Int J Parallel Prog (2018) 46:1139–1156
https://doi.org/10.1007/s10766-018-0571-0

Parallel kd-Tree Construction on the GPU with an Adaptive Split and Sort Strategy

David Wehr[1] · Rafael Radkowski[1]

Received: 21 April 2017 / Accepted: 9 April 2018 / Published online: 16 April 2018

Abstract We introduce a parallel kd-tree construction method for 3-dimensional points on a GPU which employs a sorting algorithm that maintains high parallelism throughout construction. Typically, large arrays in the upper levels of a kd-tree do not yield high performance when computing each node in one thread. Conversely, small arrays in the lower levels of the tree do not benefit from typical parallel sorts. To address these issues, the proposed sorting approach uses a modified parallel sort on the upper levels before switching to basic parallelization on the lower levels. Our work focuses on 3D point registration and our results indicate that a speed gain by a factor of 100 can be achieved in comparison to a naive parallel algorithm for a typical scene.

Keywords GPU computing · kd-Tree construction · Parallel radix sort · 3D-point registration

1 Introduction

Sorting and searching data is a key function of many computer science applications. Thus, it is all the more important that sorting data and searching can be performed as efficiently as possible. Our research addresses real-time object tracking for augmented reality (Fig. 1a) using point datasets obtained from range cameras [18, 19]. The object of interest is typically a subset of the scene point set. To identify and track it, we

✉ Rafael Radkowski
rafael@iastate.edu

David Wehr
dawehr@iastate.edu

1 Virtual Reality Applications Center, Iowa State University, Ames, IA 50011, USA

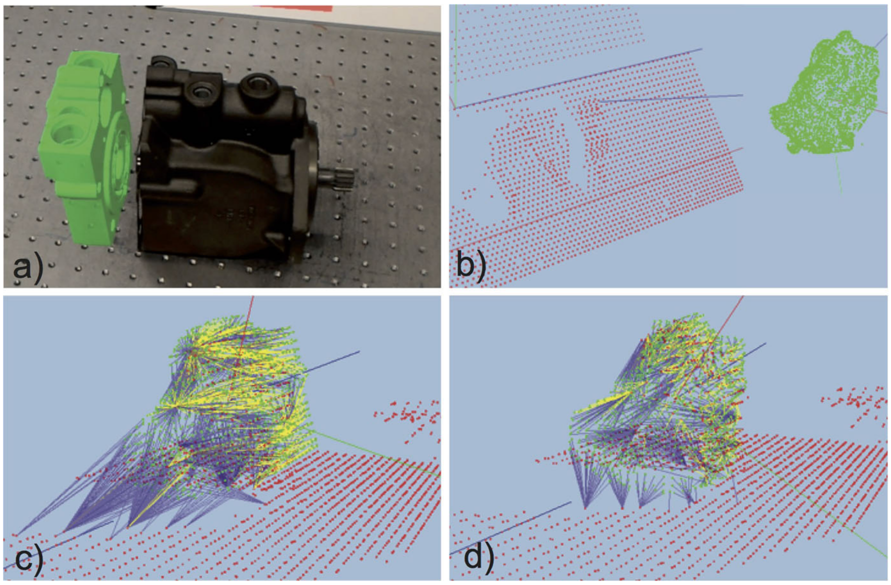

Fig. 1 **a** Tracking a piston motor allows one to register virtual objects, spatially aligned with the physical object. **b–d** A visualization of the matching and registration process in order to align a reference dataset (green) with its counterpart in the environment (red)

match a reference dataset with the related point set. Technically, we solve a matching by alignment task in two steps. First, we use feature descriptors (axis-angle descriptors) to obtain a rough alignment between a reference point set and the related points in the environment point set. This estimate is further refined using the Iterative Closest Point algorithm (ICP). The result is a reference object, which is perfectly aligned with the counterpart in the environment (Fig. 1b–d). Repeating this frame by frame allows for object tracking.

Feature descriptor matching and ICP algorithms require nearest neighbors for execution. Feature descriptors rely on adjacent neighbors to represent the surface characteristic of the object. ICP uses neighboring point pairs to align the two point sets. A state-of-the-art solution to efficiently find nearest neighbors is a kd-tree. A kd-tree represents the data in a spatial tree which increases the performance of nearest-neighbor search to $O(n \log(n))$ [3], in comparison to a brute force approach with $O(n^2)$. Although the kd-tree increases the performance of finding nearest neighbors, the procedural generation of a kd-tree is the bottleneck of the entire method.

Parallel kd-tree generation on a GPU is one approach to increase the performance. Several algorithms were already introduced [12,17,22]. Each follow different strategies; however, the most complex part remains the sort operation. The sort algorithm that yields the highest performance on a GPU for large datasets is the radix sort algorithm [20]. Its sequential implementation sorts a dataset in O(n); parallel implementations are even faster. However, the parallel radix sort solutions work well only if the dataset saturates the GPU, meaning many threads are utilized and each thread sorts an adequate amount of data. Performance is lost for large data arrays and very small ones, which typically occur on the upper and lower levels of a kd-tree.

The goal of this effort is to investigate a sort algorithm and strategy which can maintain high parallelism at all levels of the tree by using a modified parallel sort as well as switch to different sequential algorithms with respect to the number of points to sort. In contrast to the related research, we work with medium-size (300.000) point sets and switch between a parallel radix sort, sequential insertion sort, and a sequential radix sort for large arrays, normal sized arrays, and small arrays of points. We tested our approach with random data and a typical scene setup and compared it to a naive GPU solution. The results indicate a speed increase by factor 100.

The remaining paper is structured as follows. Section 2 reviews the related research and provides the required background information for this paper. Afterwards, we explain our GPU realization in Sect. 3. Section 4 describes performance tests and comparison. The last section closes the paper with a conclusion and an outlook.

2 Background and Related Work

A kd(imensional)-tree is a data structure to organize points with k dimensions, in our case, 3-dimensional points $\mathbf{x} = \{x, y, z\} \in \mathbb{R}^3$. It belongs to the binary search tree family and was originally examined in [3–5]. In difference to a binary tree, it constrains the dimensions on each level of the tree: Each level splits all points along one specific dimension (x, y, or z) using a hyperplane which is perpendicular to the related axis. The hyperplane separates all points into two areas: All points lower than the split value can be considered left of the hyperplane, the other points at the right. The split dimension changes at each level, starting with x at the root, to y, to z, and re-starting at x again. To identify the hyperplane, a kd-tree algorithm needs to find a pivot point for each node of the tree, which incorporates sorting the area and determining the median value, which is typically used as the split value (the hyperplane). Sorting a large dataset is the performance sink of a kd-tree algorithm and subject for optimization.

In recent years, plenty of GPU solutions to speed up the construction and the nearest neighbor search with a kd-tree were introduced. In an early work, Garcia et al. [6] compared a brute-force nearest neighbor search implemented on a GPU with a MATLAB implementation. The authors report a speedup up to a factor of 100.

Zhou et al. [22] introduced a kd-tree construction algorithm for a GPU with a focus on computer graphic primitives and ray-tracing. The algorithm constructs the tree nodes in breadth-first search order. The authors examined a strategy for upper level nodes, i.e. nodes with many points to sort, in order to maintain fine-grained parallelism of a GPU. They employ a combination of spatial median splitting and empty space maximizing, to sort the arrays of large nodes, which first computes bounding boxes per node and further splits the array into chunks to yield an optimal GPU saturation [11]. Although the authors introduce a similar strategy as we follow, their dataset is assembled to support ray tracing. The paper reports a speedup factor in the range of 9–13.

The authors of [17] published a nearest neighbor search solution for a GPU. A kd-tree is built on the CPU and copied to the GPU to execute the search. The authors' work is on 3D object registration, thus, they have to solve the same problems we address. However, they did not construct a kd-tree on a GPU.

Leite et al. [14,15] construct a kd-tree on a gpu using a grid spatial hashing. The grid size can be set by users to tailor the algorithm for different scenarios. The authors also perform the search on the gpu to minimize the amount of data that need to be copied between the gpu and the host memory.

Karras [13] suggested an in-place algorithm to construct binary radix trees. The approach sorts the data for the tree in parallel and the connection between the segments of the array is maintained by assigning indices to nodes. In [8], Ha et al. present a 4-way radix sort algorithm for a gpu, which demonstrate a significant performance increase. Singh et al. [21] published a survey addressing several sort algorithms optimized for a gpu.

Although the results of the previous work show a high performance gain in comparison to a brute-force approach, the efforts focus on 3-dimensional datasets. Recently, Hu et al. [12] introduce parallel algorithms for KD-tree construction and nearest neighbor search with a focus on high dimensional datasets. In their implementation, each node at the same level of the tree is processed in an individual thread, computing the split index and dimension for the current node and redistributes all points. The authors' algorithm yields a speedup ranging from a factor of 30 to a factor of 242 for point sets of different sizes.

Our effort is inspired by Hu et al. [12] and Zhou et al. [22]. We maintain the strategy to compute each node in a single thread. However, we design a sort algorithm that utilizes many threads, regardless of the number or size of arrays to sort, which yields further speedup. Sorting the point set along the split dimension is the most costly procedure of the kd-tree generation. Running a single sort algorithm for each node limits the performance, especially when the number of points per array are small. At the upper levels of a kd-tree, the array is split into just two or four nodes, so assigning each node to a thread does not yield a significant performance gain, since the GPU remains under-saturated. Designing the algorithm to utilize all available threads throughout the entire construction yields the highest performance, as the GPU can remain fully saturated.

3 GPU Realization

The following section describes our parallel kd-tree construction algorithm. The tree is built iteratively in a top-down manner. Input data is an array of N points, and the output is a kd-tree data structure.

Sorting the points along a dimension for each node causes the largest performance bottleneck during kd-tree construction. This is the performance limiter which we optimize in our algorithm. Sorting is performed in parallel to utilize the increased computation resources on the GPU. However, the workload is not trivially parallelizable, especially at the beginning of the construction process, when only a few arrays

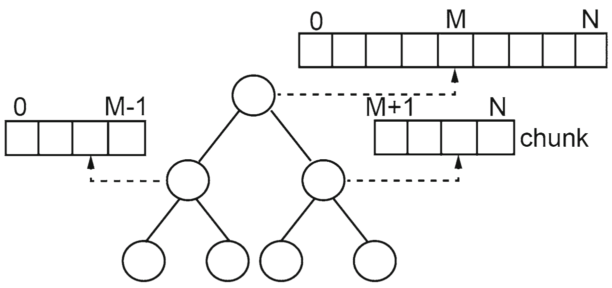

Fig. 2 Level 1 handles the entire array, and splits on the median between 0 and N. Nodes in level 2 each handle corresponding sub-arrays ("chunks") from their parent. This pattern continues until each node only handles a single element

need to be sorted. To prevent bottlenecks in the upper levels, we use a modified parallel radix sort to evenly distribute the work across the GPU (Sect. 3.2). In the lower levels, there are enough nodes being independently created that parallelization is achieved by simply running a single standard sort per node. Additionally, to avoid the overhead of storing the median and bounds of each node during construction, we use a closed-form median approximation formula, as in [7].

The following subsections provide an overview of the parallel kd-tree construction, explanation of memory management, and a closer look at the particular parts of the algorithm. We utilize CUDA and CUB[1] for our implementation.

3.1 kd-Tree Construction for Vector Data in $\mathbb{R}^3$

Algorithm 1 describes the strategy for our parallel kd-tree construction. The main algorithm runs on the CPU and invokes work units to run in parallel on the GPU. First, each d-dimensional point is split into d separate datasets and scaled by a factor F to quantize the values so they can be sorted by an integer sort. Separating the points by dimension coalesces memory access since only one dimension is sorted at a time, which improves GPU performance [9,16]. We also compute the maximum element of the dataset per dimension to minimize the number of iterations required during radix sort. At each level of the kd-tree, we sort the points by dimension, compute the median splitting elements, and add the median elements to the tree. Figure 2 shows how each node corresponds to both a subsection of the array and a splitting element. The proposed algorithm recursively constructs a kd-tree using linear-time radix sort on the upper layers of a kd-tree, resulting in $O(n\log(n))$ runtime. The time spent performing insertion sort on the final levels (see Algorithm 2) can be considered overall linear, since only arrays less than a fixed size (16 in our case) are sorted using insertion sort, and the number of sub-arrays less than a constant size is linear in N.

[1] https://nvlabs.github.io/cub.

Algorithm 1: The construction algorithm

```
Input  : Array of N points in $\mathbb{R}^3$
Output: kd-tree data structure
initialization (memory allocation);
$(x, y, z, index) \leftarrow$ parallel index generation: store an array index for each point $\mathbf{p}_i$
$(p_0, p_1, p_2) \leftarrow$ parallel split, scale, and quantize: split array in $\mathbb{R}^3$ into components of 3 arrays in
$\{0 \dots F\}$;
$(max_0, max_1, max_2) \leftarrow$ Find max values;
$l \leftarrow 0$
1: for $l = 0$ to $l = \lfloor \log_2 N \rfloor$ do
2:     $(p_d, index) \leftarrow$ Sort (different strategies);
3:     $(p_0, p_1, p_2) \leftarrow$ Parallel reorganization: Re-organize all other points to align with the sorted $p_d$ ;
4:     Parallel create tree nodes for current level
5: end for
```

3.2 Sorting Strategy

The key reason for the performance of our approach is the ability to evenly distribute the sorting load across the GPU, since our profiling shows that 95% of the time constructing the tree is spent sorting.

The challenge in evenly distributing the load is that the number of arrays and their size change with each level of the kd-tree data structure. For example, with N points, the first level requires 1 sort of N points. The second level requires 2 sorts of $\frac{N-1}{2}$ points each, and so forth, until the final level requires $\frac{N}{2}$ sorts of 1 point each. Typical parallel sorting algorithms typically work best sorting large arrays, but have too much overhead for sorting smaller arrays.

Our primary method for optimizing sorting to work with the wide range of array sizes is to use a variation on parallel radix sort that can operate on the entire array of N points by handling each sub-array independently. The second optimization is to switch to executing multiple sequential sorts in parallel when the number of sorts is large enough to saturate the threads.

Algorithm 2: Sort

```
Input  : Array of N points in $\mathbb{N}$
Output: $p_d, index$
$(chunksize) \leftarrow \frac{N}{2^l}$
1: if $chunksize \geq cutoff$ then
2:     $(p_d, index) \leftarrow$ Parallel radix sort
3: else if $16 \leq chunksize < cutoff$ then
4:     $(p_d, index) \leftarrow$ Sequential radix sort
5: else
6:     $(p_d, index) \leftarrow$ Sequential insertion sort
7: end if
```

Algorithm 2 depicts the overall sorting strategy. The utilized sorting algorithm depends upon *chunksize*, where a *chunk* is the sub-array that is handled by a particular node (Fig. 2). The total number of points to be sorted is constant, thus $chunksize$ can be computed from Eq. 1, where l is the level of the kd-tree to be constructed and the root node is at level 0.

$$chunksize = \frac{N}{2^l} \tag{1}$$

If the chunk size is at least as large as a specified cutoff point, we sort the entire array at once using our parallel radix sort algorithm (Algorithm 3). When the $chunksize$ drops below *cutoff*, we sort each chunk using a sequential radix sort, with one chunk per thread. The final levels with a $chunksize$ of less than 16 are sorted with a sequential insertion sort running on a separate thread for each chunk.

The explanation for a certain *cutoff* size is further elaborated in Sect. 3.3. The number 16 as threshold between radix sort and insertion sort is inspired by common techniques among hyrbid sorts for small arrays, since the $O(n^2)$ performance is offset by a low constant factor. For instance, the C++ standard template library (libstdc++ specifically) switches to insertion sort when arrays fall below 15 elements. We chose a similar value in an attempt to have good performance over a variety of GPUs.

Algorithm 3: Parallel Radix sort

```
Input  : Array of N points in $\mathbb{N}$, $max_d$
Output: $p_d$, $index$
$magnitude \leftarrow 1$
1: while $\left\lfloor \frac{max_d}{magnitude} \right\rfloor > 0$ do
2:     Clear memory $H$
3:     $(H) \leftarrow$ parallel histogram creation
4:     $(H') \leftarrow$ parallel inclusive prefix sum of $H$.
5:     $(T) \leftarrow$ parallel distribute counts
6:     $(N) \leftarrow$ parallel copy of T
7:     $magnitude \leftarrow magnitude \cdot base$
8: end while
```

3.3 Parallel Radix Sort

Algorithm 3 describes the main steps in our parallel radix sort. It can be viewed as a segmented version of standard parallel radix sort [9]. Radix sort requires multiple iterations of a stable sort—in our case, count sort. As in standard radix sort, we loop over increasing magnitudes of *base* until we have sorted the largest magnitude which occurs in our dataset.

To allow for sorting segments independently, each chunk has a separate set of histograms kept in global memory, visualized as layers in the diagram (Fig. 3). Each block handles *TPB* (threads per block) number of points in the array and computes a block-local histogram of the digits. These histograms are stored in block-local memory (called shared memory in CUDA) during construction for fast access.

The amount of block-local memory required per block is proportional to the number of chunks it overlaps. Because chunk size decreases by half at each level, the number

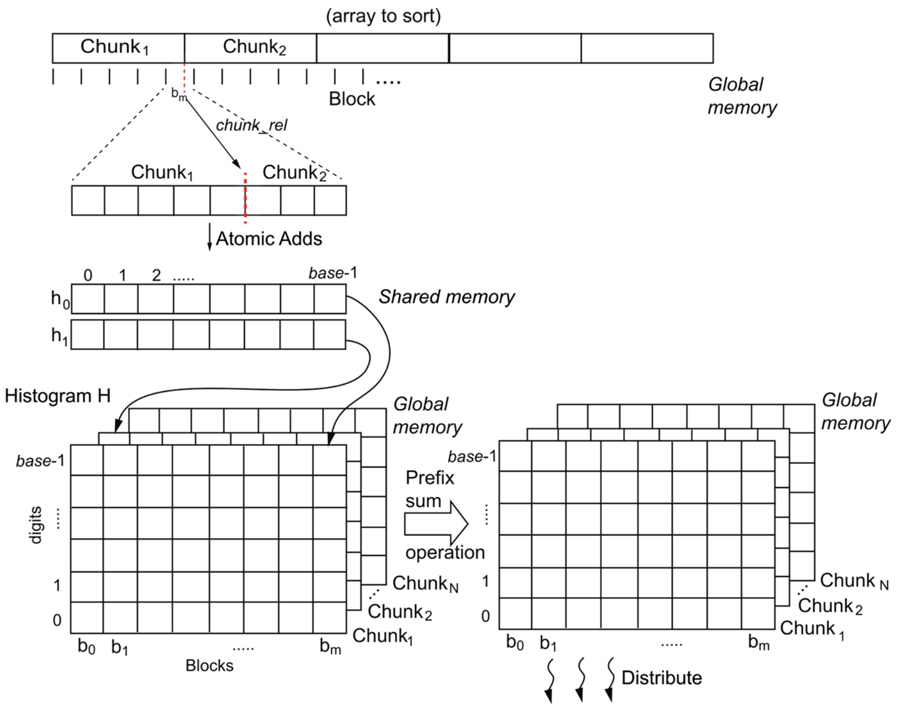

Fig. 3 Schematic of the memory management for the parallel radix sort

of chunks that are overlapped, and thus the memory required, grows exponentially in the final levels. To prevent this exponential memory requirement, we limit each block to overlapping at most two chunks, resulting in two histograms: h_0 and h_1. This implies the *cutoff* parameter for switching must be at least *TPB*. We set *cutoff* = *TPB* to allow the parallel sort to execute on as many levels of construction as possible. Setting *cutoff* as low as possible is validated in Sect. 4.1.

Once the local histograms are computed, they are copied to the corresponding location in the global histogram, H. As in [20], we store the histograms in column-major order to allow a prefix sum operation to compute the final location of each point in the array. The chunks are stored back-to-back in memory to uphold this property across the entire array.

Lastly, the points are distributed to their corresponding final location in a temporary array (Algorithm 5) using the offsets from the prefix-sum of H. Because the distribution must be stable, we distribute each local histogram (column) in a sequential manner, requiring *TPB* iterations of a loop.

The parallel histogram creation is shown in Algorithm 4. It is executed with one thread per point. The general approach is to take advantage of fast atomic operations on block-local memory for computing a block-local histogram, then copying that into global memory. Each thread handles a single point, but works cooperatively with the other threads within the block via block-local memory. The initialization of h_0 and h_1

is done once per block, as is the transfer of h_0 and h_1 to global memory. Determining which chunk a point belongs to is described in the appendix.

Algorithm 4: Parallel histogram creation

```
Input  : Array of N points in $\mathbb{N}$, $max_d$, $chunksize$, $H$, $e$, $thread\_index$
Output: $H$
 1: if $thread\_index = 0$ then
 2:     $(h_0, h_1) \leftarrow$ Initialize block-local memory arrays, each of size $base$, to 0
 3: end if;
 4: thread barrier;
 5: $i \leftarrow$ Point index in $N$ this thread is assigned to
 6: $split \leftarrow$ Splitting if $chunk(i) \neq chunk(i+1)$
 7: $chunk\_rel \leftarrow chunk(i) - chunk$(first point in this block); $chunk\_rel \in \{0, 1\}$
 8: if $\neg split \wedge i < N$ then
 9:     $d \leftarrow digit$ of $N_i$ for magnitude $e$
10:     atomic($C_{chunk\_rel}[d] \leftarrow C_{chunk\_rel}[d] + 1$)
11: end if
12: thread barrier;
13: if $thread\_index = 0$ then
14:     $H \leftarrow$ Add $h_0$ and $h_1$ to their corresponding locations in the histogram
15: end if
```

Algorithm 5 describes the parallel distribution of values in detail. It is similar to standard radix sort, with the exception that each thread only handles a single histogram (columns in Fig. 3). We additionally populate an index array so the dimensions that are not being sorted can be rearranged to correspond to the currently sorted dimension array.

Algorithm 5: Parallel Distribution

```
Input  : Array of N points in $\mathbb{N}$, Temporary array $N'$ of N points, Array of indices $I$ $H$, $e$, $TPB$
Output: $N'$, $I'$
 1: $i_b \leftarrow$ Index of block assigned to this thread;
 2: $i_c \leftarrow$ Chunk index for block $i_b$;
 3: $n_{cb}, n_{ce} \leftarrow$ Start and end indices in $N$ for chunk;
 4: $h_b \leftarrow max(n_{cb}, i_b \cdot TPB)$ (Begin index of histogram);
 5: $h_e \leftarrow min(n_{ce}, (i_b + 1) \cdot TPB)$ (End index of histogram);
 6: for $i = h_e$ to $i = h_b$ do
 7:     $digit \leftarrow digit$ of $N_i$ for $magnitude$
 8:     $H \leftarrow$ Decrement value in H for block $i_b$ and digit $digit$
 9:     $i_{out} \leftarrow$ Value from H for block $i_b$ and digit $digit$
10:     $N'[i_{out}] \leftarrow N[i]$
11:     $I'[i_{out}] \leftarrow I[i]$
12: end for
```

The number of threads per block chosen (the value of *TPB*) has an impact on the overall performance. Theoretically, we expect that running a higher TPB count is more beneficial for larger point sets, while a low TPB is faster for smaller sets. This can be understood by noting that there are two steps contributing to the majority of the runtime: the local histogram creation, and the global histogram distribution. For higher numbers of threads per block, fewer blocks in the histogram creation allow for better parallelization when many (hundreds of thousands) of threads are involved, as well as fewer writes to global memory. On the other hand, higher numbers of threads per block causes the distribution step to run fewer threads that each take longer, which is a disadvantage for parallelism.

Theoretically, it is possible to identify an ideal value analytically, however, there are many factors that contribute to the performance, including GPU scheduling, memory access latencies, and cache timing, i.e. the GPU at hand. Therefore, the ideal *TPB* is dependent upon the number of points to sort, and we recommend a search among candidate values to find it. Any multiple of 32 up to 1024 is a viable block size candidate for TPB, since GPU warps are of 32 threads, and blocks cannot exceed 1024 threads. Searching among potential dataset sizes with a granularity of 10 results in $10 \cdot 32 = 320$ combinations of hyper-parameters to test. As the algorithm is meant for real-time applications, 320 tests amounts to very low total time, and the hyper-parameter search data can be collected only once for a particular GPU. See Sect. 4.1 for results of hyper-parameter searching.

3.4 Create Node

Node creation is done once per level, with one thread per newly created node. Each node needs to record the median element that it splits along, as visualized in Fig. 2. The computation for the median is done by computing the start index of the right child chunk, using the method outlined in the appendix. The array for the tree is pre-allocated, so we simply copy the point information into the corresponding location within the tree array.

3.5 Nearest Neighbor Search

For searching the kd-tree, we first copy the query points to the GPU so the search can be conducted in parallel with one thread per query point. For the application of object tracking, an approximate nearest neighbor search is sufficient, so we use a priority queue to handle the backtracking required for nearest neighbor search [1]. We extend this approach to use a double-ended priority queue, allowing us to place a bound on the number of enqueued nodes and only keep the most likely candidates [2]. The search is performed by descending the tree, and at each branch, placing the node not taken into the queue. If the queue is full and the new node is closer, then the furthest enqueued item will be removed and replaced. Once a leaf node is reached, the next nearest item in the queue is removed, and search continues from that point. Traversal of a node can be terminated based on the “bounds-overlap-ball” test [5].

4 Performance Tests

We ran several experiments to measure the performance of our approach. For variety, we chose three point sets for testing, including simulated and real-world data.

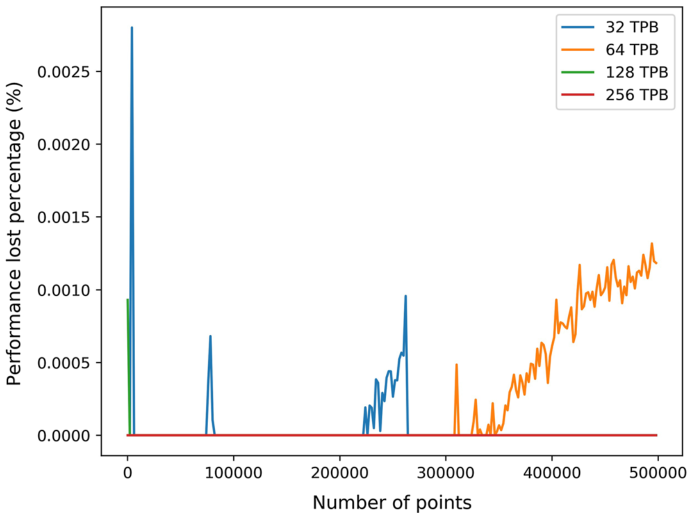

Fig. 4 Performance loss from setting *cutoff* = *TPB* versus the fastest experimental *cutoff* value for various data set sizes

First, we empirically chose the hyper-parameters which yield the best performance (Sect. 4.1). We further compared our approach with a current state-of-the-art method and a naive parallelization approach (Sect. 4.2). All experiments, except where noted, were executed on an NVIDIA Titan X GPU using CUDA 8.0.

4.1 Performance Analysis

Our approach has three adjustable parameters—namely the number of threads per block (*TPB*), the base for radix sort, and the cutoff point between parallel and sequential sorts. We ran a hyper-parameter experiment to verify our assumptions regarding the expected behavior.

We choose to set the cutoff point equal to the number of threads per block to maximize the time spent using the parallel sorting algorithm. Figure 4 shows the performance difference between setting *cutoff* = *TPB* versus the fastest experimental value of *cutoff*. We find that setting *cutoff* to *TPB* is nearly always ideal, with a loss of at most 0.0025%, which matches our expectations.

Our tests showed that setting the *base* parameter to 32 was optimal for all scenarios. All following experiments were done with $base = 32$.

The only hyper-parameter remaining is *TPB*, or threads per block. Figure 5 shows the execution times for different *TPB* values with mean and one standard deviation marked. Each experiment was repeated 25 times to obtain statistically significant results.

The results show that on average, 2.5 ms are required to construct a kd-tree with 2000 points and 30 ms for 500,000 points. It is also noticeable that 32 and 64 TBP yield the lowest runtime for a low number of points, up to 200,000. For larger numbers of points, 128 or even 256 threads per block are preferable.

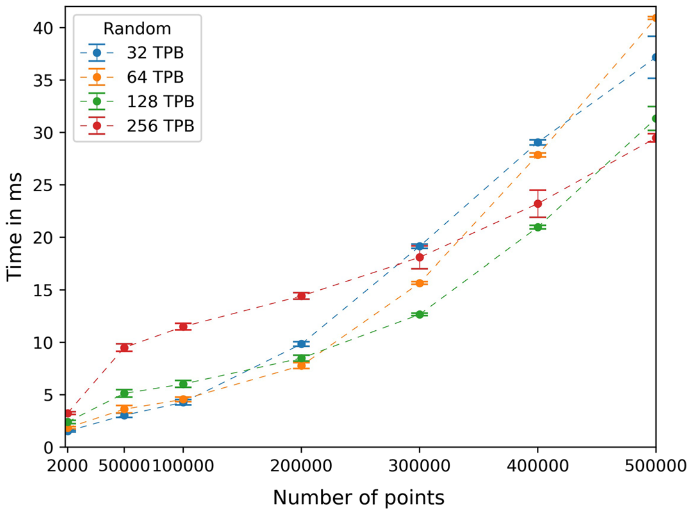

Fig. 5 Comparison of the performance for different numbers of randomly generated points and the numbers of threads per block (TPB) using the suggested method

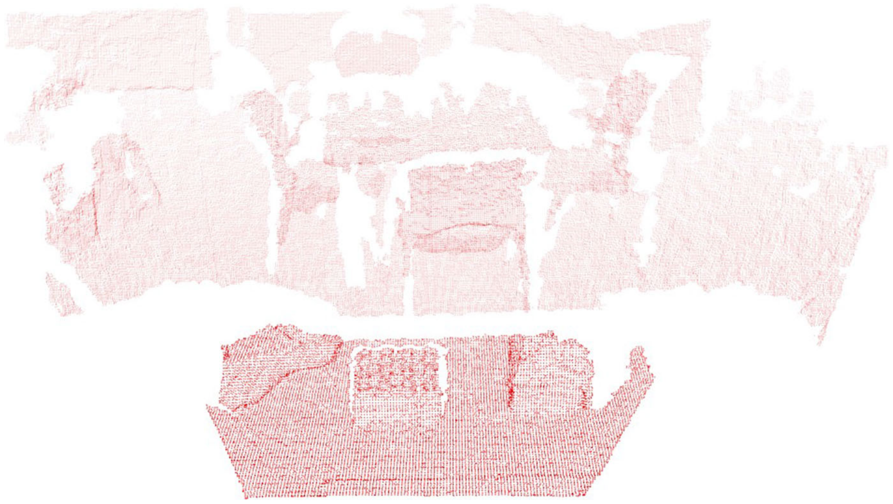

Fig. 6 Point set of a typical work environment

As mentioned above, we work on rigid object tracking which requires finding a small object of interest in a larger point dataset as described in [18,19]. Therefore, to verify the results on a practical application, a scan of a typical work environment was captured (Fig. 6) for testing. Note that a typical scene scan contains up to 280,000 points, which is the limit of our camera hardware. For repeatability, we also performed tests with the Stanford bunny. These results are depicted in Fig. 7. We find that the performance is similar to random points.

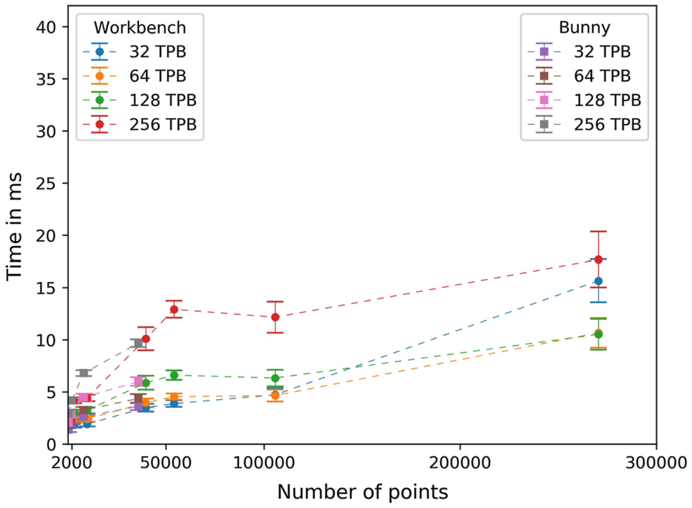

Fig. 7 Comparison of the performance for different numbers points from a typical scene with the numbers of threads per block (TPB) using the suggested method

The quantitative data for construction and nearest-neighbor search experiments are shown in Table 2 in the appendix.

To see the impact of each sort on the overall time, we recorded the amount of time spent executing parallel radix sort, sequential radix sort, and sequential insertion sort. The proportion of total time spent in parallel radix sort, serial radix sort, and serial insertion sort are 96.5 ± 0.36, 0.86 ± 0.39, and $2.64 \pm 0.39\%$, respectively, with one standard deviation. This, in conjunction with the observation that best times are achieved by switching from parallel radix sort as late as possible (staying within memory constraints), indicates that the modified parallel radix sort is the major contributor to the performance.

4.2 Comparisons

We compared our approach with the method suggested by Hu et al. [12] and ran experiments using the same GPU model (NVIDIA GeForce GTX 660) as they tested with. The times for each method are shown in Table 1. The results suggest that for small datasets ($< 102{,}400$), the proposed method is faster than current state of the art methods, which is reasonable. Hu et al. gain performance from a larger number of threads, thus, on lower levels of a kd-tree. For small numbers, the suggested method yields better performance due to the independent number of threads on each level of the tree.

We also anticipate that our approach may be able to better take advantage of modern GPUs for two reasons—firstly, it is able to utilize all threads (which modern GPUs have more of), even when there are few nodes being constructed. Secondly, the NVIDIA Maxwell architecture (CUDA Compute Capability 5.x) introduced hardware support for shared memory atomics [10], and fast atomic shared memory operations are crucial

Table 1 Time to construct kd-tree for 3-dimensional randomly generated points on an NVIDIA GeForce GTX 660

Number of points	Time (ms)	
	Proposed method	Hu, Nooshabadi, Ahmadi
3200	3.5	4.6
6400	4	7.2
1280	4.8	8.2
25,600	5.9	8.8
51,200	9.5	12.4
102,400	22.2	20.2

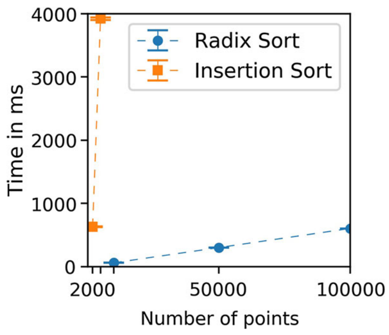

Fig. 8 Comparison of the performance for different numbers of randomly generated points and the type of sort using a simple parallelization technique

to our algorithm. Therefore, when running on a GeForce GTX 660 (CUDA compute capability 3.0), the histogram creation is significantly slower than on a newer GPU. Investigating these hypotheses remains as future work.

We further compared our approach with a simple parallelization technique by sorting each chunk independently on a single GPU thread. Radix sort and insertion sort were chosen because we use them in our hybrid sorting method. Figure 8 shows the results. Generating a kd-tree with only 10,000 points requires 61 ms for radix sort, and for 5000 points using insertion sort, already 4000 ms. Testing insertion sort with data sizes comparable to what we tested on the GPU was infeasible due to the quadratic runtime of insertion sort.

Lastly, we compared to a typical kd-tree construction on the CPU, shown in Fig. 9. The data were collected by running on an Intel Core i7-7700HQ processor.

Comparing the results with the suggested method, we yield a performance gain by a factor of up to $\sim$ 150 against both CPU construction and a simple parallelization method.

4.3 Discussion

Our results are roughly in line with what we expected regarding runtime growth, as radix sort is a linear-time algorithm. One of the interesting results is how higher *TPB*

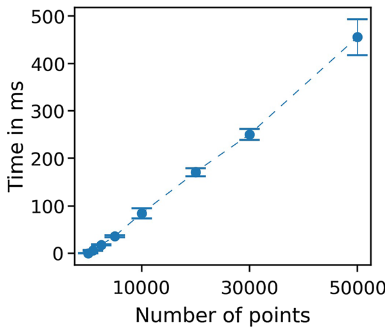

Fig. 9 Time for constructing a kd-tree on the CPU for different numbers of randomly generated points

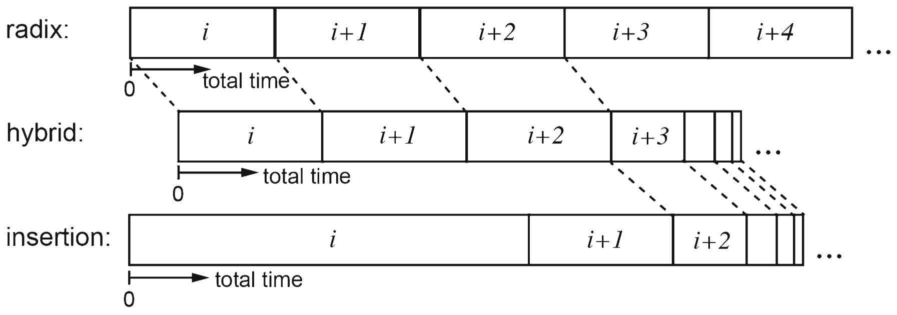

Fig. 10 Times for sorting final levels, using different sequential algorithms

values result in better performance for large data sizes, but worse for smaller datasets. As noted in Sect. 3.2, higher TPB makes the histogram creation more efficient since more work is done in block-local memory, but makes the distribution step slower, as it needs to loop over a larger number of points.

The final levels of the tree must be sorted using a non-parallel version, because of memory limitations, as described in Sect. 3.2. The switch to insertion sort for the final levels, although not a new idea, has a measurable impact on runtime, particularly for real-time systems. Figure 10 compares the amount of time to sort each level of the tree using only radix sort, only insertion sort, and the hybrid approach. As the level gets deeper, each sort is of fewer points, and thus the overhead of radix sort becomes more significant. We verified this behavior with the NVIDIA Visual Profiler.

5 Conclusion and Outlook

This paper introduced an approach for kd-tree generation on a GPU which improves the performance when working with medium-size point datasets ($\leq$ 500,000) by focusing on sorting, which is the major performance bottleneck. Typical sorting algorithms are not tailored to work with the highly variable data sizes encountered in kd-trees, making full GPU saturation difficult. Our approach to improve previous work is to adapt a parallel sorting algorithm to sort sub-sections of the data independently. Although the algorithm is limited to sorting larger sub-sections because of memory constraints, our approach mitigates this by switching to a simple parallelization model for the final levels of construction. The time is further decreased by switching from radix sort to insertion sort for very small arrays.

A comparison with a typical approach on the CPU indicates a performance gain of up to a factor of 150. Performance is also on comparable to a current state-of-the-art GPU kd-tree construction, with reason to believe it may be better on newer GPUs. Therefore, we conclude that a modified parallel radix sort in combination with switching sorting strategies improves the GPU performance of kd-tree construction in comparison to previously reported approaches.

The kd-tree implementation is currently used as part of our tracking software TrackingExpert [19] to find nearest neighbors to refine object alignment with ICP. In future work, we intend to use it for feature descriptor matching as well.

Appendix

Median Splitting Determination

I To determine which chunk a point belongs to, we use the technique described in [7] to compute a median. In brief, we consider the width w of a chunk to be a real number $w = \frac{N}{2^l}$, where l is the zero-indexed tree level. Therefore, given a particular index i, we can determine the chunk by $c = \frac{i}{w}$.

II Determining whether a point is a median splitting element is also necessary during the histogram calculation. That can be determined with the following criteria, as per [7].

$$splitting = \begin{cases} \lceil \frac{i}{w} \rceil < \frac{i+1}{w} \land i \neq 0; & true \\ \text{else}; & false \end{cases}$$

III Finally, we need the ability to calculate the starting index of a chunk, excluding the splitting element. Because the chunk width is constant, the starting index i_s of a chunk c can be computed by:

$$i_s = \begin{cases} c = 0; & 0 \\ i \neq 0; & \lfloor (w \cdot c) \rfloor + 1 \end{cases}$$

Experimental Data

See Tables 2 and 3.

Table 2 Quantitative results for all construction operations

Points	TPB	Comparison			Suggested method		
		R	WB	B	R	WB	B
2000	32	61	24	18	1.5	1.8	1.9
	64	61	24	18	1.8	2.2	2.2
	128	61	24	18	2.4	3.0	2.8
	256	61	24	18	3.2	4.2	4.2
50,000	32	302	485	315	3.0	3.8	3.6
(36,000)[a]	64	302	489	315	3.6	4.5	4.4
	128	302	489	315	5.1	6.6	6.0
	256	301	489	315	9.5	13	9.7
100,000	32	605	958	–	4.2	4.7	–
	64	605	958	–	4.6	4.7	–
	128	605	958	–	6.0	6.3	–
	256	605	958	–	11	12	
300,000	32	1918	2545	–	19	16	–
	64	1916	2478	–	16	11	–
	128	1917	2511	–	13	11	–
	256	1912	2579	–	18	18	–
400,000	32	2559	–	–	29	–	–
	64	2555	–	–	28	–	–
	128	2559	–	–	21	–	–
	256	2555	–	–	23	–	–
500,000	32	3245	–	–	37	–	–
	64	3249	–	–	41	–	–
	128	3251	–	–	31	–	–
	256	3249	–	–	29	–	–

The time in ms shows the mean value of all tests per item. *R* Random, *WB* Workbench Scene, *B* Stanford Bunny

[a]Bunny has only 36,000 points rather than 50,000

Table 3 Quantitative results for all nearest-neighbor queries

Tree size	Query size			
	100	10,000	50,000	100,000
2000	676 ± 71	1024 ± 103	4897 ± 115	9672 ± 128
5000	762 ± 104	1055 ± 90	5421 ± 160	$10{,}781 \pm 174$
50,000	871 ± 154	1502 ± 234	7851 ± 190	$15{,}769 \pm 174$
100,000	849 ± 158	1466 ± 152	8454 ± 109	$17{,}104 \pm 145$
200,000	844 ± 102	1763 ± 250	9293 ± 172	$18{,}896 \pm 180$
500,000	899 ± 122	1976 ± 348	$10{,}082 \pm 283$	$20{,}535 \pm 255$

The time in ms shows the mean and standard deviation of all tests

References

1. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. J. ACM **45**(6), 891–923 (1998)
2. Atkinson, M.D., Sack, J.R., Santoro, N., Strothotte, T.: Min-max heaps and generalized priority queues. Commun. ACM **29**(10), 996–1000 (1986)
3. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
4. Bentley, J.L.: Multidimensional divide-and-conquer. Commun. ACM **23**(4), 214–229 (1980)
5. Friedman, J.H., Bentley, J.L., Finkel, R.A.: An algorithm for finding best matches in logarithmic expected time. ACM Trans. Math. Softw. **3**(3), 209–226 (1977)
6. Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using GPU. In: 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (2008)
7. Garrett, T., Radkowski, R., Sheaffer, J.: Gpu-accelerated descriptor extraction process for 3d registration in augmented reality. In: 23rd International Conference on Pattern Recognition, Cancun, Mexico (2016)
8. Ha, L., Kruger, L., Silva, C.: Fast four-way parallel radix sorting on GPUs. Comput. Graph. Forum **28**(8), 2368–2378 (2009)
9. Harada, T., Howes, L.: Introduction to GPU radix sort. In: Heterogeneous Computing with OpenCL. Morgan Kaufman (2011)
10. Harris, M.: Maxwell: The Most Advanced CUDA GPU Ever Made. NVIDIA, Santa Clara (2014)
11. Havran, V.: Heuristic ray shooting algorithms. Ph.D. thesis, Czech Technical University, Czech Technical University (2001)
12. Hu, L., Nooshabadi, S., Ahmadi, M.: Massively parallel KD-tree construction and nearest neighbor search algorithms. In: 2015 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 2752–2755 (2015)
13. Karras, T.: Maximizing parallelism in the construction of BVHs, Octrees, and k-d trees. In: Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics, EGGH-HPG'12, pp. 33–37 (2012)
14. Leite, P., Teixeira, J.M., Farias, T., Reis, B., Teichrieb, V., Kelner, J.: Nearest neighbor searches on the GPU. Int. J. Parallel Program. **40**(3), 313–330 (2012)
15. Leite, P.J.S., Teixeira, J.M.X.N., de Farias, T.S.M.C., Teichrieb, V., Kelner, J.: Massively parallel nearest neighbor queries for dynamic point clouds on the GPU. In: 2009 21st International Symposium on Computer Architecture and High Performance Computing, pp. 19–25 (2009)
16. Merrill, D.G., Grimshaw, A.S.: Revisiting sorting for GPGPU stream architectures. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, pp. 545–546 (2010)
17. Qiu, D., May, S., Nüchter, A.: GPU-accelerated nearest neighbor search for 3D registration. In: Proceedings of Computer Vision Systems: 7th International Conference on Computer Vision Systems, ICVS 2009, pp. 194–203. Springer, Berlin (2009)
18. Radkowski, R.: Object tracking with a range camera for augmented reality assembly assistance. J. Comput. Inf. Sci. Eng. **16**(1), 1–8 (2016)
19. Radkowski, R., Garrett, T., Ingebrand, J., Wehr, D.: Trackingexpert—a versatile tracking toolbox for augmented reality. In: IDETC/CIE 2016, the ASME 2016 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, Charlotte, NC (2016)
20. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: 2009 IEEE International Symposium on Parallel Distributed Processing, pp. 1–10 (2009)
21. Singh, D.P., Joshi, I., Choudhary, J.: Survey of GPU based sorting algorithms. Int. J. Parallel Program. (2017). https://doi.org/10.1007/s10766-017-0502-5
22. Zhou, K., Hou, Q., Wang, R., Guo, B.: Real-time KD-tree construction on graphics hardware. ACM Trans. Graph. **27**(5), 126:1–126:11 (2008)