2016 23rd International Conference on Pattern Recognition (ICPR)
Cancún Center, Cancún, México, December 4-8, 2016

# GPU-Accelerated Descriptor Extraction Process for 3D Registration in Augmented Reality

Timothy Garrett, Rafael Radkowski
Virtual Reality Applications Center
Iowa State University
Ames, Iowa, USA
Email: garrettt@iastate.edu

Jeremy Sheaffer
Department of Computer Science
Iowa State University
Ames, Iowa, USA

***Abstract*—Augmented Reality (AR) is a type of human-computer interaction that overlays virtual information on a user's natural visual perception of the environment. Determining where to place these virtual objects so they appear a part of the environment requires real-time *tracking*. One type of tracking uses point clouds, processed from commodity depth sensors, such as the Microsoft Kinect. In this format, tracking becomes a 3D registration problem which can be quickly solved using local registration methods; however, these techniques require a sufficient overlap between the object-to-track, and the sampled object points from the range camera. 3D descriptors can be used to provide this initial overlap, but require significant computational resources.**

**This work extends previous research towards real-time descriptor computation on large point sets. To do so, the 3D descriptor extraction process is broken down into fundamental algorithmic steps. These steps are optimized by computing them on the GPU using NVIDIA's CUDA framework. Results of an experiment indicate the opportunity for large speed-ups in the most computationally intensive portions of descriptor extraction.**

## I. Introduction

Augmented Reality (AR) is a type of human-computer interaction in which virtual information is seemingly integrated into the real world. It does so by overlaying the information on a user's natural visual perception of the environment [1]. This information is presented in a context-sensitive manner that is appropriate for a specific task, and provides numerous opportunities to improve how users perform tasks.

Object tracking refers to the techniques enabling the real-time calculation of an object's pose relative to a global reference coordinate system. One type of tracking processes images from a commodity range camera (such as the Microsoft Kinect), which provides distance information for each image pixel. Using the optical properties of the camera, range images can be transformed into point clouds. Exploiting this format, tracking becomes a 3D registration problem; one of estimating the rigid transformation that best aligns an a priori point cloud *reference model* with point cloud data obtained in real-time from the environment (*environment model*).

Local point cloud registration approaches, such as variants of the Iterative Closest Points (ICP) algorithm, enable real-time tracking on commodity hardware [2], [3], [4], [5], [6], [7]. However, ICP relies on a sufficient overlap of the reference model and the environment model, and is best suited for frame-to-frame tracking with small changes between frames [8], [9]. A lack of sufficient overlap introduces the opportunity to trap the model at a local minimum during the registration process [10], [11].

3D descriptors can be used for the initial alignment, however, *descriptor extraction* (computation) for one point cloud from a single range image is generally intractable [12], [13]. Despite the computational burden of extracting descriptors, few efforts have been undertaken to extend the extraction process to the GPU. While many 3D descriptors are encoded from information contained in a *Local Reference Frame* (LRF) and are thus viable candidates for task parallelism, many are not trivially approached due to scattered memory access patterns.

In this work, we extend the work in [14] to perform descriptor extraction on the GPU toward real-time 3D registration. The descriptor extraction process is broken down into three elementary steps (k-d tree generation, normal estimation, and descriptor extraction). In this work, we focus on the first and last respective steps and provide the following contributions:

1) A top-down point cloud-based k-d tree construction method on the GPU
2) A pared implementation of the Fast Point Feature Histogram (FPFH) on the GPU, using cached nearest neighbors and demonstrating significant speed-ups compared to its CPU counterpart.

## II. Related Work and Background

The descriptor extraction process extended herein processes an unorganized Point Cloud $P$ (a set of 3D points, such as those rendered in Figure 1) into a set of FPFH descriptors in three elementary computation steps. Figure 2 provides an overview of these steps and characterizes the percentage of total computation time required for the point clouds in Figure 1. First, the points are organized into a k-d tree, enabling fast $O(\log n)$ searches (Section II-A). Second, the normal estimation step generates a normal for each point in $P$. Finally, the descriptor extraction step (containing both Simplified Point Feature Histogram [SPFH] Extraction and FPFH Extraction) uses the points, normals, and local point neighborhoods to establish LRFs and encode descriptors (Section II-B). As shown, the SPFH extraction takes the majority share of time in the sequential descriptor extraction process, and thus stands to benefit the most from GPU optimization.

In this work, we focus on both k-d tree construction and descriptor extraction steps. Thus, this section covers work related in the area of 3D descriptor extraction and k-d tree construction on the GPU.

### A. K-d tree construction on the GPU

A k-d tree is well-known balanced binary tree used for accelerating searches performed on multidimensional data [15]. In one type of k-d tree, each internal node (non-leaf) represents both a container for data (e.g. triangles, primitives, points, etc.), and an axis-aligned splitting plane, spatially separating the regions of its two children. In this work, we focus on k-d trees in which both internal nodes and leaf nodes each contain a single 3D point.

Sequential k-d tree construction with 3D points generally consists of two parts that are recursively repeated. First, data are sorted on the selected dimension. Secondly, the data are split using the *median element* (which is not strictly the median of the data, but the middle element). All nodes less than that element for the current dimension are passed to build a subtree with the root node located at the median element's left child, and likewise, those with a greater value are passed to build a subtree at the median element's right child. Due to the scattered memory access patterns inherent in traversing graphs (by chasing pointers) on the CPU, k-d tree construction isn't trivially implemented on the GPU.

On GPUs, k-d trees have been implemented for several fast or interactive ray tracing approaches [16], [17], [18], [19], [20]. This method allows passing rays into the root node, and rapidly searches for intersections with primitives objects (such as triangles) in the scene at leaf nodes. In contrast to this work, we develop a k-d tree for point clouds that differs in several fundamental ways: the type of data contained in a node; the lack of a heuristic function at each internal node (such as the Surface Area Heuristic); and the storage of data at each splitting plane, allowing nearest neighbor searches at each node traversed.

Other work implements k-d trees for computing particle interactions, relaxing the binary restriction of the data structure [21]. This work extends the k-d tree to support searching in cells, and is therefore, similar in design to an oct-tree. Whereas, our approach maintains the balanced binary tree structure of the k-d tree with a single splitting plane for each inner node.

Most similar to this work is that of Gieseke et al. [22], who develop so-called *buffer kd trees* to perform nearest neighbor queries on a large amount of data. While their speed-ups are significant in an experiment for large clouds, they are limited to single nearest neighbor search results, whereas this work requires K-nearest neighbor searches.

### B. 3D Descriptor Extraction on the GPU

Although the descriptor extraction process is computationally burdensome, few efforts have been undertaken to parallelize the process on the GPU. At this time, the Point Cloud

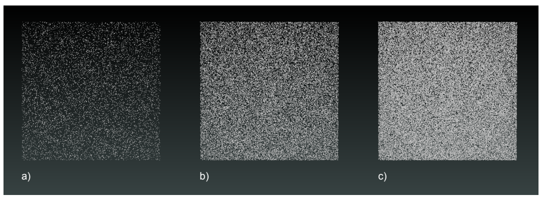

Fig. 1. Planar point cloud rendering with a) 10,000 points b) 50,000 points and c) 100,000 points

Library (PCL) version 1.8.0 [23] contains a limited number 3D descriptor extraction methods on the GPU, including a version of Point Pair Features [24], and the Fast Point Feature Histogram [25]. While the work contained here develops a GPU version of the FPFH, it differs in two main respects. First, we extract a descriptor using one radius only (thus pared), computing a fraction of the total descriptors. Secondly, we cache local nearest neighbors in a prior step to avoid redundant, time-consuming computations.

Recently, Palossi et al. [13] introduced a GPU version of the SHOT descriptor extractor for real-time speeds. However, results were limited to point clouds containing 10,000 points, which is roughly 3% of the total number of points in a single Microsoft Kinect camera frame.

## III. GPU OPTIMIZATION

This section describes the GPU optimizations developed in this work, aimed at real-time point cloud registration in 3D. To this end, GPU optimizations are implemented using NVIDIA's *Compute Unified Device Architecture* (CUDA) framework [26]. CUDA serves as an extension of the C/C++ programming languages, in which the GPU is considered a co-processor to the CPU, and runs a serial program called a *kernel* massively in parallel. This framework is capable of executing thousands of *threads* in parallel, which are organized in *blocks*.

### A. K-d tree

Construction of the k-d tree on the GPU is based on a couple key observations. First, the GPU co-processor obtains fast speeds when branch divergence is minimized, since all threads in the warp perform instructions in a SIMD (Simultaneous Instruction Multiple Data) manner. Secondly, fast speeds are obtained when memory access is coalesced. With this in mind, rather than designing an algorithm around random memory access by one thread, we develop an algorithm, that from its input parameters, can ask itself during its computation if it is the thread that should process the median element. And if the thread determines it should process the median element, it can place the point in the correct position in the tree. This concept is referred to as a *median splitting element*, and is formulated in Appendix A.

Similar to the sequential CPU implementation, the GPU implementation of the k-d tree construction relies on repeatedly sorting the points by a dimension (Algorithm 1), and then splitting the data with a median element and adding it to the tree (Algorithm 2). The sorting and splitting operations are called sequentially for each depth of the k-d tree, and is thus limited to $\lceil \log n \rceil$ calls.

The segmented sorting sequentially runs several parallel kernels, and is documented in Algorithm 1. First, a map operation fills an array with its indices, which is used later in the sorting process. To coalesce memory accesses, each point is split into a separate data set by dimension. This is followed by a map operation that creates a list of predicates, which is used in an inclusive scan [27] to identify the *sorting group* that each index belongs in. The sorting group segments the data so sorting is confined to those points between the median splitting elements. To do so, the range of the data is computed, and subsequently linearly transformed in the *range widening* step (Line 8). At this point, the sorting is performing using a fast key-value radix sort; using the data as keys to be sorted, and the original point indices that come along for the ride [28]. The result is that the indices array contains at index $i$ the location for where the point at $i$ should be moved, and a move is easily performed with a mapping.

Similar to sorting, building of the tree at a particular depth relies on the ability for the thread $i$ to determine if the point located at index $i$ should be the median and therefore split the data and insert the point into the tree (Algorithm 2). To do so, the algorithm must contend with two special cases that lie outside the assumptions formulated in Appendix A. The first case is the trivial case where $c = 0$, and hence the element at index 0 will always report it is a median splitting element. This is easily handled on the final depth of the tree, since the algorithm is formulated to only choose the element at index 0 when $\alpha < 1$. The second special case is when $\alpha < 1$ which occurs on the largest depth, where each node is thus a median splitting element. In this situation, the kernel can calculate the median and median splitting elements for the previous depth, and set elements which are neither to the tree. The elements at the final level of the tree are guaranteed to be within 1 index of their parent, since median splitting elements at the depth above are $2\alpha < 2 * 1 = 2 \Rightarrow \lfloor 2\alpha \rfloor = 1$.

The tree is stored in a linear array of points, with the root stored at index 0, its left child at index 1, and right child at index 2. Each left and right child node of a node at position $i$ is stored in positions $2i + 1$ and $2i + 2$, respectively.

## B. Descriptor Extraction

Similar to the CPU implementation, the GPU descriptor extraction is executed in two steps. First, an SPFH descriptor is extracted for each point in the cloud. Secondly, FPFH descriptors are extracted by adding Euclidean distance-weighted SPFH descriptors from points in the local neighborhood.

*1) SPFH Extraction:* The parallel algorithm used for SPFH extraction is shown in Algorithm 3. In the sequential algorithm, each point pair inside point's local neighborhood is processed in a loop. In the parallel algorithm, however, we launch a thread for each possible point-pair inside every local neighborhood. Thus, given a point cloud of $n$ points, and a maximum neighborhood size of $k$, we launch $nk^2$ threads. To prevent the influence of each point-pair twice in the sequential algorithm (e.g. once for (i,j), and once for (j,i)), we restrict the condition that $i < j$, and thus only process the point-pair once. Similarly, this restriction is extended to the parallel algorithm by enforcing the same constraint on the thread index. Since each thread calculates its own point-pair contribution to the SPFH at $i$, one atomic add operation is performed to for each angular variation (Line 3) to ensure race conditions don't occur.

*2) FPFH Extraction:* The goal of the FPFH Extraction step is to add the weighted SPFHs in the local neighborhood to the SPFH at index $i$. We leverage the fast access of shared memory

```
Algorithm 1: Sort_Dimension($P, n, depth$)
Input: A point cloud $P = \{\mathbf{p}_i\}$, the length of the point
       cloud $n$, the depth of the tree $depth$
Output: A point cloud sorted by a single
1  $indices \leftarrow$ Parallel map: element $i$ gets unique index $i$;
2  $(x, y, z, w) \leftarrow$ Parallel scatter: split each point
   $\mathbf{p}_i = \{x_i, y_i, z_i, w_i\}$ into components;
3  $predicates \leftarrow$ Parallel map: element $i$ to 1 iff $i$ is a
   median splitting element, and 0 otherwise;
4  $keys \leftarrow$ Parallel inclusive scan on $predicates$;
5  $buffer \leftarrow x, y$, or $z$ (by depth);
6  $(max\_element, min\_element) \leftarrow$ Parallel max-min
   scan on $buffer$;
7  $range \leftarrow max\_element - min\_element$;
8  $buffer \leftarrow$ Parallel map: Range widening on $buffer$
   using $predicates$ and $keys$;
9  $(buffer, indices) \leftarrow$ Parallel key-value radix sort, where
   key=$buffer$ and value=$indices$;
10 $P \leftarrow$ Parallel gather of $\{x_i, y_i, z_i, w_i\}$ into their new
   location specified by $indices$;
```

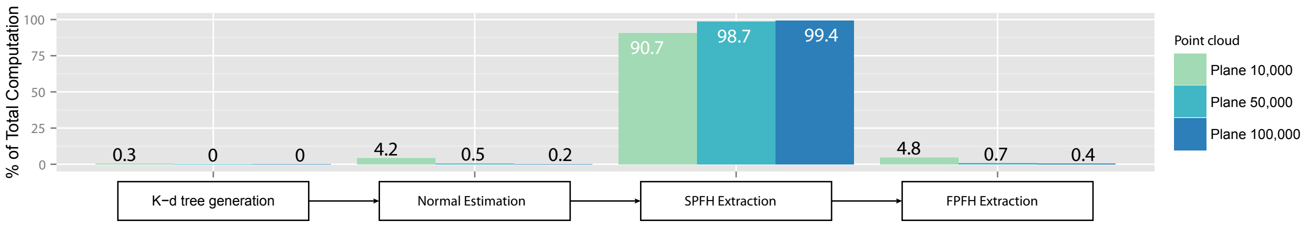

Fig. 2. Percentage of total run-time for each logical step in the descriptor extraction process

**Algorithm 2:** Build_Depth_Kernel($P, n, tree, depth$)

```
Input: A point cloud $P = \{\mathbf{p}_i\}$, the length of the point cloud $n$, the k-d tree $tree$, the depth of the tree $depth$
Output: A k-d tree stored on the GPU
1  $max\_depth \leftarrow \log_2 n$;
2  if $depth == \lfloor maxDepth \rfloor$ then
3      $depth \leftarrow depth - 1$;
4      $special\_case \leftarrow true$;
5  end
6  $\alpha \leftarrow \frac{n}{2^{depth+1}}$;
7  if $special\_case$ then
8      if $i == 0 \vee \lceil \frac{i}{\alpha} \rceil \geq \frac{i+1}{\alpha}$ then
9          Choose left or right as parent;
10         Set element $i$ in the tree;
11     end
12 else
13     if $\lceil \frac{i}{\alpha} \rceil < \frac{i+1}{\alpha} \wedge \lceil \frac{i}{\alpha} \rceil$ is odd then
14         Set element $i$ in the tree;
15     end
16 end
```

among threads in a block to do so. The FPFH extraction step is launched in a kernel with a block for each point in the point cloud, and a thread for each element in the local neighborhood. As long as the maximum number of local neighbors is fixed, and the shared memory is large enough to hold one floating point number for each local neighbor, this approach can be used.

Each thread is used to retrieve an independent SPFH and weight its bins using Euclidean distance. When that is complete, all threads in the block are synchronized. Then, for each bin of the histogram, the weighted values are loaded into shared memory, and a Scan operation with the addition operator sums the neighborhood values into the bin at thread index 0. Finally, the FPFH descriptor is written to memory.

## IV. Software and Hardware

The proposed algorithms are implemented as two C++ libraries and tested in an experiment. The CPU library uses the Eigen library [29] to align point objects to 16-byte sizes for Streaming SIMD Extensions (SSE), to implement an aligned allocator for the point cloud class, and for performing fast singular value decompositions for normal estimations. The parallel library uses NVIDIA's CUDA version 7.5.18, and is built supporting the Maxwell architecture. The Thrust Library that accompanies CUDA is used for the radix sort in Algorithm 1. Points and descriptors are aligned to 16-byte boundaries through the use of macros.

The experiment was run on a consumer-grade HP Envy laptop running Microsoft Windows 8.1 x64. The laptop has an Intel Core i7-4900MQ CPU and 8 GB of RAM. The GPU is a mid-range NVIDIA GeForce 840m (28 nm) based on the Maxwell architecture (GM108 chip), with 2 GB dedicated memory, and 2 GB shared memory.

**Algorithm 3:** Compute_SPFH_Kernel ($P$, $N$, $n$, $Neighborhood$, $kNeigh$)

```
Input: A point cloud $P = \{\mathbf{p}_i\}$, A point cloud of normals $N = \{\mathbf{n}_i\}$, the length of the point cloud $n$, the local neighborhoods for each point $Neighborhood$, and the size of each local neighborhood $kNeigh$
Output: An SPFH descriptor for each point $SPFH$
1  $i \leftarrow$ Point Index;
2  $j \leftarrow$ First neighbor index;
3  $k \leftarrow$ Second neighbor index;
4  $neigh\_size \leftarrow kNeigh[i]$;
5  if $j \geq neigh\_size \vee k \geq neigh\_size \vee j \geq k$ then
6      return;
7  end
8  $\mathbf{p}_j \leftarrow GetNeighbor(Neighborhood, i, j)$;
9  $\mathbf{n}_j \leftarrow GetNormal(N, j)$;
10 $\mathbf{p}_k \leftarrow GetNeighbor(Neighborhood, i, k)$;
11 $\mathbf{n}_k \leftarrow GetNormal(N, k)$;
12 $\mathbf{u} \leftarrow \mathbf{n}_j$;
13 $\mathbf{p}_{kj} \leftarrow \mathbf{p}_k - \mathbf{p}_j$;
14 $\mathbf{v} \leftarrow \mathbf{p}_{kj} \times \mathbf{u}$;
15 $\mathbf{w} \leftarrow \mathbf{u} \times \mathbf{v}$;
16 Normalize($\mathbf{u}, \mathbf{v}, \mathbf{w}$);
17 $impact \leftarrow$ CalculateBinsAndImpact($\alpha_1, \alpha_2, \alpha_3$);
18 AtomicAdd( $SPFH[i], impact$ );
```

## V. Results and Discussion

Preliminary results of the descriptor extraction experiments on the CPU and GPU are displayed in Figures 3, 4 and 5. On the x-axis of each chart is the computation step of the descriptor extraction process, and on the y-axis is the average computation run-time in milliseconds for a sample size of 100.

As you can see in Figure 3, the time needed to compute the k-d tree, estimate normals, extract SPFH descriptors, and extract FPFH descriptors for the smallest (10,000-point) point cloud on the CPU is not real-time. As evidenced by the chart, the major bottleneck is the SPFH Extraction step, in the best case taking 3.4 seconds.

Figures 4 and 5 show the preliminary results of descriptor extraction on the GPU on varying sized plane models and 100,000 point primitive models, respectively. The k-d tree generation step shows an improvement in speed for the GPU for both the 50,000 point plane and the 100,000 point plane, but longer run-time for the 10,000 point cloud. These results are consistent with other approaches that build k-d trees on the GPU, such as Zhou et al. [30], who report a 3x - 7x speedup in their CUDA implementation over the sequential implementation.

Most striking is the reduction in the time of the SPFH extraction process, where the CPU version runs in the best case

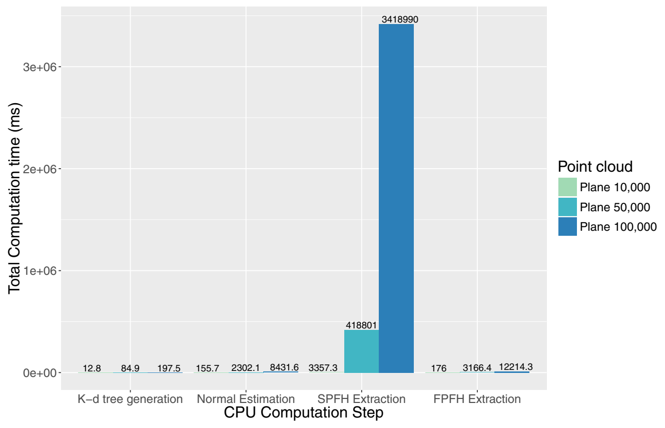

Fig. 3. Average CPU extraction time on the planar point cloud models

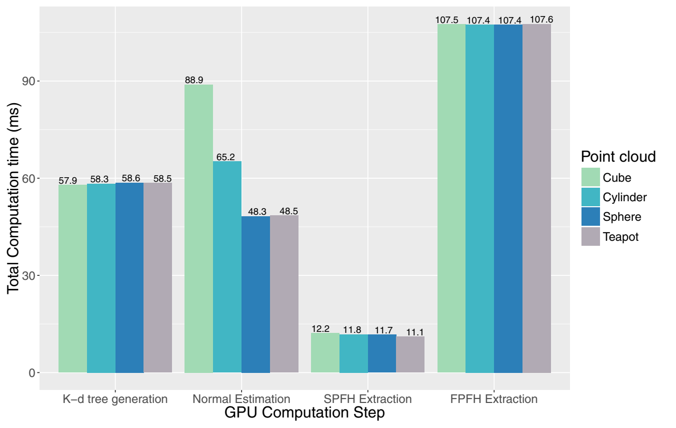

Fig. 5. Average GPU extraction time on primitive object point clouds containing 100,000 points

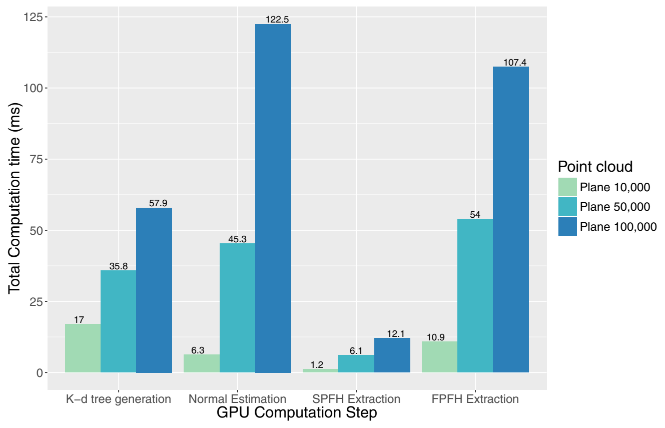

Fig. 4. Average GPU extraction time on the planar point cloud models

nearly 2800 times longer than the GPU algorithm for the plane object. Some of this speed-up can be attributed to the lack of a nearest neighbor search, since the local neighborhoods are cached in the normal estimation stage. In comparison, the FPFH extraction takes longer to compute than the SPFH extraction. One of the reasons for this could potentially be the $O(k \log n)$ processing contained within each kernel for the scan operation that reduces each dimension of the descriptor.

## VI. Conclusion and Future Work

In conclusion, preliminary results on primitive object point clouds indicate that dramatic reductions in processing time for the descriptor extraction process can be obtained by performing it on the GPU. In particular the SPFH and FPFH steps indicate that the processing time of point clouds with 100,000 points can be reduced from minutes to milliseconds.

There is much future work ahead. First, GPU memory needs to be better utilized for faster reads and writes. Texture memory should be investigated for use with unorganized point clouds, to take advantage of global access and caching. Similarly, shared memory use should be investigated further as an alternative to some much slower global reads and writes.

Threads and block layouts for kernel execution needs to be more intelligently selected. Unless otherwise noted, each kernel was executed using the maximum of 1024 threads to each block. However, there may be more optimal block layouts that produce faster descriptor extraction times.

More point cloud models need to be tested to determine if the timing results are similar in each step to those obtained for the primitive point cloud models. We will continue this investigation larger, room-sized clouds obtained from commodity depth cameras.

## References

[1] R. Azuma, “A survey of augmented reality,” *Presence*, vol. 4, no. August, pp. 355–385, 1997.

[2] P. J. Besl and N. D. McKay, “Method for registration of 3-d shapes,” in *Proc. SPIE*, vol. 1611, Boston, MA, April 1992, pp. 586–606.

[3] Y. Chen and G. Medioni, “Object modelling by registration of multiple range images,” *Image and Vision Computing*, vol. 10, no. 3, pp. 145–155, Apr. 1992.

[4] Z. Zhang, “Iterative point matching for registration of free-form curves and surfaces,” *International journal of computer vision*, vol. 152, no. 1994, pp. 119–152, 1994.

[5] A. Segal, D. Haehnel, and S. Thrun, “Generalized-ICP,” in *Robotics: Science and Systems*, Seattle, WA, June 2009.

[6] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon, “Kinectfusion: Real-time dense surface mapping and tracking,” in *2011 10th IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, Basel, Oct 2011, pp. 127–136.

[7] T. Garrett, S. Debernardis, R. Radkowski, C. K. Chang, M. Fiorentino, A. E. Uva, and J. Oliver, “Rigid object tracking algorithms for low-cost ar devices,” in *Proceedings of the ASME 2014 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, Buffalo, NY, August 2014.

[8] R. Bergevin, M. Soucy, H. Gagnon, and D. Laurendeau, “Towards a general multi-view registration technique,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, no. 5, pp. 540–547, May 1996.

[9] K. Pulli, “Multiview registration for large data sets,” in *Second International Conference on 3-D Digital Imaging and Modeling, Proceedings*, Ottawa, Ont., October 1999, pp. 160–168.

[10] G. Sharp, S.-W. Lee, and D. Wehe, “Icp registration using invariant features,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 1, pp. 90–102, Jan 2002.

[11] J. Santamara, O. Cordón, and S. Damas, "A comparative study of state-of-the-art evolutionary image registration methods for 3d modeling," *Computer Vision and Image Understanding*, vol. 115, no. 9, pp. 1340 – 1354, 2011.

[12] X. Li and I. Guskov, "Multiscale features for approximate alignment of point-based surfaces." in *Symposium on geometry processing*, vol. 255. Citeseer, 2005, pp. 217–226.

[13] D. Palossi, F. Tombari, S. Salti, M. Ruggiero, L. Di Stefano, and L. Benini, "Gpu-shot: Parallel optimization for real-time 3d local description," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2013 IEEE Conference on*, Portland, OR, June 2013, pp. 584–591.

[14] T. Garrett, "An initial matching and mapping for dense 3d object tracking in augmented reality applications," Master's thesis, Iowa State University, 2015.

[15] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep 1975.

[16] T. Foley and J. Sugerman, "Kd-tree acceleration structures for a gpu raytracer," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ser. HWWS '05, 2005, pp. 15–22.

[17] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree gpu raytracing," in *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, ser. I3D '07, 2007, pp. 167–174.

[18] S. Popov, J. Gnther, H.-P. Seidel, and P. Slusallek, "Stackless kd-tree traversal for high performance gpu ray tracing," *Computer Graphics Forum*, vol. 26, no. 3, pp. 415–424, 2007.

[19] P. Danilewski, S. Popov, and P. Slusallek, "Binned sah kd-tree construction on a gpu," *Saarland University*, pp. 1–15, 2010.

[20] Z. Wu, F. Zhao, and X. Liu, "Sah kd-tree construction on gpu," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ser. HPG '11, 2011, pp. 71–78.

[21] N. Nakasato, "Implementation of a parallel tree method on a gpu," *Journal of Computational Science*, vol. 3, no. 3, pp. 132 – 141, 2012.

[22] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel, "Buffer kd trees: processing massive nearest neighbor queries on gpus," in *Proceedings of The 31st International Conference on Machine Learning*, 2014, pp. 172–180.

[23] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[24] B. Drost and S. Ilic, "3d object detection and localization using multimodal point pair features," in *3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT), 2012 Second International Conference on*. IEEE, 2012, pp. 9–16.

[25] R. B. Rusu, N. Blodow, and M. Beetz, "Fast point feature histograms (fpfh) for 3d registration," in *IEEE International Conference on Robotics and Automation*. Kobe: IEEE, May 2009, pp. 3212–3217.

[26] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[27] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," in *Graphics hardware*, vol. 2007, 2007, pp. 97–106.

[28] D. Merrill and A. Grimshaw, "Revisiting sorting for gpgpu stream architectures," University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Tech. Rep. CS2010-03, 2010.

[29] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[30] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," Microsoft Research, Tech. Rep. MSR-TR-2008-52, April 2008.

[31] S. Henderson and S. Feiner, "Exploring the benefits of augmented reality documentation for maintenance and repair." *IEEE transactions on visualization and computer graphics*, vol. 17, no. 10, pp. 1355–1368, October 2011.

[32] B. Schwerdtfeger and G. Klinker, "Supporting order picking with augmented reality," in *Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*, ser. ISMAR '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 91–94.

[33] D. Aiger, N. J. Mitra, and D. Cohen-Or, "4-points congruent sets for robust pairwise surface registration," in *ACM SIGGRAPH 2008 Papers*, ser. SIGGRAPH '08. New York, NY, USA: ACM, 2008, pp. 85:1–85:10.

[34] N. Mellado, D. Aiger, and N. J. Mitra, "Super 4pcs fast global pointcloud registration via smart indexing," *Computer Graphics Forum*, vol. 33, no. 5, pp. 205–215, 2014. [Online]. Available: http://dx.doi.org/10.1111/cgf.12446

[35] Y. Guo, M. Bennamoun, F. Sohel, M. Lu, and J. Wan, "3d object recognition in cluttered scenes with local surface features: A survey," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 36, no. 11, pp. 2270–2287, Nov 2014.

[36] T. Hastie, R. Tibshirani, and J. Friedman, "The elements of statistical learning," 2009.

[37] M. Liu, F. Pomerleau, F. Colas, and R. Siegwart, "Normal estimation for pointcloud using gpu based sparse tensor voting," in *2012 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, Dec 2012, pp. 91–96.

[38] N. Bosner and J. L. Barlow, "Block and parallel versions of one-sided bidiagonalization," *SIAM Journal on Matrix Analysis and Applications*, vol. 29, no. 3, pp. 927–953, 2007.

[39] M. Andrecut, "Parallel gpu implementation of iterative pca algorithms," *Journal of Computational Biology*, vol. 16, no. 11, pp. 1593–1599, 2009.

[40] V. H. Andez and J. e. R. An, "A robust and efficient parallel svd solver based on restarted lanczos bidiagonalization," *Electronic Transactions on Numerical Analysis*, vol. 31, pp. 68–85, 2008.

[41] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, "Cula: hybrid gpu accelerated linear algebra routines," in *Proc. SPIE*, vol. 7705, Orlando, FL, April 2010, pp. 770 502–770 502–7.

## APPENDIX

The index of the median element $m_{idx}$ can be selected as the middle of $n$ elements using the floor function:

$$m_{idx} = \left\lfloor \frac{n}{2} \right\rfloor \tag{1}$$

Repeated application of this median selection yields the observation that at depth $d$, for $\alpha \geq 1$, the index $i$ of the median element can be expressed as one particular assignment of $c$ by

$$i = \lfloor c\alpha \rfloor \tag{2}$$

such that $c \in \mathbb{Z}$, and

$$\alpha = \frac{n}{2^{d+1}} \tag{3}$$

**Definition 1.** *Let $i$ be a median splitting element at depth $d$ if at some depth $d_i \leq d$, $i$ is the index of a median element.*

Since by definition $\forall \alpha, \alpha \geq 1$, it follows

$$i = \lfloor c\alpha \rfloor \iff i \leq c\alpha < i + i \tag{4}$$

$$\frac{i}{\alpha} \leq c < \frac{i+1}{\alpha} \tag{5}$$

$$c \in \left[ \frac{i}{\alpha}, \frac{i+1}{\alpha} \right) \tag{6}$$

**Theorem 1.** *There is at most one non-zero positive integer value $c$, such that*

$$c \in \left[ \frac{i}{\alpha}, \frac{i+1}{\alpha} \right)$$

*Proof.* Suppose, for the sake of contradiction, that both $c$ and $c+1$ were contained in the interval $[i/\alpha, (i+1)/\alpha)$. Then

$$\frac{i}{\alpha} \leq c + 1 < \frac{i+1}{\alpha} \tag{7}$$

$$\frac{i-\alpha}{\alpha} \leq c < \frac{i+1-\alpha}{\alpha} \tag{8}$$

Since $\alpha \geq 1$,

$$c < \frac{i+1-\alpha}{\alpha} \leq \frac{i}{\alpha} \tag{9}$$

which is a contradiction.

□