Performance comparison between state-of-the-art point-cloud based collision detection approaches on the CPU and GPU

Johannes Schauer (johannes.schauer@uni-wuerzburg.de) *
Janusz Bedkowski (januszbedkowski@gmail.com) **
Karol Majek (karolmajek@gmail.com) **
Andreas Nüchter (andreas.nuechter@uni-wuerzburg.de) *

* *Informatics VII: Robotics and Telematics, Julius-Maximilians-University Würzburg, Am Hubland, D-97074 Würzburg, Germany*
** *Institute of Mathematical Machines, ul. Krzywickiego 34, 02-078 Warsaw, Poland*

Abstract:
We present two fundamentally different approaches to detect collisions between two point clouds and compare their performance on multiple datasets. A collision between points happens if they are closer to each other than a given threshold radius. One approach utilizes the main CPU with a k-d tree datastructure to efficiently carry out fixed range searches around points in 3D while the other mainly executes on a GPU using a regular grid decomposition technique implemented in the CUDA framework. We will show how massively parallel 3D range searches on a grid based datastructure on a GPU performs similarly well as a tree based approach on the CPU with orders of magnitude less parallelization. We also show how each method scales with varying input sizes and how they perform differently well depending on the spatial structure of the input data.

Keywords: k-d tree, CUDA, parallel algorithms, 3D point clouds, regular grid decomposition

1. INTRODUCTION

Advanced automation and telematics focuses on computerization of processes. Factory management, the inspection of tunnels and mines as well as the automation of machinery need reliable, flexible and fast environment perception and analysis methods. Three-dimensional perception of environments emerged in robotics applications. 3D scanners such as the projection based kinect sensor or the time-of-flight sensor kinect2 are available as consumer products and professional high-end 3D laser scanners are state-of-the-art in surveying. Combined with recent development in the area of simultaneous localization and mapping, these sensor systems deliver precise large scale 3D point clouds of environments.

This paper focuses on the analysis of 3D point clouds obtained in different environments. Given a 3D point cloud of the environment, a 3D point cloud of a model, and some trajectory, we move the model through the environment and calculate collisions. This means, we calculate where the model will interact with the environment. The problem is highly parallelizable, as all points can be treated in parallel. The paper compares algorithms using a search tree, namely a k-d tree, running on a CPU with OpenMP with a GPU implementation that exploits a regular grid decomposition.

Throughout the paper, we use a couple of realistically-scaled real-world data sets to test the performance of our collision detection methods. The first two data sets have been acquired with a mobile mapping system based on a FARO Focus3D sensor in factories of the Volkswagen AG in Hannover and Wolfsburg. Aim of the research is to support model switching by testing if new car models fit the production line (Elseberg et al., 2014a). The third data set has been acquired with a Riegl VZ400 laser scanner in the El Teniente mine, the largest underground copper mine in Chile. The fourth data set was acquired by a LYNX mobile mapping system in a tunnel for trains. For the latter two applications, telematics methods should be used for testing if equipment can pass the environment.

2. RELATED WORK

Collision detection, which is also called interference detection or intersection searching, is a well studied topic in computer graphics (Jiménez et al., 2001; Lin and Gottschalk, 1998; Bender et al., 2014; Mainzer and Zachmann, 2014; Tang et al., 2011) because of its importance for dynamic computer animation and virtual reality applications (Tzafestas and Coiffet, 1996; Muñoz et al., 2014; Hummel et al., 2012). On the other hand, their work is lim-

⋆ List of whom we have to thank

ited to collision detection between geometric shapes and polygonal meshes whereas most sensor data is acquired as point clouds. While collision detection is also relevant for motion planning in the field of robotics, it is a less studied problem there. Collision detection between point clouds was for example researched by Klein and Zachmann (Klein and Zachmann, 2004) who use the implicit surface created by a point cloud to calculate intersections. Another example is the recent work by Hermann et al. (Hermann et al., 2014) who use voxels to check for spatial occupancy for robot motion planning.

Existing techniques make use of very similar approaches. One method is to apply a spatial hierarchical partitioning of the input geometry using octrees (Jung and Gupta, 1996; Fan et al., 2013), AABB-trees (Wang and Liu, 2014), BSP-trees (Ar et al., 2000) or k-d trees (Held et al., 1995). Other solutions apply regular partitioning using voxels (Garcia-Alonso et al., 1994; Hermann et al., 2014; Faas and Vance, 2011). The goal of any partitioning is to be able to quickly search and check only the relevant geometries in the same or neighboring cells. The CPU based method presented in this paper will make use of a hierarchical k-d tree for the environment in combination with a regular partitioning of the model into a grid of bounding spheres.

The k-d tree implementation bears similarities to R+-trees (Sellis et al., 1987) insofar it recalculates a new bounding box for each child node. In contrast to R+-trees, the k-d tree implementation presented here does not make efforts to create a balanced tree. In (Elseberg et al., 2012) our k-d tree implementation was benchmarked against three nearest-neighbor search libraries based on the k-d tree data structure: ANN (Mount and Arya, 2010), libnabo (Magnenat, 2014) and FLANN (Muja and Lowe, 2012) and came out amongst the fastest implementations.

GPU enabled collision detection algorithms are mainly used in computer graphics for ray tracing. The algorithms utilize GPUs using shader language programming, OpenCL or Nvidia CUDA. The first GPU ray tracer was using a uniform grid for acceleration and was implemented in shader language (Purcell et al., 2002). Stackless k-d tree packet GPU ray traversal implementation was introduced in Popov et al. (2007). In Zhou et al. (2008) an algorithm of constructing k-d trees using CUDA enabled GPUs is shown. To cope with large datasets a method for incremental construction of Bounding Volume Hierarchies (BVH) that incrementally constructs a BVH with quality comparable to the best surface area heuristic (SAH) (MacDonald and Booth, 1990) builders was introduced in Bittner et al. (2015).

In the context of nearest-neighbor search in 3D point datasets k-d tree (Qiu et al., 2009) and regular grid decomposition are used (Bedkowski et al., 2012). The performance of these two data structures is compared in Bedkowski et al. (2013).

3. DESIGN AND IMPLEMENTATION

In the following section we describe the design and implementation of the two methods we present in this paper. The first method is based on a k-d tree search which runs entirely on the CPU. The second method is based on a regular grid decomposition and runs on the GPU. Both methods take the following inputs:

- a set of points making the environment E
- a set of points making the model M
- a set of 6DOF transformations making a trajectory T and
- a search radius r

Both methods will find all points in the environment E which fall within a certain radius r around any point of the model M at any point on its trajectory T. The CPU method works with double precision (eight bytes) while the GPU based method is limited to floating point precision (four bytes).

3.1 3dtk k-d tree

The collision detection method using a k-d tree was described in detail in Schauer and Nüchter (2015). We are using the method called kd-CD-simple from that publication in these benchmarks. Using the nomenclature for E, M, T and r from above, the basic algorithm is as follows:

$K \leftarrow create_kd_tree(E)$
$c \leftarrow [false \forall p \in E]$
for all $t \in T$ **do**
  for all $m \in M$ **do**
    $m' \leftarrow transform(m, t)$
    $s \leftarrow rangesearch(K, m', r)$
    $update_colliding(s, c)$
  end for
end for

In other words: Create a k-d tree K from the environment and create an array c which stores for each point in the environment whether it collides with the model at any point on the trajectory or not. Then do the following: For each 6 DOF transformation t on the trajectory and for each point m of the model, apply t to m, producing m' and find all points s in K that lie within radius r around m' and update c to set these colliding points to $true$. Since searches in the k-d tree are a read-only operation and since even updating the same point in c from different threads does not lead to any race conditions (because values are set to true irrespective of their former value), the algorithm is embarrassingly parallel. In other words, if one could run $|T| * |M|$ threads in parallel, then the whole algorithm would only take as long as the longest search in the k-d tree would take.

3.2 regular grid decomposition

The GPU accelerated implementation of collision detection is based on regular grid decomposition and GPGPU accelerated nearest-neighbor search. The collision detection algorithm, using the nomenclature from above with E as the environment, M as the model, T as the trajectory and r as the search radius is is follows:

$R_e \leftarrow create_RGD(E)$
$S = \emptyset$
$c \leftarrow [false \forall p \in E]$
for all $t \in T$ **do**
  $M' \leftarrow transform_in_parallel(M, t)$

$S' = find_corresponding_cells_in_RGD(M', E)$
if $S' \neq S$ **then**
$sync_data_with_GPU_memory(S')$
$R'_e \leftarrow create_RGD(S')$
$S = S'$
end if
for all $m \in M$ **do** ▷ In parallel
$s \leftarrow rangesearch(R'_e, m', r)$
$update_colliding(s, c)$
end for
end for

The core concept of the algorithm is to use regular grid decomposition R_e to split large environment point cloud E into smaller cells and then use only the cells which intersect with the bounding box of the transformed model for collision detection at each point of the trajectory T. For each point of the trajectory, all points of the model M are transformed (in parallel) using t, producing M'. The axis aligned bounding box of M' is compared to R_e to find all cells S' containing possibly colliding points, ie. all points in S' must be more than r away from the boundary of the axis aligned bounding box defined by S'. If there are new cells in S' compared to S, or some are no longer relevant, points are copied to or removed from GPU memory respectively. If the data in GPU memory was updated, for all points in S' a regular grid decomposition R'_e is carried out. After the data on the GPU has been made current, for all points m in M the range search is performed with radius r in R'_e. Finally the array c is updated and copied back to host memory. In this approach parallelism is limited by two factors, device memory and number of cores. All transformations are done theoretically in parallel, the serialization process is low level and provided by the device driver and depends on used device.

4. EXPERIMENTAL METHOLOGY

We tested our methods on expensive high end hardware as well as on standard end-user hardware. That way, we will be able to visualize the best-case performance and compare it with performance on more affordable setups.

4.1 CPU tests

The tests of the 3dtk k-d tree implementation have been carried out on two systems. We call the first system "e5-2630 v3" which is a modern desktop system with a 2.4 GHz 8 core processor and 32 GB of RAM. The second test system is dual CPU server system with two 2.8 GHz 10 core CPUs (for a total of 20 cores) and 256 GB of RAM. We call the second system "e5-2680 v2". Both systems are based on Intel Xeon processors and support Hyper-Threading with 16 and 40 threads, respectively. The operating system in both cases was Debian unstable with GCC 5.3.1 and Linux 4.3.5 on the amd64 architecture.

4.2 GPU tests

To test the GPU accelerated implementation 3 different Nvidia graphic cards were used. The first GPU is a Geforce Titan X with 3072 CUDA cores (base clock: 1000 MHz, boost: 1075 MHz) and 12 GB GDDR5 384-bit memory. The second one is a Geforce GTX980 with 2048 CUDA cores (base clock: 1126 MHz, boost: 1216 MHz) and 4 GB GDDR5 256-bit memory. The third one is Tesla K40 with 2880 CUDA cores (base clock: 745 MHz, boost: 810/875 MHz) and 12 GB GDDR5 384-bit memory. The GPU performance is tested with the first system "e5-2630 v3".

4.3 Datasets

We used four very heterogeneous datasets in our benchmarks. Table 1 shows a comparison of their most important characteristics. Each dataset is comprised of the pointcloud of the model, the pointcloud of the environment and the 6 DOF trajectory that the model takes through the environment (the first four columns in table 1). The trajectory is a sequence of transformation matrices describing the rotation and translation (but not scaling or shearing) of the model at each step. In all experiments we used a search radius of 10 cm. The shown timings are for the collision search only and do not include the fixed times per dataset that is needed to create the necessary initial datastructures like the k-d tree for the CPU based method or the regular grid decomposition of the environment for the GPU based method.

The "El Teniente" dataset was collected in a stop-and-go fashion in the El Teniente underground copper mine in Chile. The trajectory follows the path along which the scanner was moved in a closed loop. Due to the stop-and-go scanning method, the point density is highest around the positions where a scan was carried out. A synthetic point cloud of a 3D model of a front loader was moved through this dataset. The trajectory was produced by fitting a B-Spline through the individual scanning positions.

The "Hannover" and "Wolfsburg" datasets were collected in production facilities of the automotive company Volkswagen in their factories in Hannover and Wolfsburg, respectively. Both datasets were collected using continuous laser scanning on a mobile platform which was moved on the production conveyor (Elseberg et al., 2014b). The characteristics of these two point clouds are very similar because of the similar environment and the similar scanning methods. The main difference is, that the "Wolfsburg" dataset is much larger and produced the longest run times in our test due to its size, the high sampling rate along its trajectory as well as using a model with twice the amount of points. We used pointclouds extracted from actual CAD models of the Volkswagen Crafter and Tiguan car bodies for the "Hannover" and "Wolfsburg" datasets, respectively. The trajectory was retrieved from the scanner positions and transformed such that a realistic simulation of the movement of the car body along the production line is achieved.

The fourth dataset called "Train Tunnel" was recorded in a continuous fashion from a laser scanner on the back of a train (Schauer and Nüchter, 2014). The dataset contains a tunnel environment as well as through an open outdoor environment before and after the tunnel. The model moved through the environment was a manually scanned train wagon which we moved along a trajectory that allowed us to simulate a bogie size of 20 m of that train wagon,

Table 1. Datasets used for collision detection. The first column shows the name of the dataset, the second column shows the number of points in the environment, the third column the number of points in the model, the fourth column the number of discrete positions along the trajectory and the fifth and sixth column the number of colliding points on the CPU and GPU, respectively.

Name	#Environment	#Model	#Trajectory	#colliding CPU	#colliding GPU
El Teniente	806183400	100000	17795	35225149	35225399
Hannover	55872714	214489	17234	2495803	2495804
Wolfsburg	350109065	434700	398999	26089196	26089208
Train Tunnel	18919000	28622	19392	1627225	1627233

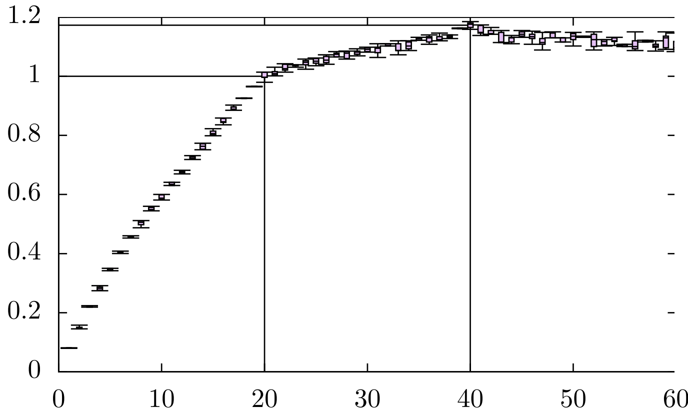

Fig. 1. Box-and-whisker plot of 3dtk runtime on the Hannover dataset by number of threads on the "e5-2680 v2" setup (20 cores). The x-axis shows the number of threads. The y-axis is scaled to show relative runtime compared to using 20 threads. Values indicate the multiple of the runtime per number of threads compared to 20 threads. Higher values mean faster computation. The runtime at 40 threads is close to 1.2 times the runtime with 20 threads.

leading to collision that could not have been detected with a structure gauge based method.

The last two columns of table 1 show a difference between the number of colliding points that were found with each method. An analysis showed that both methods produce at least 99.999 % common points given our datasets. The differing points were found to lie on the give search radius. The number of common points is higher for datasets where the input points are given with low precision values. This lets us conclude that the differences stem from floating point errors due to our differing algorithms as well as from the CPU method using double precision while the GPU method uses float precision. Both methods reliably produce the same set of points between different runs and are thus fully deterministic.

5. RESULTS

In this section we present benchmarks of our algorithms in varying setups. We first present results specific to our CPU based implementation, then GPU specific results and lastly compare the runtimes of our CPU and GPU implementations with each other.

5.1 CPU specific benchmarks

In an earlier publication (Schauer and Nüchter, 2015) we claimed that our k-d tree collision detection algorithm would scale completely linearly with increasing number of threads. In figure 1 we show proof of this claim. As the test system had 20 individual CPU cores, performance increases with the same slope until that number of threads. After that, performance increases with a less steep slope until 40 threads which can be explained by Intel Hyper-Threading which is able to boost performance by an additional 17.3 % compared to the 20 thread case. No further performance improvement is gained after 40 threads, which lets us conclude that the best CPU utilization is achieved by using the exact same amount of threads as virtual cores are available.

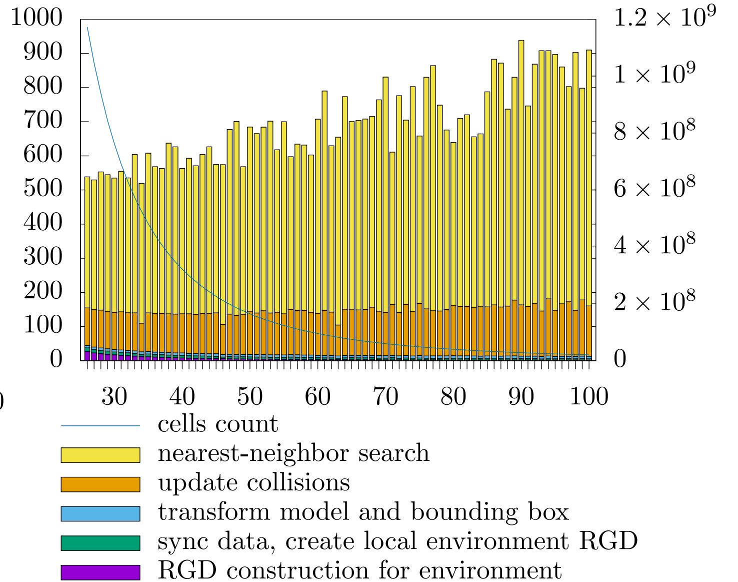

Fig. 2. Performance of the GPU method by grid resolution on the Hannover dataset. The x-axis shows the grid size. The right-hand-side x-axis belongs to the line plot and marks the resulting number of cells. The left-hand-side x-axis belongs to the stacked bar chart and indicates the computation time in seconds for each step of the GPU computation.

5.2 GPU specific benchmarks

In figure 2 we show the influence of grid resolution on the performance of the GPU accelerated implementation. For high number of cells the construction time of regular grid is increased, but is still below 5% of the total computation time. The most time is spent during nearest-neighbor search. The computation time increases with the grid cell size but also heavily fluctuates. These fluctuations can be explained by slight variations in the decompositioning resulting in different number of cells being needed on the GPU for collision detection. These fluctuations in run-time

are thus deterministic and depend on the input model and trajectory.

5.3 CPU versus GPU benchmarks

Figure 3 shows four graphs comparing the run-time of the CPU and GPU implementations on our four datasets. The CPU approach using the 3dtk k-d tree implementation uses as many threads as the respective machines have virtual CPU cores. The GPU approach using regular grid decomposition uses 50 as the grid size. It can be seen that the CPU and GPU based methods perform differently well, depending on the dataset. The GPU based method on the "titan X" platform is the fastest on the "Wolfsburg" dataset but the CPU based "e5-2680 v2" platform vastly outperforms the GPU based methods on the Train Tunnel dataset. We attribute the spatial differences between the datasets to this effect. The "El Teniente" and "Train Tunnel" datasets are similar in that the bounding box of the environment is mostly empty space in both cases, but more so for the "Train Tunnel" dataset which does not contain a loop like the "El Teniente" dataset. Thus, the regular grid decomposition for these datasets will yield a high number of empty cells which will never be queried and the cells with points which end up getting needed comparatively large and overapproximate the actual space to check for collisions. The Hannover and Wolfsburg datasets on the other hand are indoor datasets where the points are more or less evenly distributed over the constructed grid cells.

The last comparison we carried out was to evaluate the two approaches by how they behave depending on the number of points of the environment. For this purpose we created 111 random samplings of the environment point cloud of the "Hannover" dataset in from 500000 points up to 55500000 points in steps of 500000. We then executed our algorithms on the "e5-2680 v2" as well as on the "titan X" platform for each of the resulting 111 datasets, each with the original model and trajectory. The resulting runtimes can be seen in figure 4. The graphs indicate a nearly linear behaviour, but we suspect that a very shallow logarithmic function underneath. More research is needed to properly attribute the behaviour seen in the graph.

6. CONCLUSION AND OUTLOOK

This paper has presented a comparison of CPU and GPU implementations for the collision detection problem. With clever implementations, the run time is lowered to an acceptable level for telematics applications.

Needless to say a lot of work remains to be done. While we have now presented flexible CPU and GPU implementations, we further aim at improving run time. To this end, we will look into the issue of regular resampling of the trajectories through B-Spline approximation, support of double precision calculations for the GPU method as well as enhancing the GPU method with more collision detection methods from 3dtk (see Schauer and Nüchter (2015)). Another useful feature would be a heuristic which is able to pick a good cell size for the regular grid decomposition or a way to work around the limitations of the GPU methods for very sparse environments. Lastly, more experiments are needed to verify the actual dependence of our algorithms on the input pointcloud.

ACKNOWLEDGEMENTS

The authors greatfully acknowledge the funding from the German Academic Exchange Service DAAD (grant ID 57155328) – Sensor data processing using General Purpose Computations on GPUs.

REFERENCES

Ar, S., Chazelle, B., and Tal, A. (2000). Self-customized bsp trees for collision detection. *Computational Geometry*, 15(1), 91–102.

Bedkowski, J., Majek, K., and Nüchter, A. (2013). General purpose computing on graphics processing units for robotic applications. *Journal of Software Engineering for Robotics*, 4(1), 23–33.

Bedkowski, J., Maslowski, A., and De Cubber, G. (2012). Real time 3d localization and mapping for usar robotic application. *Industrial Robot: An International Journal*, 39(5), 464–474.

Bender, J., Erleben, K., and Trinkle, J. (2014). Interactive simulation of rigid body dynamics in computer graphics. In *Computer Graphics Forum*, volume 33, 246–270. Wiley Online Library.

Bittner, J., Hapala, M., and Havran, V. (2015). Incremental bvh construction for ray tracing. *Computers & Graphics*, 47, 135–144.

Elseberg, J., Borrmann, D., Schauer, J., Nüchter, A., Koriath, D., and Rautenberg, U. (2014a). A sensor skid for precise 3d modeling of production lines. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-5, 117–122. doi:10.5194/isprsannals-II-5-117-2014. URL `http://www.isprs-ann-photogramm-remote-sens-spatial-inf-sci.net/II-5/117/2014/`.

Elseberg, J., Borrmann, D., Schauer, J., Nüchter, A., Koriath, D., and Rautenberg, U. (2014b). A sensor skid for precise 3d modeling of production lines. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 2(5), 117.

Elseberg, J., Magnenat, S., Siegwart, R., and Nüchter, A. (2012). Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics*, 3(1), 2–12.

Faas, D. and Vance, J.M. (2011). Brep identification during voxel-based collision detection for haptic manual assembly. In *ASME 2011 World Conference on Innovative Virtual Reality*, 145–153. American Society of Mechanical Engineers.

Fan, W., Wang, B., Paul, J.C., and Sun, J. (2013). An octree-based proxy for collision detection in large-scale particle systems. *Science China Information Sciences*, 56(1), 1–10.

Garcia-Alonso, A., Serrano, N., and Flaquer, J. (1994). Solving the collision detection problem. *Computer Graphics and Applications, IEEE*, 14(3), 36–43.

Held, M., Klosowski, J.T., and Mitchell, J.S. (1995). Evaluation of collision detection methods for virtual reality fly-throughs. In *Canadian Conference on Computational Geometry*, 205–210. Citeseer.

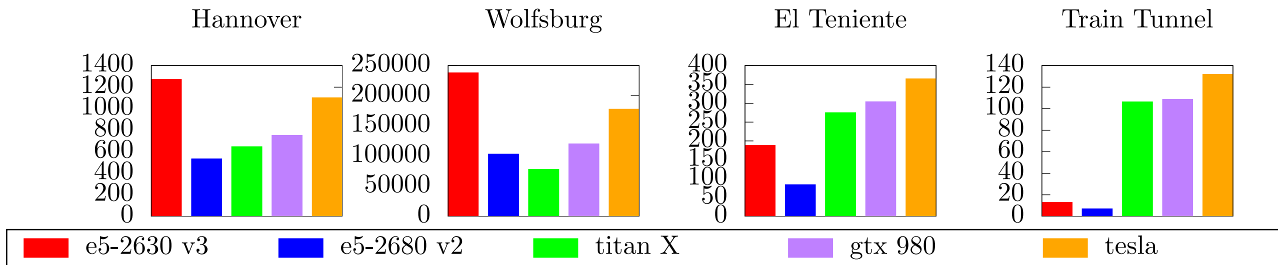

Fig. 3. Performance of CPU vs. GPU implementations on all four datasets. The y-axis denotes the runtime in seconds on each platform. The red and blue bars on the left represent the runtime of the CPU implementation while the remaining three bars show the GPU runtimes.

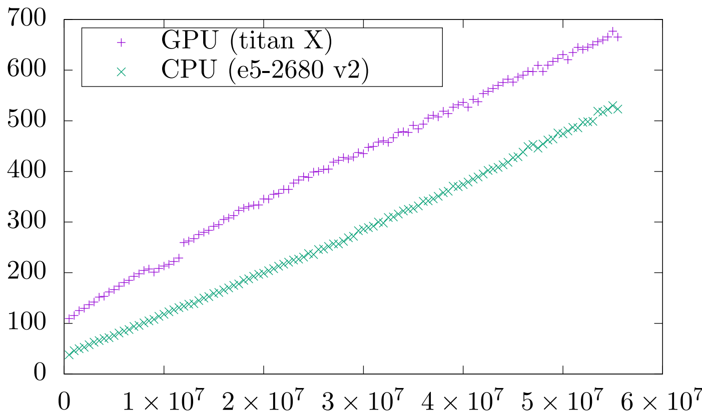

Fig. 4. Performance with varying numbers of points in the environment, based on random sampling of the Hannover dataset. The x-axis shows the number of points in the environment while the y-axis shows the runtime in seconds.

Hermann, A., Drews, F., Bauer, J., Klemm, S., Roennau, A., and Dillmann, R. (2014). Unified gpu voxel collision detection for mobile manipulation planning. In *Intelligent Robots and Systems (IROS), 2014*.

Hummel, J., Wolff, R., Stein, T., Gerndt, A., and Kuhlen, T. (2012). An evaluation of open source physics engines for use in virtual reality assembly simulations. In *Advances in Visual Computing*, 346–357. Springer.

Jiménez, P., Thomas, F., and Torras, C. (2001). 3d collision detection: a survey. *Computers & Graphics*, 25(2), 269–285.

Jung, D. and Gupta, K.K. (1996). Octree-based hierarchical distance maps for collision detection. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, 454–459. IEEE.

Klein, J. and Zachmann, G. (2004). Point cloud collision detection. In *Computer Graphics Forum*, volume 23, 567–576. Wiley Online Library.

Lin, M. and Gottschalk, S. (1998). Collision detection between geometric models: A survey. In *Proc. of IMA conference on mathematics of surfaces*, volume 1, 602–608.

MacDonald, J.D. and Booth, K.S. (1990). Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3), 153–166.

Magnenat, S. (2014). libnabo. URL `https://github.com/ethz-asl/libnabo`.

Mainzer, D. and Zachmann, G. (2014). Collision detection based on fuzzy scene subdivision. In *Symposium on GPU Computing and Applications (Singapore, 2013)*, volume 3.

Mount, D.M. and Arya, S. (2010). Ann: a library for approximate nearest neighbor searching, 2005. URL `http://www.cs.umd.edu/~mount/ANN/`.

Muja, M. and Lowe, D.G. (2012). Flann - fast library for approximate nearest neighbors. URL `http://www.cs.ubc.ca/research/flann/`.

Muñoz, J.S., León, C.A.D., and Gómez, H.T. (2014). Development of a hierarchy collision detection algorithm in order to implement a laparoscopic surgical simulator. *Revista QUID*, (19).

Popov, S., Günther, J., Seidel, H.P., and Slusallek, P. (2007). Stackless kd-tree traversal for high performance gpu ray tracing. In *Computer Graphics Forum*, volume 26, 415–424. Wiley Online Library.

Purcell, T.J., Buck, I., Mark, W.R., and Hanrahan, P. (2002). Ray tracing on programmable graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 21, 703–712. ACM.

Qiu, D., May, S., and Nüchter, A. (2009). Gpu-accelerated nearest neighbor search for 3d registration. In *Computer Vision Systems*, 194–203. Springer.

Schauer, J. and Nüchter, A. (2014). Efficient point cloud collision detection and analysis in a tunnel environment using kinematic laser scanning and kd tree search. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 40(3), 289.

Schauer, J. and Nüchter, A. (2015). Collision detection between point clouds using an efficient k-d tree implementation. *Advanced Engineering Informatics*, 29(3), 440–458.

Sellis, T., Roussopoulos, N., and Faloutsos, C. (1987). The r+-tree: A dynamic index for multi-dimensional objects.

Tang, M., Manocha, D., Lin, J., and Tong, R. (2011). Collision-streams: fast gpu-based collision detection for deformable models. In *Symposium on interactive 3D graphics and games*, 63–70. ACM.

Tzafestas, C. and Coiffet, P. (1996). Real-time collision detection using spherical octrees: virtual reality application. In *Robot and Human Communication, 1996., 5th IEEE International Workshop on*, 500–506. doi: 10.1109/ROMAN.1996.568888.

Wang, H.Y. and Liu, S.G. (2014). A collision detection algorithm using aabb and octree space division. In *Advanced Materials Research*, volume 989, 2389–2392. Trans Tech Publ.

Zhou, K., Hou, Q., Wang, R., and Guo, B. (2008). Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, 27(5), 126.