B$^+$-Tree Optimized for GPGPU

Krzysztof Kaczmarski

Warsaw University of Technology,
ul. Koszykowa 75, 00-662 Warsaw, Poland
k.kaczmarski@mini.pw.edu.pl

Abstract. GPGPU programming requires adjusting existing algorithms and often inventing new ones in order to achieve maximum performance. Solutions already created for supercomputers in nineties are not applicable since SIMD GPU devices are in many aspects different than vector supercomputers. This paper presents a new implementation of B$^+$-tree index for GPU processors. It may be used in cases when processing parallelism and order of elements are equally important in computation. The solution is based on data representation optimal for GPU processing and an efficient parallel tree creation algorithm. We also deeply compare GPU B$^+$-tree and other solutions.

1 Introduction

General Purpose Graphic Processor Unit (GPGPU) programming allowing significant acceleration in various algorithms focused great attention of research community. Hundreds or maybe thousands of applications already demonstrated even two orders of magnitude speed up when compared to classical CPU based sequential algorithms. An efficient index structure based on B$^+$-tree allowing GPU algorithms to maintain an ordered key-value collection is another crucial structure for many uses not only for data intensive algorithms in databases and OLAP but also in geometrical computation, graphics, artificial intelligence, etc., wherever a sorted collection of keys with ultra fast search, insertion, deletion and parallel element access is needed. This paper advocates a novel implementation of a tree index based on B$^+$-tree on GPU. We also compare it to already well known of CSB$^+$-tree and other GPU solutions.

Our previous studies in this field involved experiments with parallel kernels working with B$^+$-tree pages. That preliminary work motivated us to improve the overall memory representation as well as search and insertion. We gained large speed up also utilizing GPU streaming multi-processors (SM) much better.

In the rest of the paper we assume that the reader is familiar with general concepts of GPGPU programming like: SIMD, SIMT, thread, thread block, host, device, global and shared memory, synchronization at block and global levels, coalesced memory access and others. For deeper study of these topics one should consult programming guide [1], best practices guide [2] or other public tutorials.

R. Meersman et al. (Eds.): OTM 2012, Part II, LNCS 7566, pp. 843–854, 2012.

1.1 Motivation

Our previous works shown [3] that GPU B$^+$-tree is excellent for parallel access of all the elements being much better than CPU based STX B$^+$-tree [4] and for searching being faster other GPU collections like Thrust [5]. Similar results in case of searching were achieved by [6], where authors leverage logarithmic power of B-tree searches and parallel nature of tree page search. However, a new element insertion time was unsatisfactory due to the following reasons:

- Each inserted element had to be copied between host and device. Inserting single element initializes this very expensive transfer for small amount of data making insertion process very slow when compared to pure CPU solutions.
- Inserting a single element into a large ordered array is poorly conditioned for GPU. Array elements shifting requires a lot of memory transfers equipped with global threads synchronization.

This bad behaviour could be improved by bulk insertion of large number of elements at once which is also better for parallel vector processor like GPU. However, this required to completely redesign all memory structures and rebuild all algorithms. As a result we propose a new B$^+$-tree parallel creation strategy working bottom up. This also allows for merging two trees or inserting a large number of elements into already existing tree.

1.2 Other Solutions

In [6] authors present interesting analysis of B$^+$-treesearching. They propose to search for one element per thread block and use brute force search inside single page which is similar to our ideas from [3].

A CPU bottom-up batch construction of a B$^+$-tree is presented in [7] while binary tree creation was studied in [8]. The first one is similar to our approach except for not being parallel. Authors do not explain how to keep node pages in all the levels correctly filled. We show how to solve this problem for various number of elements by performing repagination of two rightmost pages in each level.

FAST tree creation from 64 million tuples is done in less then 0.07 seconds on a similar test environment. We have to indicate that this value is not including time of copying data from CPU to GPU memory and input set of tuples is already sorted. Our system is able to perform similar operation on 30 millions of elements in 0.1 milliseconds. Our result is possible due to lower trees (binary tree compared to B$^+$-tree) and therefore fewer creation steps.

Indexing Structures and SIMD Processors. Indexing structures based on various trees are studied for years. Its utilization is now a key part of most of commercial database systems followed by many studies on improving their efficiency. Interesting properties are demonstrated by cache conscious trees and cache conscious

B+-trees for the main memory [9,10]. Among other tree structures CSS-Trees are closest to GPU requirements. Important optimizations of binary trees for SIMD processors were presented by Kim et al. [8]. We further analyse both works in comparison to our GPU B+-tree.

Generally the task of maintaining index structures for a SIMD GPU processors is under research and not yet offered publicly for programmers as a ready to use library, except for Thrust [5] implementing a very flexible and efficient collection implementation which mimics C++ STL and several libraries offering array sorting for GPU [11,12]. They offer only flat arrays and are not really solving the problem we took them as one of reference points for our work because of lack of other ready to use solutions.

Cache Conscious Structures. The main goal of a CSS-Tree [9,10] is the best possible utilization of memory cache by ordering nodes and leafs and also saving cache line by removing unnecessary pointers. Thanks to these techniques memory reads are much faster due to more accurate cache hits. The more memory can be cached, the better performance one may get. In a CSB+-tree each node page points only to a single child page. Other children locations are found by pointer arithmetic. This saves memory but also requires child pages to be placed in certain places managed by higher level structure called a node group. Optimum insertion performance can be achieved for preallocated node groups which is in the worst case consuming a lot of unused memory.

GPU B+-tree could also benefit from cache improving technique but the most important optimization (especially for not Fermi devices without global memory caching) is coalesced memory access. In our implementation it is achieved with array based node and leaf pages plus on-chip shared memory. A child pointer is read only once to find proper child page to jump to. Therefore saving main memory by child pointers removal would not increase performance. However, we agree that memory optimization are critical for GPU because of significant space limitations. Since our tree is placed in memory by layers a technique similar to CSB+-tree could be used.

FAST [8] presents a solution for SIMD processor optimization by organizing binary sub-trees in the memory in such a way that cache misses and memory latency is minimized. Although this solution is very flexible and can be used for multi-core CPUs with SSE extensions and various SIMD architecture it is limited only to binary trees. The same results are obtained in our system but with trees of higher orders. We can utilize different processor SIMD lanes by changing page order. In case of CUDA and GPU architecture with half warp of 16 threads an optimal page would have therefore order $p = 16$. It allows for single instruction coalesced memory read and perform all in page binary key search just in on-chip ultra fast shared memory. Practice showed that even higher page orders like 32 or 64 perform very well due to sequential nature of binary search. It takes from several steps ($\log_2 64 = 6$) to finish searching inside a page. Therefore even bigger pages which need more than one memory access to be copied to shared memory are justified (2 reads may copy up to 64 keys, 4 bytes each). When simultaneously searching for keys also memory latency may be hidden if number

of queries exceeds number of SMs by factor of 4. In case of NVIDIA GTX 280 it is 120 keys searched in the same time. Also bigger numbers are available as we present in our experiments.

2 Optimized GPU B$^+$-tree

B$^+$-tree contains unique keys and values for the keys stored in pages of two types: nodes and leaves. If a tree has order n then leaves contain $k_{0\ldots n}$ keys and $v_{0\ldots n}$ values. Node pages contain $k_{0\ldots n}$ keys and $p_{0\ldots n+1}$ pointers to child pages. Keys in a child page pointed by p_0 are smaller than key k_0. Keys in a page pointed by p_{i+1} are greater or equal to k_i. Detailed description of all the properties of the B$^+$-tree may be found in the literature [13,14].

An important feature of B$^+$-tree (opposite to B-tree [15,14] and other tree structures) is that all the leaf pages may be traversed from left to right to get all the values in order of keys. There is no need to perform recursive searches from root node to bottom of the tree. It is enough to get a pointer to the first leaf page and keep pointers to the next page (to the right) in order. Visiting all elements in order is then very similar to going through an ordered list. This property, essential for many algorithms, is also present in our solution.

2.1 Architecture

Let us p denote order of the tree or the number of elements stored in one page. In each leaf we need $2p$ compartments for keys and values plus a counter for number of occupied places and a pointer to the next page in order. In the nodes we need p places for keys and $p+1$ places for child pointers plus a counter but no horizontal pointer to the next page. Additionally, we allocate one cell for storing minimal key in a subtree beginning at given node page, which is used during bulk nodes creation. Due to GPU optimization requirements these structures need to be split into separated arrays which will be concurrently accessed by SIMD threads as shown in figure 1.

Node pages are divided into four arrays (*keys, child pointers, page size, min keys*). First two arrays contain elements grouped for pages which do not have to be stored in order, in opposite to a traditional breadth-first fashion. This allows for many simplifications in parallel algorithms. In our example root page has got index 2, while the two children indexes 0 and 1 respectively. The last two arrays are indexed by page numbers.

Leaves are similarly divided into four arrays (*leaf keys, leaf values, next leaf pointer, leaf size*). Again keys and values are grouped for leaves. In our example (fig. 1) all leaves are ordered from left to right which again is not obligatory. Pointers to a next leaf and leaf sizes are indexed by leaf numbers but the first element in the pointers array contains index of the first leaf, so leaf of index 0 keeps its next pointer in element 1, leaf of index 1 keeps pointer in element 2, etc.

The representation of leaf pages in a flat array allows also for concurrent access of all keys and values which is one of the most important advantages of GPU B+-tree over a CPU one and also other collections.

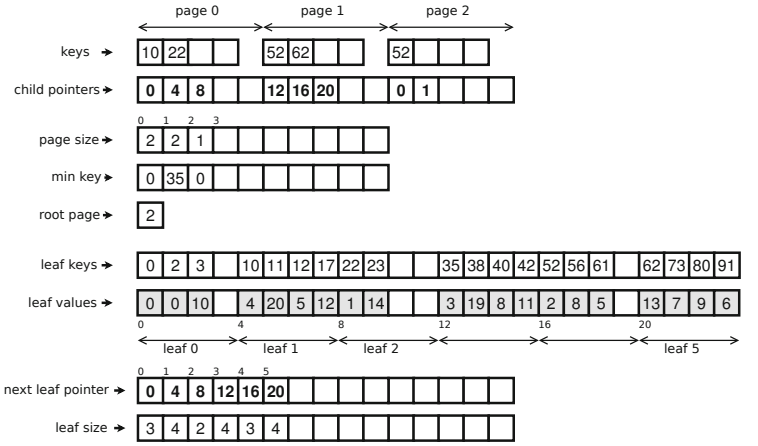

Fig. 1. A sample array based GPU B+-tree of order 4 and height 3 with 20 random elements forming 6 leaf pages

2.2 Bulk and Single Element Insertion

Single Insert. When inserting a single element into a B+-tree we follow the classical algorithm defining the procedure of inserting k and v into a tree:

1. Finding leaf page for insertion.
2. Inserting an element by sorting it into page array.
3. If necessary perform splitting in a loop going up the tree and move dividing elements into parent pages.

In details, it is composed of more steps due to necessity of threads synchronization. The first part works as it is explained in the section 2.3. This part ends by pointing to a page which is a leaf and contains the key being inserted (if it was already existing in a tree) or to the proper leaf to put the key and value in. The second step, inserting an element, must be further divided into:

1. Check if the element is present in the page or not. This step must finish completely before proceeding to the next one. Similarly to other steps it is a separate GPU kernel.
2. Make space for the new element in the proper place: copy elements greater then k aside, when all threads finished, copy elements back making a gap for the new element. This requires two kernels.
3. Finalize insertion: put the new element in place, increase element counter and decide if splitting of the page must be done.

Splitting itself is much simpler than an element insertion, does not require threads synchronization and may be done in a single step. However, inserting a new element into a parent page, which is a consequence of child page splitting, again needs several steps due to making a free place in the sorted page array. We can only skip checking if the element exists in a page, because it is guaranteed by the properties of the tree that it is not present there.

An important and quite common improvement on the device side is tracking pages when going down during the search procedure. Later if splitting is necessary it is much faster to go up to parent page using this path.

Bulk Creation. The above method of inserting a single element is extremely inefficient on GPU because of necessity of several kernels synchronisation, many memory copying and low utilization of GPU cores. This can be greatly improved in cases when we index a large set of key-value pairs. In the contrast to single element insertion, it uses all GPU cores with coalesced memory access if possible.

The idea is based on an observation that a tree creation may be done bottom-up if we start from sorting all keys in leaf pages rather than insert elements one by one. Then having leaves ordered properly as if they were created by a normal top-down insertion we can build the tree level by level from the lowest pages to the top. Due to balanced character of a tree this process is simple and what is more important for us may be parallel. The algorithm assuming that we insert n keys and n values (page order is p) contains the following stages:

1. Allocate new device side leaf array, CPU
2. Copy keys and values into the leaf array, CPU + GPU
3. Sort leaves, GPU
4. Reorganize leaf pages if necessary, GPU
5. Create the lowest level node pages, GPU
6. Until reaching root node create subsequent node pages from lower level, GPU
7. Repair pages if necessary to keep B$^+$-tree properties, GPU

The first three steps are obvious and need no comments. Step 4 is done due to improving future insertions. After step 3 all keys tightly fill keys array leaving no free spaces in leaf pages (see fig.2 point 1). Reorganization of keys in pages may change this by creating from 1 to $\frac{p}{2}$ empty elements in each page ($f \in [\frac{p}{2}, p]$) for the price of more leaf pages. Later when an element is inserted there will be no need to split the leaf page (at least at the beginning). Amount of empty space may vary in different situations.

Step 5 and 6 create concurrently node pages upon pages from the lower level. Each thread reads value from appropriate index in the leaf keys or node keys arrays and puts it into a new page in proper place calculated upon properties of the tree: page order p, page filling factor f and current level number. Only very simple pointer arithmetic is done in this step and all the process is fully concurrent within one tree level. This process takes $\log_{f+1}(n)$ steps.

The last step 7 may be necessary in cases when number of elements in a page violates tree property of containing from $\frac{p}{2}$ to p elements in each page except the root node. This is presented in fig. 2 where one leaf page and one node page

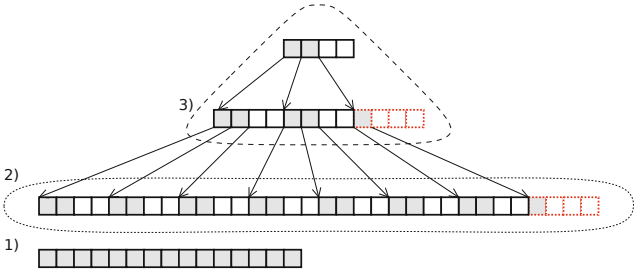

Fig. 2. Stages of bulk creation, page order $p = 4$ and filling factor $f = 2$: 1) after leaf keys sorting, 2) after leaf keys reorganization, 3) after node pages creation. Due to properties of B+-tree the last page at each level except root may have to be repaired (marked by dotted pages).

contain just one element instead of required two. Page repairing may be done by moving some of the keys from the predecessor page or by joining two last pages. The strategy depends on the page filling factor chosen á priori. Due to possible changes in pages on all levels this process must be done from top to bottom in synchronized steps.

Experiments show that although sorting is the most expensive operation, overall performance is anyway much better then on CPU counterpart.

Bulk Insert. If a tree is already created and we need to insert another portion of elements then we can still use the bulk algorithm. Actually, the task may be reduced to merging two arrays of key-value pairs. Since all elements in the tree are kept in leaf array (however leafs are not physically in order there) we may append new set of element, sort them and recreate node pages. The overall algorithm may look like this:

1. Dispose page nodes on the device, CPU
2. Allocate new leafs array if necessary, CPU
3. Copy new key-values pairs appending leafs array, CPU + GPU
4. Dispose old leafs array, CPU
5. Sort leaves array, GPU
6. Mark duplicates GPU
7. Remove duplicates and unused space, GPU
8. Allocate new page nodes array, CPU
9. Create page nodes for leafs array, GPU

As previously the most costly parts are sorting and removing duplicates. Duplicates may be marked in a single parallel SIMD operation. Removing them can be done with scan based parallel primitives which are very efficient for GPU (for example in Thrust library shipped with CUDA SDK).

2.3 Searching

Binary Searching. In a standard B^+-tree searching is done by a single thread going from root node down to leaves and doing binary search in each node page. This algorithm is not efficient for GPU because it can use only one thread and memory latency cannot be effectively hidden. It is also not using all memory bandwidth since it reads only one memory location at a time. However, among its advantages is a very simple kernel construction. It only contains two nested loops. One going top-down in a tree and the second one performing binary search in a tree page. Although theoretically not efficient proved to be the best option due to no synchronization needed and minimal instructions set (see fig. 4, left).

Combined Searching. Another possibility is not to use binary search inside a page but rather use all GPU bandwidth by trying to locate searched element by reading all the page node keys by parallel threads. This brute-force searching could be faster for very small pages in which all keys can be read in single coalesced memory access. We have to admit that this method although promising was not very efficient in experiments due to complicated nature of parallel localization of a key in a sorted array. Let us say that the page order is 4, we are looking for 9 and threads have read four keys respectively: $\{4, 5, 10, 20\}$. No thread can decide what to do without a cooperation with other threads. Actually each thread have to read its key and also the key of the neighbour. If the searched key lays between the two then proper place in the array is located. In our case thread t_1 reading keys $k_1 = 5$ and $k_2 = 10$ may decide that searching should continue in a child page pointed by pointer p_2. Due to half-warp based coalesced read and block based synchronization this method may be efficient only for trees with pages smaller than 16 elements. This strategy is almost one order of magnitude faster that searching in a flat Thrust sorted array (see fig. 4, left).

Brute Searching. Much simpler way to use all the possible GPU memory bandwidth is to perform a brute force search inside leaf keys without touching the tree nodes at all. In the naive solution, we run as many threads as there are available key slots in leaf pages. Each thread reads one key. If it matches the searched one then this thread reports success. If other threads have not started searching yet, may be cancelled in such case (improving performance only in optimistic scenarios). Obviously this method although the simplest is the most expensive since it does not use the information that the keys are sorted inside the pages. We implemented it mostly for comparison with other methods (see fig. 4, left).

Multiple keys searching. A completely different approach to parallelism is searching for multiple keys in the same time. Since each binary search uses one thread and all threads in a block should perform same computation as much as possible avoiding expensive conditional statements and thread dependent loops, we decided to run one minimal block of 16 threads for each searched key. This strategy cannot utilize fully all streaming processors but may at least try to hide memory latency. Also, as in case of normal binary search, coalesced memory reads cannot

be done (see fig. 4, right). Another possibility would be to search for 2 keys (1 key per half-warp) in single block as shown in [8].

2.4 Elements Traversal

This operation is done by visiting logically all values in a single SIMD parallel operation. Similarly to brute search we run as many threads as keys in the leaf pages. Each thread performs desired operation and exits. The implementation is clean and very simple due to *embarrassingly parallel* characteristics of this operation and the fact that we store all leaf keys and values in single array. This kind of tasks are known to scale excellent for many streaming processors since threads blocks are automatically aligned along SMs. This operation may be further modified to allow in order traversal by using single block of threads and visiting only one leaf page at a time for the cost of sacrificing massive parallelism.

3 Runtime Results of GPU B+-tree

The test environment constituted a machine with Intel's 3Ghz CPU Core2 Duo, 4GB of RAM and single NVIDIA GTX 280 card (1.242 GHz, 240 cores and 896MB of DDR3 memory). As reference points in the tests we used GPU Thrust library in version 1.3.0 [5] and for the CPU B+-tree implementation the fastest RAM based read-only solution: STX B+Tree C++ Template Classes in version 0.8.3 [4].

All the programming was done in NVIDIA CUDA C [16], currently probably the most widely used language for GPGPU programming but it may be easily ported to any other environment e.g. OpenCL making it fully operational on all hardware platforms.

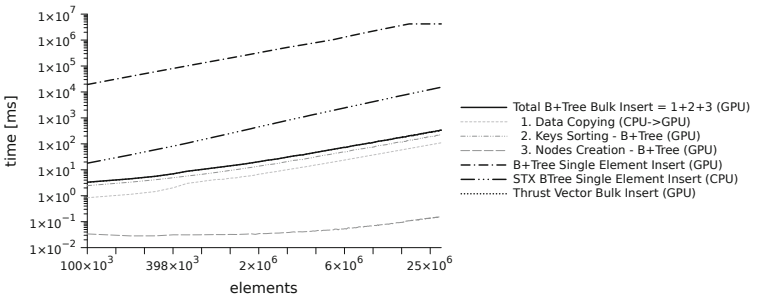

Fig. 3. The plot shows average time creating a tree from a set of elements. In case of single element insertion elements are inserted one by one. Bulk procedures takes all elements at once. Thrust performance not visible due to being covered by B+-tree line. Axis Y: size of a tree. Axis X: time in milliseconds.

Tree Creation and Element Insertion. In our case GPU B$^+$-tree bulk creation from a set of random keys is around one order of magnitude faster than CPU STX which cannot do this kind of bulk operation. Figure 3 shows that bottom-up tree creation is dominated by keys sorting and host to device memory copying. Page nodes creation itself takes only about 0.1ms even for 30 millions of keys.

An interesting way of creating an index quad-tree on GPU was published by Zhang et al.[17]. The process uses GPU to go bottom-up of a pyramid. Although the algorithm is rather data intensive and many helper arrays must be used the parallel procedure is almost an order of magnitude faster than CPU based sequential one, which is similar to our results.

Single element insertion is the most complicated and expensive operation. Page shifting requiring threads synchronization slows down GPU. This badly conditioned task's performance is much worse than in case of CPU STX (see fig. 3).

Because of a bug in Thrust library, we could not test its behaviour in inserting single element into sorted vectors. FAST tree do not consider single element insertion at all.

Searching and Simultaneous Searching. We implemented three different methods for single key searching and a parallel massive variant of the binary search. In the first case we gained speed up over already known sorted collections search by both binary and combined searching.

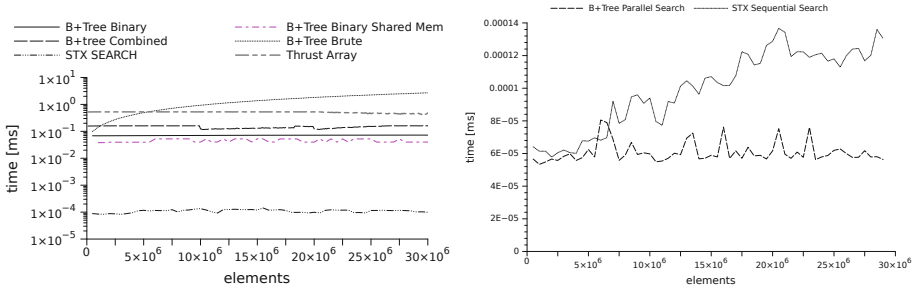

Fig. 4. Comparison of searching times. **Left:** Single element search. CPU STX outperforms other methods. Binary B$^+$-tree is the fastest on GPU. Brute search quickly becomes slower than any other method. **Right:** Simultaneous key search done for a set of 10.000 random keys.

GPU B$^+$-tree (including copying keys from CPU to GPU and returned values back) without shared memory utilization and global memory caches can do 16.5k queries per second and 25.6k with shared memory and code tuning (see fig.4, left). FAST GPU queries achieves excellent result of 85 millions of queries per second [8]. It is not known if this value includes memory copying.

GPU B$^+$-tree simultaneous search for 10k keys achieved 17.75M of searches per second. When compared to the fastest CPU library we achieved better performance even for smaller trees. In this case CPU shows time increasing with growing tree size while GPU time remains almost constant (see fig.4, right).

Albeit, all our queries are repeated 100 times and the measured time value was averaged we observed much noise from the operating system which was tremendously changing results. Even changing kernel version in the operating system could influence results by 10%.

All Elements Traversal. B+-tree pages are similar to normal arrays but they are usually not filled in 100% – in the worst case even half of the threads are wasted. This is why deviation between Thrust and GPU B+-tree increases linearly with growing number of pages in the tree. We observe that B+-tree performance is very close to Thrust flat device array. FAST and CSS-Tree research do not report this kind of experiments.

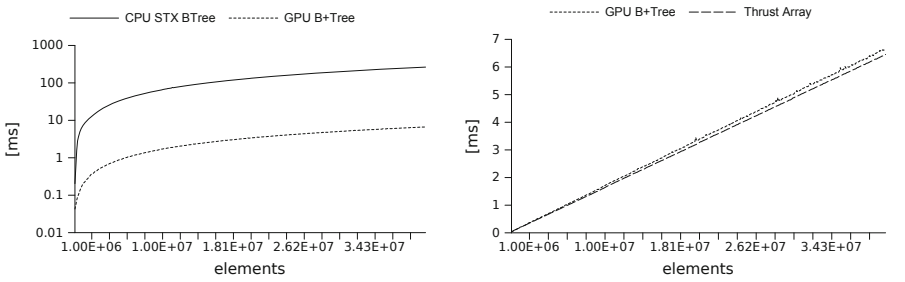

Fig. 5. Results of an experiment visiting all values of a collection and performing a scalar operation. **Left:** CPU B+-tree cannot stand competition with GPU and becomes much slower very quickly. **Right:** Thrust device vector and B+-tree are very close together. Even for large sets B+-tree is not negatively influenced. Test run from 100k to 40 millions of elements.

4 Conclusions and Future Works

The presented GPU B+-tree implementation achieved better results than other GPU solutions and CPU B+-tree. Comparing to our previous works we effectively improved insertion time by a new bottom-up algorithm and searching by improved memory management and alignment. Thanks to fast parallel sorting we may create a tree or merge trees much faster than it can be done on CPU.

An index creation from a random set of keys is not really observable slower than a flat array sorting done on GPU by Thrust library. It is around 100× faster than FAST GPU index based on a binary tree and around 10× faster than CPU STX library. Also in simultaneous elements traversal CPU code is 100× outperformed. Parallel search of 10k elements in a tree of 30 million keys is about 2× faster than ultra fast CPU code.

Further usage of constant memory, texture buffers and minimization of data transfer to kernels may significantly improve the solution. Also unused threads for non existing elements inside pages may be saved. If a list of all the pages

occupation factor is stored on the host side then a kernel execution processing single page could use this information to calculate optimal number of threads for each page individually.

We experienced that CPU B$^+$-tree implementation may achieve impressive speed in single key searching. GPU obviously wins in parallel computations done on all elements in the collection stored in leafs. An almost straightforward idea would be to combine the two models of computation and create a collection with indexing, i.e. upper $h-1$ levels of the tree stored on the CPU side and leaf pages with values stored in GPU device. This could allow for faster searches and parallel elements visiting without any compromise.

References

1. NVIDIA Corporation, NVIDIA CUDA C programming guide version 4.0 (2011)
2. NVIDIA Corporation, CUDA C best practices guide (2011)
3. Kaczmarski, K.: Experimental b+-tree for gpu. In: ADBIS 2011, vol. (2), pp. 232–241 (2011)
4. Bingmann, T.: STX B+ Tree C++ Template Classes v 0.8.3 (2008), `idlebox.net/2007/stx-btree`
5. Hoberock, J., Bell, N.: Thrust CUDA Library v.1.3.0 (2011), `code.google.com/p/thrust`
6. Fix, J., Wilkes, A., Skadron, K.: Accelerating braided b+ tree searches on a gpu with cuda. In: Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance, A4MMC (2011)
7. Kim, S.-W., Won, H.-S.: Batch-construction of b+-trees. In: Proc. of the 2001 ACM Symposium on Applied Computing, SAC 2001. ACM, USA (2001)
8. Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A.D., Kaldewey, T., Lee, V.W., Brandt, S.A., Dubey, P.: FAST: fast architecture sensitive tree search on modern cpus and gpus. In: Proc. of the 2010 Int. Conf. on Management of Data, SIGMOD 2010, pp. 339–350. ACM, New York (2010)
9. Rao, J., Ross, K.A.: Cache conscious indexing for decision-support in main memory. In: Proc. of the 25th Int. Conf. on Very Large Data Bases, VLDB 1999, San Francisco, CA, USA, pp. 78–89. Morgan Kaufmann Publishers Inc. (1999)
10. Rao, J., Ross, K.: Making b+-trees cache conscious in main memory. ACM SIGMOD Record (January 2000)
11. Cederman, D., Tsigas, P.: Gpu-quicksort: A practical quicksort algorithm for graphics processors. ACM Journal of Experimental Algorithmics 14 (2009)
12. Harris, M., Owens, J.D., Sengupta, S.: CUDA Data Parallel Primitives Library (2008), `code.google.com/p/cudpp`
13. Knuth, D.E.: The Art of Computer Programming, Vol. III: Sorting and Searching. Addison-Wesley (1973)
14. Comer, D.: The ubiquitous b-tree. ACM Comput. Surv. 11(2) (1979)
15. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. Acta Inf. 1, 173–189 (1972)
16. NVIDIA Corporation, CUDA C Toolkit and SDK v.3.2 (January 2011), `developer.nvidia.com/object/cuda_3_2_downloads.html`
17. Zhang, J., You, S., Gruenwald, L.: Indexing large-scale raster geospatial data using massively parallel gpgpu computing. In: Proc. of the 18th SIGSPATIAL Int. Conf. on Adv. in Geographic Inform. Systems, GIS 2010. ACM, USA (2010)