Parallelizing Approximate Search on Adaptive Radix Trees

Tobias Groth[1], Sven Groppe[1], Martin Koppehel[2], and Thilo Pionteck[2]

[1] University of Lübeck, Ratzeburger Allee 160, 23562 Lübeck, Germany
{groth, groppe}@ifis.uni-luebeck.de
[2] Otto von Guericke University Magdeburg, Universitätsplatz 2,
39106 Magdeburg, Germany
{martin.koppehel, thilo.pionteck}@ovgu.de

Abstract. Efficient searching in a number of strings is a common task in computer science and radix-trees are often used as a compact storage for string saving. One variant is the Adaptive-Radix-Tree (ART). ART has adaptive node sizes for more compact and cache-friendly memory layout. Graphical Processing Units (GPUs) can be used as hardware accelerator for massively parallel tasks and in addition they use very fast memory. We propose a parallel approximate search in the ART on CPU and GPU to optimize the throughput of queries and speed up applications that depends on these algorithms. Thereby we use the edit distance to compare two search keys in the tree and select appropriate values. We use the CPU for experimental comparison with the GPU, which have several thousand cores and modern processors typically have four to several dozens cores, but theses cores and RAM are more flexible. We propose several variations of the CPU algorithm like fixed vs. dynamic memory layouts and pointer vs. pointer-less data structures. In our experimental evaluation with OpenCL on ROCm 3.0, AMDs platform for GPU-Enabled HPC and Ultrascale Computing, the speedup and throughput of the GPU implementation for the approximate search in comparison with the best CPU variant are in the maximum up to factor 4.16 depending on the size of the tree and batch size. The speedup between the best and the worst CPU algorithm is up to factor 11.67, depending on tree and batch size.

Keywords: Adaptive-Radix-Tree(ART) · CPU acceleration · GPU acceleration · OpenCL · edit-distance · parallel · approximate search

1 Introduction

Modern multicore processors have many cores and further techniques of parallelism. Furthermore GPUs have a higher computing power and are much more

optimized for parallel processing. They have up to several thousand cores, but these cores are different from the CPU ones. They are more specialized to perform a single operation on multiple data [1] (SIMD). Also a graphics card contains multiple gigabytes of very fast high-bandwidth memory, but data transfers and fixed memory hierarchies can have negative implications on the performance if not considered during algorithm design. In computer science, key-value pairs are often used, e.g. indices in database or as tags in geographic information retrieval systems [17]. These algorithms benefit from an efficient and fast access to the values by a certain key. The keys are often strings in most times. An efficient data structure for string keys is the PATRICIA-tree [15] and the more general radix tree, which is a memory-efficient storage for the keys and widely used for indexes. The memory-efficient storage is the big advantage of radix trees in comparison to other structures like the skiplist, where keys are stored independently from each other. One variant of the radix tree is the Adaptive-Radix-Tree [12] (ART), which has adaptive node sizes and a lower memory consumption than the other variants because the shared prefix of multiple keys is only stored once.

Besides searching for an exact matched key, an approximate search is interesting as well. Approximate searches are widely used to retrieve information from databases and dictionaries. Also an approximate search can be used to find an optimal sequence alignment and matching parts of DNA sequences [10]. For an approximate search in ART, we need a metric to distinguish string keys. There are several different approaches like for example the Jaccard-Coefficent [6], the Cosine-Similarity [16] and the Edit-Distance [13]. The Jaccard-Coefficent uses set operations on sets of n-grams in comparisons. This n-grams can be for example, words or characters. The Cosine-Similarity is based on the frequency of words in a text. The edit distance represents the number of insert, replace and delete operations, which are required to transform a string into another [13] and is hence very intuitive for humans. Because the edit distance is a widely used metrics for string similarity, we propose approximate search algorithms to determine edit distances between a search term and a given set of keys and we use ART in order to avoid multiple computations for shared prefixes. These algorithms are using CPU and GPU acceleration and are optimized for massively parallel execution.

Our contributions are

- highly efficient approximate search algorithms for ART on CPU and GPU,
- variants of these algorithms on the CPU, including pointer versus pointer-less data structure and using fixed versus dynamic memory allocation,
- the most efficient GPU variant with pointer-less data structure and using fixed memory allocation, and
- an extensive experimental evaluation and analysis of these algorithms.

First we look at related work in Section 2. Next the ART is explained in Section 3.1 and then the concept of the searches is described in Section 4. Finally we present a comprehensive experimental evaluation in Section 5 and conclude in Section 6.

2 Related Work

In many papers approximate searches, GPU acceleration and trees are a research topic, but the focus is different. Papers like [10] describe the basics of approximate algorithms and the calculation of the edit distance. A problem for most tree structures on GPUs is that pointers aren't optimal for GPUs. Approaches to handle pointer data structures exist in [11] and [14]. The contribution in [14] describes a tree structure for fast copying of different types of trees. A common and widely used tree is the B^+. The authors of [7] have focused on this tree and optimize their algorithms for GPGPU. It is used wherever the order of elements and parallel computing are important. Another tree structure is FAST [9], which is a binary and architecture sensitive tree. It is optimized for low latency and uses thread and data-level parallelism and explore them on CPU and GPU. Therefore the sizes for page, cache line size and SIMD width are variable. GPU LSM [4] is a data structure for dynamic dictionaries on GPU. It has support for fast insertion and deletion based on the LSM-tree. For retrieval lookup, count and range queries are supported. For the Adaptive-Radix-Tree the GPU-based radix tree (GRT) [1] variant has been developed, which is written in CUDA and supports exact and range based search. A mapping is used by the search algorithm to transfer the tree structure to the GPU memory. The authors of [3] compare the Adaptive-Radix-Tree with Judy, two variants of hashing via quadratic probing and three variants of Cuckoo hashing in an extensive experimental evaluation. The results are that hash tables perform better in OLAP and OLTP scenarios, but for range queries ART is significantly faster. In comparisons with a B^+-tree ART is slower in performing range queries. ART is two times faster than Judy, which is also an adaptive radix tree variant, but ART needs the double space of memory. Besides trees hash tables are an important data structure. Therefore [2] introduces an algorithm for building large hash tables in real time. For this purpose, they used a data-parallel approach and build the tables on GPU. They also applied their hashing methods to graphic applications. In comparison to the discussed works we consider an GPU accelerated approximate search in ART as our main contribution, which hasn't been considered for this data structure in the existing literature to the best of our knowledge. This includes the development of different parallel variants for modern CPU and GPU features.

3 Basics

3.1 The Adaptive-Radix-Tree

The Adaptive-Radix-Tree (ART) [12] is a tree designed for in-memory-databases. ART has four different node types with three different memory representations for an efficient memory layout for different node sizes. Nodes are created at the deepest possible level of the tree; this is called lazy expansions. Further path compression is used to reduce the size of the ART and to speed up the search. We present an example ART in fig. 3. The tree contains 10 keys and two node types Node4 and Node16 are used. These types contain a maximum of four and

16 children. All inner nodes except one have a shared prefix of length one. The leafs contain numbers as values.

Nodes are only created if they are required and superfluous nodes with only one child are removed. If the maximum number of children is exceeded, a bigger node is used. If the number is lower than the minimum, a smaller node is chosen. Path compression uses a hybrid approach. In every node there is space to store a fixed number of characters. If this space is exceeded, a modified approach is applied. Instead saving subsequent characters, the length of the string is saved. This length is used by the search algorithm to jump over characters in the search term and to select the next character for the branch decision. This approach leads to wrong results in some cases, because a wrong search term is marked as correct even if different characters are in the missing part of the string. This problem can be avoided if the complete key is also saved in the leaf. Hence if the search ends in a leaf node the whole key has to be compared. [12]

3.2 Edit-Distance

The edit distance can be calculated with a simple algorithm. For two keys u,v with the length m and n, a matrix M can be calculated where the initial case is $M_{0,0} = 0, M_{i,0} = i, 1 \leq i \leq m, M_{0,j} = j, 1 \leq j \leq n$. Then the further cells can be determined by applying the rules given in fig. 1a.

$$
\begin{aligned}
&\text{If } \; u_i = v_j \\
&\text{then} \\
&\quad M_{i,j} = M_{i-1,j-1} \\
&\text{else} \\
&\quad M_{i,j} = min \begin{cases} M_{i-1,j-1} + 1 \\ M_{i,j-1} + 1 \\ M_{i-1,j} + 1 \end{cases} \\
&\text{endif} \\
&1 \leq i \leq m, 1 \leq j \leq n
\end{aligned}
$$

(a) Rules for computing the edit distance

		i	n	s	t	i	t	u	t	e
	0	1	2	3	4	5	6	7	8	9
i	1	0	1	2	3	4	5	6	7	8
n	2	1	0	1	2	3	4	5	6	7
c	3	2	1	1	2	3	4	5	6	7
o	4	3	2	2	2	3	4	5	6	7
m	5	4	3	3	3	3	4	5	6	7
i	6	5	4	4	4	3	4	5	6	7
n	7	6	5	5	5	4	4	5	6	7
g	8	7	6	6	6	5	5	5	6	7

t != i
min(5+1,
6+1,
4+1)
= 5

e.g.
5 operations are needed to transform "institu" into "incom" or vice versa

(b) Edit distance example and example computation of one cell

Fig. 1. Rules and an example matrix for the edit distance

Without backtracking, the algorithm requires only the last column to compute the next. This will be used to reduce the memory consumption. We can reduce the size further and can use only one column and a variable for computation, but this takes away the possibility to revert the last computation if a branch occurs.

4 Approximate Search Algorithm

In this section we develop an approximate search for CPUs and GPUs. Furthermore, we optimize the approximate search for better throughput and to handle bigger trees.

4.1 Memory Organization

If ART is used on a GPU, a tree that is stored in a continuous memory is required because copying a big chunk of memory is faster than many small pieces of memory. Nodes are linked by offsets in order to avoid pointers, which would be invalidated by the transfer on the GPU. An example of the memory mapping is shown in fig. 2. In this example we traverse the tree with a depth-first-search algorithm. We reserve memory on the way down and insert the data on the way up. This is necessary because the offsets of all children have to be known before saving a node. The complete memory object will grow dynamically because the size of the tree is unknown. However, the memory object is continuous, so it has a fixed size and needs to be extended in case of an overflow. Therefore the overflow has to be detected and a new memory object has to be created. Then the algorithm copies all data to the new memory object, but the offset remains unchanged because it is relative to the beginning position of the memory. After the algorithm finishes, a continuous memory for the tree is created. The root node is placed at the first address of the continuous memory.

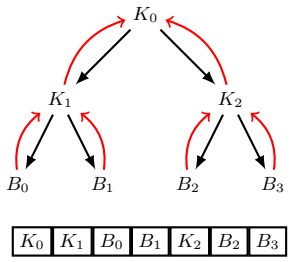


Fig. 2. Mapping of a tree in a continuous memory, red edges are back jumps. We start at K_0 and reserve memory for the node. Then we visit K_1 and reserve memory. Then we reserve memory for B_1 and we reach the end of a path. Now we store B_0 in the reserved memory and jump back. The same happens to B_1. Then all children of K_1 are stored and we store the node in the reserved memory and go to K_0. Because K_0 has another child K_2, we are processing the subtree in K_2 in an analogous way.

4.2 Approximate Search on ART utilizing CPUs

The CPU has flexible cores combined with dynamic memory and can calculate many threads in parallel. To address these advantages, we develop an algorithm

(see fig. 3a), which uses these techniques to calculate the edit distance and to choose a key that is below or equal this distance. First we can use threads to run multiple searches in parallel. Furthermore, we have to find out if we can also use parallelism inside the tree. Because for calculation of the edit distance only the previous and the current columns are necessary, with dynamic memory, we can simply copy the column for each branch. Then we start a new thread with this column and the current position in the tree. We use tasks with an automatic management of threads to avoid problems with the overhead of thread start, and stop operation and with an overload if there are too many threads. If we reach a node, where all edit distance values are above the maximum edit distance, we prune the whole subtree, and use backtracking in order to visit not already visited branches of previous nodes. Note that previous columns are already determined. We only have to reset the position and remove all columns, except of the previous ones. If we reach a leaf we calculate the edit distance for the last characters and look at the last element in the column. If it is lower than or equal to the maximum edit distance, we have found a proper key and we have to save the value in a list. Because the algorithm runs in parallel, we have to lock the list and then save the value. Because we search in parallel, we need to lock the list with a write-lock. This is sufficient, because we require unique search terms in one batch and hence it does not occur that they reach the same leaf. Alternatives are that we can use multiple lists and merge them at the end. Please note that we can't allocate fixed sized memory for lock free access, because the result size in each search is different and unknown.

4.3 Approximate Search on ART utilizing GPUs

For the GPU algorithm, please see fig. 3b for an example. Because the GPU has a different architecture, our algorithm is more complex than the one on the CPU. We use parallelism for multiple search requests and for root node processing. Because we don't have dynamic memory we can't create or remove columns. Therefore, we use a static memory buffer with two columns with the maximum length of a search term and calculate the edit distance alternating the roles of two columns. First all work-items calculate the edit distance of the root node in parallel and save this value in a sequential reset buffer. Then all work-items process the deeper nodes sequentially. To save the progress and to have a proper ending condition, the maximum path length in the tree is necessary. With this length, we create a buffer, where we save the number of processed children in a node for the current path. If we reach a leaf or the maximum edit distance, we increment the counter for the processed children in the level above. Because we don't have a history of the edit distance, we have to jump to the sequential root and calculate all edit distances again. Then we reach the position again and we select the next child. To reduce this overhead for long paths we additionally reserve memory for a reset buffer in the half length of the path.

A major problem is the result buffer, because the number of results is unknown, but the buffer size has to be known before the executions starts. The result of a search can be zero to number of leafs in the tree. We limit the amount

of memory and if we detect an overflow, then we abort. This leads to unfinished executions and missing results. Hence we modify the search algorithm to filter missing queries and execute them sequentially with enough memory for all results. Furthermore, we use a three level result buffer shown in fig. 4 to optimize the fill level. The size increases with the shared level. Level 1 only has space for a few results and the third level has space for millions of results. Because sharing memory between work-items require concurrent access, lock and unlock functionality is indispensable. Therefore atomics are used to increment a counter and get the old value. This value is used to exclusively access the buffer cell. For a better bandwidth and GPU usage, multiple threads can be started on the CPU and every thread gets a chunk of search queries and executes them on the GPU in parallel. Therefore every thread has separate buffers except of the tree buffer, which is always shared read-only.

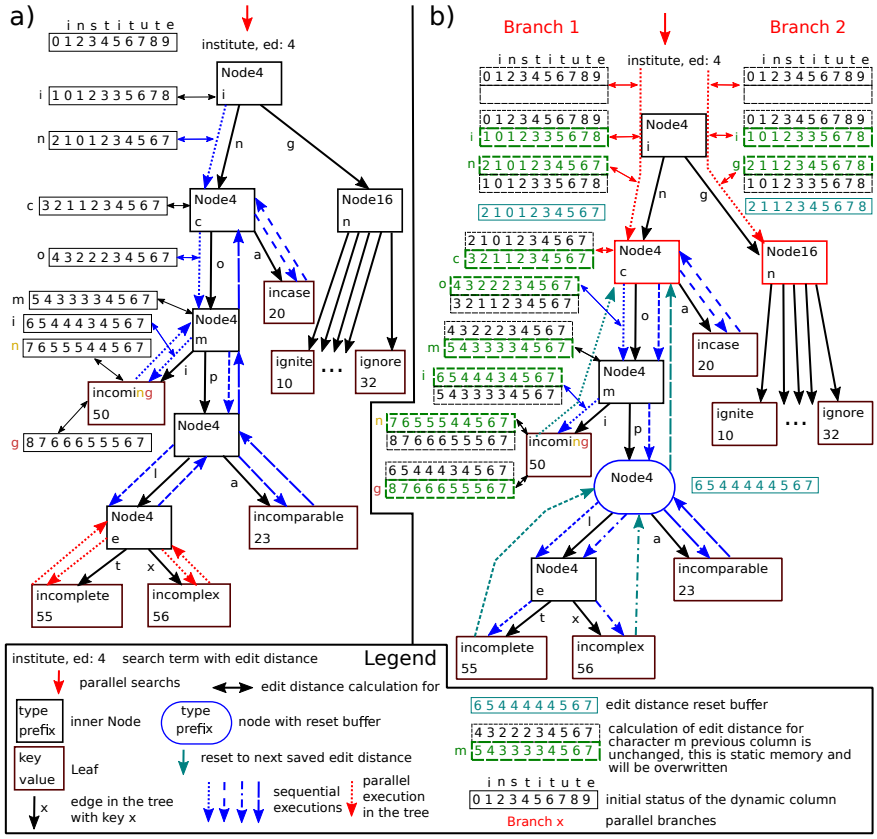


Fig. 3. Comparison of tree traversal with edit distance in a) CPU and b) GPU algorithms. We search for all keys that have a maximum edit distance of two to the search term "institute". To visualize the calculation of the edit distance, the columns for the path to the node with the key "incoming" is provided

5 Experimental Evaluation

We evaluate the proposed algorithms for CPU and GPU in this section.

5.1 Experimental Environment

The experimental evaluation runs on a Ryzen 1700X with 8 cores and 16 threads and a RX Vega 56 with 8GB HBM2-memory and 56 compute-units with 1024 processing elements each. Therefore 57344 work-items can be executed in parallel. The local memory on the GPU is 64KB large. The evaluation system is equipped with 16GB RAM and an SSD. Ubuntu 18.04 is used as software platform and we compile the code with the GCC compiler in version 8. OpenCL acceleration is provided via ROCm 3.0.

5.2 Benchmark Data

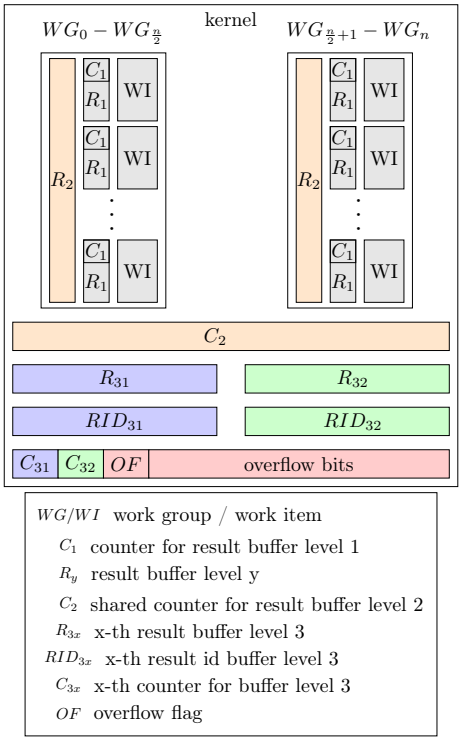


Fig. 4. A kernel work-item can store the data of unknown length using different memory levels. First it uses the level 1 buffer directly and exclusively. If this memory is full, level 2 memory is used and shared with all work-items in a work-group. If this memory is depleted, level 3 buffer is used. There are two buffers, the first half of the work groups share the first half and the second half share the second. If all buffers are depleted, the work-item tags the search with an overflow bit and the search is sequentially executed afterwards

For our experiments, a compilation of synthetic and real test data is used. The load times of a search depends on various parameters, whereby the most important ones are the number of search terms, the number of branches in a node, the depth of the tree and the length of the shared prefix. Each of these properties is examined in this evaluation. For a single experiment the other properties are fixed and then the examined property is changed. We repeat a search mostly 10 times except for single runs that take longer than one hour. These searches are only repeated three times. If a search takes longer than four hours, we used

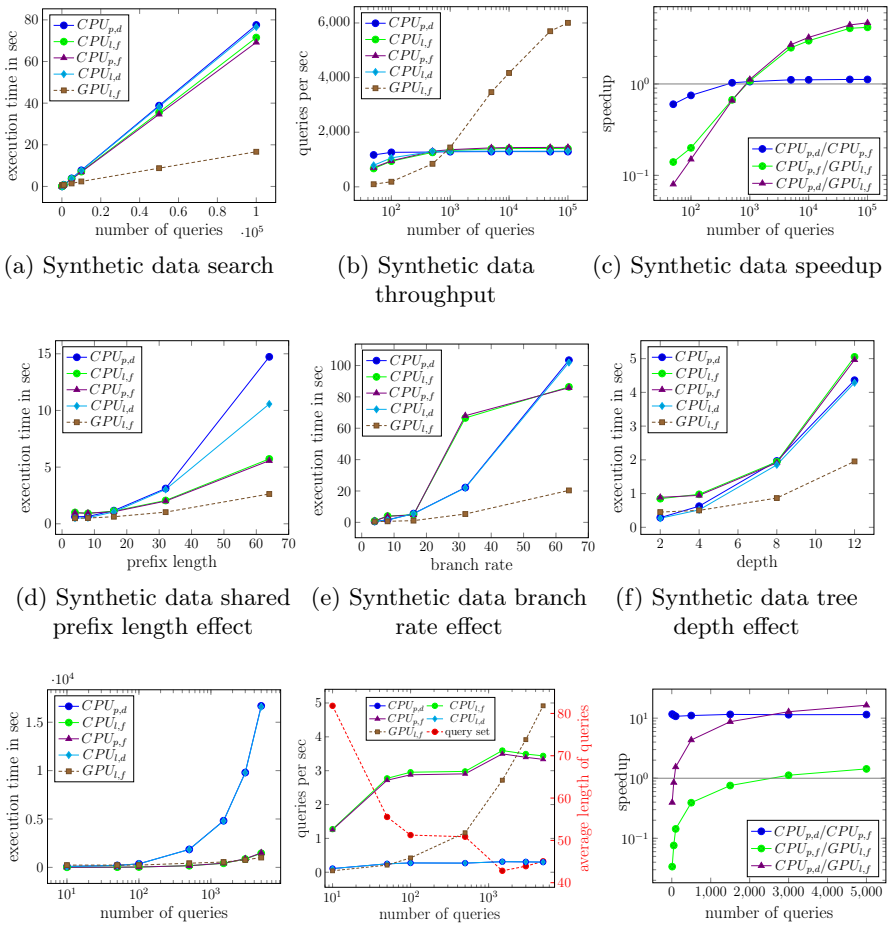
https://www.amd.com/de/products/cpu/amd-ryzen-7-1700x
https://www.amd.com/de/products/graphics/radeon-rx-vega-56
https://www.ubuntu.com/desktop
https://gcc.gnu.org
https://www.khronos.org/opencl
https://rocm.github.io

two executions. The duration is measured and the average is calculated and displayed in graphs. If the number of search requests isn't displayed in the graph of the approximate search, 10,000 requests are made per iteration. The search terms are created by randomly choosing a key from the dataset, which is modified by a specific number of characters. Additionally to the described synthetic data and queries, and for more realistic data we use a data set from the billion triples challenge (BTC) [5]. This real-world data set contains 19,655,239 triples with 4,352,096 unique keys and the longest key has a length of 32,628. The resulting tree has a maximal path length of 34. Further the tree and the algorithms have several important parameters. The maximum edit distance describes, which max distance between a key in the result and the search term is acceptable. For the algorithm the maximum path length (MPL) and the maximum search term length (MSL) are important. The MPL describes the longest path from the root to a leaf. It is important for the stack, which describes the next child and the exit condition. The MSL describes the maximum length of a batch of search terms and is required for the length of the column. The important parameters for the performance of the CPU algorithm are the number of parallel search terms and tasks in tree. They describe how many parallel searches are made and how many parallelism is inside a tree. For the GPU there are the number of GPU threads and the kernel size. The first parameter describes how many parallel command queues will be used and the second one, how many search terms are processed in a kernel execution and will be computed in parallel.

5.3 Approximate Search on ART

For better distinction, we introduce a naming schema for the developed algorithms. Dynamic memory allocation is represented by d, fixed memory by f, pointer based ART structure by p and pointer-less offset structure by l. Hence we use the following naming schema e.g. $CPU_{p,d}$ represents the developed CPU algorithm and $GPU_{l,f}$ the GPU algorithm. Further variants are $CPU_{l,d}$, $CPU_{p,f}$ and $CPU_{l,f}$. To get proper evaluation results, we have to fine-tune the settings for all algorithms. $CPU_{p,d}$ and $CPU_{l,d}$ are much faster if we process as many queries as possible in parallel: The best configuration uses 250 parallel queries and spare on parallel computations inside the tree. Because the root node is parallelized in $GPU_{l,f}$, $CPU_{p,f}$ and $CPU_{l,f}$, the number of parallel queries depends on the root node type. For performance the optimal number of parallel queries is up to 5000 for $GPU_{l,f}$ and up to 30 for $CPU_{p,f}$ and $CPU_{l,f}$. More tasks would produce more overhead on the CPU and on the GPU the number of executions is limited by the GPU memory.

The original ART has to be modified for the approximate search, because the calculation of the maximum edit distance is only possible if all characters of the shared prefix are saved. The original ART only saves a prefix with fixed length. By replacing the fixed length prefix for correct results of the algorithm with a prefix with variable size we have to deal with a higher memory consumption. The variable size prefix is implemented by a separate buffer, whereby an offset is used to access single elements in the buffer.



(a) Synthetic data search (b) Synthetic data throughput (c) Synthetic data speedup

(d) Synthetic data shared prefix length effect (e) Synthetic data branch rate effect (f) Synthetic data tree depth effect

(g) BTC data search (h) BTC throughput (i) BTC speedup

Fig. 5. Evaluation results of the approximate search implementations

In the experiments (see fig. 5(a-i)), the $CPU_{p,d}$ is the slowest algorithm and performs worse in comparison to the variants as more queries are processed. This is independent from the dataset and its properties. We achieve analogous results for the $CPU_{l,d}$, but the algorithm has advantages if the length of the common prefix increases. $CPU_{p,f}$ and $CPU_{l,f}$ have a better performance in all situations except of a higher branch rate. In this case, the limited parallelism is a problem and the $CPU_{p,d}$ and $CPU_{l,d}$ are initially faster, but this is changed if the number of searches is further increased. The $GPU_{l,f}$ is slower at a small number of queries because the transfer and execution overhead are higher, but if the number of queries is increased, the algorithm is faster than the other algorithms. The speedup is up to 4.16 on synthetic data and up to 1.43 on real-world BTC-

data. Because the creation of queries depends on random, the average length of the used queries varies. As shown in fig. 5h (red line and axis on the right), the throughput depends on the length of search terms. If the length is increased, the edit distance calculation increases in the same way and the throughput is decreased. This effect is more significant if the parallelism is very limited. If the parallelism is higher, we can compute more long terms in parallel in the same time. Because of this effect, $CPU_{p,f}$ and $CPU_{l,f}$ are more affected than the other algorithms and have a non-linear trend.

6 Summary and Conclusions

We introduce approximate search algorithms, which run on CPU and GPU. Thereby the memory situation is observed: The CPU algorithm uses dynamic memory allocation and the GPU memory layout is adaptive but static during a run. The approximate algorithm that runs on the CPU uses tasks as an implementation of parallelism. Tasks provide a solution by splitting the workload to multiple threads. In contrast, the GPU uses a hierarchical result memory concept and processes the data in chunks. In our experiments, the approximate search on the GPU has a higher throughput and speedup compared to the CPU implementation. Inspired by the approximate search designed for the GPU, we develop and evaluate CPU variants using pointer versus pointer-less data structures and using fixed versus dynamic memory allocation. The speedup between the best CPU and the GPU variant is about 1.43, while the speedup between the worst CPU variant and the GPU is about 16.41. The current GPU implementation can only handle trees that fit into the GPU memory. It is promising for future work to extend the algorithms and enable larger trees to be processed by investigating compression for ART. Furthermore with OpenCL 2 Fine-Grained System SVM [8] the original pointer structure could be kept and the CPU and GPU can share the tree and data directly. Our future work covers also hybrid approaches with APUs, which are combinations of CPU and GPU on the same chip with shared memory.

Acknowledgements

This work is funded by the German Research Foundation (DFG) project GR 3435/15-1.

Funded by DFG Deutsche Forschungsgemeinschaft German Research Foundation

References

1. Alam, M., Yoginath, S.B., Perumalla, K.S.: Performance of point and range queries for in-memory databases using radix trees on gpus. In: 18th IEEE International Conference on High Performance Computing and Communications; 14th IEEE International Conference on Smart City; 2nd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS, Sydney, Australia, December 12-14. pp. 1493–1500 (2016)

2. Alcantara, D.A., Sharf, A., Abbasinejad, F., Sengupta, S., Mitzenmacher, M., Owens, J.D., Amenta, N.: Real-time parallel hashing on the GPU. ACM Trans. Graph. **28**(5), 154 (2009)
3. Alvarez, V., Richter, S., Chen, X., Dittrich, J.: A comparison of adaptive radix trees and hash tables. In: 31st IEEE International Conference on Data Engineering, ICDE, Seoul, South Korea, April 13-17. pp. 1227–1238 (2015)
4. Ashkiani, S., Li, S., Farach-Colton, M., Amenta, N., Owens, J.D.: GPU LSM: A dynamic dictionary data structure for the GPU. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS, Vancouver, BC, Canada, May 21-25. pp. 430–440 (2018)
5. Harth, A.: Billion Triples Challenge data set. Downloaded from http://km.aifb.kit.edu/projects/btc-2012/ (2012)
6. Jaccard, P.: Étude comparative de la distribution florale dans une portion des alpes et des jura. Bulletin del la Société Vaudoise des Sciences Naturelles **37**, 547–579 (1901)
7. Kaczmarski, K.: B $^{+}$ -tree optimized for GPGPU. In: On the Move to Meaningful Internet Systems: OTM, Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE, Rome, Italy, September 10-14. Proceedings, Part II. pp. 843–854 (2012)
8. Kaeli, D.R., Mistry, P., Schaa, D., Zhang, D.P.: Heterogeneous Computing with OpenCL 2.0. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2015)
9. 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: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD, Indianapolis, Indiana, USA, June 6-10. pp. 339–350 (2010)
10. Krugel, J.A.: Approximate Pattern Matching with Index Structures. Phd thesis, Technical University of Munich, Munich, Germany (2016)
11. Lefohn, A.E., Sengupta, S., Kniss, J., Strzodka, R., Owens, J.D.: Glift: Generic, efficient, random-access gpu data structures. ACM Trans. Graph. **25**(1), 60–99 (Jan 2006)
12. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: Artful indexing for main-memory databases. In: 29th IEEE International Conference on Data Engineering, ICDE, Brisbane, Australia, April 8-12. pp. 38–49 (2013)
13. Levenshtein, V.I.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady **10**, 707 (Feb 1966)
14. Lu, Y., Yang, L., Bhavsar, V.C., Kumar, N.: Tree structured data processing on GPUs. In: 7th International Conference on Cloud Computing, Data Science Engineering - Confluence, Noida, India. pp. 498–505 (2017)
15. Morrison, D.R.: PATRICIA - practical algorithm to retrieve information coded in alphanumeric. J. ACM **15**(4), 514–534 (1968)
16. Salton, G., Buckley, C.: Term-weighting approaches in automatic text retrieval. Information Processing & Management **24**(5), 513 – 523 (1988)
17. Yousaf, M., Wolter, D.: How to identify appropriate key-value pairs for querying osm. In: Proceedings of the 13th Workshop on Geographic Information Retrieval. GIR '19, Association for Computing Machinery, New York, NY, USA (2019)