A Hybrid (CPU+GPU) Working-set Dictionary

by

Ziaul Choudhury, Suresh Purini

in

15th International Symposium on Parallel and Distributed Computing (ISPDC 2016).

Report No: IIIT/TR/2016/-1

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

A Hybrid CPU+GPU Working-set Dictionary

Ziaul Choudhury*, Suresh Purini†, Shiva Rama Krishna‡
*Siemens Corporate Research, Bangalore, India
Email: ziaul.choudhury@siemens.com
†International Institute of Information Technology, Hyderabad, India
Email: suresh.purini@iiit.ac.in
‡Siemens Corporate Research, Bangalore, India
Email: i.sivaramakrishna@siemens.com

***Abstract*—In this paper, we propose a hybrid CPU+GPU data structure, that optimizes search operation for frequently accessed search keys. This is based on the working-set structure due to Bdoiu et al. [1]. The main 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, search queries are processed in batches of size 1K to 16K ($\mathbf{K} = \mathbf{2^{10}}$). We measured the query throughput of our data structure using Millions of Queries Processed per Second (MQPS) as a metric, on different key access distributions. On distributions, where some keys are accessed more frequently than others, we achieved *2x* higher MQPS when compared to a highly tuned hash map provided by C++ BOOST library, and *1.5x* higher MQPS against the B+ tree implementation in the Rodinia GPU benchmark. We further showed the effectiveness of our structure, when it is used to store visited vertices information in breadth-first search traversal of graphs. Here, we achieved *1.2x* and *1.5x* speedups when compared to the BOOST hash map and the GPU B+ trees respectively.**

I. Introduction

A dictionary data structure is a key-value store which supports insert, delete and search operations based on keys. AVL trees, B+ trees and hash tables are examples of such data structures. These structures do not lay any special emphasis on search operation which could be the more common operation in many applications. Further, the search algorithms used are oblivious to any key access patterns. However, the key access sequences in real world applications tend to have some structure. For example, during a window of time a small subset of keys could be more frequently accessed than the rest. We can design data structures which use the structure in the key access sequences to give sub-logarithmic search times. For example in splay trees [2], keys which are recently used tend to be closer to the root node and hence have lesser search times when compared to keys residing in leaf nodes. The *working set* property defined below, captures one such access pattern optimization.

- Working set property: The working set property states that it requires at most $O(\log[\omega(x) + 2])$ time to search for an element x, where $\omega(x)$ is the number of distinct elements accessed since the last access of the element x.

Intuitively, the working set property states that recently accessed elements are cheap to access again. Whereas the splay trees satisfy the working set property in an amortized sense [2], the working-set structure satisfies the same in the worst case sense [1]. The working-set structure is an ordered collection of balanced binary search trees. It forms the basis of our current work and is described in detail in Section 2.

There has been several attempts to design data structures which effectively use the underlying architectural features of CPUs and GPUs. Kim et al. [3] proposed a cache aware memory layout for storing binary search trees in main memory and further used SIMD units within a CPU to expedite search operations. The proposed memory layout is similar to the Van Embde Boas layout used in cache oblivious search trees [4]. Fix et al. [5] proposed search and range finding algorithms on B+ trees using GPUs by performing parallel key comparisons within a single B+ tree node. However, there has not been any work on architecture aware working-set structures for CPUs, GPUs or their hybrid combination. Based on this, we claim that the following are the main contributions of this paper.

1) We proposed a hybrid CPU+GPU working-set dictionary data structure which optimizes search operation for frequently accessed search keys. The search queries are bundled for batch processing.
2) Second, we compared the performance of the proposed structure in terms of Millions of Queries Processed per Second (MQPS) against pure CPU based working-set structures, AVL trees and BOOST hash map; and GPU based Rodinia B+ trees on different key access distributions.
3) Finally, we studied the impact of our structure in an application such as breadth-first search graph traversal wherein the newly discovered nodes will be accessed again for neighborhood exploration in the near future.

The layout of the paper is as follows. Section 2 provides necessary background on GPUs, CPU+GPU heterogeneous computing and the working-set dictionary data structure. We propose our main data structure design in Section 3 and the corresponding optimizations in Section 4. Section 5 contains experimental results and finally we conclude with Section 6.

II. Background

In this section, we present the necessary background information on the heterogeneous CPU+GPU computing framework and the working-set dictionary data structure.

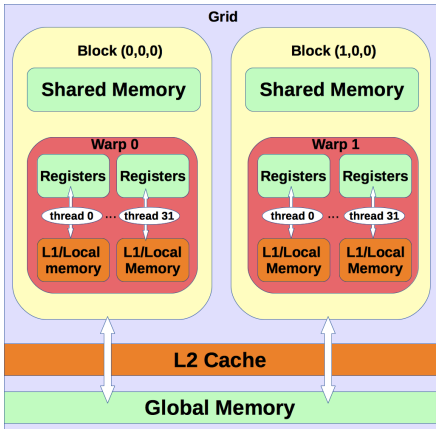

Fig. 1. A schematic diagram of the CUDA hardware model.

A. Heterogeneous computing

Graphics Processing Units (GPUs) are finding a place in wide spectrum of devices ranging from mobile phones to super computers due to their remarkable performance-price ratio and competitive performance-per-watt in some GPU variants. They can be used for general purpose computation (GPGPUs) also using programming frameworks such as CUDA [6] and OpenCL [7]. Among the many applications ported onto these GPUs, database engines are relevant to the current work discussed in this paper. GPUs have been used to accelerate database operations like select and join by storing the keys on them using indexing structures like hash tables or B+ trees [8], [5]. Before we discuss the issues involved in porting such structures to GPUs, we briefly explain the typical GPU architecture below.

1) GPU Architecture: GPUs are composed of multiple streaming multiprocessors (SMs). Each SM in turn contains a number of light weight primitive cores (refer Figure 1). Whereas the SMs execute in parallel independently, the cores within a SM consume the same instruction stream generated by the threads running on the SM in a lock step fashion. This is the Single Instruction Multiple Thread (SIMT) model which is similar to the SIMD model. A high end Kepler GPU from NVIDIA, for example, has an overall core count of 2880 which are distributed equally among multiple SMs. The memory subsystem is composed of a global DRAM and an L2 cache shared by all the SMs. There is also a small software managed data cache called shared memory attached to each SM and as the name suggests is shared by all the cores within an SM. The access time of the shared memory is close to that of registers.

2) Programming Model: A compute kernel on a GPU is organized as a collection of thread blocks. Each thread block can be 1D, 2D or 3D and every thread within the block is given a unique coordinate the dimension of which depends on the block organization. Thread blocks are in turn organized into a grid of 1D, 2D or 3D dimension. Every thread is uniquely determined based on its grid and block coordinates. All the threads within the same block are scheduled for execution on the same SM. However, different thread blocks can be scheduled on different SMs. A group of 32 threads within a block, called a warp, runs in lock step. A warp of threads is the basic unit of execution in a GPU. Decomposing a problem on the GPU involves mapping each input point of the program to grids, blocks and warps.

3) Hybrid Computing: The GPU is embedded in a system as an accelerator device connected to the CPU through a low bandwidth PCIe express bus. Hybrid 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 cores, 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. Another solution is to overlap the data transfer operations with computation, using an asynchronous API in the CUDA framework. The CPU after transferring the data and launching the kernel mostly sits idle during the computation. The main objective behind the hybrid computational model is to make both the CPU and GPU contribute towards the computation.

4) Pipelining: Pipelining is a common design strategy used to achieve high throughput at a micro-architectural level in processors, and at a task level in parallel programs involving multiple cores and accelerators. In fact, it has been identified as a common design pattern in parallel computing [9] and has been used in the hybrid set up also [10] in the following way.

1) Overlap the computation and transfer of data between CPU and GPU memory.
2) Execute different parts of an algorithm on either of the devices based on their architecture.
3) Load balance the computation by executing the algorithm in different states on the CPU and the GPU.

In this paper, we use the the first two techniques while designing our pipeline.

5) Discussion on Search Structures: Data structures that use both the CPU and GPU simultaneously have been reported in literature. Kelly and Breslow [11] proposed a hybrid approach to construct quad trees by building the first few levels in the CPU and the rest of the levels in the GPU. Breirbart [12] showed that spatial data structures perform better when partitioned between CPU and 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 [13] 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 RAM. Finally, due to the latency involved in launching a GPU kernel, we have to accumulate many search queries in a query bundle and process them on the GPU in parallel.

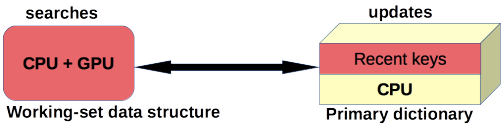

Fig. 2. The figure shows the relation between our working-set structure and the primary dictionary. Each node in the working-set structure has a key and address field, where the address refers to the location of the key with in the primary dictionary.

B. Working-set Structure

A working-set structure on N elements is a dictionary satisfying the working set bound [1]. It consists of $\log\log N$ balanced binary search trees and the elements within each tree are connected through a doubly linked list. Let $T_1, \cdots, T_l$ be the trees and $L_1, \cdots, L_l$ be the corresponding doubly linked lists, where $l = \log\log N$. For $1 \leq i \leq l-1$, $|T_i| = 2^{2^i}$ and $0 < |T_l| \leq 2^{2^l}$. Elements in a tree occur in the most recently used (MRU) order in the corresponding linked list. The element at the head of the linked list is the most recently used element. By concatenating the linked lists from L_1 to L_l, we get a global MRU list of elements. All the operations on the working-set structure are engineered so that this property is satisfied. By maintaining this property, we can find an element x in time bound $\log\omega(x)$, where $\omega(x)$ indicates the position of the element x in the global MRU list. An element is searched in the trees T_1 to T_l, in that order. If the element is found in tree T_i, then it is deleted from T_i, and reinserted into the tree T_1. Before reinsertion, an element has to be moved from tree T_j to T_{j+1}, for $1 \leq j \leq i-1$, in order to maintain the tree size constraints. Whenever an element has to be moved from tree T_j to T_{j+1}, the element at the tail of the list L_j is chosen for deletion and it occupies the head of the list L_{j+1} after insertion into the tree T_{j+1}. During an insert operation, an element is shifted from tree T_j to T_{j+1}, for $1 \leq j \leq l-1$. Then the new element is inserted into tree T_1 and is placed at the head of the list L_1. If the tree T_l reaches it full size during this process, then a new tree will be created. During a delete operation of an element present in tree T_i, an element will be moved from T_j to T_{j-1}, for $i < j \leq l$. The most recently used element in T_j moves to the position of the least recently used element in tree T_{j-1}. If the tree T_l gets empty during this process, it will be destroyed. This structure is designed in the pointer machine model and an analogous structure in the binary search tree model is proposed by Bose et al. [14].

III. Proposed Hybrid Working-set Structure

In this section, we first give an overview of our data structure and its memory layout across both the devices (CPU and GPU). We then describe the algorithms for insert, delete and search operations in detail.

A. Overview

The primary goal of our data structure is to support fast search operations on recently used keys as defined by the working-set property. In fact, we intuitively expect that every key k with $\omega(k) \leq cM$ where M is the size of GPU memory and $0 < c \leq 1$ is some constant, is available in the GPU memory and can be quickly accessed. The value of c depends on the key-value pair record size. Towards this we maintain a primary dictionary which resides completely in the CPU memory and a secondary hybrid working-set structure which spans across CPU and GPU memories, see Figure 2. The insert and delete operations are performed on the primary dictionary, and the search operation is performed on the working-set structure. The search and update operations include in them logic to maintain coherency between the primary and secondary structures.

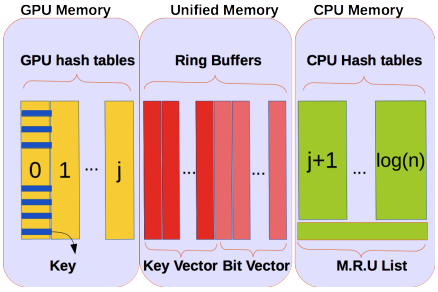

Fig. 3. Layout of the working-set structure across CPU and GPU memories

We used a hash table as our primary dictionary and it can be any of the standard dictionary structures. This hash table completely resides in the CPU memory. The secondary dictionary is a modified working-set structure from Section 2. Every entry in this structure contains a pointer to its equivalent entry in the primary dictionary. Instead of $\log\log N$ trees, the modified working-set structure consists of $\log N$ hash tables such that $|H_i| = 2^i$ for $0 \leq i < \log N$ and $|H_{\log N}| \leq 2^{\log N}$. An MRU list is maintained for keys within each hash table and from which a global MRU list can be deduced. The size of the i^{th} hash table is maintained at 2^i instead of 2^{2^i} as it allows us to use GPU memory more effectively. A key k occupies the position $\omega(k)$ in the global MRU list and will be present in the $H_{\log\omega(k)}$ hash table. The working-set structure is distributed across GPU and CPU memory as follows.

1) *GPU Memory:* The first j hash tables such that $\Sigma_{i=0}^{j}|H_i|$ is less than the available GPU memory reside in it. We flatten out all these hash tables into one contiguous array H. The keys within H are sorted according to their access time stamp.
2) *CPU Memory:* The rest of the hash tables from H_{j+1} to $H_{\log N}$ are stored in the CPU memory.

The CPU and GPU communicate through a set of four ring buffer queues stored in the unified memory which both the devices can access simultaneously. Figure 3 shows the memory layout of our modified working-set structure stored across CPU and GPU memories.

B. Search and Update Operations

The input search queries are batched into a query bundle Q of size $|Q| = q$. The search algorithm is modeled as a two stage pipeline consisting of the stages *gpu_search* and *cpu_search*. These two stages are scheduled for computation on the GPU and CPU respectively. The *gpu_search* and *cpu_search* operations communicate via the following four bounded buffer ring queues located in the unified memory, see Figure 4.

1) *gpu_in:* It is an input queue to the GPU wherein each entry is a vector of q keys corresponding to a query bundle.
2) *gpu_out: gpu_search* communicates the output of its search operation to *cpu_search* through this queue. Each entry of this queue is a vector of size q. Each element of this vector is a tuple (b, ptr) where $b \in \{0, 1\}$ is equal to 1 if and only if the corresponding key in the query bundle is available in the GPU memory. If $b = 1$, then ptr indicates the location of the key in the primary dictionary.
3) *cpu_in:* As the GPU memory gets populated with the most recently used keys, it may become necessary to overflow some keys back to CPU memory. This overflow happens in batches of size q keys each. Each entry in the *cpu_in* queue corresponds to such a key overflow batch.
4) *cpu_out:* During a transient period of time, a key may be present in the MRU list of the GPU but not in the primary dictionary. This happens as the GPU optimistically enters a search key in its MRU list. However, *cpu_search* validates every successful search operation of GPU on a key by checking the primary dictionary. A search operation is complete only after a successful validation step by CPU. Further, if a key is not found in the primary dictionary, then this information is communicated to GPU, via *cpu_out* queue, upon which it will delete the optimistically entered key from its MRU list.

***gpu_search**:* Each query bundle is first processed by the *gpu_search* operation by consuming it through the *gpu_in* queue. It also reads the *cpu_out* queue to correct its MRU list which is optimistically updated during an earlier search operation. Let us denote the GPU MRU list as H. *gpu_search* performs its function by launching the following GPU kernels from CPU.

1) Pre-processing Kernel: First a *cpu_out* element is dequeued. It is bit vector which contains whether a key obtained by GPU from a previous search is valid. If a key is not valid, then the corresponding key in the MRU list of GPU is marked as empty. The position of the key in the MRU list is obtained by using a relative indexing scheme which is explained later in the paper. The i^{th} bit in the bit vector is processed by the i^{th} thread of the kernel. We keep track of the number of keys and empty spaces available in the MRU list by maintaining counters.

After fixing the MRU list, the threads move on to create a hash table Q^h, which contains the keys from the query bundle. Each thread i inserts the i^{th} key from the query bundle into Q^h by applying a hash function and any collisions are resolved through linear probing [8]. The hash table Q^h is stored as an array in the GPU global memory.

2) Search Kernel: The search kernel is launched after the pre-processing kernel finishes execution. Each thread t of the GPU is mapped to an index $m(t)$ of the MRU list H where m is a mapping function. The mapping function is designed in such a way that all the threads within a warp map to the same index of H. If w is the warp size, each thread i, $0 \leq i < w$ within a warp search for the element located at its index, say x, by probing the location $Q^h[\alpha(x) + i]$, where α is the hash function used to create Q^h. If the warp is successful in locating x inside the hash table, the location of H from which the warp read the value of x is marked as an empty space.

After the search process is over, the threads synchronize and the first q threads insert the q keys in the query bundle at the beginning of H. Every invocation of the search kernel leads to addition of q new keys to H. Since the size of H is fixed, at some point we may run out of space. At this point, the q keys at the tail of H will be moved to the *cpu_in* queue and their positions in H are marked as empty spaces. Even if the MRU list has empty spaces available, they may be spread out. We fix this problem by compressing H using a prefix scan operation whenever the empty space count reaches $q \log |H|$. All the threads in the search kernel take part in the shrinking process. The shrinking starts by first creating a temporary array A of size $|H|$. Each entry $A[i]$ is set to 1 if $H[i]$ is a key. It is set to 0 if it is a space. Then a prefix scan operation is done on A. Finally using A all the keys in H are packed in to one contiguous chunk of the GPU memory. The value of $q \log |H|$ is chosen to balance the cost of the prefix scan operation. Since the time complexity of parallel prefix sum is $O(\log n)$, the amortized cost of the prefix scan operation is $O(q)$.

Finally, the *gpu_search* operation ends by inserting an entry into the *gpu_out* queue. That entry is a vector of elements such that the i^{th} element contains the relevant information about the i^{th} key in the query bundle.

***cpu_search**:* This stage handles both the partial working-set structure and the complete primary dictionary residing in the CPU memory. Recall that the partial working-set structure in the CPU memory is a collection of hash tables. We used linear probing to resolve collisions. Since the maximum number of keys that can be accommodated by each of these hash tables is known in advance, their sizes are set so that the collisions and therefore the clustering effects prevalent in the linear probing scheme is minimized. The keys in the hash tables are linked through a double linked list L which maintains the MRU order on the keys.

This stage starts by removing two elements, one each from the two queues *gpu_out* and *cpu_in*. Each element (b, ptr) in the *gpu_out* vector entry is analyzed in parallel by a set of q CPU threads. Each thread executes either of the two steps

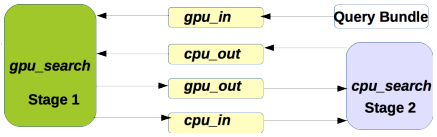

Fig. 4. The pipeline stages in the working-set structure. The direction of the arrows depict the read and the write operations on the communicating buffers in the unified memory.

described below.

- The bit b is set if the key has already been found by the GPU. In this case, the thread uses ptr to check if the element is present in primary dictionary. A key can be found in the optimistic MRU list maintained by GPU but not in the primary dictionary during a transient period. The result of this step is communicated via *cpu_out* queue to fix the GPU MRU list.
- If the bit b is not set, the thread searches for the key in the working-set structure by going through the hash tables H_{j+1} to H_{logN} and finally in the primary dictionary.

After this, the *cpu_in* entry is processed by shifting the overflow keys into the beginning of the working-set structure as described below.

Shift operation: The CPU threads insert the q overflow keys into the hash table H_{j+1}. In terms of the MRU list L, the keys will occupy the front positions and the relative order between them is not strictly enforced to enhance concurrency. Due to this operation, the hash table H_{j+1} may violate its size constraint. In this case, q elements from H_{j+1} will be moved to the next hash table and so on, until the hash table size constraints are no longer violated. We may end up creating a new hash table during this process.

Summary: The query bundles are generated in the CPU and are collected in the *gpu_in* queue. *gpu_search* operation collects the validation feedback from *cpu_search* about query bundles that have already been processed through *cpu_out* queue. If a query bundle occupies the position $gpu\_in[i]$, then its validation feedback from CPU is available at $cpu\_out[i]$. Then the entries corresponding to that query bundle start from the location qi in the MRU list H. These entries are suitably modified using the validation feedback. Then the *gpu_search* operation processes the next available query bundle from the *gpu_in* queue. Meanwhile, the CPU is busy processing elements from *gpu_out* and *cpu_in* inside the *cpu_search* stage. Recall that these queues are populated by the *gpu_search* stage.

Insert(x)/Delete(x)/Search(x): Insertion and deletion operations are performed on the primary dictionary. The search operation starts in the working-set data structure and moves to the primary dictionary in case the search key is not yet a part of the working-set structure. A key found in the primary dictionary but not in the working-set structure is copied into the GPU MRU list at this point of time.

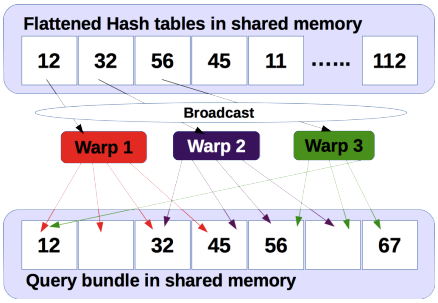

Fig. 5. A view of the search kernel in progress. The GPU hash table and the query bundle is located in the shared memory of each block.

IV. Optimizations

In this section, we describe the optimizations used to further improve the throughput of our hybrid working-set data structure.

A. Overlapping Operations

As the preprocessing kernel reads an entry from *cpu_out* queue and processes, an asynchronous memory transfer operation is initiated to transfer the query bundle from the *gpu_in* queue to the GPU memory (refer Figure 6). Zero copy memory is used to transfer the query bundle from the CPU to the GPU. This is a non paged memory present in the CPU. The size of this memory is set to the size of a query bundle.

B. Memory Optimization

The search kernel reads data from two memory locations, namely H and Q^h. Both of them are stored in the slow GPU global memory. The memory throughput can be improved if these two arrays are maintained in memories closer to the streaming multiprocessor. H is divided into x chunks, where x is the number of blocks inside the search kernel. Each block loads Q^h and the corresponding chunk in H into its shared memory. The shared memory is a highly banked memory unit [6], where the memory addresses are divided into different memory banks. When all the threads in a warp accesses the same location from the shared memory, that memory read operation is broadcasted to all the threads in the warp. On the other hand if only a few threads in the warp access the same location then it leads to bank conflicts which leads to serialization of the memory read operation. A memory broadcast in our structure happens when all the threads inside a warp read the same location from H, which is now located in the shared memory. Also, while reading Q^h the successive threads of a warp read successive locations inside Q^h. This minimizes the bank conflict scenario (refer Figure 5).

V. Results and Discussion

In this section, we show the effectiveness of our hybrid working-set structure on certain key access distributions when compared with other standard dictionary structures. The metric used for performance comparison is query throughput which is measured in Millions of Queries Processed per Second (MQPS). The throughput of our hybrid working-set

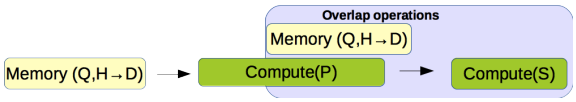

Fig. 6. The figure shows the memory/compute overlap in the *gpu_search* stage. Here P and S denotes the pre-processing and the search kernel respectively. The direction of memory transfer is from the host(H) CPU to the device(D) GPU.

Kepler(K20) GPU parameters

Property	Value
Total amount of global memory	5760 MBytes
# of SM's	14
# cores per SM	192
GPU Clock rate	0.73 Ghz
Memory Clock rate	2600 Mhz
L2 cache size	1572864 bytes
Shared memory per SM	49152 bytes
Max # of threads per SM	2048
Max dim of a block (x,y,z)	(1024,1024,64)
Concurrent copy and kernel exec	yes with 2 copy engines

Fig. 7. The specifications of the Kepler GPU used for the experiments in this paper.

data structure is compared to a hash map implementation provided by the BOOST library [15] and a GPU B+ tree in the Rodinia benchmark [16]. For the sake of completeness, our data structure is also compared to an AVL tree and a simple working-set structure which is described in Section II-B.

A. Experimental Framework

Our experimental setup consists of an NVIDIA Tesla K20 GPU and an Intel Xeon E-5506 CPU. Both the CPU and GPU are connected through a PCI express bus with 8GB/s peak bandwidth. The GPU has 5GB/6GB of GDDR5 memory. It provides upto 3.95 TFLOPS single-precision and 1.33 TFLOPS double-precision floating point performance. An overview of the GPU specifications is presented in Figure 7. The GPU runs on CUDA framework 6.5 with driver version 12.8. The host is an Intel Xeon E-5506 four core CPU running at 2.13 GHz with 4 GB DDR3 SDRAM. The operating system used is a 64-bit version of Linux (Ubuntu 12.4). The CPU threads were created using OpenMP version 4.1. All the results are averaged out over 100 runs.

The primary dictionary linked to our hybrid working-set structure is a hash table on 100M keys. The data structures used for comparison are built on the same key set of 100M keys respectively. The query bundle sizes are varied from 1K to 16K keys in the experiments. The keys in a query bundle are generated using various strategies which are described shortly. These strategies model real world search scenarios.

B. Performance

In our experiments, we set the size of the query bundle to 8K keys. The performance of our hybrid working-set structure on uniform and non-uniform random distributions are presented below. Further, its performance when used in the breadth-first search graph traversal algorithm is also presented.

1) Uniform Random Distribution: A total of 4M to 64M search queries are generated. The keys in the data set are written to a buffer along with some random keys which are not present in the data set. Then a random number generator is used to choose search keys from this buffer. The random number generator ensures that all the keys in the data set are accessed with equal probability.

As can be seen in Figure 8, our data structure has almost the same MQPS value as the GPU B+ tree for lesser number of search queries. The simple working-set tree and the AVL tree have the least MQPS value. The GPU B+ tree has better throughput as the number of search queries increase. Our structure performs better than other CPU based dictionaries and approaches the performance of the GPU B+ tree. This is due to the uniform random key access distribution which does not allow the GPU MRU list to stabilize resulting in continuous memory transfers between CPU and GPU. However, if the dictionary size is small enough to fit into the GPU memory then the performance would be independent of the key access distribution subject to other in-built synchronization overheads.

2) Non-uniform Distribution: In this experiment, certain keys are accessed more frequently than others resulting in non-uniform key access distribution. Out of the set of 100M keys, the probability that a key is chosen from a certain set f is set to 0.8. The size of set f is 4M. The number of search queries are varies from 4M to 64M.

From Figure 9, we can observe that our structure processes 94 MQPS on an average while the GPU B+ tree and the BOOST hash map processes 60 MQPS and 42 MQPS respectively. Our structure out performs the GPU B+ tree in this case, as the size of the array H is of the same order as the working set of the primary dictionary. The keys in B accumulate on the GPU after some number of searches. Thus the memory transfer overhead of accommodating the overflowing keys is minimized and the CPU is mostly involved in checking the validity of the keys found in the GPU.

3) Non-uniform Distribution on Graphs: Graph traversal is an important primitive in almost all graph algorithms. Breadth-first search (BFS) is a commonly used traversal procedure. BFS uses a bit vector to store the visited information of the graph vertices. A dictionary is used in cases where the value range[1] of the vertices is very large, as this saves considerable space. Let the output of executing BFS on a graph be an be a array of vertices $v_1, v_2, v_1 ..... v_m$. This array contains the vertices in the order they were visited by the BFS procedure. This array contains many duplicate entries. In this experiment real world graphs from the SNAP [17] data set are traversed in the breadth first order. The bit vector data structure in BFS is replaced by the dictionaries used in this paper including our

[1]Value range here denotes the numeric values used to represent the vertices of a graph.

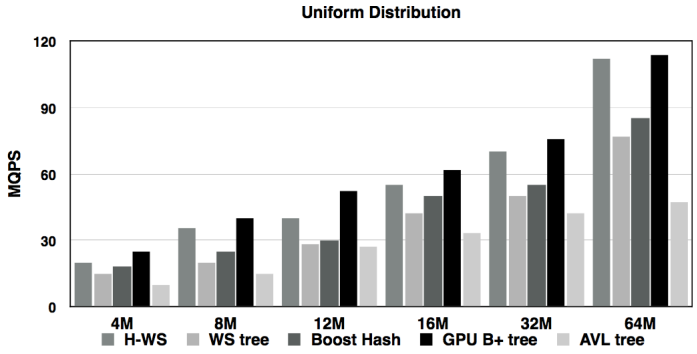

Fig. 8. The graph plots the MQPS values of the various dictionary data structures against the number of search queries processed. Here H-WS is our hybrid working-set data structure.

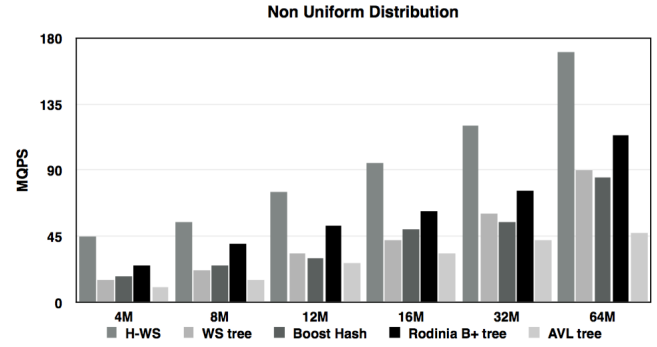

Fig. 9. The graph plots the MQPS values of the various dictionaries data structures against the number of search queries processed. The distribution of the search keys are non uniform as described in the experiment.

hybrid working-set data structure. The graph is stored in the CPU memory and the BFS runs on the CPU. If the dictionary used by BFS has a GPU implementation, then it is invoked. Graphs with varying diameter are chosen from the graph data set. For each graph, a metric G, $G = \frac{v}{d}$, where v is the number of vertices in the graph and d is the graph diameter, is calculated. The BFS timings are reported against this metric. This metric is varied from 10K to 1M in the experiment.

It is observed that the BFS implementation with our structure takes 300 milliseconds on average while the GPU B+ tree and the CPU hash table implementations take 440 and 500 milli seconds on average respectively. The values for the other two dictionary implementations i.e the simple working-set structure and the AVL tree are also provided in Figure 10. The speed up in BFS with our data structure increases with the value of the graph metric. The reason for this behaviour is the temporal locality in the vertex access pattern. When the frontier of a vertex is expanded at a level, all its adjacent vertices get accessed and starts to form a part of the working set. If the G value is high, then the frontier will be large and hence more vertices in the working set. This can lead to better temporal locality which can be well exploited by our hybrid working-set structure.

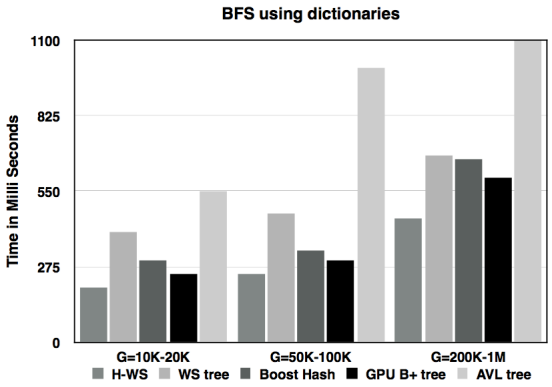

Fig. 10. The graph plots the timings of the BFS procedure against the changing G values. The BFS here uses dictionaries to store the visited information of the vertices.

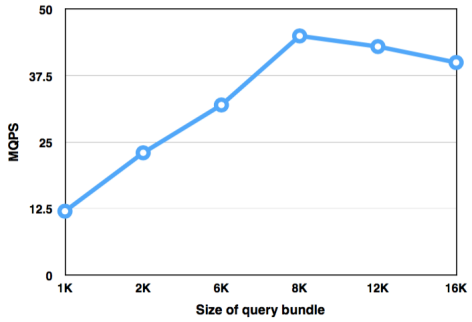

Fig. 11. The figure shows the variance in throughput of the hybrid working-set structure with the change in size of a query bundle.

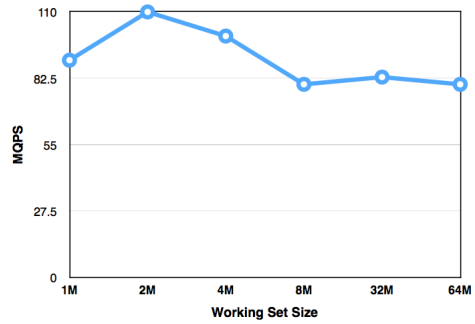

Fig. 12. The figure shows the variance in throughput of the hybrid working-set structure with the change in the working set size of the primary dictionary

Percentage time split

Number of queries	Uniform Distibution			Non Uniform Disribution		
	% CPU	% GPU	% MEM	% CPU	% GPU	% MEM
4M	45	65	10	10	70	8
8M	46	66	12	35	75	10
12M	54	63	17	31	80	11

Fig. 13. The table shows the percentage time split of the devices and the memory transfer operations.

C. Analysis

Figure 11 shows the impact of varying the query bundle size on the throughput of our structure. We perform 8M search queries in the non-uniform search key distribution model, with the working set of the primary dictionary set to 1M keys. The MQPS value steadily increases from 12 to 45 as the the size of the query bundle changes from 1K to 16K. The MQPS value starts to saturate at 45 and possibly decrease after that. The more the number of search keys in the query bundle, the more search keys the data structure is able to process in near constant time. Assumption is that there are enough GPU threads and the keys in the hash table Q^h are evenly distributed. As the size of the query bundle increase significantly, the effort spent by the GPU threads inside the pre-processing kernel to create the hash table increases. The hash table now cannot reside wholly inside the shared memory. It has to be brought in chunks like the array H. The memory copy overhead of copying the query bundle across the CPU and GPU also increases. This factors brings down the overall throughput of our structure.

Figure 12 shows the impact of varying the working set size value of the primary dictionary. We perform 32M search queries with a query bundle size of 8K keys. The MQPS value starts to increase till a point and then it decreases with the increasing value of the working set, i.e. number of frequently accessed keys. This happens since the GPU is no longer able to accommodate the frequently accessed key set after a certain point and starts to spill over the keys over to CPU.

Figure 13 shows the total percentage of time spent by the CPU, the GPU and the memory copy operations, in processing a set of queries. The numbers are reported for the uniform and non-uniform random search key distributions as described earlier. The working set of the primary dictionary was fixed at 4M keys and the query bundle size is set at 8K keys. Both the devices spent almost the same amount of time in the uniform case. In the non-uniform case most of the work happens in the GPU.

VI. Conclusions and Future Work

In this work, we proposed a hybrid working-set dictionary data structure whose layout spans across GPU and CPU memories. The dictionary is organized such that the most recently used keys reside in the GPU memory to the extent it accommodates. The queries are bundled, and first searched in a batch on GPU, and those keys which are not found in the GPU memory are searched for in the CPU memory. We experimentally showed that for key access distributions which show temporal locality property, our proposed structure performs better than pure CPU based data structures such as AVL trees, working-set structures and hash maps; and pure GPU based B+ trees. We further explored the effectiveness of our structure by using it in the breadth-first search graph traversal algorithm which shows locality properties while exploring the graph nodes.

In our future work, we plan to investigate the challenges involved in using multiple accelerators including GPUs and FPGAs[18], [19]. 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] M. Bdoiu, R. Cole, E. D. Demaine, and J. Iacono, "A unified access bound on comparison-based dynamic dictionaries," *Theor. Comput. Sci.*, vol. 382, no. 2, pp. 86–96, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2007.03.002

[2] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, pp. 652–686, Jul. 1985. [Online]. Available: http://doi.acm.org/10.1145/3828.3835

[3] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "Fast: Fast architecture sensitive tree search on modern cpus and gpus," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 339–350. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807206

[4] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious b-trees," *SIAM J. Comput.*, vol. 35, no. 2, pp. 341–358, Aug. 2005. [Online]. Available: http://dx.doi.org/10.1137/S0097539701389956

[5] J. Fix, A. Wilkes, and K. Skadron, "Accelerating braided b+ tree searches on a gpu with cuda."

[6] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007. [Online]. Available: http://developer.download.nvidia.com/compute/cuda

[7] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2010.69

[8] D. A. F. Alcantara, "Efficient hash tables on the gpu," Ph.D. dissertation, Davis, CA, USA, 2011, aAI3482095.

[9] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, *Sourcebook of parallel computing*. Morgan Kaufmann Publishers San Francisco, 2003, vol. 3003.

[10] F. Khalid, F. Feinbube, and A. Polze, "Hybrid CPU-GPU Pipeline Framework," in *20th International Conference on Parallel and Distributed Processing Techniques and Applications*, 2014.

[11] M. Kelly and A. Breslow, "Quad-tree construction on the gpu: A hybrid cpu-gpu approach," *Retrieved June13*, 2011.

[12] J. Breitbart, "Data structure design for gpu based heterogeneous systems," in *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*. IEEE, 2009, pp. 44–51.

[13] 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*, ser. SCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 240–247. [Online]. Available: http://dx.doi.org/10.1109/SC.Companion.2012.40

[14] P. Bose, K. Douïeb, V. Dujmović, and J. Howat, "Layered working-set trees," *Algorithmica*, vol. 63, no. 1-2, pp. 476–489, 2012.

[15] B. Schling, *The Boost C++ Libraries*. XML Press, 2011.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2009.5306797

[17] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[18] R. Mueller, J. Teubner, and G. Alonso, "Data processing on fpgas," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 910–921, Aug. 2009. [Online]. Available: http://dx.doi.org/10.14778/1687627.1687730

[19] Y.-H. E. Yang and V. K. Prasanna, "High throughput and large capacity pipelined dynamic search tree on fpga," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 83–92. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723128