.R.U List
Overflow
x x x 12 0 34 1 45 1 65 1 72 0 83 1 x x x
Prefix successful keys from query
12 1 98 1 72 1 12 0 34 1 45 1 65 1 72 0 83 1 x x x
Relocation indices for the list after compaction
3 4 5 5 6 7 8 8 9
Final state of the list
To CPU
x x x 12 1 98 1 72 1 34 1 45 1 65 1 83 x x x

Figure 2: The figure shows a reorganize operation on a simple MRU list with the overflow and the query sections. The value part is omitted for simplification.

3.2 A Simplified S-Hash Table

The simplified S-hash table (SS-hash) eliminates the CUDPP hash and operates on the MRU list directly. The step to separate the queries based on operations is no longer necessary as the GPU now handles mixed operations together in one batch. The algorithm for maintaining the MRU list in the GPU remains the same. The only difference is the replacement of the CUDPP hash with our MRU list processing logic described below (Figure 3).

MRU List Processing: After the query vector is filled with query elements, a GPU kernel is launched. The thread configuration of the kernel is adjusted to launch W warps, where W equals the size of the active middle section in the MRU list. A warp is assigned a single MRU list element. Each block in the kernel loads a copy of the query vector in its shared memory. The i^{th} warp processes the $j \times 32 + i^{th}$ key in the MRU list, here j is the index of the block containing the warp and each block has a maximum capacity of 32 warps. The warp reads the assigned key in the list and all the threads in the warp linearly scan the query vector in shared memory for the key. If a thread in the warp finds a match, it first reads the operations bits to identify if it is a search or a delete operation. For a successful search operation, a thread sets the working set bit of the key in the query vector and unsets the corresponding bit in the MRU list. The thread uses the copy of the query vector in the global memory for this step. This bit manipulation is done using the bit-wise OR/AND primitives. For a successful delete, the key-value pair along with the working set bit in the MRU list is set to 0. The working set bit in the query vector is left unset. The success of a concurrent search for the same key that is getting deleted is determined by whether the search read the key before the delete started modifying the bits in the key-value pair. Insert operations need not be processed as they will be taken care by the reorganize step that was described before. This is a pretty straightforward code without any optimization. Listed below are some optmizations which are intrinsic to our algorithm and some others which are incorporated with minor modifications.

- **Memory Bank conflicts:** Bank conflicts within a block occur when a few threads within a warp read the same shared memory location. In our algorithm all the warp threads read adjacent memory locations of the shared memory therefore preventing bank conflicts.
- **Global Memory coalescing:** Coalescing happens when all the threads from a warp read successive global memory locations. Consequently all the read requests are served by a single memory transaction. In our algorithm all the warp threads read a single global memory location so the scope of coalescing reads is lost. As an optimization, before all the warps in a block starts reading keys from the MRU list, warp 0 in each block reads in a set of contiguous keys from the MRU list and places them in a shared memory buffer. After this the warps, including warp 0, starts executing and fetching the keys from this buffer instead of the MRU list.
- **Warp Serialization:** If the threads from two different warps read the same shared memory location, the two warps are scheduled one after another on the respective SM. There is a high probability for this to happen as all the warps within a block scan the query vector linearly starting from the beginning. To reduce this probability each warp chooses a random location in the query vector to start the scan from and wrap around in case it over flows the size of the query vector.
- **Launch configuration:** The number of warps, thereby blocks, launched can be reduced if more work is assigned to a single warp. Instead of processing a single key from the MRU list, each warp can pick up a constant number of keys to look for inside the query vector.
- **Redundant work:** There might be scenarios where all the query elements are serviced by a very small fraction of the warps while the majority of the warps do redundant work of simply reading the query vector and the keys before expiring. To combat this issue, each warp on successfully processing a query decrements an global atomic counter initialized to B. Now a warp only starts its execution cycle if the value of this counter is greater than 0.

In this design, the `Boost` hash is replaced by a lock free cuckoo hash from [14]. The overflow keys are now added in parallel by individual CPU threads to the CPU hash table.

3.3 A Cache Partitioned SS-hash Table

In this section, the focus is shifted to the CPU hash table design. The lock free cuckoo hash is replaced by our own implementation in the SS-hash table. The developed hash table is optimized for multi-core caches using the technique of partitioning, hence we call it `CPSS` hash table. The work in [12] designed a shared memory hash table for multi-core machines (CPHASH), by partitioning the table across the caches of cores and using message passing to transfer

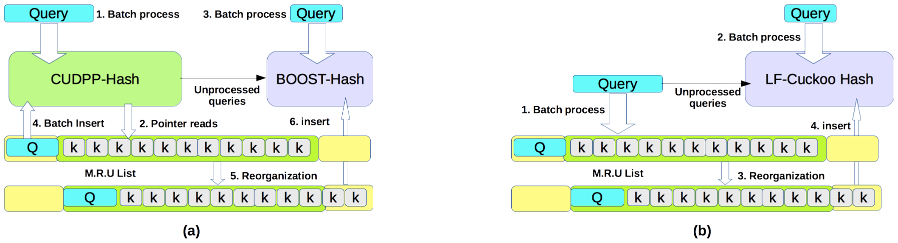

Figure 3: The overall design of the S-hash and SS-hash tables.

search/insert/delete operations to a partition. Each partition was handled by a separate thread on a core. We design our hash table along similar lines. The hash table processes queries in batches and operates in a rippling fashion during query processing.

Our hash table is implemented using a circular array. The array housing the hash table is partitioned into $\mathtt{P}$ different partitions, $\mathtt{P}$ is the number of CPU threads launched. In each partition $\mathtt{P}[\mathtt{i}]$, a fixed number of buffer slots, $\mathtt{R}[\mathtt{i}]$, are reserved. The rest of the slots in each partition are used for hashing the keys. Within a partition the collisions are resolved using closed addressing. A mixed form of cuckoo hashing and linear probing is used. Each partition uses two hash function $\mathtt{h}_1$ and $\mathtt{h}_2$, each operating on half the slots reserved for hashing. Each partition is serviced by a thread and handles $\frac{\mathtt{Q}}{\mathtt{P}}$ queries, $\mathtt{Q}$ is the total number of queries batched for processing on the CPU.

Operations: The query element $\mathtt{m}$ in the batch is assigned to the partition $\mathtt{k} = \mathtt{H}(\mathtt{m})\%\mathtt{P}$, here $\mathtt{H}$ is a hash function and $\mathtt{H} \neq \mathtt{h}_1, \mathtt{h}_2$. The assignment is completed by writing the contents of the query element to a slot in $\mathtt{R}[\mathtt{k}]$. After this the threads execute a barrier operation and come out of the barrier only if there are no more entries in the buffer slots of each thread's partition. Each thread $\mathtt{i}$ reads a key from its buffer and hashes it to one of the hashing slots using $\mathtt{h}_1$. If the slot returned is already full, the thread searches for an empty slot in the next constant number of slots using linear probing. If this scheme fails the threads replaces the last read key from its slot and inserts its key into this slot. The ”slotless” key is hashed in the other half of the partition using $\mathtt{h}_2$. The same process is repeated here also. If the thread is unsuccessful in assigning a slot to the removed key, it simply replaces the key from last read slot and inserts the just removed key in $\mathtt{R}[\mathtt{i} + 1]\%\mathtt{P}$. The insertion to the buffer slots of an adjacent partition is done using lock free techniques. All the insertions and deletions happen at the starting slot of these buffers using the atomic compare-and-swap primitive. This is the same mechanism used by a lock free stack [6]. For *search* and *delete* operation, each thread probes for a constant number of slots within its partition. Unsuccessful queries are added to the adjacent partition's buffer slots for the adjacent thread to process. There is a major issue with concurrent cuckoo hashing in general. A search for a specific key might be in progress while that key is in movement due to insertions happening in parallel, thus the search returns false for a key present in the hash table. Note that in our case the overall algorithm for the hash tables is designed in such a way that the CPU side insertions always happens in a separate batch before the searches and deletes.

4 Performance Evaluation

This section compares the performance of our heterogeneous hash tables to the most effective prior hash tables in both sequential and concurrent (multi-core) environments. The metric used for performance comparison is query throughput which is measured in Millions of Queries Processed per Second (MQPS).

The experimental setup consists of a NVIDIA Tesla K20 GPU and an Intel Xeon E5-1680 CPU. Both the CPU and GPU are connected through a PCIe express bus with 8GB/s peak bandwidth. The GPU has 5GB of global memory with 14 SM and 192 cores per SM. The GPU runs on latest `CUDA` framework 7.5. The host is a 8 core CPU running at 3.2 GHz with 32 GB RAM. The CPU code is implemented using the latest `C++11` standard. All the results are averaged out over 100 runs. Our hash tables are compared against a concurrent lock free cuckoo hash implementation (LF-Cuckoo) from [14] and a serial hash table from the `Boost` library. For completeness we also compared the results with *Lea's* concurrent locked (LL) hash table.

4.1 Query performance

We use micro-benchmarks similar to the works in [7],[14]. Each experiment uses the same data set of 64 bit key-value pairs for all the hash tables. The results were collected by setting up the hash tables densities (load factor) close to $90\%(0.9)$. Figure 4 compares the query throughput of the hash tables on 10M queries. All the hash tables are filled with 64M key-value pairs initially. The results are shown for two types of query mixes, one has a higher percentage of search queries and the other has more update operations. Two types of key access patterns are simulated for the search and delete queries. A *Uniform* distribution generates the queries at random from the data set while the *Gaussian* distribution generates queries where a fraction of the keys are queried more than the others. The standard deviation of the distribution is set such that 20% of the keys have higher access frequency. As each warp in the GPU processes a single MRU list element, it is treated as a single GPU thread in the plots. The number of $\{\texttt{GPU}, \texttt{CPU}\}$ threads is varied linearly from $\{32, 4\}$ to $\{1024, 128\}$. The size of the MRU list is fixed at 1M key-value pairs and each query vector has 8K entries. The `Boost` hash being a serial structure always operates with a single CPU thread.

As can be seen in Figure 4, for search dominated uniform key access patterns the heterogeneous hash tables out performs the `Boost` hash and *Lea's* hash and the throughput scales with the increasing number of threads. The hash tables has the same query throughput compared to the lock free cuckoo hash. For the insert dominated case, the heterogeneous hash tables out performs all the other hash tables. The reason is the simplified insert operation where the simple reorganize operation inserts the keys into the MRU list and thereby to the hash tables. The CPU only handles the overflow inserts which have less probability of occurrence. For the *Gaussian* distribution case, our hash tables out performed the others by a significant margin. They can process `10` times more queries compared to the `Boost` hash and `5` times more compared to the lock free cuckoo hash. The frequently accessed keys are always processed on the GPU. The CPU only processes the unprocessed queries in the query vector and the overflow keys. The probability of CPU doing work is less, as most of the queries are satisfied by the GPU without generating any overflow. Figure 5 shows the cache misses per query for the *Uniform* and the *Gaussian* distribution case. The `CPSS` hash has fewer number of cache misses compared to the `Boost` hash and the lock free cuckoo hash. As the GPU has a primitive cache hierarchy and most of the memory optimizations has already been taken care of, only the CPU cache misses are reported. In the *Gaussian* distribution case the `CPSS` hash performs much better compared to the other case as most of the queries are resolved by the GPU itself and the CPU has less work to do.

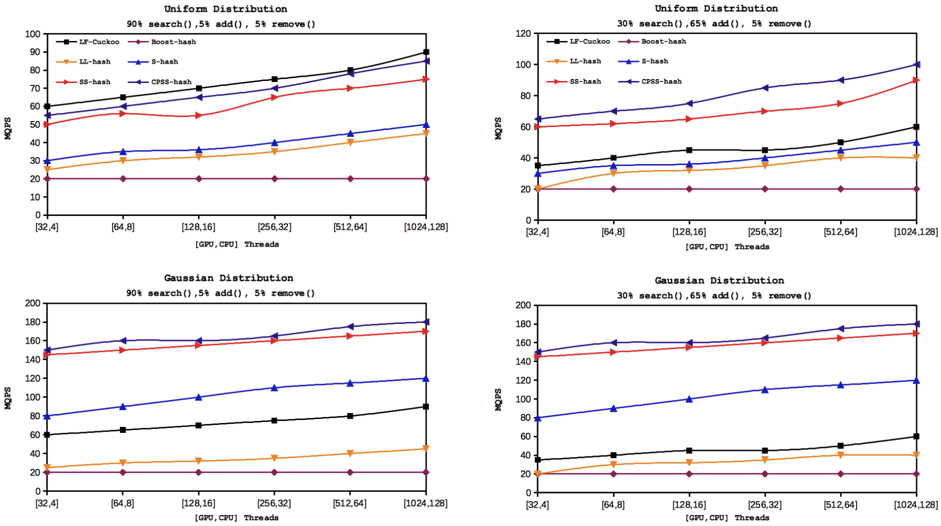

Figure 4: The query throughput of the heterogeneous hash tables on different key access distributions and query mixes.

4.2 Structural Analysis

The experiments in this section are carried out on the `CPSS` hash to find out the reasons for the speed up that was reported earlier. In Figure 6 the number of queries were varied with $\{1024, 128\}$ threads in total. The other parameters are same as before. As can be seen, for the uniform scenarios half the time is spent on memory copies. These cover both `DeviceToHost` or `HostToDevice` implicit memory transfers. These memory transfers were captured with the help of the

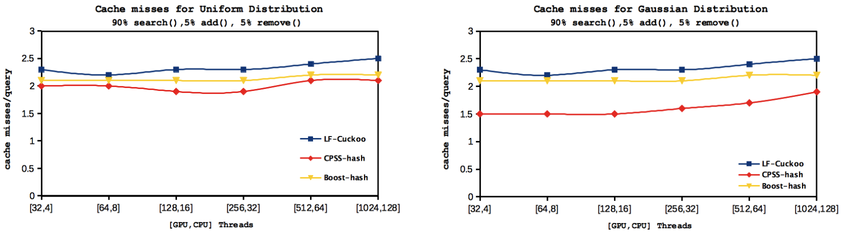

Figure 5: The cache misses/query comparison for the hash tables.

CUDA profiler. In the *Gaussian* distribution case the GPU performs most of the work with minimum memory transfer overhead and hence the expected speed up is achieved. As can be seen in Figure 6 the maximum throughput is achieved when our hash tables are configured with MRU list and query vector size of 1M and 8K respectively. With increasing size of the MRU list and the query vector, the time spent by the GPU in the *Reorganize* operation and the time for the DeviceToHost memory transfers increases. This is the reason for the diminishing query throughput at higher values of these parameters.

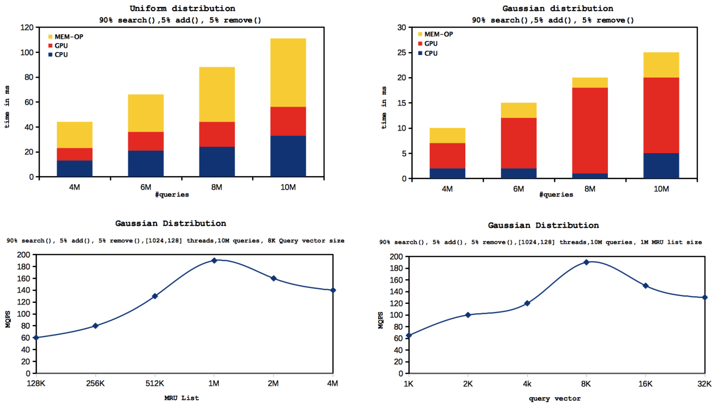

Figure 6: The top two graphs shows the time split of the CPU,GPU and memory transfers for processing different number of queries under different key access distribution. The bottom graphs show the variance of the query throughput with the size of the MRU list and the query vector respectively.

5 Conclusion

In this work, we proposed a set of heterogeneous working-set hash tables whose layout spans across GPU and CPU memories, where the GPU handles the most frequently accessed keys. The hash tables operates without any explicit data transfers between the devices. This concept can be extended to any set of interconnected devices with varying computational powers where the most frequently accessed keys lies on the fastest device and so on. For non-uniform key access distributions, our hash tables out performed all the others in query throughput. In our future work, we plan to investigate the challenges involved in using multiple accelerators including GPUs and FPGAs. We envisage that maintaining a global MRU list spanning across all the devices could be computationally expensive. So suitable approximations that give the right trade-off have to be made.

References

1. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.
2. D. A. F. Alcantara. *Efficient Hash Tables on the Gpu.* PhD thesis, Davis, CA, USA, 2011. AAI3482095.
3. Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan. GpuCV: A GPU-accelerated framework for image processing and computer vision. In *Advances in Visual Computing*, volume 5359 of *Lecture Notes in Computer Science*, pages 430–439. Springer, Dec. 2008.
4. M. Bdoiu, R. Cole, E. D. Demaine, and J. Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theor. Comput. Sci.*, 382(2):86–96, Aug. 2007.
5. M. Daga and M. Nutter. Exploiting coarse-grained parallelism in b+ tree searches on an apu. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 240–247, Washington, DC, USA, 2012. IEEE Computer Society.
6. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. ACM.
7. M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *22nd Intl. Symp. on Distributed Computing*, 2008.
8. J. Hoberock and N. Bell. Thrust: A parallel template library, 2010. Version 1.7.0.
9. M. Kelly and A. Breslow. Quad-tree construction on the gpu: A hybrid cpu-gpu approach. *Retrieved June13*, 2011.
10. F. Khorasani, M. E. Belviranli, R. Gupta, and L. N. Bhuyan. Stadium hashing: Scalable and flexible hashing on gpus. 2015.
11. D. Lea. Hash table util.concurrent.ConcurrentHashMap, revision 1.3, in JSR-166, the proposed Java Concurrency Package.
12. Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. Cphash: A cache-partitioned hash table. *SIGPLAN Not.*, 47(8):319–320, Feb. 2012.
13. M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.
14. N. Nguyen and P. Tsigas. Lock-free cuckoo hashing. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 627–636. IEEE, 2014.
15. R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.
16. B. Schling. *The Boost C++ Libraries.* XML Press, 2011.
17. D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.