B-tree Based Query Engine for GPUs

Shipra Dureja, Pruthvi Vikram, Narender Pulugu,
Mayura Gurjar Cheepinahalli Vasudeva, Muralidhar Reddy Kuluru
Faculty of Computer Science
Otto von Guericke University
Magdeburg, Germany

***Abstract*—With the rapid increase in the size of data being stored in databases, it has become a necessity to implement a highly efficient data retrieval paradigm using powerful data structures and massively parallel computing systems. This laid a foundation to use B-Trees which are dynamic and mutable data structures along with GPGPUs (General-Purpose Graphic Processing Unit) to implement such a paradigm.**

In this paper, we extend the retrieval capacity of one such implementation to perform selection (range search) and aggregation (count and sum) operations using NVIDIA's CUDA programming toolkit. This paper highlights the performance of the implemented database operations in terms of its required computational time with respect to number of search keys in the query. We evaluate our B-tree implementation with synthetically generated random data set.

We observe all our implemented database operations take approximately equal computational time for a given input size of the search keys. We also propose the optimal input size of the search keys for better GPU computation utilization concerning the hardware used. This paper also makes further observations regarding the impact of using randomly generated synthetic data set with non-identical key-value pairs to the identical key-value pairs.

***Index Terms*—B-tree, GPU, CUDA, WCWS, data structure, dynamic, mutable, threads, query, atomic operations**

I. Introduction

The amount of data generated and collected is increasing rapidly, escalating the complexity of handling it. There has been extensive research to accelerate data handling. Recent researches [7], [8] have shown that mutable data structures combined with GPUs have significantly improved the performance of data handling and accessing data. The development of mutable data structures is considered a difficult task due to the large number of memory access restrictions put in place to preserve the parallel computing capacity of the GPU's.

Very recently, few researchers have implemented a B-tree probe on GP-GPU and is proved to be very efficient [7]. However, the implemented probe is limited in the number of database operations. The implemented probe included only basic functionality like insertion and point search. It can only accept identical integer key-value pairs. The practical application of B-trees in GPUs for databases require more database operations and also the ability to work on non-identical key-value pairs.

To integrate the implementation into DBMS, we extended this probe with additional functionalities, precisely for database operations like selection query (range search) and aggregation query (sum and count). We also intended to enhance the probe to work on a synthetically generated data-set.

The further understanding of the paper requires the basic background information about the B-trees and GPUs. The section [II] provides the necessary basic knowledge to the user regarding B-tree capabilities and architecture of the GPU.

As the planned design and implementation uses the existing probe as its base model, all the design choices and strategies employed in the existing probe are strictly adhered to be compatible with it. These design choices of the implementation are further explained in detail in section [III].

The implementation of selection and aggregation operations utilizes the already existing point search operation to compute their respective results. The detailed information on the implementation and its respective pseudo-code are explained in the section [IV]. This section also provides the challenges faced and limitations observed while implementing these operations.

To evaluate the performance of our implementation, we chose the computational time as the metric. The computational time of each query for varying input sizes of search keys is recorded. The performance of queries for synthetically generated data (stored as non-identical key-value pairs) and randomly generated sequence of numbers (stored as identical key-value pairs) are observed individually. Further, a detailed evaluation plan and the observed results are discussed in the section [V].

We observed with our evaluations, all three database operations with the same input size of search keys, take almost equal time to fetch results from the B-tree stored inside GPU. With the increase in the number of search keys up to the size 2^{10}, the computational time increases linearly but not exploiting the GPU parallelism. However, the optimal query size (number of search keys) for our GPU evaluation ranges from 2^{10} to 2^{15}, as the computational time stays approximately the same, hence, utilizing the high parallelism property of GPUs. Further, increasing input size beyond 2^{15}, the computational time almost doubles, decreasing the performance. The detailed variation in the computational time with different search key sizes can be seen in section [VII].

II. Background

A. GPUs

Traditionally GPUs render images and videos. However, current gen GPUs can process non-graphical applications due to their parallel programming capabilities [4]. Therefore, in

this section, we provide a brief background on the execution of a GPU...

The GPUs support several thousands of concurrent threads. These threads are organized into thread groups/blocks. The threads within a thread block share the computation resources such as registers. Each of the thread blocks is divided into multiple schedule units that are dynamically scheduled on the streaming multiprocessors (SMs), each having its own dedicated local resources. The assignment of a thread block to an SM is done by hardware. And, all SMs, therefore all threads have access to some global memory. Each of the threads has low context-switch and low creation time when compared to CPU threads. Databases that are accelerated with GPUs provide significant benefits compared to mainstream databases, notably when repetitive queries on massive amounts of data are required [5] [6] [7].

Thus, we see that GPUs have a complex parallel processing mechanism. Therefore, we need to re-implement B-trees to fit this architecture. In next section, we detail the traditional B-tree structure, which we later modify to fit for GPUs.

To achieve higher GPU performance, we need to fulfill some of the criteria discussed below. The first criterion is to avoid discrepancies between the instructions given to neighboring threads, which can be achieved by avoiding load imbalance and branch divergence between threads. The kernel execution is set to be completed and the resources allocated to its execution are released only after all the threads executing the kernel complete their execution. If there is a branch divergence in one of the threads, though other threads have completed their execution, the resources are still held until the diverged thread completes its execution. Hence, it is necessary to avoid branch divergence to obtain better GPU performance. The second criterion is to reduce the memory transactions needed to access the thread data, which can be achieved by allowing threads to access consecutive memory addresses. Keeping the above two criterion in mind and to obtain higher GPU performance, we opted to use the 'Warp Cooperative Work Sharing strategy' [8] which will be discussed in later stages of the paper.

B. B-Trees

B-trees are dynamic and mutable in nature and can be used to organize and manage large chunks of data [7]. B-trees are generalized binary search trees which allow the nodes to have more than 2 children [1]. Fig [1] shows a very simple structure of a B-tree with a root node and four leaf nodes. The root node has pointers to the leaf nodes and separator keys. The leaf nodes have the data with keys in disjoint key ranges. The separator keys in the root node divide these key ranges. When the number of leaf nodes exceeds the number of pointers and is equal to the number of separator keys, a new layer of 'branch' nodes is introduced. Now, the separator keys of the new branch node divide the key ranges in leaf nodes, while the separator keys of the root node divide the key ranges in the branch node/nodes. For a large collection of data, a B-tree with multiple layers of the branch nodes can be used [2].

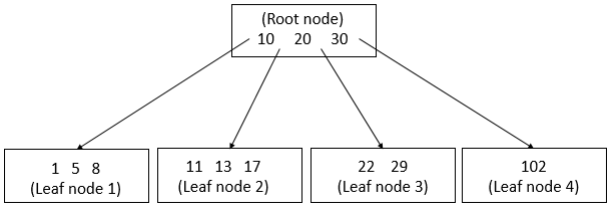

Fig. 1. A simple B-tree with a root node and four leaf nodes

B-trees in databases

B-trees are 'self-balanced', that is, they maintain levels of the index appropriate for the size of the data being indexed [4], making them ideal for use in a database systems. When a B-tree is used to store all columns of a table, it is called a primary index. Here, the data of each row is stored in particular leaf nodes of the B-tree. In a secondary index, each entry will have a reference to a particular record in the primary index. The reference can be a search key or a record identifier. A record is fetched by a root-to-leaf search in the primary index after search in the secondary index. In databases, all the B-tree keys must be unique, even though the user-defined columns are not. For the primary index, this uniqueness is required for correct retrieval while for the secondary index, it is required for correct deletion [3].

III. Design Decisions

The design decisions of our implementation are made upon observing previous works and recommendations from the documentation provided by GPU manufacturer NVIDIA. So, in our design we are considering all the leaf nodes in B-tree will store the key-value pairs, while internal nodes store the pivot-pointer pairs (section III-1), and an additional pair has a pointer to the node's high key and its right sibling offset (side-link strategy of B-link trees). This is required to check by each thread to determine whether the thread is in the intermediate node or the leaf node [7]. Read and write operations are decoupled to enable other warps to continue with their operations while a particular warp is on read mode (section III-2). Proactive splitting is used instead of traditional latch coupling while splitting a node (section III-3). We use the warp-cooperative work-sharing strategy (section III-4), based on generating the work per thread but performing it per warp.

Further this section explains about the detailed significance of the decisions made.

1) Choice of B: The size of a cache line in an NVIDIA GPU is 128 bytes[1]. Each thread in a warp needs 4 bytes (size of integer) of memory, and so, 32 threads can be used to read a node in a coalesced manner. Out of these 32 threads, one is reserved for reading the node's high key (to determine if level-wise traversal is required for each thread) and one more is reserved for reading its right sibling offset. Thus, 8 bytes are consumed, which leaves us with 120 bytes. Each pivot-pointer pair in intermediate nodes or each key-value pair in leaf nodes consumes 8 bytes, which leads us to have the value of B as

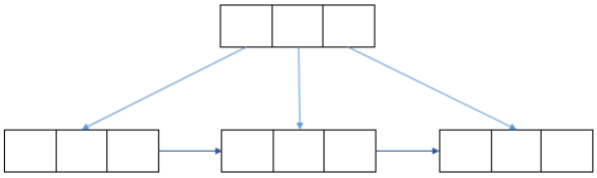

Fig. 2. A B-link-tree Structure (B=3)
Source: Adapted from [7]

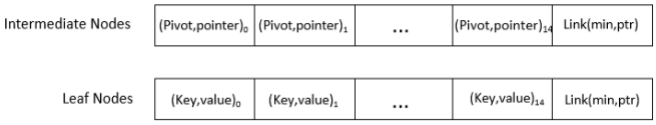

Fig. 3. Our B-tree node structure (B=15)
Source: Adapted from [7]

15. Fig [2] shows us a simple B-tree structure with B=3. Fig [3] shows the B-tree node structure which we have assumed and used in the implementation [7].

2) Decoupled read and write modes: To perform the database operations (range, count, and sum), we require only read operation. Thus, we decouple read and writes. In read mode, warp performing the tree traversal for any update operation does not require latching [7]. Exclusive write latches are used to modify the content in the node. However, once warp decides to switch the mode from read to write, an additional read is required to check whether the node has the most recent content, as the other warps may have later modified the content [7].

3) Side-link strategy and proactive splitting: In a traditional implementation, inserting a new item includes locking a safe path exclusively during insertion. But such locks limits the concurrency and bottleneck the performance of GPUs. This lock can be avoided by using the side-link strategy of B-link-tree [9]. In the traditional method, during a split operation, a node is divided into two and the parent node is updated with the new node information. In case a read occurs after the split, but before the update of the parent node, the read will not be able to find a correct path after that parent node. This can be resolved using the side-link strategy. After the split, if the required item is not in the new left node, the read operation traverses to the new right node. In a B-link-tree, each level acts as a linked list. So, each node in a level stores a key to its right member and that member's lowest key value. Thus, no longer locking the upper node during the split and this makes it easy to maintain level-wise links. Splitting of a node in the B-tree is required when a node is full. In certain cases, the split may propagate till the root node. Latch coupling is the traditional approach to splitting. In latch coupling, an exclusive sub-tree is locked, which starts at a 'safe' node, which guarantees that any splits in the future will not further propagate to the upper part of the tree. But this disallows both read and write operations in that particular sub-tree. This leads to a condition where other threads accessing the particular sub-tree will not be able to perform any operations. This leads to the loss of concurrency ultimately leading to very poor performance in a GPU. An alternative to this latch coupling is proactive splitting. This proactive splitting along with side-links of a B-tree will increase concurrency. In proactive splitting, when there is a need to split a node, the first splitting stage is processed without latching the upper node. But it is latched just before committing the changes to the newly created node and its sibling. Then the parent node is checked, whether it has become full subsequently after committing the changes. This is done using restart as described below. Together, proactive splitting and side-link strategy will help in increasing the concurrency.

4) Warp Cooperative Work Sharing Strategy: GPU works on SIMT (single instruction multiple threads) principles, a single instruction is provided to all the threads of a GPU, and each thread executes this instruction on different data. During the execution of instructions, thread-based operations (intra-thread communication and memory accesses) needs to be controlled.

Warp level cooperative work sharing strategies(WCWS) provide the necessary functionality to work or operate in a group of 32 threads called warps. In this strategy, the complete task is shared to the warps and the execution of each warp is controlled rather than individual threads. It is a bit less flexible than working with individual threads but is very efficient in memory managing.

In the block-level and grid-level strategies, the complete task is shared block wise and grid wise respectively. A block is made of multiple warps and a grid contains multiple blocks. As the hierarchy (number of threads) increases, the flexibility to control the execution decreases. In block-level and grid-level strategies, communication between threads happens through shared and global memories which makes them computationally expensive and making them less suitable for our implementation. Due to these issues with block level and grid level cooperative work sharing strategies, we used warp cooperative work sharing strategy, which offers the required flexibility and has less computational complexity than other available sharing strategies.

The implementation focuses on using GPU for parallelism where each thread tries to generate single and independent access to the B-tree to perform operations. That is, we try to input the work via these threads but, the work is processed inside a warp, which has 32 threads. This means that all the 32 threads inside a warp have individual work, but the warp summarises the individual work of all those 32 threads. These 32 pieces of work are serialized in a queue, working on each piece at a time. This strategy has the following benefits: (i) while reading or writing in a tree node, this helps to achieve coalesced memory access (ii) helps in avoiding thread

[1]Memory transactions: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/sourcelevel/memorytransactions.htm

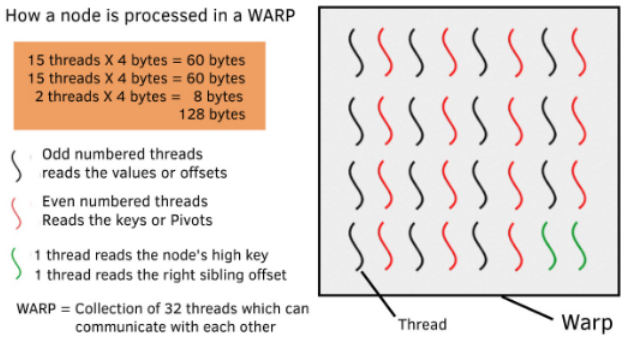

Fig. 4. An example of the WCWS strategy on a B-tree

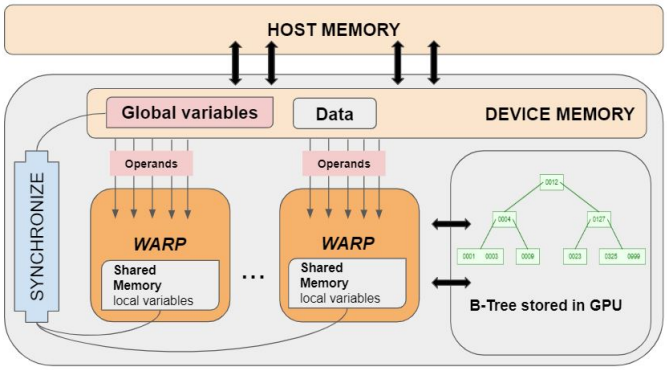

Fig. 5. Internal working of WCWS Strategy

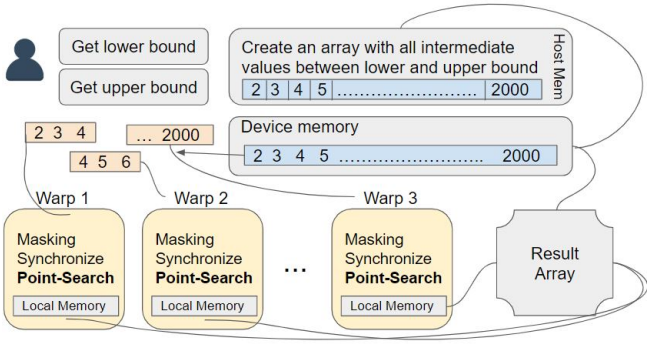

Fig. 6. An example of working of Range Query

divergence inside a warp (iii) helps in reducing the need for load balancing

It allows neighboring threads to perform different operations. Not all resident threads in an SM are executed in parallel. Each SM executes instructions in a warp in Single-Instruction-Multiple-Data (SIMD) fashion. Each thread within a warp fetches 4 consecutive bytes, thus the memory transactions are performed in units of 128 bytes. As a simple example, Fig [4] shows how a node is processed in a warp.

The existing work explains the basic database operations like search, insert, update, and delete on a B-tree in a GPU. It also discusses how a warp traverses the tree by comparing the lookup key and the intermediate node pivots using warp-wide comparison [7]. This lays the foundation for range query processing by explaining how a warp traverse tree searching for the location of the lower bound key and use side-links to perform level-wise traversals until it locates the upper bound key for selection and aggregation operations.

IV. Implementation

As per the choice of warp cooperative work sharing strategy (WCWS), also described in section [III-4], we divided each warp of 32 threads into odd-even pairs. The first 15 odd indexed threads read the values or offsets while the first 15 even indexed threads read the keys or pivots. Remaining 2 threads in the warp read node's high key and right sibling offset respectively [7]. Each thread in warp takes 4 consecutive bytes of memory, making it a total of 128 bytes for performing memory transactions warp-wide. This is incorporated using warp-wide instructions available in the CUDA tool kit.

We are using CUDA's intra-warp communication instructions namely warp voting functions like **__ballot_sync()** and **__shuffle()** for efficient use of WCWS strategy. This is described in the Fig [5]. Using ballot sync, a warp performs reduction operation with the comparison of pivots of its threads, and then the key to be searched is broadcasted to these reduced threads using shuffle.

A. Range Query

Using Fig [6] to describe our implementation design of the range query, we generated all possible key-value pairs between the lower and upper bound values of the given range. The complete search query is then divided between the warps, which means each warp gets an equal part of the search query. The range search is initiated using an existing point search query with multiple threads. Warp uses side links to traverse and find the upper bound of the range.

As the entire process follows WCWS strategy, all the key-value pairs found using these threads are stored at the local memory of each warp. After all the threads of different warps complete their search query, the result present at local memories of each warp is written to global memory using atomic functions (and synchronization) and is then transferred to the host memory. If the key-value pair is not present in the B-tree, we drop that particular thread and no information from it is sent to memory. We have described the implementation of our range query in Algorithm [1].

Limitation

Each thread performs a point search on keys to fetch results of range query instead of providing the lower and upper bound query values to each thread. Key-value pairs are stored in coalesced memory and each warp occupies 128 bytes. Side links fetch the memory addresses in coalesced structure as internally the warps have threads maintaining continuous memory addresses. If the input size of number of search keys (query size) is less than 1024, all the inputs can be run by a single SM within a single block. As all threads in a single block can communicate, we can associate the link between

Algorithm 1: Algorithm for Range Query

Data:
$Input \leftarrow Lowerbound(LB), Upperbound(UB)$
$NumOfQueriesInRange \leftarrow UB - LB + 1$
$resultArr \leftarrow empty[]$
$Allocate \leftarrow MemAlloc(), copytoDevice()$
while *(threadX.Id < NumOfQueriesInRange)* **do**
  searchInBtree(threadX.Id, search)
  // appending the values found to the result array
  if *value is found* **then**
    resultArr.pushback(value)
  else
    Ignore result
Result: $Result \leftarrow copyToHost(resultArr)$

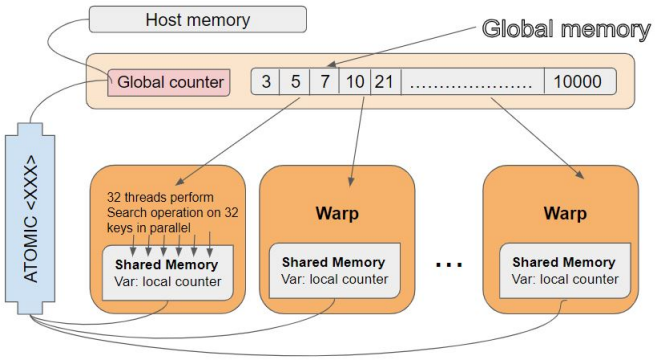

Fig. 7. An example of working of Aggregation Query

memory addresses and collect all the found values. Once the input size increases beyond 1024, multiple SMs and blocks are launched. The threads between different blocks cannot communicate through the local memory and has to depend on the global memory which is quite computationally expensive. It also becomes difficult to maintain memory addresses from different blocks. Hence, we have to assign each thread its search query and they perform multiple point searches for range query.

B. Aggregation Queries

Our general idea of implementing aggregation operations like count and sum are based on range search queries to fetch the results. Range queries are executed by scanning the sorted data in B-trees to find the respective values. The aggregation functions are then performed on these obtained values. The database aggregation operations like count and sum might face the issue of synchronization while accessing memory. This is resolved with atomicAdd() function from the atomic functions available in CUDA toolkit. Atomic functions are used whenever multiple threads are trying to access the same memory location requiring synchronization of these operations. So, atomicAdd() here uses locks to keep all threads in sync. This is easily explained with the example in Fig [7].

1) Count Query: Count query is based on the range query to find the number of key-value pairs present in the B-tree. Both shared and global memories are used to communicate between the threads and update the count value. The performance of the count query depends upon the size of the number of search keys which is discussed in the section [V]. However, overlooking the input size of search keys, all threads work in parallel and writes to global and shared memory are managed with atomicAdd() function. We have described the implementation of our count query in Algorithm [2].

Algorithm 2: Algorithm for Count Query

Data:
$Input \leftarrow Lowerbound(LB), Upperbound(UB)$
$globalCount \leftarrow 0$
$Allocate \leftarrow MemAlloc(), copytoDevice()$
forall *SM and block:* **do**
  while *(threadX.Id < queryInRange)* **do**
    $\_\_shared\_\_sharedCount \leftarrow 0$
    searchInBtree(threadX.Id, search)
    if *value is found* **then**
      atomicAdd(sharedCount, 1)
    else
      Ignore result
  atomicAdd(globalCount, sharedCount)
Result: $Result \leftarrow copyToHost(globalCount)$

2) Sum Query: Sum query works similar to count query. It uses range queries instead of just depending on point search as we expect more than one query to sum up. So, values for the respective keys (queries) are first found using this range search and then uses _shfl_sync ("shuffle") to broadcast that variable to all the threads in a warp followed by atomicAdd() to store the values in the shared sum needed as final output. Different threads run in parallel to access the shared memory. Once the number of search keys grows to a specific limit (1024), also mentioned in the count query in section [IV-B1], we access global memory.

We have described the implementation of our sum query in Algorithm [3].

V. Evaluation

In this section, we are going to evaluate the performance of our implementation. For the implementation, we have used NVIDIA RTX 2060 GPU. For all our experiments, we have used 32-bit keys and values. We consider the time (in milliseconds) as our performance metric. Initially, we randomly generated a sequence of numbers, ranging from 0 to n (where maximum value for $n = 2^{15}$, due to the system memory limitations), and used these numbers for the evaluation. The numbers generated are shuffled and then used as the key-value pairs. This shuffling is done in the CPU. Here, the keys and values are identical. The required amount of memory to store these key-value pairs is allocated in the GPU, then these pairs are transferred from CPU into the GPU, and B-tree is constructed using the GPU's parallel computational capacity.

Algorithm 3: Algorithm for Sum Query

Data:
$Input \leftarrow Lowerbound(LB), Upperbound(UB)$
$globalSum \leftarrow 0$
$Allocate \leftarrow MemAlloc(), copytoDevice()$
forall *SM and block:* **do**
  while *(threadX.Id < queryInRange)* **do**
    $\_\_shared\_\_sharedSum \leftarrow 0$
    searchInBtree(threadX.Id, search)
    if *value is found* **then**
      __shfl__sync() atomicAdd(sharedSum, value)
    else
      Ignore result
  atomicAdd(globalSum, sharedSum)
Result: $Result \leftarrow copyToHost(globalSum)$

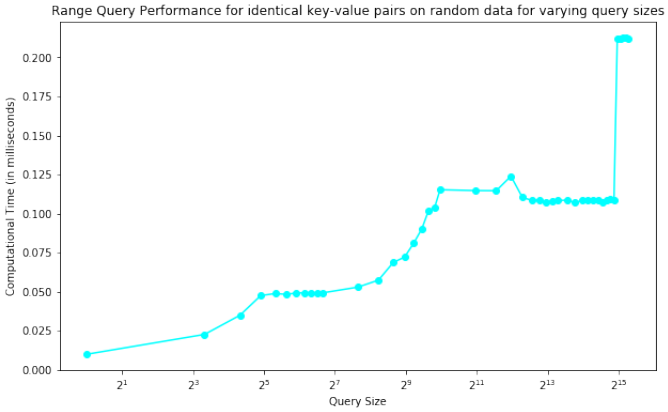

Fig. 8. Execution time for identical key-value pairs for range query for random data on varying query sizes (number of search keys)

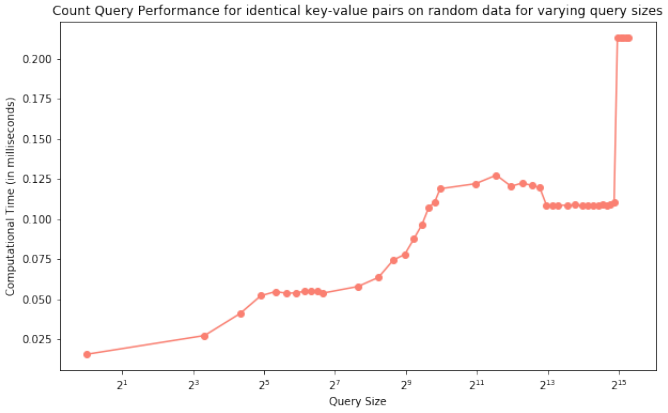

Fig. 9. Execution time for identical key-value pairs for count query for random data on varying query sizes (number of search keys)

As part of the evaluation of our implementation, we evaluated the performance of range, sum and count queries individually for the varying size of number of search keys. In each case, the computational time required for particular queries are recorded and a line graph is plotted to visualize the effect of varying input sizes on the query execution time. The input size of the B-tree (key-value pair) is fixed to the 2^{15}, due to our system limitations.

Fig [8], Fig [9] and Fig [10] are the pictorial representations of range query, count query and sum query respectively, in the form of line graphs.

We observed, for a key range of 2^{15}, all the implemented queries have shown an approximately similar performance. The computational time required for the queries increased gradually for the number of search keys till 2^{10}. As the number of search keys is less than or equal to 1024 (2^{10}), all the queries are executed by a single streaming multiprocessor and the atomic operations used for synchronizing the writing operations of various threads into the shared memory serializes the thread execution. Due to this serialization, we observe an almost linear increase in the query computational times.

From 2^{10} to 2^{15} we can observe a flat curve, this is range for optimal number of search keys for our GPU evaluation. As the number of search keys crosses the mark of 2^{10}, multiple multiprocessors are launched depending on the input size, and work is shared between them. Hence, even though there is an increase in the number of search keys, the required computational time for the execution of queries does not change. This clearly shows the effect of the parallel computational abilities of GPUs.

From the figures, Fig [8], Fig [9] and Fig [10], we also observed that the computation time is doubled when the number of search keys are increased beyond 2^{15}. All the multiprocessors available for the GPU are being used to execute the queries of size 2^{15}. Hence, for the queries beyond 2^{15}, all the streaming multiprocessors execute the first 2^{15} search keys and the remaining search keys are executed after any of the streaming multiprocessors completes its execution. This doubles the query computational time.

For further evaluations, we generated synthetic data, with non-identical key-value pairs. The synthetic data is in the form of a table with multiple columns, where one column consists of primary keys and we use these keys as the input to our B-tree, also said to be primary indexes. We stored the index location of these keys as the corresponding values, making it non-identical key-value pair data, unlike the random data we used at the start of this evaluation section. When we perform queries on this query engine, the input of the query is the key/keys that fetch the desired output for further operations if required, whether it is range query, count query, or sum query. The output of this query is a key-value pair. The value that stored the index location for its respective key is utilized to retrieve the entire row from the table, hence fetching the required data. This way, we have access to all the data generated. Later, with these obtained data values, we can perform the three database operations (range, sum, and count) we implemented.

The Fig [11] shows the performance of the range query, where the query size (number of search keys) is plotted against computational time (in milliseconds). We can see the

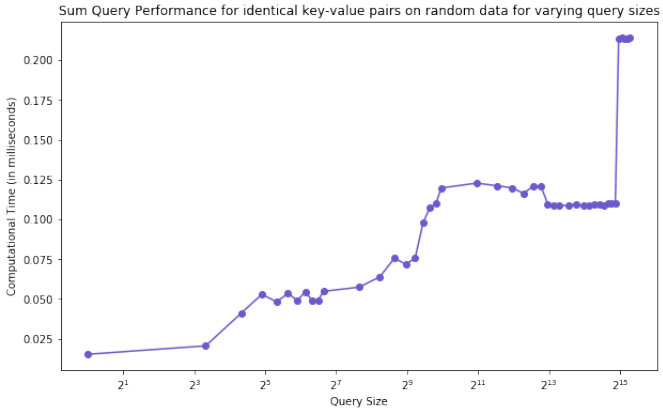

Fig. 10. Execution time for identical key-value pairs for sum query for random data on varying query sizes (number of search keys)

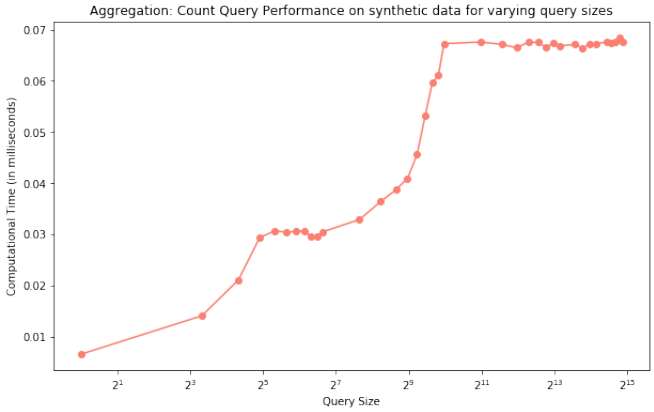

Fig. 12. Execution time for count query for varying query sizes (number of search keys) for synthetic data

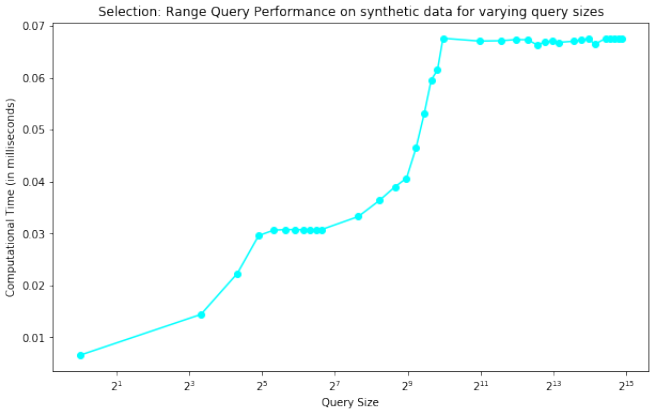

Fig. 11. Execution time for range query for varying query sizes (number of search keys) for synthetic data

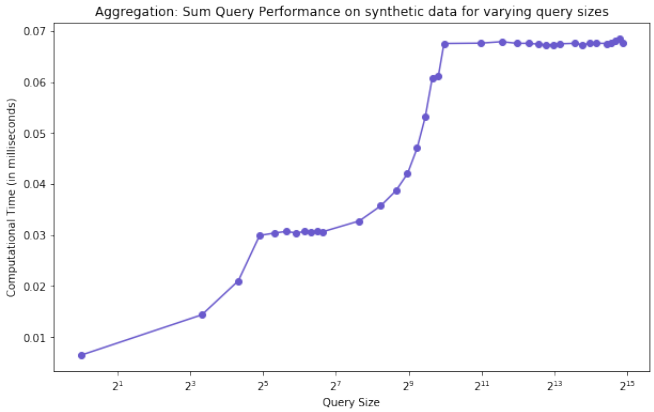

Fig. 13. Execution time for sum query for varying query sizes (number of search keys) for synthetic data

gradual increase in the computational time for search keys up to 2^5, and then we can observe an exponential rise in the computational time for search keys up to 2^{10}. After this point, the time computed becomes almost constant irrespective of the number of search keys. Similarly, Fig [12] and Fig [13] shows the performances of sum query and count query respectively.

We also evaluated the impact of randomly generated synthetic data(table with multiple columns) to the data set containing only identical key-value pairs (sequence of numbers). We observed that both data sets follow the similar path for computational time graph. However, the computational time for synthetic data is always a bit higher than the randomly generated sequence of numbers. This is because in the synthetic data set, we first obtain the index from the B-tree and then retrieve all the rows corresponding to these index values.

VI. Related Work

In "Engineering a high-performance GPU B-Tree" [7], the authors have implemented a B-tree that supports concurrent querying operations such as point query, and update operations like insertions and deletions. They have also discussed their idea on warp-centric implementation of range and successor queries which laid a foundation for our implementation. Authors also provided their evaluation results comparing with various data structures especially with GPU log-structured merge tree (LSM) and a GPU sorted array and the outcomes show that their implementation outperforms LSM for small and medium-size batch insertions(approximately 100k queries per insertion). Even though LSMs are known for their insertion sizes, they have the worst query performance when compared to B-trees. These insertion sizes are critically important for fundamental data structures because if the data structure performs well on large batch volumes only, it won't be much use as a general-purpose data structure. The proposed implementation addresses challenges like delivering full utilization of global memory bandwidth, full utilization of available GPU cores and a careful design of data structure that achieves mutability and high performance for query processing. The principal focus is not on implementing a novel B-tree but design decisions that support high-performance queries and obtain maximum concurrency. The paper also includes a GPU friendly cache-aware design of B-tree and a new strategy called Warp-Cooperative Work-Sharing strategy (WCWS) in which the work is done in terms of warps with each warp

consisting of 32 threads to attain coalesced memory accesses avoiding branch divergence and allowing the neighboring threads to run different operations. The proposed WCWS strategy supports the entire warp to irregular tasks arising due to node-splitting in B-trees facilitating avoidance to load-balance work across threads. The search, insertions and incremental updates on B-trees yield greater speedups when compared with the sequential search and batch insertions.

Bingsheng He, Ke Yang and other authors of "Relational query co-processing on graphics processors" [5], implemented relational join algorithms for GPUs and designed data-parallel primitives such as split and sort. Implementing sort-merge and hash joins using these primitives as GPUs provide higher computational capabilities, better latency tolerance and higher memory bandwidth. They have concentrated on using the high parallelism as well as the high memory bandwidth of the GPU, and use parallel computation yielding a parallel database that also utilizes the row-column format. The results show that these primitives provide high-level abstractions for data-centric operations and are highly tuned to fully utilize the architectural features of graphics processors. The join performance was compared with CPU based in-memory join algorithms and have achieved speedups up to 2-7 times on GPU.

In the paper "Optimizing GPU-accelerated Group-By and Aggregation" [13], the authors discuss the aggregation operations and compare different strategies for aggregation operations. In the papers "GPU-accelerated database systems: Survey and open challenges" [12], and "Fast Computation of Database Operations using Graphics Processors" [14], the authors discuss the design space of the GPU accelerated database management system. Along-with, they also present algorithms for other operations like conjunctive selections and semi-linear queries.

The take away from these papers is to first compute the search clause filter and obtain a column of values. These obtained values can be further used as an input to the actual aggregation instead of using the entire query's work into a single kernel.

VII. Conclusion

We have analyzed and implemented selection (range queries) and aggregation (count and sum) database operations on GPUs using NVIDIA's CUDA toolkit. Fig [11], Fig [12], and Fig [13] shows the time taken by different queries with varying input query sizes (number of search keys). It has been observed from our experiments that all three different queries with the same inputs are taking almost equal time to fetch results from the B-tree that is stored inside GPU. This is due to the use of global and shared memory concepts of GPUs. In this implementation, we were able to use the full capacity of the GPU used for this experiment which is evident from the selectivity range of the number of search keys used. Until 2^{10} input size, the task is assigned to only 1 SM (streaming multiprocessors). As we are increasing the number of search keys from 0 to 2^{10}, using atomic add to write into the local variable, the queries are serialized. This sequential execution of queries results in the gradual increase of computational time until 2^{10} queries. As the number of search keys increases above 2^{10} queries, extra queries go to the next SM. Each SM (with our GPU configuration) can run 2^{10} queries at one clock cycle. With the increase in number of search keys, multiple SMs are launched. All the SMs work in parallel and the computational time remains approximately the same. As they work in parallel, the computational time required for the execution of the queries until 2^{15} does not change, it remains constant. For 2^{15} queries all the SMs within our GPU are getting utilized (full utilization of GPU's capacity) and the next increase in queries should have to wait until one of the SM completes its execution. Hence, the computational time is almost twice after 2^{15} queries and would stay approximately the same until 2^{30}.

VIII. Future Work

With the feasibility of implementing a query engine for database operations on B-trees in GPUs, has opened doors for new ideas for future work. Until now, we have computed the performance metrics for these implementations on two types of data, randomly generated sequence of numbers and synthetic data for both identical and non-identical key-value pairs. For the necessity of finding these results to be the optimal performance values, we can apply different data structures to implement similar operations in GPUs and compare the desired results. If the results on comparison look promising for our implementation of query engine in B-trees in GPUs as compared to other implementations, we can include and execute more database operations like joins.

In our current implementation, the data used is in integer format. Further, we can extend this implementation to support data inputs like strings, alphanumeric, and date to be the key indexes for internal sorting in B-trees. Also, the primary index in B-trees can be extended to secondary index taking into consideration more data and not restricting to just primary keys. We can additionally make it appropriate to support conjunctive query processing. The performance on varying number of search keys for such parametric changes in B-trees in GPUs can be observed with new evaluation results.

References

[1] Comer, Douglas. "Ubiquitous B-tree" ACM Computing Surveys (CSUR) 11, no. 2 (1979): 121-137.

[2] Graefe, Goetz, and Harumi Kuno. "Modern B-tree techniques." In 2011 IEEE 27th International Conference on Data Engineering, pp. 1370-1373. IEEE, 2011.

[3] Bayer, Rudolf. "B-tree and UB-tree." Scholarpedia 3, no. 11 (2008): 7742.

[4] Hammer, Joachim, and Markus Schneider. "Data Structures for Databases." Management 60 (2001): 1.

[5] He, Bingsheng, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. "Relational query coprocessing on graphics processors." ACM Transactions on Database Systems (TODS) 34, no. 4 (2009): 1-39.

[6] Arora, Manish. "The architecture and evolution of cpu-gpu systems for general purpose computing." By University of California, San Diago 27 (2012).

[7] Awad, Muhammad A., Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. "Engineering a high-performance GPU B-Tree." In Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, pp. 145-157. 2019.
[8] Ashkiani, Saman, Martin Farach-Colton, and John D. Owens. "A dynamic hash table for the GPU." In 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 419-429. IEEE, 2018.
[9] Lehman, Philip L., and S. Bing Yao. "Efficient locking for concurrent operations on B-trees." ACM Transactions on Database Systems (TODS) 6, no. 4 (1981): 650-670.
[10] Karnagel, Tomas, René Müller, and Guy M. Lohman. "Optimizing GPU-accelerated Group-By and Aggregation." ADMS@ VLDB 8 (2015): 20.
[11] Mahapatra, Tushar, and Sanjay Mishra. Oracle parallel processing. O'Reilly and Associates, Inc., 2000.
[12] Breß, Sebastian, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. "Gpu-accelerated database systems: Survey and open challenges." In Transactions on Large-Scale Data-and Knowledge-Centered Systems XV, pp. 1-35. Springer, Berlin, Heidelberg, 2014.
[13] Karnagel, Tomas, René Müller, and Guy M. Lohman. "Optimizing GPU-accelerated Group-By and Aggregation." ADMS@ VLDB 8 (2015): 20.
[14] Govindaraju, Naga K., Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. "Fast computation of database operations using graphics processors." In ACM SIGGRAPH 2005 Courses, pp. 206-es. 2005.