CUBPT: Lock-free bulk insertions to B+ tree on GPU architecture

Yulong Huang[1]*, Benyue Su[1], Jianqing Xi[2]

[1] *School of computer & information, Anqing Normal University, Anqing 246401, China*

[2] *School of Software Engineering, South China University of Technology, Guangzhou 510006, China*

Received 15 June 2014, www.cmnt.lv

Abstract

B+-tree is one of the most widely-used index structures. To improve insertion process, several batch algorithms are proposed, which all use one thread to complete one node insertion and cannot make full use of GPU's parallel throughput. So, a batch building and insertion method on GPU named CUBPT is proposed in this paper. During the process of bulk building and insertion, CUBPT use one thread to insert one key, which can maximize the performance by GPU. The experimental results show that when build a 10M tree, the overall performance of CUBPT improved 25.03 times compare with four threads PBI. When insert 10M uniform keys into a 10M tree, the overall performance of CUBPT improved 13.38 times compare with four threads PALM; when insert 10M highly skewed keys into tree with same size, the overall performance of CUBPT improved 15.23 times compare with four threads PALM.

Keywords: in-memory B+-tree, bulk build, Lock-free batch insertion, GPGPU

1 Introduction

Recently, the performance of single-core CPU is suffering a bottleneck and traditional architecture cannot promote the performance of multi-core CPU rapidly. Existing research result [1] shows that eight-core or above CPU cannot obtain any breakthrough on computing performance. Meanwhile, with the rapid development of GPU technology, it is ideal for high-performance computing task. Especially for the task which handles large-scale single-instruction and multiple data streams, the performance of GPU far beyond multi-core CPU. Hence, there has been a growing trend in leveraging the high parallel throughput of GPU for general purpose computing in parallel computing field. CUDA [2] is a programming tool, which allows programmers to write programs run on GPU rapidly. So, a large number of researchers use GPU to accelerate database operations such as sort, scan and achieved great results [3].

As memory capacity has increased dramatically, many database tables and related indices can reside in main memory completely now. So, more and more researchers devoted to improve operational performance of in-memory indices [4, 5]. The B+ tree is one of the most widely-used indexes. To improve its insertion, several batch algorithms are proposed [6, 7, 8]. These algorithms attempt to utilize different architecture processor to optimize the insertion process. Meanwhile, they cannot utilize parallel throughput of GPU. So, an experimental method on GPU is proposed [9]. However, this method just inserts one record at one time, which cannot make full use of the parallel throughput too. For this reason, a novel lock-free batch insertion algorithm on GPU called CUBPT is proposed in this paper. It utilizes one thread to handle one key's insertion, which can take full advantage of the parallel throughput of GPU.

2 Related Works

In this section, the structure of B+-tree and serial insertion method are described firstly. On this basis, several batch algorithms are reviewed.

2.1 THE STRUCTURE AND SERIAL INSERTION OF B+ TREE

B^+ tree is a balanced search tree consists of internal and leaf nodes. To improve efficiency, a modified structure is proposed [10]. Here, each internal node contains the maximum key of its sub-trees and the pointers to these sub-trees. All leaf nodes are located on same layer and contain the keys and pointers to corresponding records. Leaf nodes are linked by order of the keys which makes it convenience to retrieve. To an internal node in B^+ tree with order m, its structure is as follows: $(P_0,K_0,P_1,K_1,\ldots,P_i,K_i)$ $0\leq i<n$, where, K_i represents the ith key and $K_{i-1}<K_i$. P_i is the pointer to the root node of ith sub-tree in which all the keys are lower than K_i. n is the number of keys stored in the node. For root node, the range of n is $[2,m)$. For internal nodes, the range of n is $[\lceil m/2\rceil, m)$. The structure of leaf node is similar with internal node. It contains $n(\lceil m/2\rceil\leq n\leq m)$ keys and pointers to corresponding records and also a pointer to adjacent node. A B^+ tree with order 3 is as follows.

* ***Corresponding author*** e-mail p3vsea2002@126.com

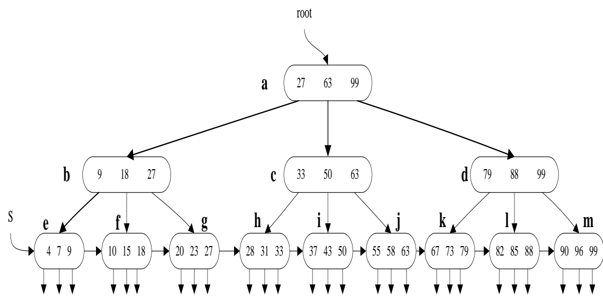


FIGURE 1 The structure of a 3-order B^+ tree

As we all know, traditional B^+ tree is constructed with one by one insertion, which start from an empty tree. So, only serial insertion algorithm with above B^+ tree is described as an example. The insertion process of key 16 is as follows: First, compare 16 with *a*. Because 16<27, then compare 16 with *b*. Because 9<16<18, 16 should insert into *f*. Because there is no enough space, *f* should split into *f* and *f'*, where *f* contains keys {10, 15} and *f'* contains keys {16, 18}. Meanwhile, key 15 should insert into *b*. Similarly, *b* should split. Repeat such process until *a* split.

2.1 THE STRUCTURE AND SERIAL INSERTION OF B+ TREE

In traditional algorithm, the insertion process of each record needs to visit a path from root to leaf. If the size of records is very large, this process needs to execute frequently. So, a batch construction algorithm on single-core CPU is proposed [6]. Here, sort the records firstly and then build them bottom-up, once a layer from leaves to root. Because the path traversal times are reduced largely, the performance has improved greatly. However, this method cannot utilize the parallel throughput of multi-core CPU. So, another batch construction and insertion algorithm for B-link tree is proposed [7]. Similar with above algorithm, B-link tree is constructed with same way. But, this algorithm uses one thread to handle the construction process of one node. In batch insertion process, it also uses one thread complete one node insertion because the data contention. To leverage the parallel performance of many-core CPU, a new latch-free modifications algorithm called PALM is proposed [8]. Combined with BSP model [11] and auxiliary structures, PALM divides the insertion process into three stages. In every stage, it uses one thread to handle the insertion process of one node. To reduce synchronization cost, a point-to-point strategy is applied. For leveraging parallel throughput of GPU, an experimental insertion method is proposed [9], which can improve insertion efficiency with thread block. However, it only inserts one record by one time. In this basis, A B+ tree batch insertion algorithm on GPU is further proposed [14].Due to the requirement of this algorithm, it use many arrays to store the leaf nodes and internal nodes. So, the management of data structure is very complex. At the same time, since the space cost is very expensive, it cannot complete batch insertion process in GPU when the scale of records is very large.

3 CUBPT lock-free batch insertion algorithm

3.1 THE STORAGE STRUCTURE OF CUBPT

As we all know, the design of data structure is one of the key problems for GPGPU programming and the array of structures is most suitable for GPU. So, we use array of nodes to store B^+ tree. To optimize tree storage, we observe that all nodes have similar structure. Let m denote the order of tree. For every node, the size of array *keys* is m and array *ptrs* is m+1. In *ptrs*, the first m elements store the index of relevant nodes and the last has different usages. For leaf nodes, it is used to store the index of next node. For other nodes, it is used to store the first address of new storage space which generated in node split that can accelerate key insertion. All nodes contain a *type* to identify different type node. For our algorithm, a *parent* for parent node's location and a *kNum* for keys number in node are required. So, the size of every node is 2*(m+2) in device memory. The device memory structure of B^+ tree described in Figure 1 is as follows.

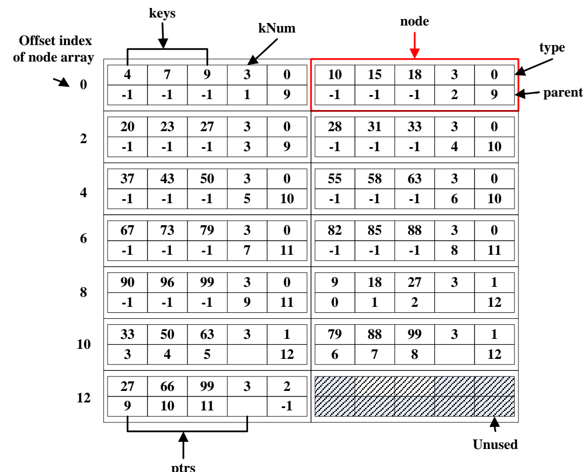


FIGURE 2 The storage structure of B^+ tree in GPU

For B^+ tree, the number of nodes increases by record insertion, which makes the expansion of nodes array necessary. However, CUDA does not support dynamic expansion. So, we adopt the following two-level structure to implement dynamic expansion.

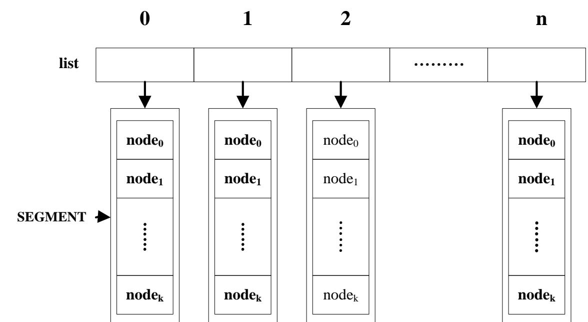


FIGURE 3 Two-level structure of nodes array

Here, nodes array is divided into several *segments* with fixed size and an array *list* is used to store every *segment's* address. Hence, only *list* needs to be expanded statically. So, the cost is very small. Expansion needs three stages: (i) create new *segments*; (ii) expand *list*; (iii) add address of new *segments* into *list*.

3.2 THE DESCRIPTION OF CUBPT ALGORITHM

To leverage parallel throughput of GPU to accelerate insertion process, a lock-free batch algorithm on GPU called CUBPT is proposed in here, which can improve this process very largely. The main idea is as follows: Firstly, allocate a buffer in main memory to store records to be inserted. When the number of records reaches threshold α, copy them from host to device. Then, use parallel primitive [12] to sort them. At last, batch insert these records into B^+ tree in device memory. The details are shown in Figure 4.

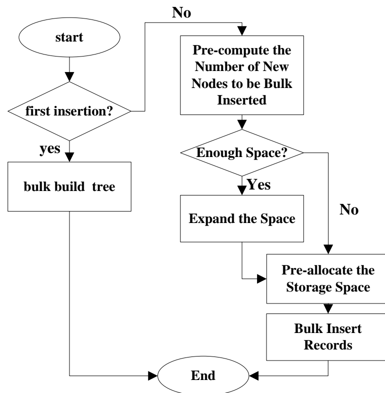


FIGURE 4 The flow chart of CUBPT

As depicted in Figure 4, when insert first batch records, we need to batch construct tree because B+ tree is null. After that, we may continue batch insertion process. So, our method consist of two parts which are batch construction and batch insertion. The detail of these two parts is as follows.

3.2.1 The process of batch construction

For an ordered keys set K, there are two stages to construct tree. Firstly, use $|K|$ threads to divide K into n leaf nodes. Then, use n threads to insert maximum key of every leaf node into its parent. Repeat this process until root node constructed. To distribute the keys more uniformly, we utilize the following Formula 1 and 2 to compute the number of nodes to build in every layer and the number of keys to stored in every node.

$$node_num \leftarrow (|K|+m-1)/m, \qquad (1)$$

$$\begin{cases} ave_keynum \leftarrow (|K|+node_num-1)/node_num \\ h \leftarrow |K|-node_num*(ave_keynum-1) \end{cases}. \qquad (2)$$

With above formulas, K is inserted into *node_num* leaf nodes, in which the first h leaf nodes contain *ave_keynum* keys and the others contain *ave_keynum*-1 keys. The internal nodes are constructed by the same way. The detail of batch build process of CUBPT is as follows.

Batch_build: bulk building b+ tree in global memory
Input: an ordered key set K
Output: b+ tree in global memory T_{dm}
Begin //starting from the leaf layer and construct every layer of T_{dm} iteratively
1. *nodeNum*←(|*K*|+*m*-1)/*m*;//compute the number of leaf nodes. *m* is the order of T_{dm}
2. **For each** *tid*∈[0, *nodeNum*) **parallel do**
3. set the value of *kNum* and *prts*[*m*]of leaf node T_{dm}->node[*tid*];
4. **End for**
5. **For each** *tid*∈[0,|*K*|) **parallel do** //bulk inserts the keys into the leaf nodes
6. calculate index *node_index* and insert location *node_loc* of the leaf node that *K*[*tid*] should be inserted;
7. T_{dm}->node[*node_index*].keys[*insert_loc*]←*K*[*tid*];
8. **End for**
9. **If**(*nodeNum*==1) **return** T_{dm};
10. **While**(*nodeNum*>1)
11. *cur_num*←(*nodeNum*+*m*-1);//compute the number of nodes to be constructed in current layer
12. Parallel set *kNum* of the nodes in current layer by using |*cur_num*| threads;
13. **For each** *tid*∈[0,*nodeNum*) **parallel do**
14. get the maximum key *max_key* of node[*tid*] in preceding layer;
15. with formula 1 and 2, get the index *node_index* of the node that *max_key* should be inserted;
16. insert *max_key* into T_{dm}->*node*[*node_index*] and set the related pointer of child and parent node;
17. **End for**
18. *nodeNum*←*cur_num*;
19. **End while**
20. Set the root pointer of T_{dm} and **return**;
End

As mentioned above, the difference between our algorithm and existing algorithms is that our algorithm uses one thread to handle one key's insertion, which can leverage the parallel throughput of GPU fully.

3.2.2 The process of batch insertion

In above process, CUBPT can maximally accelerate the construction process of b+ tree by using the parallel throughput of GPU. Similarly, to accelerate insertion process, a large-scale batch insert algorithm on GPU is proposed in here. It consists of the following stages:

(i) **Search**: find out the index of leaf nodes to be inserted.

(ii) **Batch insert leaf nodes layer:** According to the search results, batch insert the records into leaf nodes. If overflow, then split it.

(iii) **Batch insert internal nodes layer**: In previous stage, if leaf nodes split, then the maximum keys of newly increase nodes need to be inserted into parent nodes layer. During this process, if the nodes also split, continue insert into the next layer. If necessary, perform this process iteratively until the root node split.

(iv) **Create root node**: In 3rd stage, if root node split, then create a new one and insert related keys into it.

The details of these stages are as follows:

Search: according to the ordered keys set K, search the index of leaf nodes to be inserted by $|K|$ threads.

Starting from the root node, every thread uses binary search method to match the key to be inserted with current node, and then the next search node is obtained. Repeat this process until the index of leaf nodes to be inserted is found. Different with existing algorithms, this stage does not need to search leaf nodes. Hence, this reduced somewhat search cost.

Before batch insert *K*, we should ensure that there has enough space to accomplish current insertion, the total number of increase nodes in current batch insertion should be pre-counted. If no enough space, expand it with the way described in 3.1.The pre-count process consists of two steps: Firstly, parallel reduce the search result to array *mNodeIndex* and *mNode_keynum*. Then, use the following algorithm to count the total number of newly increase nodes.

CountAddNodeNum: Parallel count the number of newly increase nodes in current batch insertion
Input: index array *mNodeIndex* and increase keys number array *mNode_keyNum* of the leaf nodes to be inserted
Output: The total number of newly increase nodes *totalAddNum*
Begin
1. *mNode_keyNum*← get total keys number in related node after inserted by *mNodeIndex* and *mNode_keyNum*;
2. parallelly count the newly increase nodes number by formula 1 and reduce the result to *addNodeNum*;
3. if the newly increase nodes number in leaf layer is greater than 0,then iteratively count every internal layer with similar way.
4. If root node splits, then count the number of newly increased nodes
5. Reduce the results of above steps to *totalAddNum*
End

After the nodes array expanded, the following describe how to batch insert *K* into leaf nodes layer. The detail of this process is as follows:

Batch insert leaf nodes: If there has enough space, then batch insert *K* into the leaf nodes layer. This stage contains five steps as follows.

Step 1: According to the search result *mNodeIndex*, obtain original keys and indices of related leaf nodes with braid parallel method [5]. Then store them in array *ori_keys* and *ori_index*. Here, every thread block contains *m* threads. So, we can use one thread to get one key.

Step 2: Use following *merge_by_key* algorithm to merge *ori_keys, ori_index*, *K* and *mNodeIndex* into array *tmp_keys* and *tmp_index* by order of keys. For simplicity, we use Thrust [11] to implement which as follows.

---

merge_by_key: parallel merge four ordered arrays into two arrays
Input: *K*; index array *mNodeIndex*, original keys *ori_keys* and indices *ori_index* of leaf nodes to be inserted
Output: temporary keys array *tmp_index* and index array *tmp_index*
Begins
1. construct a virtual <*key*, *index*> pair array *part1* with *K* and *mNodeIndex*;
2. use similar method to construct a virtual <*key*, *index*> pair array *part2* with *ori_keys* and *ori_index*;
3. parallel merge *part1* and *part2* into *tmp_keys* and *tmp_index* respectively;
End

Step 3: according to *tmp_index*, use above allocation strategy to get the increase nodes number of every leaf node, and store them in *addnode_num*. Figure 5 shows the details of step3 with a 3-order B^+ tree.

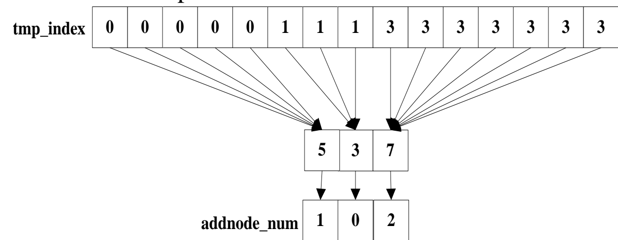


FIGURE 5 The details of Step 3

Step 4: with *addnode_num*, use $\sum addnode_num[i]$ ($0 \le i < |addnode_num|$) threads to allocate continuous space for newly increase nodes. Here, every thread need to set some values such as *ptrs[m]*, *kNum* and *parent*.

Step5: After that, according to *tmp_index*, use */tmp_keys/* threads to parallel insert *tmp_keys* into the leaf nodes .In here, use *tmp_index* to split *tmp_keys* and store the result in array *partition_index*. Then, assign a thread for one key in *tmp_keys* to calculate index (*insert_index*) and insert location (*insert_loc*) of leaf node that the key to be inserted. Finally, batch inserts *tmp_keys* into the leaf nodes. The detail of perform this step on a 3 order b+ tree is as Figure 6.

Now, *K* is inserted into leaf nodes. The following shows how to batch inserts into internal nodes layer.

Batch insert internal nodes: Similar with previous stage, this stage also consists of five steps as follows:

Step 1: Obtain index and maximum key of newly increase nodes in previous level and Store them in array *addnode_index* and *addnode_maxkey*.

Step 2: According to *addnode_index*, the index of corresponding parent nodes are parallel obtained and store in array *parent_index* .On this basis, braid parallel method [5] is used to obtain index and maximum key of child nodes and store in array *ori_index* and *ori_maxkey*.

Step 3: Use the same way with step 2 in previous stage to merge *ori_maxkey*, *ori_index*, *addnode_index* and *addnode_maxkey* into array *tmp_keys* and *tmp_keys*.

Step 4 & Step 5: The process of these two steps is similar with the last two steps in previous stage. The only difference is that every thread in here also needs to modify the related child and parent node pointer.

At this point, all internal nodes split completely. If necessary, a new root node needs to be created.

Create a new root node: obtain the maximum key and index of newly increase nodes in the root node level and store in array *ori_maxkey* and *ori_index.* Then, use *merge_by_key* algorithm *to* merge *ori_maxkey, ori_index*, the maximum key and index of original root node into array *tmp_maxkey* and *tmp_index*. Finally use the same way with our batch construction algorithm to create a new root node.

In summary, according to the split situation of current nodes layer, our algorithm can use different number of threads to handle batch insertion process, which can make full use of the high parallel throughout to further accelerate the insertion process of B+-tree.

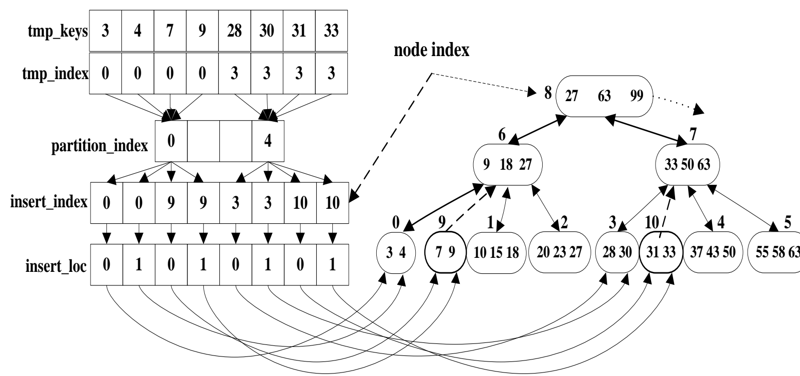


FIGURE 6 The details of Step 5 in stage of batch insert leaf nodes

4 Experimental results and analysis

To prove the effectiveness of our algorithm, we have implemented and tested on a PC with a NVIDIA GTX570 GPU and a recently-released Intel Corei7 Quad-core processor. The hardware configuration is shown in Table1 and software configuration is as follows: The operation system is WindowsXP professional sp3 and IDE is VS2008 with CUDA4.0; the detailed analysis of our algorithm is as follows.

TABLE 1 Hardware configuration

Hardware	GPU	CPU(quad-core)
Processors	780MHz *15 * 32	2.9GHz x*4
DRAM(GB)	1.25	6

4.1 BATCH CONSTRUCTION PROCESS ANALYSIS OF CUBPT

Here, randomly generate a group of datasets which size are 5M, 10M, 15M, and 20M respectively. To prove the effectiveness of our algorithm, three batch construction algorithms are selected to compare:(1) Kim's single-core CPU algorithm *SingleBuild*;(2)Liao's multi-core CPU algorithm *PBI*;(3)our GPU algorithm *CUBPTBuild*; For analysing the impact of different number of threads on *PBI* ,we use two (*PBITwo*) and four threads(*PBIFour*) to perform. As mentioned before, these algorithms all consist of sort and batch insertion stage. *CUBPTBuild* also contains data transfer stage. In sort stage, *singleBuild* use the sort function in STL[15], *PBI* use parallel_sort in TBB [13] and *CUBPTBuild* use a sort primitive in Thrust [12]. Meanwhile, the transfer stage is optimized by pinned memory. Figure 7 shows the elapsed time of above algorithms in batch construction stage when the order of B+-tree is 512.

From Figure 7 we know that, with the size of keys set increases, the performance speedup of *PBITwo* and *PBIFour* almost no changes, which are approximately 1.68 and 2.16. Meanwhile, our algorithm's speedup increase from 26.79 to 29.66. The reason is that *PBI* use one thread to complete one node's construction. When the size of keys set is very large, it needs more iterations. On the contrary, our algorithm uses one thread to handle one key's insertion and construct one layer by one time. So, with increasing scale of keys set, our advantage will further increase.

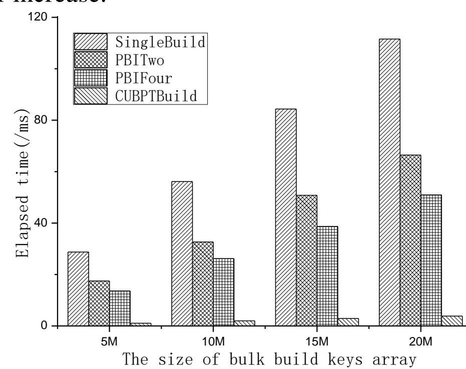


FIGURE 7. The build performance of above comparison algorithms

As above analysis, these comparison algorithms also contain other stages such as sort and transfer stage etc. So, we need to compare the overall performance which shows as Figure 8.

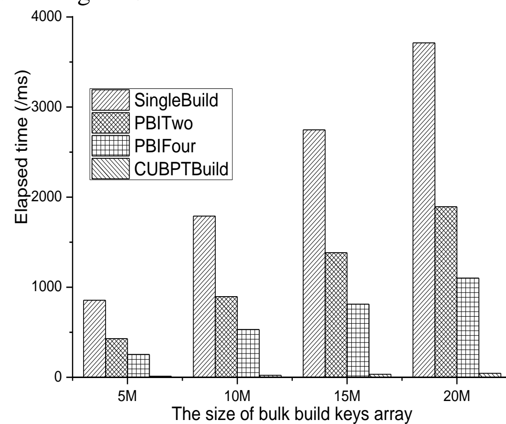


FIGURE 8 The overall performance of above comparison algorithms

As we can see from Figure 8, with the size of keys set increases, the overall speedup of *PBITwo* and *PBIFour* also no change, which are approximately 1.99 and 3.37. The overall speedup of our algorithm increases from 73.34 to 84.36. The reason is that GPU is more suitable for sorting than multi-core CPU, which brings a significant gain for our algorithm.

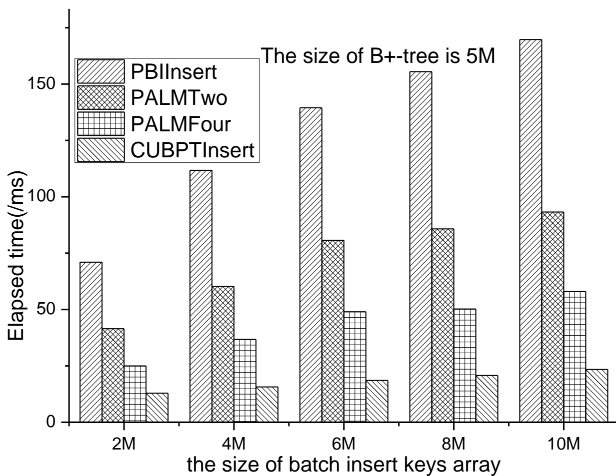
In summary, compare with *singleBuild* and *PBI*, our algorithm has distinct advantage in construction stage and overall performance. With the size of keys set increases, our algorithm's advantage continues to expand.

4.2 BATCH INSERTION PROCESS ANALYSIS OF CUBPT

Similarly, to prove effectiveness of the insertion process of our algorithm, three algorithms are selected to compare:(1) Liao's batch insertion algorithm on single-core CPU *PBIInsert*;(2) Jason's batch modification algorithm *PALM*;(3) our batch insertion algorithm *CUBPTInsert*. To analyse impact of the number of threads on *PALM*, two (*PALMTwo*) and four threads (*PALMFour*) are used to perform. These algorithms also consist of sort and insertion stage. In addition, our algorithm also contains transfer stage. In sort stage, these algorithms use the same method with above. We also optimize transfer with pinned memory. From section 3.2.2, we know that the impact of keys distribution is very large on our algorithm. So, we compare performance with different distribution.

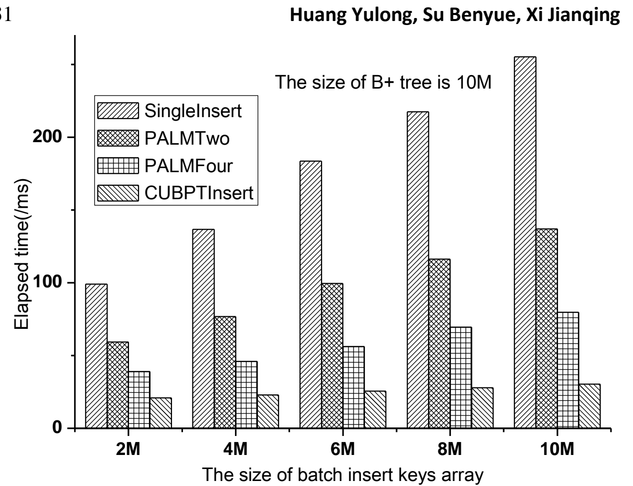
4.2.1 Uniform distribution

Here, generate four uniform data sets. In them, use 5M, 10M, 15M and 20M to build tree respectively. On this basis, batch insert 2M, 4M, 6M, 8M and 10M into the tree, which constructed in front respectively. In this process, all leaf nodes need to be inserted. For above algorithms, we also compare the insertion performance and overall performance. When batch insert keys with different size into a 5M B+-tree, the elapsed time of above comparison algorithms in insertion stage shows as Figure 9.



FIRURE 9 The insertion stage performance of above algorithms when insert uniform keys into a 5M B+-tree

When batch insert keys with different size into a10M B+-tree, the elapsed time of above comparison algorithms in insertion stage shows as Figure 10.



FIRURE 10 The insertion stage performance of above algorithms when insert uniform keys into a 10M B+-tree

In here, the elapsed time of all algorithm do not contain the search time. From figure 9 and figure 10 , we know that when the tree size is 5M, with the size of keys set increases, the speedup of *PALMTwo* and *PALMFour* in insertion stage are almost no change which are approximately 1.8 and 2.9. Meanwhile, our algorithm's speedup increases from 5.5 to 7.5.When the tree size is 10M, the speedup of *PALMTwo* and *PALMFour* almost no change too which are approximately 1.8 and 3.1. Meanwhile, the speedup of our algorithm increases from 4.76 to 8.43. According to this, we know that with the size of B+ tree increases, our algorithm's speedup increases by a small margin. The reason is that our algorithm use one thread to complete one key's insertion. So, it is more suitable to batch insert larger scale keys. Here, we also compared the overall performance of above comparison algorithms. When batch insert keys with different size into a 5M B+-tree, the overall performance of above comparison algorithms shows as Figure 11.

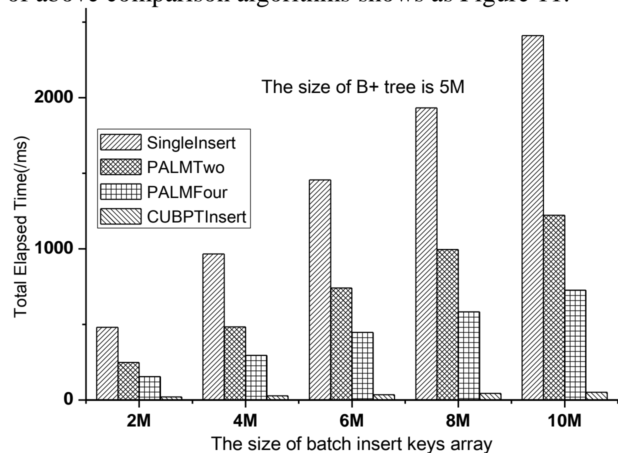


FIGURE 11 The overall performance of above algorithms when insert uniform keys into a 5M B+_tree

When batch insert keys with different size into a 10M B+-tree, the overall performance of above comparison algorithms shows as Figure 12.

From figure 11 and figure 12, we know that when tree size is 5M, with the size of keys set increases, the overall speedup of *PALMTwo* and *PALMFour* are almost no

change which approximately 1.95 and 3.2.However, our overall speedup increases from 24.4 to 47.4; when tree is 10M,the overall speedup of *PALMTwo* and *PALMFour* are almost no change too which approximately equals to above. Meanwhile, our speedup increases from 18.5 to 42.8. By analyzing the insertion and overall performance,we know that overall speedup is far more than the speedup of insertion stage for our algorithm. This is because performance gains in search and sorting stage far more than transfer cost.

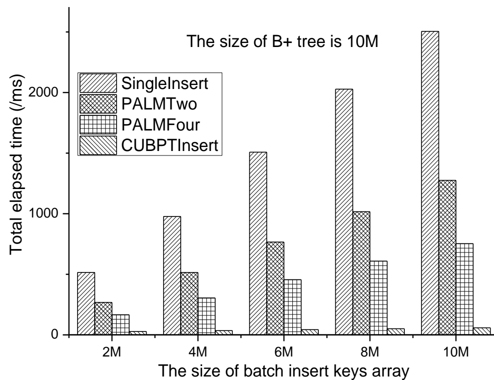
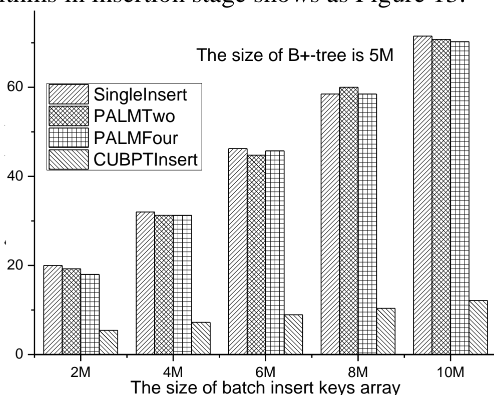


FIGURE 12 The overall performance of above algorithms when insert uniform keys into a 10M B+_tree

4.2.2 Highly skewed

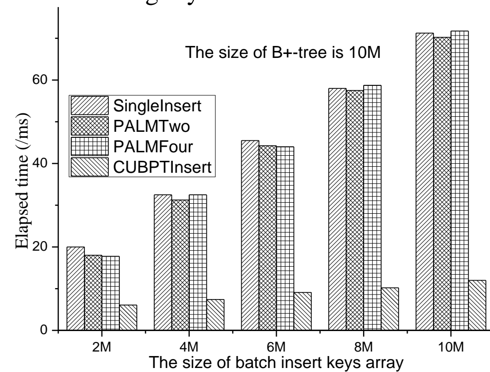
Here, the size and usage of data sets is similar with uniform distribution. The difference is that the number of leaf nodes which to be inserted are less than 5% and 80% or more keys are inserted to one leaf node. In the same way, we also focus on insertion stage and overall performance of above algorithms. Next, we analyse the elapsed time of above algorithms in insertion stage firstly. When batch insert highly skewed keys with different size into a 5M B+-tree, the elapsed time of above comparison algorithms in insertion stage shows as Figure 13.



FIRURE 13 The insertion stage performance of above algorithms when insert highly skewed keys into a 5M B+-tree

When batch insert different scale highly skewed keys into a 10M B+-tree, the elapsed time of above comparison algorithms in insertion stage shows as Figure 14.

Here, all of the above comparison algorithms do not contains the search time too. With the size of keys set increases, the performance of *PALM Two* and *PALMFour* have received almost no improvement. The reason is that Performance gains, which obtained from multiple threads execution are offset by cost of threads creation and synchronization. On the contrary, our speedup increases very obvious; when tree size is 5M, the performance speedup of our algorithm increases from 3.8 to 5.91. When tree size is 10M, the performance speedup of our algorithm increases from 3.7 to 6.1.The reason is same with uniform distribution. However, the performance promotion effect of our algorithm has a little decline. This is because the amount of data which processed by GPU is reduced slightly.



FIRURE 14 The insertion stage performance of above algorithms when insert highly skewed keys into a 10M B+-tree

To compare the overall performance of above algorithms, the total elapsed time when batch insert highly skewed keys with different size into a 5M B+-tree is described in Figure 15.

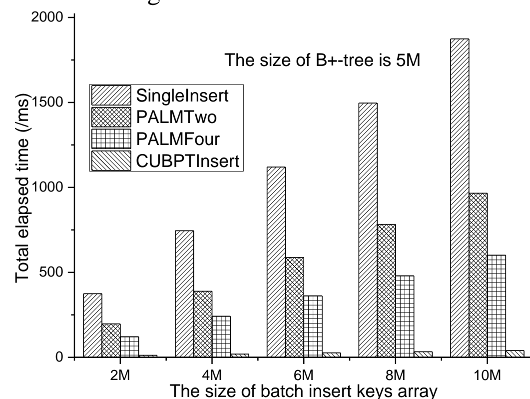


FIGURE 15 The overall performance of above algorithms when insert highly skewed keys into a 5M B+_tree

When batch insert highly skewed keys with different size into a 10M B+-tree, the total elapsed time of above comparison algorithms is described in Figure 16.

In figure 15 and figure 16, our algorithm contains transfer time. From them, we know that when the size of keys set increases, the overall performance speedup of *PALMTwo* and *PALMFour* remain unchanged which approximately 1.9 and 3.1. Meanwhile, our overall speedup continues to increase. If tree size is 5M, our overall speedup increases from 30.5 to 47.1. If the tree size is 10M, our overall speedup increases from 28.9 to 47.2. However, from above analysis, PALM can not get

any performance gains in insertion stage. Here,the overall speedup is mainly obtained by sort and search stages.For the same reason, our overall speedup is far more than the speedup of insertion stage.

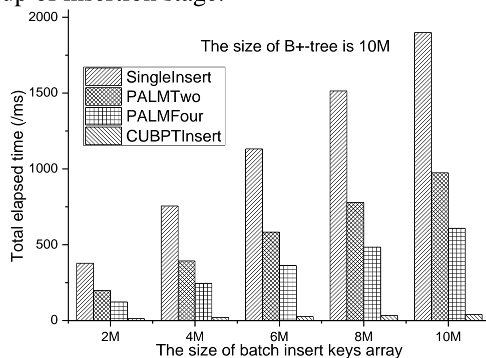


FIGURE 16 The overall performance of above algorithms when insert highly skewed keys into a 5M B+_tree

All in all, our algorithm has obvious performance advantage in both stages. Meanwhile, our speedup increases with the size of keys set which different with *PALM*. Especially, if keys distribution is highly skewed, our speedup is more obvious. So, our algorithm is more suitable for highly skewed keys insertion.

5 Conclusion and future research

B+-tree is one of the widely used index structures. To improve the process of keys insertion, several batch algorithms are proposed. Meanwhile, they cannot make full use of the parallel throughout of current GPU. So, a lock-free batch Insertion algorithm on GPU is proposed in here, which consists of two stages. The experimental results show that when batch construct tree, the total speedup of our algorithm can achieve 73.34 ~ 84.36. When batch insert keys, the total speedup of our algorithm can achieve 18.5 ~ 47.4. Here, we only research on batch construction and insertion algorithm. In the future, we will focus on batch search and deletion algorithm on GPU.

Acknowledgments

We would like to thank the financial supports of the National Science Foundation of China (No.61340016) and Strategic Emerging Industry projects of Guangdong Province, China (No.2011A010801008).

References

[1] Moore S K 2008 Multi-core is bad news for supercomputer *IEEE Spectrum* **15**
[2] *NVIDIA Corporation. NVIDIA CUDA Programming Guide, version 2.3* 2009 7-15
[3] BingSheng He, Ke Yang, Rui Fang, etc. 2008 Relational Joins on Graphics Processors *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* 511-24
[4] Changkyu Kim, etc. 2010 FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs *Proceedings of the 2010 international conference on Management of data* 339-50
[5] Fix J, Wilkes A, Skadron K 2011 Accelerating Braided B+ Tree Searches on a GPU with CUDA *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance*
[6] Sang-Wook Kim, Hee-Sun Won 2001 Batch Construction of B+-Trees *Proceedings of the 2001 ACM symposium on Applied computing* 231-5
[7] Jiangmiao Liao, Hu Chen, Yixia Yuan, et al. 2010 Parallel Batch B+-tree Insertion on Multi-core Architectures *Fifth International Conference on Frontier of Computer Science and Technology* 30-5
[8] Sewall J, Jatin Chhugani, et al. 2011 PALM: Parallel Architecture Friendly Latch-Free Modifications to B+ Trees on Many Core Processors *37th International Conference onVLDB* 795-806
[9] Kaczmarski K 2011 Experimental B+-tree for GPU *ADBIS2011 Research Communications, Austrian Computer Society* 232-40
[10] Yan Weiming, Wu Weimin 2007 *Data Structures* Tsinghua University Press 238-47
[11] Vliant L G 1990 A bridging model for parallel computation *Communications of the ACM* **33**(8) 103-11
[12] Hoverock J, Bell N 2011 *Thrust: A parallel template library*
[13] Reinders J 2007 *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism* O'Reilly Media 78-9
[14] Kaczmarski K 2012 B-Tree Optimized for GPGPU *In Proceeding of OTM Conferences* 843-54
[15] Josuttis N M 2011 *The C++ Standard Library* Addison-Wesley

Authors

Yulong Huang, born in 1979, in Jian city, Jiangxi Province, China

Current position, grades: a lecturer of Computer Science and technology in School of Computer and information, Anqing Normal University.
University studies: He has received B.S degree in Computer Scicence and Technology from Wuhan University in 2002, received MSc and PhD in computer application technology from Guizhou University and South China University in 2005, 2013 respectivly.
Scientific interest: Parallel Computing, Internet of Things, Database Technology and Distributed System.

Benyue Su

Current position, grades: Professor and the Chair of the School of Computer and Information, Anqing Normal University in China. He is the senior member of China Computer Federation (CCF) and council member of Technical Committee of Geometric Design and Computing, CSIAM. He is also a senior member of the ACM.
University studies: Ph.D. in Computer Science from the Hefei University of Technology (HFUT), China, in 2007.
Scientific interest: visual computing, data analysis techniques, computer aided geometric design, computer graphics and digital image processing.
Publications: more than 60 refereed journal and conference papers in these areas.

Jianqing Xi

Current position, grades: a professor of software engineering in South China University of Technology.
University studies: MSc in software engineering and PhD in computer architecture from National University of Defense Technology in 1988 and 1992 respectively.
Scientific interest: Database and Data Warehouse, Distributed system based on P2P technology, Chinese information process, Data management and Software Development technology.