Experimental B$^+$-tree for GPU

Krzysztof Kaczmarski

Warsaw University of Technology,
pl. Politechniki 1, 00-661 Warsaw, Poland
k.kaczmarski@mini.pw.edu.pl

Abstract. The main intention of this work is to create a dictionary structure which could benefit from massive parallelism of threads when performing computation on all or a selected set of elements, while having an ability to search for and insert keys very quickly, yet preserving the order of elements. So far, no such structure dedicated for GPU exists. This paper presents results of the first to our knowledge B$^+$-tree implementation in CUDA C language.

1 Introduction

Porting a database application to GPU faces complicated problems like data storage organization, indexing structures, parallel processing of queries, semi-structured data representation and many more. A fast sorted collection implementation would be especially important due to a need of fast searching and inserting algorithms. In our research we faced this problem when dealing with Multidimensional On-Line Analytical Processing (MOLAP) structures for streams of data which not only have to be created and updated but also need to be searched and queried very quickly. In classical CPU database systems in case of unknown data domain, this is addressed by dedicated structures e.g. B-trees.

This paper presents preliminary results of B$^+$-tree structure implementation for a GPU device. The main intentions of this work were to create an ordered key-value collection with the following properties:

1. near constant time of single element insertion in a collection with tens millions of elements
2. fast search for a key or key range
3. prepared for vector data processing

Since NVIDIA CUDA C [1] is currently probably the most widely used language for General Purpose GPU (GPGPU) programming, we chose this standard for our work. However, it may be easily ported to any other solution e.g. OpenCL, making it fully operational on all hardware platforms.

1.1 Sorted Collections for GPU

There are several libraries offering array sorting for GPU [2, 3], but to our best knowledge only Thrust [4] implements inserting an element into a sorted collection in order. It is a highly efficient C++ templates library providing a very

flexible yet efficient collections implementation which mimics C++ STL. The most important container class `device_vector` allows for creating and maintaining a collection of objects on the graphical device. It is still under heavy development but already offers a wide variety of functions. A set of insertion and searching procedures are especially interesting and were used as a reference point for this work.

General task of inserting an element into a sorted array is not well suited for GPU. An efficient implementation for GPU requires all data to be stored in flat tables in order to benefit from coalesced memory accesses. Insertion in an array requires elements shifting which is obviously very expensive and when done by parallel threads requires additional synchronization.

This paper describes an experimental implementation of a B$^+$-tree for GPU improving this situation. An experiment was intended to check if a GPU device is able to perform tree operations efficiently enough to compete with other robust implementations. Although a lot of work on optimizations and improvements can still be done, the results are interesting.

2 B$^+$-tree Stored in a GPU Device

2.1 General B$^+$-tree

B$^+$-tree differs from a normal B-tree [5] by storing values only in leaf pages [6]. It is known for substantial memory savings for the price of theoretically more expensive searches which are not really observable in practice.

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 $k_{0\ldots n-1}$ keys and $v_{0\ldots n-1}$ values. Node pages contain $k_{0\ldots n-1}$ keys and $p_{0\ldots n}$ 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 mainly in [7]. Here we present only the basic algorithms for insertion and searching to make the rest of the paper more clear. All the symbols are also used in the next sections. Procedure of searching for key k:

1. Start from root page.
2. On given page: if $k < k_0$ then $i = 0$ otherwise search for such an index i in keys array that $k_i \leq k < k_{i+1}$.
3. If the page is a leaf page and $k_i = k$ then return success and value is in v_i.
4. If the page is a leaf and there is no $k_i = k$ then return negative result
5. If the page is not a leaf then navigate to child page pointed by pointer at position p_{i+1} and go to 2.

A general key-value (k, v) insertion procedure may result in tree growing (keeping the tree balanced) and is defined by the following steps:

1. Search for page to insert k into. If k is found in one of the leaves then depending on the tree properties replace the existing value with a new one or perform any other operation on the values.

2. If k was not found then a place for insertion is given by k_i in the last searched page (leaf). Insert k and v into this page. If number of elements in the page is greater then allowed n then split the leaf page by allocating a new brother page to the right, move the middle key and all greater keys to this new page and also insert the middle key and the pointer to the new page into the parent. Middle key in the split page becomes the smallest key in the new page. Put horizontal pointers from the old page to the new one and from the new one the the one which was previously next to the old one.
3. If the parent page is full then split it, but without leaving the middle key in the newly allocated page. It is only inserted in the parent page. Continue splitting if necessary.
4. If the root page has to be split, then create two new child pages and leave one middle element in the root.

An important feature of B^+-tree is that all the values pages may be walked in order of keys using pointers between leaf pages. No recursive searches from root node are needed. Figure 1 demonstrates this property of the B^+-tree.

On the other hand, if one needs to visit all keys or values concurrently it is also possible since a list of pointers to all leafs may be easily created while splitting leaf nodes.

2.2 GPU Specific Solutions

In order to optimize the tree storage for the GPU we have to observe that all the pages are similar and regularly built. Let n denote the order of the tree or the number of elements stored on one page. In the leaves we need $2n$ 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 n places for keys and $n+1$ places for pointers plus a counter but no horizontal pointer to the next page. Additionally, we allocate one more place for a key and one for a value for insertion purposes. Summing up, each page requires the same number of cells: $2(n + 2)$. All the B^+-tree may be allocated as a two dimensional array or as a set of these arrays, preserving limited GPU memory. Each page occupies two rows while elements in pages form columns of the array. This is showed in fig. 1: dark grey cells indicate normally unused places reserved for insertion; light grey color marks values stored in leaves. Each leaf page has got one cell with pointer to the next page in order. In the last leaf page this pointer points to null.

Here we identify what is the optimal page order. Properties of multi-core GPU processors guarantee maximum performance, if we are able to keep all the cores busy during all the processing time. If we assign a large number of thread blocks to single streaming processor, compiler and hardware organizes the processing automatically in such a way that if one block of threads waits for data, others may continue. Memory latency is then hidden and significant speed-up may be noticed. A drawback of this solution is that the order of blocks execution is unknown, requiring additional care in implementation of non trivial algorithms. Therefore maximal instructions throughput may be impossible

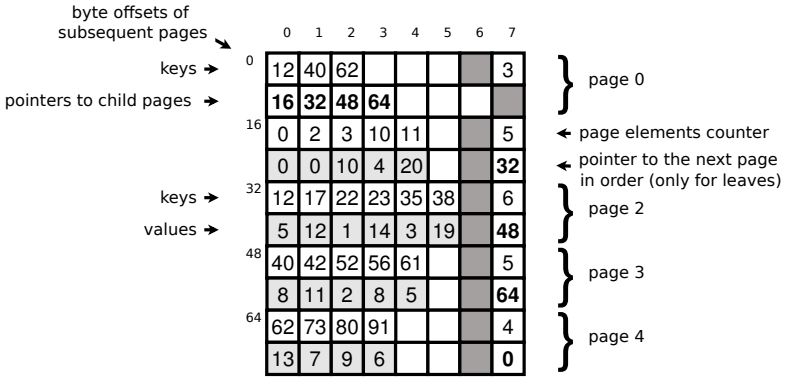

Fig. 1. A sample B$^+$-tree of order 6 and height 2 with 20 random elements forming 3 leaf pages.

because of complicated synchronization between threads running in different blocks.

B$^+$-tree processing may be parallel inside a single page but should not be concurrent on may pages since for example insertion changes the structure of a tree and may conflict with another thread performing insertion on a different page. A complicated page locking mechanism would be necessary. This could be dangerous for efficient GPU threads execution.

GPU may access memory faster if all threads in a block read or write bytes within the same segment. In such cases coalesced memory access is possible. If we allow each thread to read from a distant place in memory (for example during a parallel search procedure) all memory accesses would be sequential degrading memory bandwidth. Therefore GPU threads parallelism should be involved in a single page processing but navigation and control of processing in the whole tree should be on the side of the host. It is also obvious that if we want to use all power of streaming processors, number of elements in a single page should be large enough to keep all cores busy when processing a single page. On the other hand the shorter page the faster searching and inserting elements inside.

We performed experiments with different number of elements plus different number of threads in a block for insertion of 5K, 30K, 100K and 1M of elements. Our observations (fig. 2) follow the above statements. GPU B$^+$-tree has better performance for pages which are almost full. At some point when adding more elements the tree is not growing as often as it used to, but rather filling in existing pages. We can observe this for 1M and 100K of elements. There is an optimal page size of about 2048 or 4096 elements for which pages splitting does not happen very often, but also single page search is not very expensive and may be done by parallel threads in just a few steps. 4096 elements in a page and 256 threads gives 16 blocks of threads and 256 half-warps running concurrently. This configuration used all the resources of our system. We also noticed that for much

bigger trees (40M of elements) it is better to make the page 16384 elements and keep the tree not higher than 2 levels. Then actually size of the page should correspond to predicted number of elements stored in a tree.

For smaller trees we can see that the optimum page size depends on the number of elements in such a way, that all elements of the tree should be in a single page. This is because page splitting is too expensive when compared to a relatively fast page search. Number of threads does not influence performance. The difference between configurations, from 32 to 512 threads is at the level of 1.5% to 1.8%.

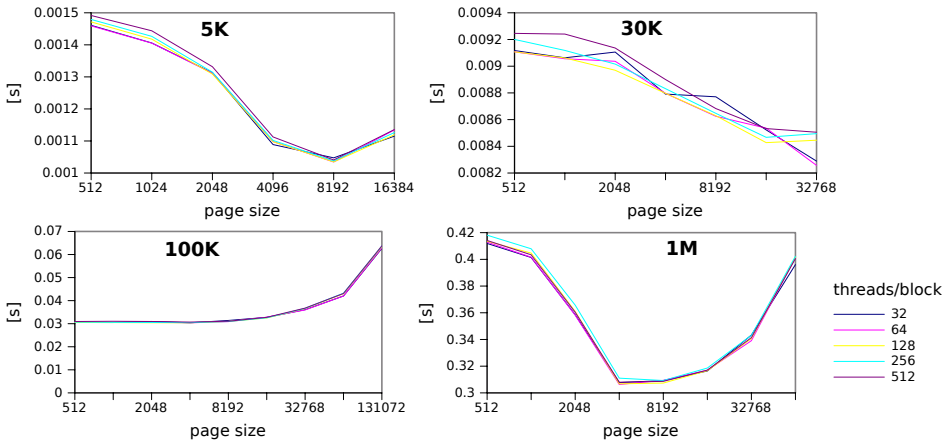

Fig. 2. Testing tree creation time for different configurations of page sizes and threads per block. Four experiments for: 5K, 30K, 100K and 1M of elements. Time is normalized: time of tree creation divided by number of elements.

3 Basic Operations Implementation

For the purposes of the experiment we only implemented the most important operations, i.e.: single element insertion, single key searching and visiting all elements. Deletion and range queries will be analysed in the future.

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 [4] and for the CPU B$^+$-tree implementation the fastest RAM based read-only solution: STX B+ Tree C++ in version 0.8.3 [8].

3.1 Element Insertion

When inserting a new element into a B$^+$-tree we follow the algorithm sketched earlier but divided into more steps due to necessity of threads synchronization.

The first, element searching, part works as it is explained in the section 3.2. This part ends by pointing to a page which is a leaf and contains the inserted key (if it was already existing in a tree) or to the proper page to put the key and value in. The second part, inserting an element must be further divided into the following steps:

1. Check if the element is present on the page or not. This step must finish completely before proceeding to the next one. Like the 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 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. Inserting a new element into the parent page again needs several steps to make 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.

In order to minimize communication between host and device we created an additional structure to keep state of the insertion process between subsequent kernel calls on the device side. In each step a kernel performs the task and writes results to this structure. Then, in the next step another kernel may quickly access the final state of the previous kernel and react properly. Host in most situations does not need to read this information and time consuming transfer over PCI is minimized. Another important improvement is tracking pages when going down during the search procedure. Later, if splitting is necessary it is much faster to go back using this path. If this path is kept on the device side we may again save time.

Runtime results Element insertion is the most complicated and expensive operation. Page shifting slows down GPU parallel processing, because it requires threads synchronization. The performance is much worse than in case of CPU counterpart. However, we may clearly notice that from certain point insertion becomes faster and faster. This is because the tree is already well formed, no new pages are added and new elements are put in existing empty spaces. When inserting random elements B$^+$-tree properties lead to a tree with pages initially half filled. This situation last until root page is full and must be split resulting in tree growth. This can be observed in the plot (fig. 3) as a sudden slow down in different places for different page sizes. The smallest page tree grows up to height 3 quite quickly. The tree with height equal to 2 and page size 2048 can contain maximally $2048^2 - 2048^1 = 4,192,256$ values. Then around 20M elements root splitting is done again and the tree becomes 3 levels high. From this point splitting of leaf nodes happen quite often but not touching the total height. After about 30M of random elements inserted there are more elements inserted

without splitting than with splitting and insertion becomes faster. From this point insertion time tends toward constant value and pages just fill with keys which would be continued until the first node page division would cause root page split and the tree would grow another level. The same situation happens for a tree with page of length 8196 but it reaches height 3 much later while a tree with page of order 16384 keeps the height 2 during all the experiment.

To comment the speed of CPU based STX B$^+$-tree library we have to note that single element insertion is really badly conditioned for massively parallel hardware and additionally the inserted element must be copied over PCI-E bus from the host to the GPU device which is not done in pure CPU solutions. Anyway, STX B$^+$-tree speed is really impressive. Authors claim to focus on speed optimization and the results are really ultra fast but paying with read-only access.

Due to a bug in Thrust library we could not test its behaviour in inserting elements into sorted vectors. We have to note that the error was reported and will be corrected in the next release. We will certainly publish comparison of the two structures as soon as possible.

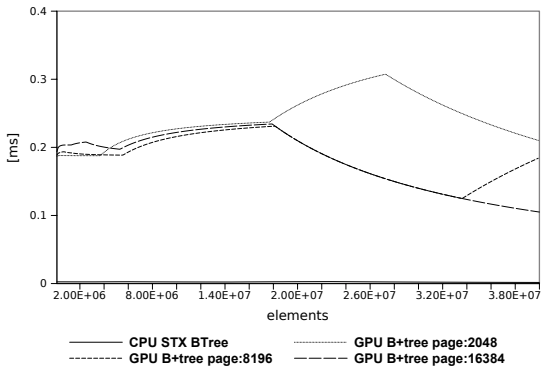

Fig. 3. The plot shows average time of a single random element insertion in an already build tree. Axis Y: size of a tree. Axis X: time in milliseconds. CPU STX outperforms GPU B$^+$-tree and can hardly be seen at the bottom of the plot.

3.2 Searching

Searching procedure for given k must answer a question if the key exists in a collection and if positive, it must return the value stored for this key. Typically searching in a sorted array by one thread is implemented by a binary search [7]. Single thread checks key in the middle of searched array continuing search in left or right part of the array for elements smaller or greater than k. In each step searched area decreases by 2 and locates, in time $O(\log n)$, the wanted element or

assumes that it is absent in the set. This procedure will not work well on a vector machine. First of all single thread cannot by any means load the multiple core processor resulting in very poor memory bandwidth an instruction throughput. Even if we decide to do multiple searches and run threads looking for many keys in the same time we get all of them going into different part of the searched array since they look for different keys. Again a tremendous degradation of memory bandwidth would be observed because coalesced reads are impossible in such case.

In other parallel libraries an element is often found by a brute force algorithm checking all elements in the collection. This could work for a GPU because of its high memory bandwidth. We observed that this approach is working quite well for smaller arrays. In one of the experiments our kernel was able to select a single value from a collection of 8M values in less than 10 milliseconds. Since this solution does not benefit from the fact that our collection is sorted we hope to improve this result in B$^+$-tree. When the collection is divided into pages, we perform parallel search with portion of data and then move to another page.

Runtime results GPU B$^+$-tree searching is done exactly as described before in the general algorithm of B$^+$-tree searching. We created one kernel which reads all keys in a page. If it is a node page then the kernel finds a pointer to the appropriate child page. If it is a leaf then the key must be there or is not existing. Host executes this kernel from root to leaf level of the tree. Kernel autonomously decides where to go next. Host only checks the result when the kernel reaches the bottom of the tree. Again this approach suffers from pages being not filled completely (fig. 4).

3.3 All Elements Visiting

This operation is done by visiting all values in a single parallel operation. For our B$^+$-tree implementation of this task is straightforward assuming that we have the host keeps pointers to all leaf pages.

Runtime results From the GPU's point of view, we need to run as many thread blocks as there are pages l. Number of threads in each block is limited to 512 so it is possible that one thread will have to visit more than one element.

Obviously this strategy can never be faster than visiting a standard array by parallel threads. Although, B$^+$-tree pages are similar to normal arrays they are usually not filled in 100%. Therefore possibly in the worst case even half of the threads are wasted. Hopefully, if we run one block for single page we load the streaming processors to the limits and allow for maximal multitasking hiding the treads which do not do anything. For very small trees this strategy does not work, but for bigger ones it keeps B$^+$-tree performance close to Thrust flat array (see fig 4. Unfortunately very big trees are again slowed down by copying the list of leaf pages to be visited from host to device. It may contain thousands of pointers. This is why deviation between Thrust and GPU B$^+$-tree increases linearly with growing number of pages in the tree.

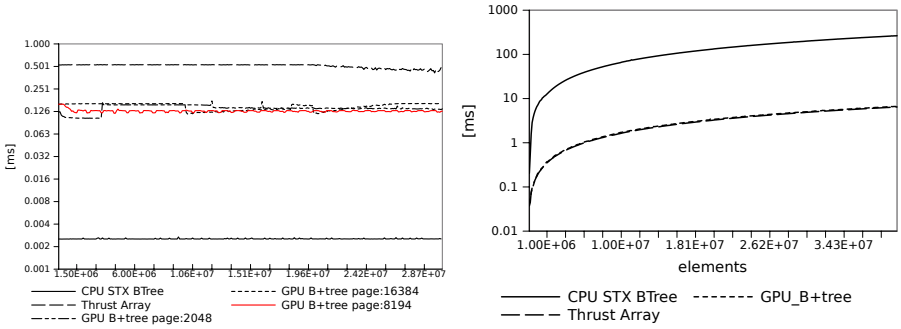

Fig. 4. (left) Comparison of searching times. Each point is an average of 100 random searches. A searched key was not guaranteed to be in the collection. The experiment run for collections from 100k to 30 millions of elements with page sizes: 2048, 8196 and 16384. **(right)** Results of an experiment visiting all values of a collection and performing a scalar operation. CPU B^+-tree cannot stand competition with GPU and becomes much slower very quickly. 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, page size 16384.

4 Conclusions

As it was predicted, B^+-tree structure may be efficiently ported to GPU opening another field of possibilities for databases supported by robust multi-core graphical devices. Our experimental implementation showed that we may achieve better searching time when comparing to a classical array based solutions. Also parallel access to all elements in the collection is not significantly slower than in case of a single flat array. CPU is outperformed here by two orders of magnitude.

Unfortunately, element insertion is highly dependent on searching, repeated descending page lookup and very slow internal page insertion. Therefore the overall tree is the fastest for two or at most three levels. Therefore page size must be big enough to guarantee that the data can fit into the collection. However, if we enlarge the page we achieve better parallelism inside. Running more GPU threads in the same kernel improves performance. On the other hand larger pages slow down searching. Definitely more research must be done to find a sensible trade off, optimization methods or new algorithms.

4.1 Future Works

Future works will begin with improvements in the implementation by better memory allocation and more general page construction. Another place for many optimizations is host and device cooperation. 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. Right now it is hard to predict if this CPU side calculation can really improve performance or not. Maybe a group of not usable threads (at most half of the page size but statistically one forth) is not a big deal when compared to additional CPU memory accesses and floating point operations. This needs further investigation.

Another open space is improvement in the tree itself. We could observe that for smaller collections optimal page size was about 2048 or 4096 elements and bigger pages are significantly slower in searching. An interesting option would be to create a tree which could automatically reconfigure making pages longer at some point when adding more elements to prevent splitting and keep the overall tree as small as possible.

The experiment showed that CPU B^+-tree implementation may be much faster than GPU in searching. That is mostly because of better caching when accessing memory and optimizations when reading single memory location at a time. GPU obviously wins in parallel computations done on all elements in the collection. 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.

We also predict that the latest NVIDIA Fermi could improve performance by utilization of new global memory caches.

References

1. NVIDIA Corporation, "CUDA Toolkit and SDK v.3.2," Jan. 2011. `developer.nvidia.com/object/cuda_3_2_downloads.html`.
2. D. Cederman and P. Tsigas, "Gpu-quicksort: A practical quicksort algorithm for graphics processors.," *ACM Journal of Experimental Algorithmics*, vol. 14, 2009. GPU Quicksort Library Download: `http://www.cse.chalmers.se/research/group/dcs/gpuqsortdcs.html`.
3. M. Harris, J. D. Owens, and S. Sengupta, "CUDA Data Parallel Primitives Library," 2008. `code.google.com/p/cudpp`.
4. J. Hoberock and N. Bell, "Thrust CUDA Library v.1.3.0," 2011. `code.google.com/p/thrust`.
5. R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indices.," *Acta Inf.*, vol. 1, pp. 173–189, 1972.
6. D. Comer, "The ubiquitous b-tree.," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979.
7. D. E. Knuth, *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
8. T. Bingmann, "STX B+ Tree C++ Template Classes v 0.8.3," 2008. `idlebox.net/2007/stx-btree`.