Are van Emde Boas trees viable on the GPU?

Benedikt Mayr, Alexander Weinrauch, Mathias Parger, and Markus Steinberger
Graz University of Technology, Austria

***Abstract*— Van Emde Boas trees show an asymptotic query complexity surpassing the performance of traditional data structure for performing search queries in large data sets. However, their implementation and large constant overheads prohibit their widespread use. In this work, we ask, whether van Emde Boas trees are viable on the GPU. We presents a novel algorithm to construct a van Emde Boas tree utilizing the parallel compute power and memory bandwidth of modern GPUs. By analyzing the structure of a sorted data set, our method is able to build a van Emde Boas tree efficiently in parallel with little thread synchronization. We compare querying data using a van Emde Boas tree and binary search on the GPU and show that for very large data sets, the van Emde Boas tree outperforms a binary search by up to 1.2x while similarly increasing space requirements. Overall, we confirm that van Emde Boas trees are viable for certain scenarios on the GPU.**

***Index Terms*— van Emde Boas tree, vEB tree, GPU, parallel construction, CUDA, GPU querying**

I. Introduction

Searching data sets efficiently has always been of interest for data processing as it is one of the key operations in many fields. Binary search [1], [2] is probably the best known search algorithm. It is also known as interval search, as at each iteration the search interval is halved until the desired element is found resulting in a search time bound of $O(\log_2(n))$. Over the years many different search algorithms have been proposed to improve upon this search bound.

One of those proposed structures is the van Emde Boas (vEB) tree [3]–[5] structure which requires one constraint on the data set. The full data set must exist within the range $U = [0, 2^{2^d} - 1]$. This allows for an interval search on the bit-representation of a value by splitting it into an upper and lower half at each iteration step. Therefore, the vEB tree performs searches in $O(\log_2(\log_2(|U|)))$ time. When working with a 32 bit environment, a vEB tree needs at most 5 iterations for querying a single value from the structure.

A big drawback of vEB trees is that the original construction algorithm foresees that each element is iteratively inserted into an existing tree, starting from an empty tree. The insertion process expands the space needed for the tree dynamically as new elements are inserted. The dynamic memory allocations and serial nature of the construction make it not a good fit for a highly parallel computing device like a modern GPU.

Instead, we propose a highly parallel construction algorithm that works on a sorted data set, which allows for parallel construction without the use of dynamic allocation and avoids synchronization primitives in a parallel compute environment. Our main focus lies on evaluating the query performance of vEB tress on the GPU. While the theoretical bounds clearly point towards the efficiency of vEB trees, they are traditionally associated with high constant factors and thus see less use in practice. With our work, we try to answer the question, whther vEB trees are viable on the GPU for general data query. To this end, we compare the efficiency of our vEB tree implementation with binary search across different data distributions.

II. Related Work

When constructing a vEB tree using the traditional insertion algorithm, the data inserted is implicitly sorted based on the divide and conquer method. Similar sorting methods like quicksort [6] and mergesort [7] employ divide and conquer, but are bound with $O(n \cdot \log_2(n))$ in their sorting time as they operate on the length of the data set.

A method with similar query time characteristics is the Tango Tree [8]. It is an online algorithm which achieves a competitive ratio of $O(log_2(\log_2(n)))$ for queries. The Tango Tree operates in a similar way to the Splay Tree [9], in that it has a preferred path, which speeds up queries to elements that have been queried prior. Additionally, red-black trees [10] represent the preferred paths as auxiliary trees, only resulting in an overhead of $O(log_2(\log_2(n)))$ bits.

While the GPU is traditionally not well suited for hierarchical data, building spatial data structures, especially motivated by hardware ray-tracing showed that building trees on top of sorted (multi-dimensional) arrays or binned data directly on the GPU is a viable option [11]–[18]. When building and evaluating hierarchical data structures on the GPU, task-based models may even allow for dynamic adjustments [19]–[23]. Further domains with specialized tree structures on the GPU include classification trees [24], decision trees [25], and min-max trees [26]. Querying for k-nearest neighbors can also be done efficiently on top of hierarchical structures on the GPU [27], [28]. Recently, there has been a move towards more efficient general data structure implementations on the GPU, including hash tables [29], dynamic dictionaries [30], R-trees [31] and B-trees [32]. In this work, we for the first time look at vEB tress on the GPU.

A use case for vEB trees are range queries. In [33] vEB trees are used in multiple areas to find the locations of points for several dimensions. Furthermore, they are popular in areas where fast query times are beneficial, like network routing [34], [35]. For smaller universe sizes U the query times have been optimized to a theoretical $O(1)$ using pipelining [36].

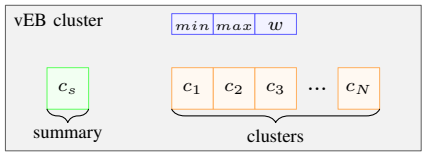

Fig. 1. Basic structure of a vEB cluster c. Each cluster stores the minimum min and maximum max value as well as an array of all clusters smaller clusters. c has a fixed word length w. All contained clusters within c have word length $\frac{w}{2}$.

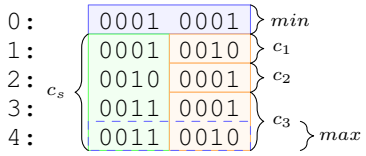

Fig. 2. Basic structure of a vEB cluster c with word length $w = 8$ shown on data. The summary cluster c_s is marked in green. The smaller cluster c_1 to c_3 that are contained in c are marked in orange. Blue marks the first element min and last element max.

III. Method

We start by discussing the basic structure of the vEB tree, the query functions, and how to insert new elements. We then analyze the time and space complexities of vEB trees, before presenting our novel GPU algorithm for vEB trees.

A. van Emde Boas tree

The vEB tree uses divide and conquer as a method to allow searches within a given data set. The vEB tree splits values on a bit level into an upper and a lower half and uses these halves to search on the word length recursively.

1) Structure: The vEB tree nodes are called clusters. A cluster operates on a specific word length, in other words bit-count, w. The word length for the top-level cluster is always $\log_2(U)$, which is the bit-count needed to represent the maximum value in the universe U. Each level in the tree halves the word length, which allows for a depth up to $\log_2(\log_2(U))$.

Each cluster c contains its minimum and maximum value, its word length w, as well as a list of children clusters $c_1 ... c_N$, and a summary cluster c_s (see Fig. 1). The children clusters form the nodes in the next level of the tree. In Fig. 2, the structure of a cluster is visualized. Values are placed into clusters $c_1 ... c_N$ based on the upper half bits. This gives an upper bound on possible clusters for a given word length w of $N = 2^{\frac{w}{2}}$. The lower half of a value is recursively passed to the children cluster which halves the word length until a word length of one is reached. The summary cluster stores information about non empty clusters and is used to find the next non empty cluster for a given value during query operations. The process of grouping values into clusters based on the upper half is shown in Fig. 2.

2) Helper functions: To explain the process of querying elements in the tree, we define two functions $high$ and low, which for each value x and a given word length w_c compute the respective upper and lower half:

Input : Query value x
Output: Predecessor for query value x

```
1  predecessor (x);
2  if max ≠ NULL and x > max then
3  |  return max
4  end
5  if x ≤ min then
6  |  return NULL
7  end
8  if wordlength = 1 then
9  |  return 0
10 end
11 highx ← high(x);
12 lowx ← low(x);
13 c ← clusters[highx];
14 if c ≠ NULL and lowx > c.min then
15 |  lowx ← c.predecessor(lowx);
16 else
17 |  highx ← c_s.predecessor(highx);
18 |  if highx = NULL and x > min then
19 |  |  return min
20 |  end
21 |  lowx ← clusters[highx].max;
22 end
23 return concat(highx, lowx)
```

Algorithm 1: Recursive predecessor search within vEB trees.

$$high\,(x, w_c) = \left(x \;\&\; \left(ffffffff_{16} \ll \frac{w_c}{2}\right)\right) \gg \frac{w_c}{2} \quad (1)$$

$$low\,(x, w_c) = x \;\&\; \left(\left(1_{16} \ll \frac{w_c}{2}\right) - 1_{16}\right). \quad (2)$$

Note that $ffffffff_{16}$ sinmply defines a 32 bit value with all bits being set. For different word lengths it needs adjustment. The upper half is generally denoted as $high(x)$ and the lower half as $low(x)$ with regards to the current clusters word length which is usually omitted when using this notation.

3) Query: Queries for vEB trees are $exists(\cdot)$, $predecessor(\cdot)$, and $successor(\cdot)$. All of these query functions are variations of each other. Algorithm 1 describes the recursive predecessor search starting at the biggest cluster.

The check on line 2 ensures that should a maximum value exist and the query is bigger than it, the query has been finished. Line 5 catches the case that a value smaller than the existing minimum of a given cluster is queried. Line 8 serves as recursion end. It returns 0 as lower values would be captured with line 5. The query for the subsequent smaller cluster happens on line 13. Should the cluster c exist in the list of clusters and the $lowx$ part be bigger than that clusters minimum then the algorithm is bound to find the predecessor of the query within c. On the contrary, should c not exist, then the summary cluster c_s is tasked with finding the next smaller cluster and taking its max value. The computed $highx$ and $lowx$ values are concatenated on a bit level based on the word length of the current cluster and get returned to the caller.

Equivalently, the $successor(\cdot)$ function operates by checking against maxima instead of minima and vice versa. To implement $exists(\cdot)$ it is possible to use x ==

```
Input: Value to insert x
1  insert (x);
2  if min = NULL then
3  |  min = x;
4  |  max = x;
5  |  return
6  end
7  if x < min then
8  |  temp ← x;
9  |  x ← min;
10 |  min ← temp;
11 end
12 highx = high(x);
13 lowx = low(x);
14 if wordlength > 1 then
15 |  if clusters[highx] = NULL then
16 |  |  create new cluster at clusters[highx];
17 |  end
18 |  if clusters[highx].min = NULL then
19 |  |  if c_s == NULL then
20 |  |  |  create c_s;
21 |  |  end
22 |  |  c_s.insert(highx);
23 |  end
24 |  clusters[highx].insert(lowx);
25 end
26 if x > max then
27 |  max = x;
28 end
```

Algorithm 2: Insertion into an existing vEB tree.

$c.predecessor(x+1)$ or $x == c.successor(x-1)$ since the vEB tree operates on a discrete domain where the aforementioned conditions hold true.

4) Insert: The original construction of vEB trees happens one element at a time. The pseudo code for the insertion is shown in 2. Newly created clusters always have half of the word length of the creating cluster. Furthermore, new clusters have their min and max value set to $NULL$. The minimum value of each cluster is stored in the min field of each cluster object and does not get propagated further down.

By swapping out the old min value with the new, smaller value x on line 7, it is guaranteed that the previously stored values gets propagated correctly. Similar to 1, the recursion ends as soon as $wordlength = 1$ or a cluster is empty. Note: should line 18 be true then the insert call at line 24 is constant as the check from line 2 returns in constant time and therefore maintains linear recursion.

B. Complexity

1) Domain: The discrete domain for the vEB trees is defined as

$$U = \left[0, 2^{2^d} - 1\right] : d \in \mathbb{N} \tag{3}$$

as otherwise it would be impossible to search on the word length. 2^d is the maximum word length possible, therefore if $d = 5$ the maximum word length is equal to 32 bits. The value d can be seen as the maximum depth for a vEB tree.

2) Build time: The time complexity of building a vEB tree with n elements and domain U is defined as:

$$O\left(n \cdot \log_2\left(\log_2\left(U\right)\right)\right). \tag{4}$$

This is because each recursive call to $insert(\cdot)$ has the word length halved until it arrives at $wordlength = 1$. A vEB tree with $d = 5$ describes the 32 bit domain and needs at most 5 recursive calls to $insert(\cdot)$.

3) Query time: The time complexity of searching a vEB tree with n elements and domain U is defined as:

$$O\left(\log_2\left(\log_2\left(U\right)\right)\right). \tag{5}$$

Note that the query time is not dependent on the number of elements but rather on the size of the domain. This is because the query is happening on the word length.

4) Space: The space requirements for vEB trees using a naive approach is $O(U)$ with creating every possible cluster beforehand but not storing anything in them until an insert for a value x is issued. Algorithm 2 already circumvents this by only creating clusters when they are needed. Furthermore, should only one value be stored within a cluster the fields min and max are utilized to prevent further creation of unnecessary clusters. Using this approach the space complexity of vEB trees is defined as

$$O\left(n \cdot \log_2\left(\log_2\left(U\right)\right)\right), \tag{6}$$

because each value can at most generate $\log_2\left(\log_2\left(U\right)\right)$ clusters when it is inserted using Algorithm 2. Further improvements are made by storing the entries that have a word length of 4 in a small bit map field as it only requires 16 bits (`uint16`). This is the biggest bit field that is sensible as the next bigger word length would be 8 which would require a $2^8 = 256$ bit field which is not efficient for querying whereas querying a small bit field can be done using built-in processor intrinsics.

The space requirements can be further improved by not allocating the space required for all smaller clusters which can reside within a cluster. For example, a cluster with word length 32 would need to allocate an array of 2^{16} clusters regardless of actually needing that much space. The usage of hash tables provide the same functionality with $O(1)$ access time while cutting down on unnecessary space allocation.

C. CUDA Implementation

When building a vEB tree on the GPU it is important for efficiency to avoid locking and using atomic operations when possible as they come with a big time overhead. The original algorithm 2 dictates elements to be inserted one at a time which does not lend itself to parallelization. Thus, our algorithm operates on the whole input set in parallel, i.e, assumes all data is present. A change of the data requires a rebuild—a strategy common across tree builders for raytracing on the GPU. The space requirement and cluster hierarchy is computed as a first step to avoid costly dynamic memory allocation during the construction of the vEB tree.

1) Sorting: For a given data set $D \subseteq U$ of data we use radix sort [37], [38] provided by the cub library [39] to prepare the data. This step is crucial as the following steps require the assumption of sorted ascending data.

2) Computing first-elements: Computing the number of clusters, a value x_n from the set D is going to generate, is necessary to allocate the space for clusters in a single call. The first-element (the minimum) of each cluster is not propagated into any smaller cluster or summary cluster. Therefore, the first element is the indicator of the start of a cluster. Identifying the amount of first-elements each value x_n is going to generate, as well as the starting bit position s (from the left) and word length w is important to compute the space requirement and structure of the tree ahead of time.

The first-elements, which are are shown in Fig. 3, are identified by a two-step process. The first step is to apply $fe(\cdot)$ to the whole input set, where

$$fe(x_n, s, w) = \begin{cases} true, & \text{if } x_{n-1}[0:s) \neq x_n[0:s) \\ false, & \text{else.} \end{cases} \quad (7)$$

x_n is the value that is checked, s is the starting position of a cluster in bits from the left, w is the word length of the cluster. The notation $x_n[0:s)$ implies taking the bits from bit position 0 to s, read from the left. For example in Fig. 3: $x_3[0:8) = \texttt{1100 0001}$. Eq. (7) compares the bits to the left of the start of a potential cluster. New clusters can only start where the $high(x_n)$ is different from the previous value x_{n-1}(see Fig. 2). This only holds because we operate on a sorted set of data. The values for s and w should not be arbitrarily chosen. The word length w is restrained to all possible cluster word-lengths. The starting position s is constrained to be a multiple of the value w and to be below the maximum word length. While brute forcing this check would work for all combinations of w and s, it is advised to start with the largest word length possible and start halving it. This allows to eliminate combinations of w and s for a given x_n: if a first element has been found (marked in blue in Fig. 2), no other first-elements for x_n can reside within that section of bits. Therefore, all smaller combinations which overlap with a first-element region can be ignored and be assumed as false.

It is also important to note that for element x_0 the condition $fe(x_0, 0, 32)$ is set to $true$ as it is always the first element of the whole vEB tree.

3) Propagation of first-elements: Step two is to identify the summary clusters. Every cluster has a summary cluster, if it has two or more different values in it. In case there is only a single value in the cluster it is directly stored in the min field and no sub-clusters are created. The summary clusters c_s starts at the first value that is different from the first-element (minimum) of cluster c. In this case, two clusters are generated, the summary cluster and the first cluster c_1. Here we use the already identified first elements from the first step to propagate downwards.

For a cluster c at value x_n with starting bit position s and word length w, it compares the first-element of c with the values below until it finds a k where the condition no longer holds true. Upon finding k the new summary cluster's first-element is identified as well as the first cluster which resides within cluster c. Note that in Fig. 3 below the blue box for $fe(x_2, 0, 8)$ no summary cluster is generated, as all values below are equal to the first-element in the scope of that cluster. Additionally $fe(x_2, 0, 8)$ and $fe(x_2, 8, 8)$ are products of the propagation from $fe(x_1, 0, 16)$ as $x_1[0, 16) \neq x_2[0, 16)$.

```
0: 1100 0001 0100 0110 1110 0010 0010 1010
1: 1100 0001 0101 1011 0100 1011 1100 0111
2: 1100 0001 1001 1110 0100 0101 0011 1000
3: 1100 0001 1010 0010 0011 0100 0101 0101
4: 1100 0001 1010 1000 0010 0100 0001 0100
5: 1100 0001 1100 0000 1111 1100 0111 0110
6: 1100 0001 1101 0010 1001 0110 1100 1001
7: 1100 0001 1101 0111 0000 1011 1011 1010
8: 1100 0001 1101 1001 0000 1100 0111 0101
9: 1100 0001 1101 1100 1010 0100 0100 0110
```

Fig. 3. Each blue box marks the first-element and therefore the start of a new cluster.

```
word length: 32 16  8      4
              ^  ^  ^      ^
0: 1 00 0000 00000000
1: 0 11 0000 00000000
2: 0 01 1100 00000000
3: 0 01 0000 00110000
4: 0 01 0000 00000000
5: 0 01 0000 00010000
6: 0 01 0000 00010000
7: 0 01 0000 00000000
8: 0 01 0000 00000000
9: 0 01 0000 00000000
```

Fig. 4. Encoding of the first-elements as bits for Fig. 3. Each set bit represents a starting position s and a word length w

4) Encoding for first-elements: To encode the identified first-elements for each element x_n a small bit field is used. Fig. 4 shows the same example as presented in Fig. 3 but shows the bit field for each value. The bit field represents all possible combinations of w and s and needs to be adjusted based on the domain size. In the example given in Fig. 4, x_0 has a first-element at starting position $s = 0$ with word length $w = 32$, x_1 has two first-elements, one $s = 0$ and $w = 16$ and one with $s = 16$ and $w = 16$, etc. The starting positions from here on out can be inferred by the position of the bit set in this bit field by multiplying the position with the word length.

After the two steps we have the finished structure as shown in Fig. 3 and its encoding in Fig. 4.

An additional benefit of this encoding is that the number of clusters for each value x_n is determined by the number of set bits in each bit field. To retrieve the count we can use the `popcount` intrinsic, which is present on all modern architectures. From this we can see that x_0 creates 1 cluster, x_1 creates 2 clusters, x_2 creates 3 clusters, and so on.

5) Cluster sizes: Using the first-elements bit field, we know how many clusters each value x_n is generating but it is unknown how many smaller clusters each generated cluster c contains. Using a search that finds the smallest k that fulfills

$$x_n[0:s) \neq x_{n+k}[0:s) \quad (8)$$

yields the number of smaller clusters $c_1, ..., c_N$ that are contained as well as max for c. Each cluster c with word length w can store up to $2^{\frac{w}{2}}$ smaller clusters. Therefore, the number of clusters N has an upper bound defined by

$$N = \min(k, 2^{\frac{w}{2}}). \tag{9}$$

6) Hash tables: Always storing the maximum possible amount of smaller clusters in an array unnecessarily increases the spatial requirements. We utilize hash tables with universal hashing [40], [41] to greatly reduce the space needed. Within those hash tables we store pointers to all existing smaller clusters $c_1, ..., c_N$. Since hash tables use a hashing algorithm that can lead to hash collisions, a fill ratio for the hash tables can be chosen to trade off hash collisions and space required.

7) Inserting clusters: After creating all cluster objects and assigning their respective hash table to them, the clusters need to be inserted into their parent clusters hash table. The first cluster with $wordlength = 2^d$ is always on x_0 and is regarded as the root of the tree. To avoid race conditions when inserting clusters, at first only clusters with $wordlength = 2^{d-1}$ are inserted. This iterative process is repeated until we hit $wordlength = 4$. After this, a small bit map is used to store the information about clusters, as mentioned in III-B3.

For every value x_n it is determined, using the now existing vEB structure, which cluster needs the respective bit set in its small bit map.

8) Querying: After all the previous steps of building the vEB tree the structure is ready to be queried. The algorithm for querying is equal to 1 with extra checks for $wordlength = 4$, as the recursion stops there and the bit field is queried instead. Bit field querying is trivial using the `ffs` (find first set) and `clz` (count leading zeros) intrinsics after shifting the bit field accordingly.

9) Building complexity: Iterative algorithms, such as determining the cluster sizes, search the set D with n elements by checking consecutive values have a time complexity of $O(n)$. Utilizing binary search on either the sorted set or first flags achieves $O(\log_2(n))$. Since smaller clusters are more common and their amount of clusters stored within are comparatively small, it is beneficial to only check the first 5–10 elements and then proceed to use a binary search, should the search condition have not been met yet.

Since binary search is the slowest algorithm in the full building pipeline the overall time complexity is $O(n \cdot \log_2(n))$. The biggest benefit of our method is that it is able to scale efficiently with the number of cores available, which makes it perfect for the usage on GPUs.

IV. Evaluation

For the evaluation we used a NVIDIA RTX 3090 graphics card using CUDA 11.1.1 [42] while running the algorithm in 64 bit mode, as the address space is required to be bigger than 2^{32} for the algorithm to process large data sets. To generate random data the library cuRAND [43] is used.

Fig. 5 shows the comparison between the query times when using vEB and binary predecessor search. We ran multiple

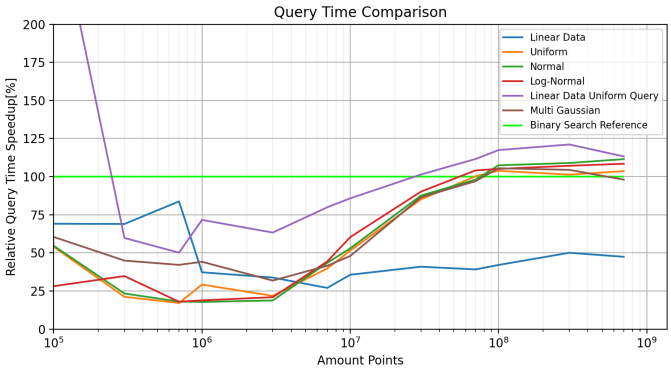

Fig. 5. vEB query times compared to binary search (100% corresponds to binary search).

queries in parallel on the GPU by using one GPU thread per query element and the same distribution function with different seeds for data generation and query unless explicitly mentioned otherwise. Linear data corresponds to fully occupied sequential data $(1, 2, 3, ...)$, uniform, normal and log-normal distributions are self-explanatory, linear data-uniform query represents random queries within a fully occupied range, and multi Gaussion uses two shifted normal distributions. When querying using binary search, we first de-duplicate the data set as the vEB tree implicitly does the same and our evaluation would otherwise be unfairly skewed.

When using linear data, the time for querying exceeds the query time for binary search for all n. Binary search outperforms the vEB tree in this case as the queries made to the data are also linear therefore benefit from caching, whereas the vEB tree suffers from a computational overhead and cannot benefit as much from caching.

In all other cases the vEB tree outperforms the binary search after reaching a certain threshold where the accesses to memory are slower for binary search than the constant computational overhead for the vEB tree. Note that we focus on 10^6 and more elements, as lower counts do not utilize the GPU well and results fluctuate significantly between query counts.

Fig. 6 shows the space overhead compared to the de-duplicated data. The complete vEB tree consists out of a single hash table array and the clusters array. Linear data shows the best case for the vEB tree space as all values are next to each other. This guarantees that a large part of the values are stored within the bit fields and therefore do not generate clusters. The closer the values are together the less space is being used. This is also noticeable for the normal and log normal distribution as the overall range of possible values stays the same and as more values are being added the gaps between values get smaller.

Fig. 7 shows the build times of vEB trees for various distributions, including the sorting of the data set. Build times are mostly independent of the distribution.

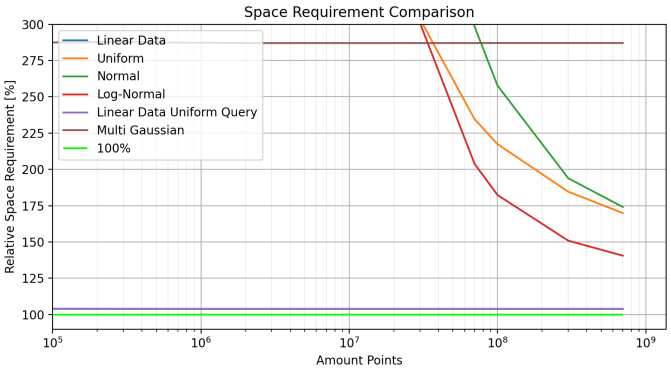

Fig. 6. vEB space overhead compared to the raw data for binary search.

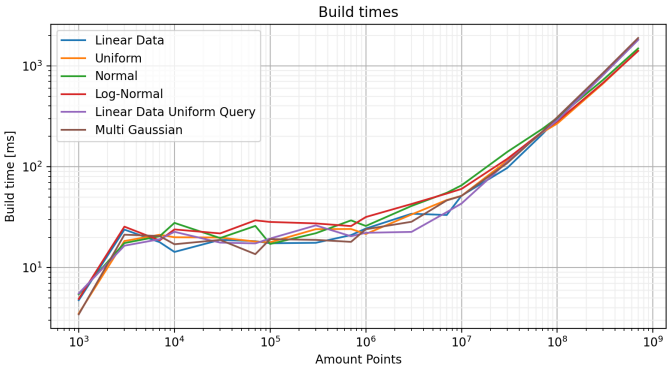

Fig. 7. vEB build times in ms.

V. Discussion

A. Performance

To our knowledge we show the first GPU-based vEB tree build algorithm. We also compared our novel construction algorithm with a traditional construction algorithm on the CPU, for which our GPU implementation outperforms the traditional by at least one order of magnitude up to 20 million values. For larger data sets the memory consumption and likely heap fragmentation resulted in even more severe speedups of our GPU implementation. While we did not run an extensive formal companion with advanced CPU vEB tree implementations, it still shows that building vEB trees directly on the GPU is viable.

Querying a vEB tree as outlined in Alg. 1 is a recursive procedure which is a heavy strain on the limited stack size of each thread on the GPU and thus limits the performance heavily. Additionally, the procedure features multiple data dependent branches, which leads to inter warp divergence and reduces performance when running multiple queries in parallel. To avoid using a stack for the recursive call on the GPU, we unroll the loop. This is possible because the maximum recursion depth can be calculated beforehand and depends only on the maximum word length.

Despite the mentioned optimizations querying a vEB tree on the GPU is only up to $1.2\times$ faster for very large data sets than binary search on the same data. For smaller data sets the memory overhead of vEB trees, combined with a higher thread divergence, does out-weight the smaller iteration count of querying a vEB tree. Thus, queries using vEB trees are only viable when working on very large data. Also, the performance gain is limited to about $1.2\times$. However, the performance gain was mostly independent of the data distribution, only when showing a very high correlation between both the data and the query elements in neighboring threads, binary search always outperforms our vEB tree—which is a very contrived case.

B. Space

Cluster sizes and hash table element sizes are the dominating factors for the space requirements. For simplicity our implementation does not specialize on the word length for storage, instead clusters are all the same size. Changing the clusters implementation to separate smaller structures for smaller word lengths could improve the space impact they have. Furthermore, these arrays could improve the performance as the overall memory consumption would decrease. The space overhead (for those data sizes where vEB trees achieve good performance) ranges between $1.5\times$ to $2.5\times$.

C. Limitations

Data locality plays an important role in space requirements of vEB trees. Big gaps between data points increase the number of clusters needed. For smaller data sizes a space overhead of more than a factor of 20 is possible. This is due to large distances between values, requiring each value to generate at least one cluster which needs at least ten-fold the memory compared to storing just the value.

VI. Conclusion

We showed that vEB trees can be built efficiently on the GPU using our novel parallel building algorithm. Building times for the vEB tree are bounded by each thread needing at most logarithmic time. We also showed that in certain cases, it is possible to keep additional space requirements comparatively low. vEB trees can outperform binary search on very large data sets for various data distributions. Thus, our vEB tree implementation has shown promising results for a certain set of use case. While one cannot rate vEB trees as universally viable on the GPU, they can be a viable tool when used in the right place. While dynamic memory allocation on the GPU is in general a costly process [44], [45], using queues [46] or dynamic hashing [29] may even allow for dynamically growing vEB trees.

References

[1] Louis F Williams Jr. A modification to the half-interval search (binary search) method. In *Proceedings of the 14th annual Southeast regional conference*, pages 95–101, 1976.

[2] Donald E. Knuth. Optimum binary search trees. *Acta informatica*, 1(1):14–25, 1971.

[3] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Mathematical systems theory*, 10(1):99–127, 1976.

[4] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 75–84. IEEE, 1975.

[5] Dan E Willard. Log-logarithmic worst-case range queries are possible in space θ (n). *Information Processing Letters*, 17(2):81–84, 1983.

[6] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[7] Mark Priestley. *Routines of Substitution: John von Neumann's Work on Software Development, 1945–1948*. Springer, 2018.

[8] Erik D Demaine, Dion Harmon, John Iacono, and Mihai P a ˘ traşcu. Dynamic optimality—almost. *SIAM Journal on Computing*, 37(1):240–251, 2007.

[9] Susanne Albers and Marek Karpinski. Randomized splay trees: theoretical and experimental results. *Information Processing Letters*, 81(4):213–221, 2002.

[10] Heejin Park and Kunsoo Park. Parallel algorithms for red–black trees. *Theoretical Computer Science*, 262(1-2):415–435, 2001.

[11] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library, 2009.

[12] Qiming Hou, Xin Sun, Kun Zhou, Christian Lauterbach, and Dinesh Manocha. Memory-scalable gpu spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):466–474, 2010.

[13] Piotr Danilewski, Stefan Popov, and Philipp Slusallek. Binned sah kd-tree construction on a gpu. *Saarland University*, pages 1–15, 2010.

[14] Kirill Garanzha, Simon Premože, Alexander Bely, and Vladimir Galaktionov. Grid-based sah bvh construction on a gpu. *The Visual Computer*, 27(6):697–706, 2011.

[15] Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. Bonsai: a gpu tree-code. *arXiv preprint arXiv:1204.2280*, 2012.

[16] Mohammad M Hossain, Thomas M Tucker, Thomas R Kurfess, and Richard W Vuduc. A gpu-parallel construction of volumetric tree. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–4, 2015.

[17] Gunther Lukat and Robi Banerjee. A gpu accelerated barnes–hut tree code for flash4. *New Astronomy*, 45:14–28, 2016.

[18] David Wehr and Rafael Radkowski. Parallel kd-tree construction on the gpu with an adaptive split and sort strategy. *International Journal of Parallel Programming*, 46(6):1139–1156, 2018.

[19] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. Softshell: Dynamic scheduling on GPUs. *ACM Trans. Graph.*, 31(6):161:1–161:11, November 2012.

[20] Markus Steinberger, Michael Kenzel, Bernhard Kainz, J"org M"uller, Wonka Peter, and Dieter Schmalstieg. Parallel generation of architecture on the gpu. *Comput. Graph. Forum*, 33(2):73–82, May 2014.

[21] Markus Steinberger, Michael Kenzel, Bernhard Kainz, Peter Wonka, and Dieter Schmalstieg. On-the-fly generation and rendering of infinite cities on the gpu. *Comput. Graph. Forum*, 33(2):105–114, May 2014.

[22] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. Whippletree: Task-based scheduling of dynamic workloads on the gpu. *ACM Trans. Graph.*, 33(6):228:1–228:11, November 2014.

[23] Andreas Derler, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. Dynamic scheduling for efficient hierarchical sparse matrix operations on the gpu. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 7:1–7:10, New York, NY, USA, 2017. ACM.

[24] Damjan Strnad and Andrej Nerat. Parallel construction of classification trees on a gpu. *Concurrency and Computation: Practice and Experience*, 28(5):1417–1436, 2016.

[25] Aziz Nasridinov, Yangsun Lee, and Young-Ho Park. Decision tree construction on gpu: ubiquitous parallel computing approach. *Computing*, 96(5):403–413, 2014.

[26] Kamil Rocki and Reiji Suda. Parallel minimax tree searching on gpu. In *International Conference on Parallel Processing and Applied Mathematics*, pages 449–456. Springer, 2009.

[27] Moohyeon Nam, Jinwoong Kim, and Beomseok Nam. Parallel tree traversal for nearest neighbor query on the gpu. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 113–122. IEEE, 2016.

[28] Yun-Ta Tsai, Markus Steinberger, Dawid Pajak, and Kari Pulli. Fast ann for high-quality collaborative filtering. In *Computer Graphics Forum*, volume 35, pages 138–151. Wiley Online Library, 2016.

[29] Saman Ashkiani, Martin Farach-Colton, and John D Owens. A dynamic hash table for the gpu. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 419–429. IEEE, 2018.

[30] Saman Ashkiani, Shengren Li, Martin Farach-Colton, Nina Amenta, and John D Owens. Gpu lsm: A dynamic dictionary data structure for the gpu. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 430–440. IEEE, 2018.

[31] Sushil K. Prasad, Michael McDermott, Xi He, and Satish Puri. Gpu-based parallel r-tree construction and querying. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 618–627, 2015.

[32] Muhammad A Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D Owens. Engineering a high-performance gpu b-tree. In *Proceedings of the 24th symposium on principles and practice of parallel programming*, pages 145–157, 2019.

[33] Timothy M Chan. Persistent predecessor search and orthogonal point location on the word ram. *ACM Transactions on Algorithms (TALG)*, 9(3):1–22, 2013.

[34] Mikkel Thorup and Uri Zwick. Compact routing schemes. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 1–10, 2001.

[35] TV Lakshman and Dimitrios Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *ACM SIGCOMM Computer Communication Review*, 28(4):203–214, 1998.

[36] Hao Wang and Bill Lin. Pipelined van emde boas tree: Algorithms, analysis, and applications. In *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*, pages 2471–2475. IEEE, 2007.

[37] Donald E Kunth. The art of computer programming: Vol. 3, sorting and searching, 2nd printing, 1975.

[38] Marco Zagha and Guy E Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 712–721, 1991.

[39] Duane Merrill. Cub. *NVIDIA Research*, 2015.

[40] D. Farrell. A simple gpu hash table. https://github.com/nosferalatu/SimpleGPUHashTable, 2020. Last accessed on: 2021-07.

[41] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[42] NVIDIA. Cuda, release: 11.1.1. https://developer.nvidia.com/cuda-toolkit, 2021. Last accessed on: 2021-08.

[43] NVIDIA. Cuda curand libary. https://docs.nvidia.com/cuda/curand/, 2021. Last accessed on: 2021-07.

[44] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. Scatteralloc: Massively parallel dynamic memory allocation for the GPU. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, May 2012.

[45] Martin Winter, Mathias Parger, Daniel Mlakar, and Markus Steinberger. Are dynamic memory managers on gpus slow? a survey and benchmarks. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 219–233, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Martin Winter, Daniel Mlakar, Mathias Parger, and Markus Steinberger. Ouroboros: Virtualized queues for dynamic memory management on gpus. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery.