Learned Index on GPU

Xun Zhong, Yong Zhang*, Yu Chen, Chao Li*, Chunxiao Xing
BNRist, DCST, Institute of Precision Medicine, Institute of Internet Industry, Tsinghua University, Beijing, China.
{zhongx21, y-c19}@mails.tsinghua.edu.cn; {zhangyong05, xingcx, li-chao}@tsinghua.edu.cn

***Abstract*—Index is a key structure created to quickly access specific information in database. Recent research on "learned indexes" has received extensive attention. The key idea is that index can be regarded as a model that maps keys to specific locations in data sets, so the traditional index structure can be replaced by machine learning models. Current learned indexes universally gain higher time efficiency and occupy smaller space than traditional indexes, but their query efficiency and concurrency are limited by CPU.**

GPU is widely used in computing intensive tasks because of its unique architecture and powerful computing ability. According to the research on learned index in recent years, we propose a new trait of thought to combine the advantages of GPU and learned index, which puts learned index in GPU memory and makes full use of the high concurrency and computing power of GPU. We implement the PGM-index on GPU and conduct an extensive set of experiments on several real-life and synthetic datasets. The results demonstrate that our method beats the original learned index on CPU by up to 20× for static workloads when query scale is large.

***Index Terms*—Learned Index, GPU, Parallel Query**

I. Introduction

With the continuous growth of data scale, new algorithms and data structures for the management of massive data are developing rapidly. Recently, the combination of artificial intelligence and database has given birth to a new research direction called "learned index" [1]. Traditional indexes are supposed to build on data with the worst distribution to gain good generality while learned indexes utilize machine learning models to learn the data distribution and predict the position of a lookup key in the dataset. Many studies [2]–[4] demonstrate that learned indexes achieve significant advantages over traditional indexes in terms of high performance and low space occupancy by extracting the patterns in the data through succinct models. The original learned index called Recursive Model Index(RMI) [1] uses a hierarchy of machine learning models. RMI's structure and implementation are very concise but it mainly focuses on read-only workloads. Various types of learned index structures were proposed after RMI, including ALEX [3], PGM-index [2] and LIPP [4], that use well-designed structures to organize machine learning models and provide support for insertions and deletions while gaining better performance.

Current learned indexes' query efficiency and concurrency are usually limited by the computing power and throughput of CPU. Machine learning models used in learned index are computing intensive tasks. As is known, Graphics Processing Units(GPU) or Tensor Processing Unit (TPU) is indispensable in machine learning and deep learning research area but current learned indexes are all assumed to be running on CPU and only adopt simple models like linear regression model or two-layer fully connected neural network. Moreover, currently learned indexes utilize low concurrency of CPU, which means they cannot process a large number of queries in an efficient way.

To tackle with these issues, we think of utilizing GPU, which has obtained broad application for its powerful computing power and high throughput [10]. Actually, researchers have tried to use GPU to improve traditional index performance, including index for query optimization in computing-intensive scenarios like index for kNN queries in road networks [12], parallel index for processing high-dimensional big data [13] and index supporting spatio-temporal queries over historical data [11]. Although these indexes performed well in these scenarios but are not universal and cannot be built on general data.Some scholars also tried to implement general traditional index on GPU like hash table [15] and B+Tree [7], [14] and also achieved good performance. However, traditional indexes mainly rely on cache and branch operations that can be processed better by CPU cores rather than GPU cores, resulting in undesirable performance loss.

Architecturally, the CPU is composed of a few cores with cache designed to handle a wide-range of tasks while GPU is composed of hundreds of cores that can only perform simple operations. Learned indexes replace branch search in tradition index with machine learning model computation, which fits in with the architecture of GPU. Despite the advantages of GPU and the correspondence between learned index and GPU architecture, developing learned index on GPU is not without challenges. There are three main problems that need to be addressed in order to attain the efficient parallel computing capability of GPU: (1) The latency for transferring the input data in main memory and retrieving the results from GPU memory is still significantly high; (2) The performance loss caused by operation synchronization between CPU and GPU kernels can be serious; (3) GPU has hierarchical memory space and unique way to access memory efficiently so the adaptability of current learned index structure in GPU architecture should also be considered.

In this paper, we focus on the scenarios with few or no update and delete operations such as blockchain, and try to deal with the last two problems. We select PGM-index [2], a concise learned index structure with contiguous space storage implementation which can be efficiently accessed in

*Yong Zhang and Chao Li are the corresponding authors.

GPU memory, migrate it to GPU and build a GPU learned index. During the migration, we design an efficient structure where index is fully stored in GPU memory to avoid the performance loss caused by synchronization between CPU and GPU kernels. As the index building algorithm needs to traverse the dataset and cannot benefit from parallel optimization, PGM-index is firstly built on CPU, then flattened to an array and finally transferred to GPU memory with additional information. In the process of query, keys are collected in CPU and transferred to GPU to execute the query operations. The results are retrieved from GPU memory after the completion of query operations. The combination of GPU and learned index can both avoid the complicated branch operation which is undesirable in GPU kernel and make full use of parallel computing capability of GPU to increase the concurrency of learned, consequently giving full play to the advantages of both sides.

In the rest of the paper, we begin by describing background on GPU and PGM-index in Section II. Then, we state our acceleration ratio computation model and describe how to implement PGM-index on GPU in Section III. After that, we perform experiments on several real-life and synthetic datasets in Section IV. Finally, we discuss our findings and conclude the paper in Section V.

II. Background

A. GPU and CUDA

GPU is not limited to processing graphics but has become an indispensable part of today's mainstream computing systems for its powerful computing power and high throughput [10]. In GPU architectures, more transistors are used for data processing, such as floating-point computation, rather than data caching and logical control. By sacrificing the complexity and independence of threading tasks, GPU can efficiently manage thousands of threads simultaneously. GPU can not be used as an independent computing platform, but needs to work with CPU, that is, CPU controls the execution sequence of programs, while GPU deals with computing intensive subtasks.

Compute Unified Device Architecture(CUDA) [6] from NVIDIA provides a powerful platform for writing parallel programs on GPU. GPU programs are organized into kernels, which are C-like functions called from within the CPU, also called the host. Kernels launch a grid of thousands of simultaneously executing threads, which are grouped into blocks. The GPU's memory space is separated from the host's, which makes it necessary to send all input data through the PCIe bus before any processing can take place in the GPU, and to send all output data from the GPU back to the host. The memory space of GPUs is also hierarchical: threads can access their own individual local memory registers; threads in a block can cooperate by using the larger block-wide shared memory; and threads across different blocks all have access to the slower but bigger global GPU memory. Memory coalescing [6] is a significant technique in GPU, which allows optimal usage of the global memory bandwidth. When parallel threads running the same instruction access to consecutive locations in the global memory, the most favorable access pattern is achieved. Consequently, continuous storage structure based on offset access is more efficient in GPU.

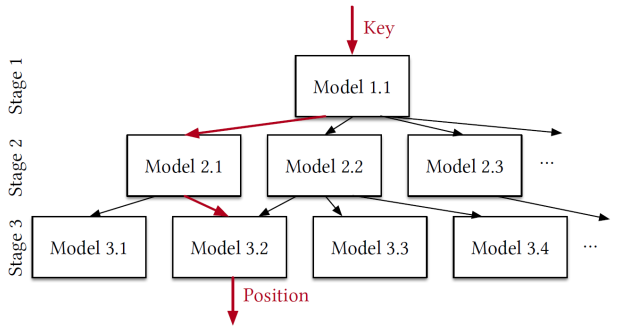

Fig. 1. RMI structure [1]

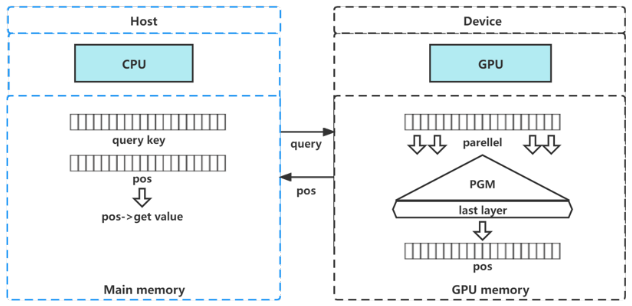

Fig. 2. GPU-PGM structure

B. GPU Traditional Index

Modern GPU architecture makes it possible to write general GPU programs and many researchers utilize the platform to optimize index query. Early GPU indexes are mainly hash index and B+Tree index. Kaczmarski implemented an experimental B+-tree for GPU [16], which is a typical implementation of traditional B+Tree. Zhang et al. used a hash index in GPU to accelerate concurrent queries [15], which collects query keys within time interval to do parallel query in GPU. There are also many structural improvements for GPU traditional index. Shahvarani et al. proposed a hybrid B+-tree on CPU-GPU heterogeneous computing platforms [14] and saved GPU memory greatly. Awad et al. engineered a high-performance GPU B-Tree [7] which has been declared as the state of the art in GPU traditional index.

C. Learned Index

The first learned index RMI proposed by Kraska et al. consists of multiple levels of learned models. As shown in Figure 1, given a search key, current level's learned model predicts which model in next level will be responsible for the key. The prediction may not be exactly precise and RMI then performs a local search in the models array to locate the desired model. The process of predicting and locally searching can be done recursively until arriving the last level. Last level's learned model will predict the position of the key in the dataset organized as a sorted array. RMI is built by training the learned model of each level from the top down recursively and models in each level can be specified. RMI is designed for read-only workloads and lack of support for updates and error range specifying, but its concise structure still provides a good reference for the follow-up study of learned index.

Some recent studies propose a number of enhanced learned index structures [2]–[4], that provide support for updates while gaining better performance and occupying less space. ALEX [3] and LIPP [4] are fully dynamic index structures. The structure of ALEX is similar to RMI, but the difference is that ALEX employs gapped arrays in the data nodes to reduce the entry movement cost for insertions. When a data node is close to full, ALEX adopts different strategies to expand or split the node according to the node size. LIPP also spares gap in node for insertions like ALEX, but redesigns ingenious structure where all nodes are treated equally and all predictions made by models are exact, resulting in better performance for queries and updates.

PGM-index [2] uses linear models and separates the keys in different linear segments with given error bound. Unlike other learned indexes, each segment of PGM-index specifies the first key covered by the segment. In this way, PGM-index recursively constructs static index structure on the sorted keys of segments in low level. To support insertions, PGM-index employs the idea of LSM-tree [5], that is building a series of PGM-index over subsets and merging them into a large subset and build a new PGM-index. In this paper, we do not implement insertions for PGM-index on GPU and leave it as future work.

As PGM-index uses contiguous space storage which is more efficient in GPU memory structure than linked-nodes storage ALEX and LIPP uses, we tend to implement the PGM-index on GPU and gain higher query acceleration ratio.

III. PGM-INDEX ON GPU

In this section, we give an overview of the structure of PGM-index on GPU(GPU-PGM) and set up a quantitative model to evaluate the speedup of GPU-PGM compared to original PGM-index(CPU-PGM) [2] in query operations. Then we describe the data structure building process and search operations of GPU-PGM .

A. The Structure of GPU-PGM

In our design, GPU-PGM is fully stored in GPU memory to avoid the performance loss caused by synchronization between CPU and GPU kernals. Because PGM-index occupies up to three orders orders of magnitude less space than B+Tree [2], we can assume this approach is reasonable.

Before executing query operation, we need to collect batch of query keys on CPU. If the query request for the database is not concurrent, we can collect query keys within a specified time interval as a batch.

Since we consider that the input queries are given in main memory, the first step is to transfer them into GPU memory, before the GPU starts executing a search operation.

Parallel query is the main advantage of GPU-PGM. GPU assigns queries to kernels for execution and each kernal is responsible for dispatching threads to perform query operations. Threads in the same kernal will access PGM-index in GPU global memory in a synchronous and aligned way and execute the same query instructions, finally get the results and write to the specified memory location in parallel.

The precise positions will be transferred into main memory after the GPU completes the search operation. In the last step, the CPU uses these positions to reach the target tuple. The structure of GPU-PGM is shown in Figure 2.

In order to comprehensively evaluate the query performance of GPU-PGM, we use total execution time including query time of batch queries in GPU kernals and time of transferring queries to GPU memory and results from GPU memory.

Similar to the analysis of Hybrid B+-tree [14], we build our cost model where T is time required for each step of the GPU-PGM structure as follows:

1) Transferring search key to GPU memory:
   $T_{input} = T_{init} + \alpha Q/B$
2) GPU traversal of all inner nodes of tree per each query:
   $T_{execute} = K_{init} + Q/SIMD_G \times P_{GPU}$
3) Transfer of positions to CPU memory:
   $T_{output} = T_{init} + \beta Q/B$

- B: Data transfer bandwidth between main memory and GPU memory.
- α: The size of single query key.
- β: The size of single position.
- Q: The Number of query keys.
- T_{init}: Data transfer initialization time between main memory and GPU memory.
- K_{init}: GPU initialization time for search operation.
- $SIMD_G$: GPU SIMD width.
- P_{GPU} : Average processing time for a query on GPU.
- P_{CPU} : Average processing time for a query on CPU.

We gain the total execution time of GPU-PGM and CPU-PGM from above:

1) Total CPU-PGM query time
   $T_{CPU} = \alpha Q \times P_{CPU}$
2) Total GPU-PGM query time
   $T_{GPU} = T_{input} + T_{execute} + T_{output}$

The speedup ratio of GPU-PGM to CPU-PGM is:

$$T_{CPU}/T_{GPU} = \frac{Q}{C} + \frac{\alpha P_{CPU}}{(\alpha+\beta)/B + P_{GPU}/SIMD_G} \quad (1)$$

where C is a constant number.

From the acceleration ratio in cost model we can inaccurately conclude that with the increase of the number of queries, the advantages of GPU become more obvious and the main constraint of query performance is the data transfer bandwidth between main memory and GPU memory. In a real scenario, the $SIMD_G$ can vary with the increase of the number of parallel tasks, which depends on the number of GPU kernals and scheduling strategy.

B. The Algorithms of GPU-PGM

Index Building. The first step of index construction is to build a piecewise linear model with error not exceeding threshold ϵ. In this step the streaming algorithm which takes $O(n)$ optimal time and space is admitted to ensure the piecewise linear model is optimal. The detail is shown in Algorithm 1.

The key idea of the algorithm is to reduce this problem to the one of constructing a convex hull of a set of points, which in this case is the set $\{key_i, pos(key_i)\}$ grown incrementally for $i = 0, ..., n-1$. As long as the convex hull can be enclosed in a rectangle of height no more than 2ϵ, the index i is incremented and the set is extended. As soon as the rectangle enclosing the convex hull is higher than 2ϵ, stop the construction and determine one segment by taking the line which splits that rectangle into two equal-sized halves(Lines 3-10). Then the algorithm restarts from the rest of the input points. This greedy approach can be proved to be optimal in the number of segments and have linear time and space complexity.

Algorithm 1 Algorithm for GPU-PGM building

```
Input: $keys$: sorted array, $\epsilon$: max error
Output: $levels$: flattened PGM-index organized as an array
        $offsets$: offsets of each level in flattened PGM-index
  Initialisation: $segnum = \infty, levels = []$, $cur$ = $keys$
 1: for $segnum \neq 1$ do
 2:     $points = []$
 3:     for $i = 0$ to $cur.size$ do
 4:         Construct convex hull over $points \cup cur[i]$
 5:         if (convex hull suits $2\epsilon$ rectangle) then
 6:             Compute the minimal rectangle cover $points$
 7:             $seg$ =line splits rectangle into two equal halves
 8:             $model.append(seg)$
 9:         end if
10:     end for
11:     $segnum = model.size$
12:     $levels.append(model)$
13:     $cur = [model[0].key, ..., model[segnum-1].key]$
14: end for
15: $offsets = [segnums[0], ..., segnums[height-1]]$
16: Flatten $levels$
17: Move flattened $levels$ and $offsets$ to GPU memory
```

After the piecewise linear model is established, the next step is to build hierarchical index structure. We start with the model constructed over the whole input array and then extract the first key of input array covered by each segment and construct another piecewise linear model over this reduced set of keys(Lines 1-14). We proceed in recursive way until the model consists of only one segment, as shown in Figure 3.

As the PGM-index building algorithm needs to traverse the dataset and cannot benefit from parallel optimization, we firstly build PGM-index on CPU. Then we organize the hierarchical index structure as a compact array and record the offset of each level, finally transfer PGM-index with useful additional information to GPU memory and gain GPU-PGM(Lines 15-17).

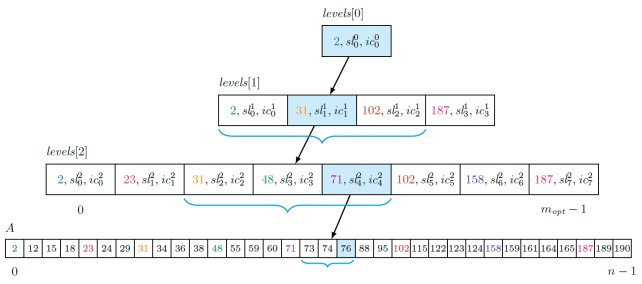

Fig. 3. PGM-index structure [2]

Algorithm 2 Algorithm for GPU-PGM query

```
Input: $keys$: sorted array, $\epsilon$: max error, $t$: threads num
       $levels$: index, $queries$: query keys
Output: $pos$: the precise position of query keys
    Dispatch queries to $t$ threads
    Each thread gains tid from multi-dimensional identifier
    Kernal executing begin:
 1: if ($tid \leq queries.size$) then
 2:     $query = queries[tid]$
 3:     $f = levels[0][0].linear$
 4:     $next\_pos = f(query)$
 5:     for $i = 0$ to $levels.size$ do
 6:         $range = [next\_pos - \epsilon, next\_pos + \epsilon]$
 7:         $segment = binary\_search(range, query)$
 8:         $f = segment.linear$
 9:         $next\_pos = f(query)$
10:     end for
11: end if
12: $gpu\_pos[tid] = next\_pos$
    Kernal executing end
13: $pos$ = Move $gpu\_pos$ to main memory
```

Index Query. The pseudocode of query operation is described in Algorithm 2. The batch of query keys is transferred from main memory to GPU memory and then dispatched to designated number of threads to execute. The most efficient operation on the GPU is that threads in the same block execute the same query instructions and access GPU global memory in a synchronous and aligned way. So it's necessary to check the number of threads in streaming multiprocessor in the current GPU when threads are dispatched. There is a multi-dimensional identifier in each thread, which can be converted into an index number tid to obtain the query key from the batch of query keys(Lines 1-2).

Now, each thread gets the corresponding query key and the query operation works as follows. At every level, we use the segment referring to the visited node to estimate the position of the searched key among the keys of the lower level(Lines 3-4). The real position is then found by a binary search in a range of size 2ϵ centered around the estimated position. Given that every key on the next level is the first key covered by a segment on that level, we have identified the next segment to query, and the process continues until the last level is reached(Lines 5-10). The results are written to the specified location and transferred back to the main memory(Lines 12-13).

IV. Experiment

In this section, we describe the datasets, hardware and the setup used in the experiments. We compare GPU-PGM against three baselines: (1) PGM-index on CPU [2]; (2) GPU-BTree proposed by Awad et al., a high-performance GPU implementation of a B-Tree which is the state of the art [7]; (3) A production quality B+Tree implementation known as STX B+Tree [8].

A. Datasets

We use four sorted datasets from Harvard Dataverse [9]: (1) Books data in library management system; (2) Part of indexes data in Wikipedia; (3) User data from Facebook; (4) Longitudes of locations in North America from Open Street Maps. These four datasets are from the actual application scenarios and have been widely used in previous researches, whose characteristics are shown in Table I.

TABLE I
DATASET CHARACTERISTICS

Dataset	Size	Key Type	Key Size	Key Num
Books	781.25MB	int32	4B	2×10^9
Wikipedia	1562.51MB	int64	8B	2×10^9
Facebook	781.25MB	int32	4B	2×10^9
OSM	1562.51MB	int64	8B	2×10^9

B. Hardware

We run all experiments on a machine with Ubuntu 20.04 using CUDA 11. The hardware used is an Intel Xeon Gold 6248 2.50GHz CPU equipped with 128GB (4x32GB) DRAM and NVIDIA RTX 2080ti equipped with 11GB memory.

C. Parameters

Max error ϵ in PGM-index is significant to query time and index size. As ϵ parameter increases, index size and height decrease. The query time is limited by the expanded search range while benefiting from reduced index size and height. We employ $\epsilon = 128$ as default parameters for our tests. In addition, the number of threads and blocks in GPU kernel are set to $query\_size/1024$ and 1024 respectively.

D. Performance Measure

In our experiments, the performance measure is the query time of a batch of query keys. For GPU index we count total execution time of query operation measured from the moment when it starts executing until the moment when the results

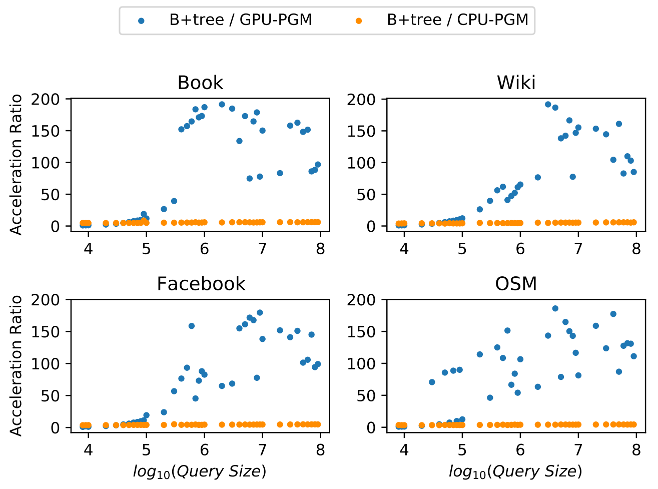

Fig. 4. Comparing GPU-PGM and CPU-PGM

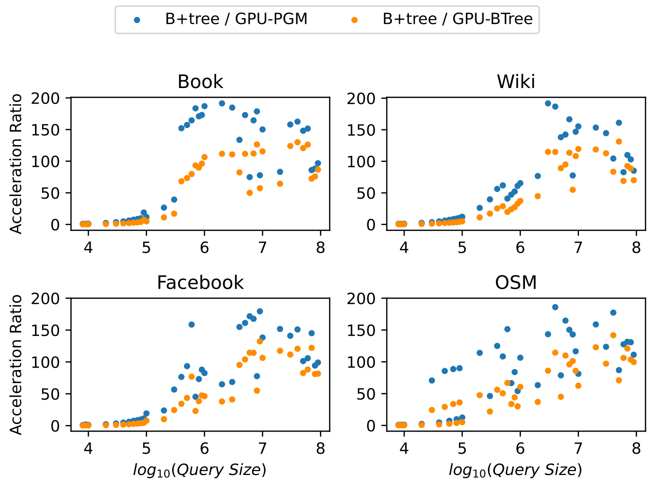

Fig. 5. Comparing GPU-PGM and GPU-BTree

are available in the host. Furthermore, We do not estimate the performance of the range query in our experiment, because keys in the last layer of index are stored in an ordered array, and the range query can be simply implemented by point query and linear search.

TABLE II
INDEX BUILDING TIME

	Building Time(s)			
Dataset	B+Tree	GPU-BTree	CPU-PGM	GPU-PGM
Books	33.34	58.23	16.67	17.99
Wikipedia	27.07	51.94	13.67	13.95
Facebook	32.17	53.88	35.44	36.77
OSM	29.22	59.50	31.14	31.43

E. Experimental Results

Query performance. Due to the large gap in query efficiency, we use an indirect measure, acceleration ratio (i.e. The ratio of index query time to STX B+Tree [8] query time) to show the results rather than the query time of a batch of query keys. Figure 4 and Figure 5 compare the acceleration ratio of GPU-PGM and CPU-PGM, GPU-PGM and GPU-BTree respectively on the four datasets.

From Figure 4 we see that the acceleration ratio of GPU-PGM is generally distributed between 50 and 200 when the query size is greater than 10^6, which is much higher than that of CPU-PGM. The performance gap between GPU-PGM and CPU-PGM does not always widen with the increase of the number of queries, as shown in equation 1 because we ignore the $SIMD_G$ which can vary with the number of parallel tasks. When the number of free threads in GPU kernels is less than the number of queries, query keys will be divided and dispatched to free threads in different batches. In addition, GPU scheduling strategy and unpredictable memory access can be influence factors.

In Figure 5, we find GPU-PGM beats GPU-BTree by up to 1.5×-3× on acceleration ratio. Compared with GPU traditional index, the advantage of GPU-PGM is that the query execution efficiency in GPU is very high. However, this advantage diminishes as the number of query keys increases because the time of data transfer between main memory and GPU memory will dominate when the number of query keys is large enough, consequently the query execution time in GPU becomes less important.

Building Time. Table II shows the total time to build indexes over four datasets. The building algorithm of PGM-index recursively calls the greedy method to build optimal piecewise linear model which takes $O(n)$ time and space. The number of levels in PGM-index is usually not large but can vary according to the distribution of the dataset. In contrast, The building time of B+Tree index is relatively stable on datasets with the same size. GPU-PGM takes a little more time than CPU-PGM for data transfer and storing additional information.

TABLE III
INDEX SPACE OCCUPATION

	Memory(MB)			
Dataset	B+Tree	GPU-BTree	CPU-PGM	GPU-PGM
Books	153.34	320.23	1.21	2.19
Wikipedia	127.07	334.94	0.74	1.32
Facebook	307.17	352.88	15.04	27.92
OSM	393.22	341.50	8.43	15.01

Space Occupation. Table III shows the total space occupied by different indexes in four datasets. It can be found that GPU-PGM also inherits the advantage of CPU-PGM in space occupation, occupies two orders of magnitude less space than STX B+Tree and GPU-BTree, because the index implementation based on continuous memory space makes its storage in GPU memory consistent with that in the main memory.

V. CONCLUSION AND FUTURE WORK

In this paper, we present a perspective to combine learned index with GPU to give full play to the advantages of both GPU architecture and learned index structure. For the case of GPU memory structure we choose PGM-index with contiguous space storage and implement the PGM-index on GPU. In four datasets with hundred million keys from practical application scenarios, the experimental results of this paper show that GPU-PGM improves the query efficiency by one order of magnitude compared with CPU-PGM while having the same advantage as PGM in space occupation.

In the future, we would like to do research on the improvement of PGM-index structure for GPU, optimization of data transfer and support for insertion.

ACKNOWLEDGMENT

This work was supported by National Key R&D Program of China (2018YFB1402701) and State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCHA202008.

REFERENCES

[1] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 489–504.
[2] P. Ferragina and G. Vinciguerra, "The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1162–1175, 2020.
[3] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann *et al.*, "Alex: an updatable adaptive learned index," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 969–984.
[4] J. Wu, Y. Zhang, S. Chen, J. Wang, Y. Chen, and C. Xing, "Updatable learned index with precise positions," *Proc. VLDB Endow.*, vol. 14, no. 8, p. 1276–1288, apr 2021.
[5] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.
[6] NVIDIA Corporation, "Programming guide: Cuda toolkit documentation," 2020. [Online]. Available: https://docs.nvidia.com/
[7] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens, "Engineering a high-performance gpu b-tree," in *Proceedings of the 24th symposium on principles and practice of parallel programming*, 2019, pp. 145–157.
[8] T. Bingmann, "Stx b+ tree c++ template classes," 2013. [Online]. Available: https://panthema.net/2007/stx-btree/
[9] R. Marcus, A. Kipf, and A. van Renen, "Searching on Sorted Data," 2019. [Online]. Available: https://doi.org/10.7910/DVN/JGVF9A
[10] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.
[11] H. Doraiswamy, H. T. Vo, C. T. Silva, and J. Freire, "A gpu-based index to support interactive spatio-temporal queries over historical data," in *IEEE International Conference on Data Engineering*, 2016.
[12] C. Li, Y. Gu, J. Qi, J. He, Q. Deng, and G. Yu, "A gpu accelerated update efficient index for knn queries in road networks," in *2018 IEEE 34th International Conference on Data Engineering (ICDE).* IEEE, 2018, pp. 881–892.
[13] M. Kim, L. Liu, and W. Choi, "A gpu-aware parallel index for processing high-dimensional big data," *IEEE Transactions on Computers*, vol. 67, no. 10, pp. 1388–1402, 2018.
[14] A. Shahvarani and H.-A. Jacobsen, "A hybrid b+-tree as solution for in-memory indexing on cpu-gpu heterogeneous computing platforms," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1523–1538.
[15] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1226–1237, 2015.
[16] K. Kaczmarski, "Experimental b+-tree for gpu," *ADBIS (2)*, vol. 11, 2011.