SLAM à la carte
GPGPU for Globally Consistent Scan Matching

Andreas Nüchter   Seyedshams Feyzabadi   Deyuan Qiu   Stefan May

***Abstract*— The computational complexity of SLAM is large and constitutes a challenge for real-time processing of a huge amount of sensor data with the limited resources of a mobile robot. Often, notebooks are used to control a mobile system and even these computing devices have nowadays graphics cards which allow general purpose computation using many cores. *SLAM à la carte (graphique)* exploits these capabilities and carries out 3D scan registrations on the GPU. A speed-up of more than one order of magnitude for precise 3D mapping is reported.**

I. Introduction

Many basic robot tasks require a globally consistent representation of the environment. Many modern robots are equipped with 3D scanners, that either work according to one of the following principles: the time-of-flight principle using pulsed or continuously emitted laser light, triangulation using projections of laser patterns or the projection of structured light, i.e., light coding. These sensors acquire 3D point clouds in a local coordinate frame from their surrounding. Local representations have to be matched to build a global map. The well-known iterative closest point algorithm (ICP) [2] puts two independently acquired 3D point clouds into one frame of reference. Instead of performing the scan matching based on extracted features, it uses closest points in Cartesian coordinates and an iterative procedure to compute the registration.

In previous work we have extended the ICP algorithm to *globally consistent 3D scan matching* [3], [16]. To yield a high-precise 3D mapping algorithm, it differs in an important aspect from other state-of-the-art simultaneous localization and mapping (SLAM) procedures. It repeatedly matches closest points of 3D scans stored in a pose graph to build a linear system of equations, that is also solved repeatedly to update the pose estimates. Thus, globally consistent 3D scan matching performs GraphSLAM in every iteration. Unlike Grisetti et al. who claims in [10, page 9] that "[...] the time to compute the linear system is negligible compared to the time to solve it" we experienced otherwise and strongly object (cf. Fig. 1). The data association is the bottleneck in scan matching based SLAM: Image to match n point clouds, each containing N 3D points. Usually it holds: $N \gg n$. For example typical 3D scans taken by the robot Kurt3D [17] contain up to 300,000 points and a typical 3D scan acquired by our robot Irma3D contains 3,000,000 points, while typical scenes contain only a few thousand scan poses. Finding nearest neigbors can be accomplished on average in $O(N \log N)$-time by using search trees [8]

A. Nüchter is with the Automation Group within the School of Engineering and Science of the Jacobs University Bremen gGmbH, 28759 Bremen, Germany a.nuechter@jacobs-university.de
S. Feyzabadi is with DFKI Bremen – Robotics Innovation Center, Germany shams.feyzabadi@dfki.de
D. Qiu is with the Shanghai Quality Supervision & Inspection Center for Medical Devices, China (PRC) deyuan.qiu@googlemail.com
Stefan May is with the Georg-Simon-Ohm University of Applied Sciences, Nürnberg, Germany stefan.may@ohm-hochschule.de

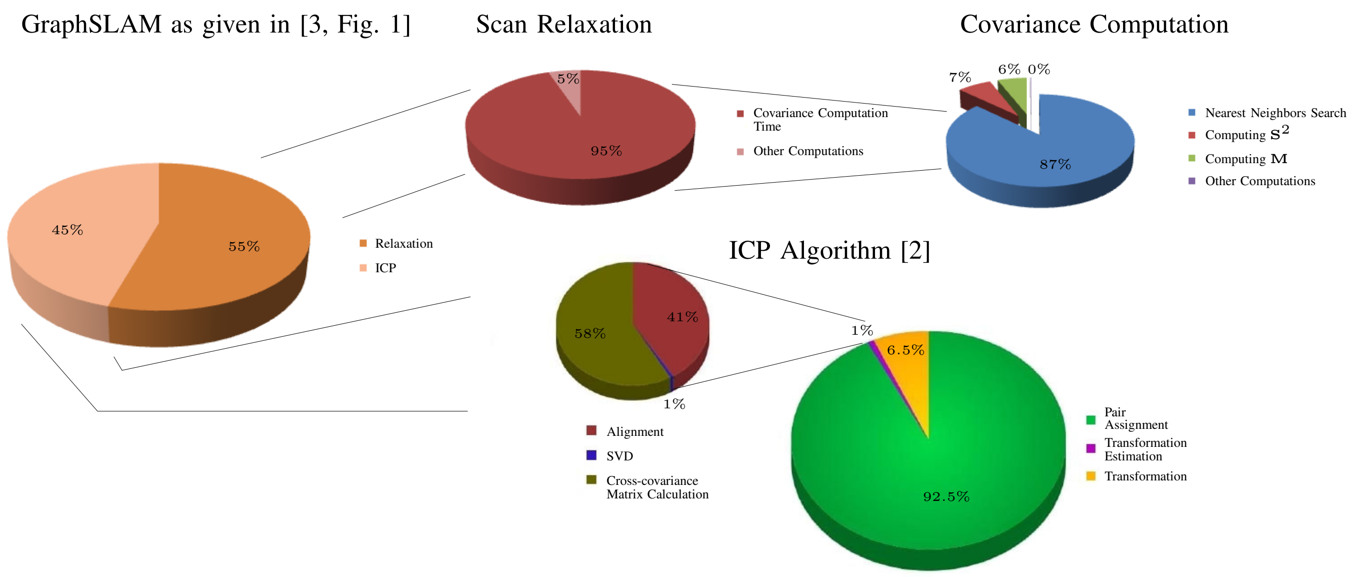


Fig. 1. Timing of GraphSLAM based globally consistent scan matching. Initial sequential ICP alignment is refined by an iterative procedure (cf. [16]). Most of the time are spent in nearest neighbor search, which is carried out on a k-d tree.

while solving the linear system is in $O(n^3)$. However, since the system of equations refers usually to a sparse graph, the letter Landau notation turns out to be not relevant, since a sparse Choleskey decomposition [6] is applicable. Therefore, speeding-up the data association, i.e., the search for correspondences, remains a central point. To this end, we have suggested in [15] to employ OPENMP to perform computing closest points in a parallel fashion. Of course, for feature based SLAM the number of equations grows with every associated feature and then the SLAM back-end becomes the bottleneck.

This paper presents results obtained by incorporating the immense computational power of GPUs (Graphics Processing Units) into our SLAM framework and into *3DTK – The 3D Toolkit* (`http://threedtk.de`). The increasing programmability of graphics cards enables general-purpose computing on graphics processing units (GPGPU) and yields a powerful massive parallel processing alternative to conventional multi-computer or multi-processor systems. Moreover, costs of commodity graphics cards are lower when measured in cost per FLOPS.

Traditional methods for computing closest points are difficult to be implemented on GPUs due to their recursive nature. We take advantage of Arya's priority search algorithm [1], to fit nearest neighbor search (NNS) in the Single Instruction Multiple Data (SIMD) model, so that it is possible to be accelerated by the use of a GPU [21]. The proposed algorithm, is implemented using CUDA, NVIDIA's parallel computing architecture [18].

II. STATE OF THE ART

An introduction to robotic mapping and SLAM is given in [24]. Recently, Grisetti et al. published a tutorial on graph-based SLAM [10]. To avoid repeating these summaries, the focus in this related work section is on related GPGPU algorithms. Up to our knowledge, this has not been applied to 3D robotic mapping. Our own previous algorithms have been published for example in [3], [16] and [15].

NNS is typically implemented using brute force methods on GPUs, which are by nature highly parallelizable. This property makes brute force based NNS methods easily adaptable for a GPU implementation and there are several implementations available. Purcell et al. used cells to represent photon locations [20]. By calculating the distances between photons in cells that are intersected with a search radius, and a query point, k nearest neighbors are located to estimate the radiance. Purcell et al. stressed that k-d tree and priority queue methods are efficient but difficult to be implemented on GPU [20]. Bustos et al. stored indices and distance information as quaternion in RGBA channels of a texture buffer. They used three fragment programs to calculate Manhattan distances and to minimize them by reduction [4]. Rozen et al. adopted a bucket sort primitive to search nearest neighbors [22]. Van Kooten et al. introduced a nearest neighbor search method based on projection for the particle repulsion problem [25], which chooses a viewport encompassing every particle (or as many as possible), and projects particles onto a projection plane. The approach takes advantage of the hardware accelerated functionalities of GPUs, such as projection and 2D layout of the texture buffer for grid calculation. Garcia et al. implemented a brute force NNS approach using CUDA [9]. When the data dimension is more than 96, and the number of points is less than around 10000, a speed up of 40 comparing to the CPU-based k-d tree implementation is reported. However, if the data dimension is less than 8 and the number of points is larger than around 19000, their algorithm becomes slower than the CPU implementation.

Other than brute force implementations, GPU-based NNS with advanced search structures are also available in the field of global illustration. In the context of ray tracing, the NNS procedure builds trees from a triangle soup, and takes also triangles but not points as the objects of interest. These algorithms cannot be used as general point-based NNS algorithms. On the other hand, the NNS algorithm for photon mapping shares a similar model with a general point-based NNS problem. Foley built a k-d tree on CPU and used the GPU to accelerate the search procedure [7]. Horn et al. extended Foley's work by restarting the search at half of the tree but not from the root [11]. Singh presented an SIMD photon mapping framework, using stack based k-d tree traverse to search k nearest neighbors [23]. Lieberman et al. used quad-tree for the similarity joint algorithm [14]. Zhou et al. implemented k-d tree based NNS on GPU for both – ray tracing and photon mapping – using the CPU as a coordinator [26]. They applied a heuristic function to construct k-d trees. In the k-d tree traverse stage, range searching is used to find the k nearest neighbors.

In recent GPU-accelerated NNS implementations, most NNS kernels are based on brute force methods. They are easy to implement but possess the natural drawback of low efficiency compared with advanced data structures. On the other hand, brute force methods mostly need reduction kernels in order to find the minimum in distance. A reduction kernel is slow due to its non-parallel nature, even implemented by highly optimized blocks. Tree-based NNS algorithms have shown a performance leap in global illumination. Hints and inspirations can be gained from these algorithms. Therefore the purpose for which they were designed makes them not easily adoptable for non-graphics purposes.

III. NEAREST NEIGHBOR SEARCH USING THE GPU

GPU-based algorithms conform to the SIMD model. In our GPU-NNS algorithm, all nearest neighbor searches (as many as the number of points in the scene point cloud) are executed simultaneously. When the number of searches is not the same as the number of threads the processor can allocate, the CUDA driver will schedule and assign the threads to the multi-processors, which is transparent to the users. Our GPU-NNS procedure features three steps.

a) k-d Trees: k-d trees are a generalization of binary search trees. Every node represents a partition of a point set to the two successor nodes. The root represents the whole point cloud and the leaves provide a complete disjoint partition of the points. These leaves are called buckets. Every node contains the ranges of the represented point set.

b) Array-based k-d Tree: Before the search procedure is performed, a piece of page-locked memory is allocated on the host side. As a convention, we refer to graphics cards as devices, and to CPU and main memory as host. A left-balanced k-d tree is built for the point set S by splitting the space always at the median of the longest axis. Being left-balanced, the k-d tree can be serialized into a flat array, and thus stored in the page-locked memory. Since the device memory cannot be dynamically allocated on the device, the array-based k-d tree is downloaded to the device before NNS. It is worth mentioning that in order to satisfy the coalescing of CUDA global memory, a Structure of Array (SoA) is used. Members of the structure are mostly 32-bit words, so that all threads of a half-warp (16 threads) are coalesced into one 64-byte global memory transaction.

c) Priority Search Method: Because recursion is not possible with CUDA, the traditional k-d tree search method cannot be used. However, the priority search method provides a way to put NNS on the GPU [1]. Priority queues are maintained in the registers of the GPU. The priority search algorithm iteratively executes the following three steps: First, extract the element having minimal distance to the query point from the queue. Second, expand the extracted node. Insert the higher node in the queue and then expand the lower node. This step is repeated till the leaf node. Third, update the nearest neighbor so far. The complete GPU-NNS algorithm is shown in Algorithm 1.

Algorithm 1 GPU-NNS Algorithm

```
Require: download the $k$-d tree, model point cloud and
    scene point cloud to GPU global memory.
 1: assign $n$ threads, where $n$ = number of query points.
 2: assign arrays pair[$n$] and distance[$n$] in GPU memory
    for results.
 3: for every query point in parallel, do
 4:     assign the query point to a thread
 5:     allocate a dynamic list for the thread
 6:     construct dynamic queue $q$
 7:     Initialize $q$ with the root node
 8:     do priority search, find: $pair$ w. shortest distance $d$
 9:     if $d <$ distance threshold then
10:         pair[threadID] = $pair$
11:         distance[threadID] = $d$
12:     else
13:         pair[threadID] = non-pair flag
14:         distance[threadID] = 0
15:     end if
16: end for
```

IV. GPU-accelerated ICP

Given two independently acquired sets of 3D points, $\hat{M}$ (model set) and $\hat{D}$ (data set) which correspond to a single shape, we want to find the transformation $(\mathbf{R}, \mathbf{t})$ consisting of a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$ which minimizes the following cost function [2]:

$$E(\mathbf{R}, \mathbf{t}) = \frac{1}{N} \sum_{i=1}^{N} ||\mathbf{m}_i - (\mathbf{R}\mathbf{d}_i + \mathbf{t})||^2 , \qquad (1)$$

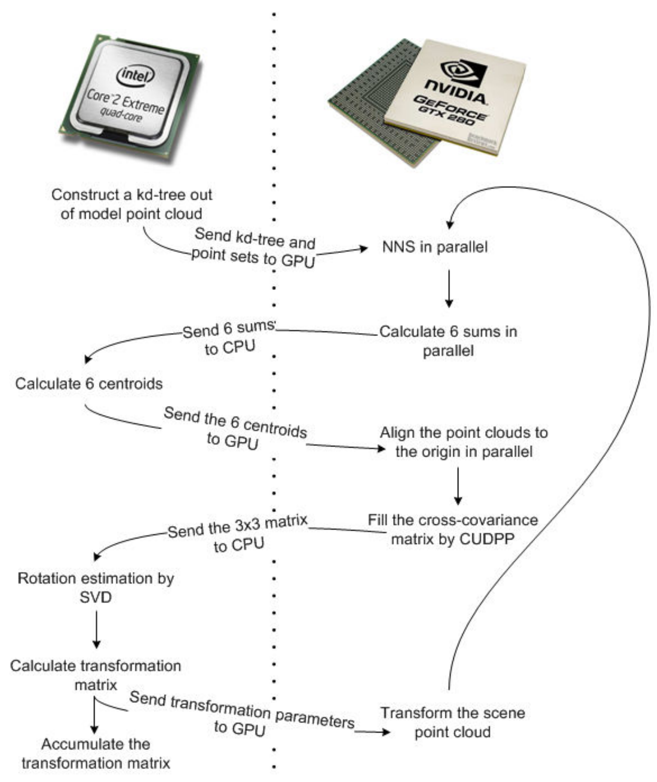
All corresponding points can be represented in a tuple $(\mathbf{m}_i, \mathbf{d}_i)$ where $\mathbf{m}_i \in M \subset \hat{M}$ and $\mathbf{d}_i \in D \subset \hat{D}$. Two things have to be calculated: First, the corresponding points, and second, the transformation $(\mathbf{R}, \mathbf{t})$ that minimizes $E(\mathbf{R}, \mathbf{t})$ on the basis of the corresponding points. The ICP algorithm uses closest points as corresponding points. A sufficiently good starting guess enables the ICP algorithm to converge to the correct minimum.

Respecting the GPU-based ICP approach, there are only a few implementations yet, due to the very recent development of the GPGPU technique. Kitaaki et al. implemented Modified ICP (M-ICP) [12] on GPU [13]. For nearest neighbor search, they took advantage of the massive raw computational power of GPU, calculating $N_M \times N_D$ times the distances between query points and model points, in which, N_M and N_D are the numbers of points in the model and data point cloud respectively. The method of transformation estimation was not presented, although the whole ICP pipeline is declared to have been implemented on GPU. As a result, their method is reported to be 2 to 3 times faster than the original one. Park et al. implemented a GPU-based pose estimation pipeline [19]. ICP was used as the last stage for fine-tuning, which is unfortunately not implemented on GPU. Choi et al. [5] applied *Iterative Projection Point (IPP)* instead of searching for the closest points to register 3D shapes. Since no searching procedure is required, a registration time of more than 200 milliseconds was reported for point clouds with 320×240 points. However, the matching accuracy might not be comparable to the ICP algorithm, which is not discussed in the publication.

Our extension of the ICP to the SIMD model accelerates not only the NNS stage but also the remaining parts of the ICP algorithm, namely the scene point cloud transformation, zero mean alignment by computing centroids [2] and the covariance matrix for the SVD based minimizer [16], which take also observable amount of time. Fig. 2 illustrates the coordination between the host and the device of the GPU-ICP algorithm. The major steps are:

Centroids Calculation: To calculate the centroids of the point clouds, the coordinates need to be summed up separately over all the points, yielding 6 sums to be divided by the number of points. To sum up the values, the reduction kernel in CUBLAS library (CUDA implementation of basic linear algebra methods) is applied, which heavily uses the shared memory.

Unlike on CPU, where the number of nearest neighbor pairs is a natural gain in the end of the loop, GPU finds the pairs by threads in parallel, so that the number of pairs has to be counted after the NNS procedure. Counting the number of pairs is implemented by the compact function of CUDPP library (CUDA Data Parallel Primitives Library),



Construct a kd-tree out of model point cloud
Send kd-tree and point sets to GPU
NNS in parallel
Send 6 sums to CPU
Calculate 6 sums in parallel
Calculate 6 centroids
Send the 6 centroids to GPU
Align the point clouds to the origin in parallel
Send the 3x3 matrix to CPU
Fill the cross-covariance matrix by CUDPP
Rotation estimation by SVD
Calculate transformation matrix
Send transformation parameters to GPU
Transform the scene point cloud
Accumulate the transformation matrix

Fig. 2. The coordination between CPU and GPU in the GPU-ICP algorithm. Notice that other than constructing the k-d tree, CPU does only negligible work. The data transfer over the PCIe is minimized. Pictures of the chips are taken from manufacturer websites.

which compacts an array according to a predefined mask array. The number of pairs is a side gain of the function. The six divisions are executed on the CPU in double precision, and then the centroids are send to GPU.

Point Cloud Alignment: The point clouds are aligned to the original points using the two calculated centroids in parallel. Similar to the principle of GPU-NNS, the number of threads is the number of points and each point is translated by a standalone thread in parallel.

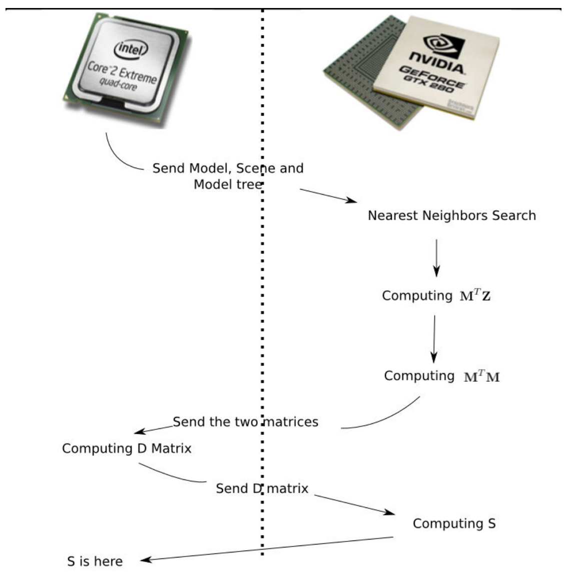
Preparation for SVD: The covariance matrix for the SVD-based ICP [16] is calculated before it is decomposed. It is filled by the dot product function of CUBLAS on GPU. Every thread, i.e., every point pair, contributes in parallel to the matrix. If a point pair is rejected, we use zero vectors for the dot product. Since the covariance matrix is small (3×3) and needs double precision, the SVD is not implemented on the GPU. We used the NewMat library to calculate the decomposition on the CPU.

Further Implementation Details: The block size is configured to be 192 or 256 to get the optimal compromise between enough active threads and enough registers per multi-processor.

V. GPU-accelerated Globally Consistent Scan Matching

In the overall scan matching based 3D mapping, global relaxation takes a longer time than ICP (see Fig. 1). Analyzing the relaxation phase, yields that the most time consuming part is again the NNS which is required for the covariance computation.

We summarize our task distribution of the covariance computation in Fig. 3. Currently, we copy the tree to GPU in each SLAM iteration to retrieve the covariance matrix for each link. This is time consuming which can not be solved easily. Since the number of calls of this method is very high, it decreases the efficiency of our algorithm. However, the procedure given by Fig. 3 effectively exploits NNS on the graphics card.



Send Model, Scene and Model tree
Nearest Neighbors Search
Computing $\mathbf{M}^T\mathbf{Z}$
Computing $\mathbf{M}^T\mathbf{M}$
Send the two matrices
Computing D Matrix
Send D matrix
Computing S
S is here

Fig. 3. The coordination between CPU and GPU in the scan relaxation algorithm. In every iteration, the pre-computed trees are transferred to GPU for NNS and the resulting covariance matrices are copied back.

VI. Experiments and Results

The computer for our experiment was a Intel(R) Core(TM)2 Quad CPU Q9450 @ 2.66GHz with a GeForce GTX 260 with 216 cores. Its graphics memory is 896 MB.

For a system test of the ICP algorithm, we use synthetic data with known ground truth poses, thus we are able to analyze the convergence of the proposed algorithm. The 3D scan registration is based on two artificial cubes (Fig. 4, left). Each of them consists of N random points lying on the surface of the cube. The side length of each cube is 10 units. The second cube is generated by translating the first cube through the vector $[1.0, 1.0, 1.0]$ and rotating it around the axis $[1.0, 1.0, 1.0]$ for 0.1 radian. The convergence, which represents the matching quality, is measured by variation as defined by root mean squared error of the distances between nearest neighbors. It is observed in Fig. 4 that although GPU-ICP with different queue lengths present different convergent rates, they end up with very similar convergent limits. The reason for this behavior is the iterative fashion of ICP. The assumption made by the ICP algorithm, is that in the last iteration, the point pairs are correct. Retrieving approximate nearest neighbors yields different point pairs for the minimization step. Due to the large number of points in the point cloud, the computed transformation $(\mathbf{R}, \mathbf{t})$ is still similar to

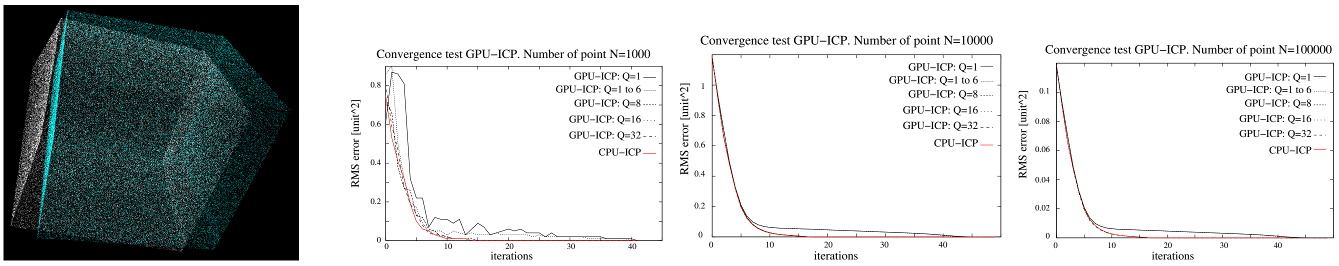


Fig. 4. The left image shows the artificial cube used for the convergence test. The remaining subfigures present the convergence test results for different number of points. Comparison of the convergence over 50 iterations between GPU-ICP implementations and the CPU-based ICP implementation. "Q" means priority queue length. Approaches with different queue lengths present different convergent rate. However, they end up with very similar convergent limits.

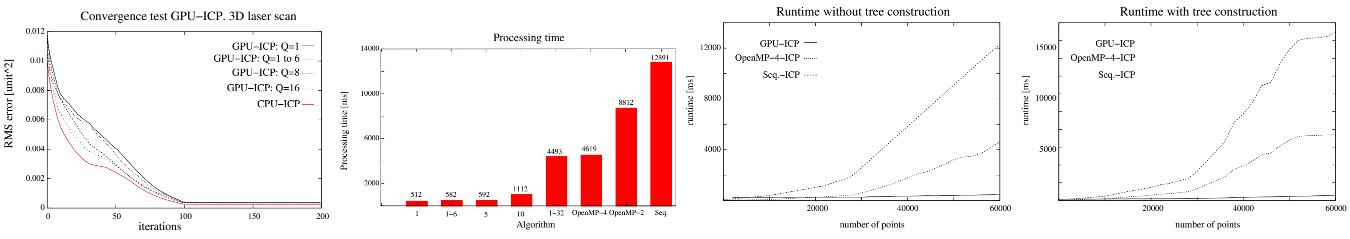


Fig. 5. From left to right: (1) Convergence test for a 3D laser scan. (2) Speed test. The result of the 3D registration experiment of 68229 points. 1-6 means the queue length is increased gradually in iterations as follows: 1, 2, 3, 4, 5, 6, and 1-32 as: 1, 2, 4, 8, 16, 32. It is observed from the speed test that GPU-ICP is 25 times faster than the sequential CPU-based ICP. In addition, the OpenMP results with 4 and 2 threads are given. (3) The runtime comparison among sequential CPU implementation, OPENMP-based implementation and GPU-ICP. Without k-d tree construction. (4) Including k-d tree construction.

the exact case. Therefore, more iterations will be needed to match the point clouds, and again, in the last iteration the point pairs are correct, since the query point will be in the bucket, where the true nearest neighbor is. If less points are present in the point cloud the effect of "initially" wrong correspondences becomes larger, however, the minimum of zero is always reached.

In all experiments a slightly better accuracy could be observed for the CPU implementation compared to the GPU implementation. The reason is that the CPU operates in a double floating mode and allows a longer queue.

The data sets used in our real-life evaluation are available in the *Robotic 3D Scan Repository* (`http://kos.informatik.uni-osnabrueck.de/3Dscans/`). The first data set is a typical indoor scan acquired by an actuated, i.e., tilting SICK LMS200 scanner. Both point clouds contain 68,229 points. The second data set refers to the first 65 3D scans of the set *Hannover1* contributed by Oliver Wulf, Leibniz Universität Hannover, Germany. These 3D scans have a field of view of 360° × 180° and each contains approximately 30,000 3D points. The third data set is the *Koblenz* data set acquired by Johannes Pellenz and Dagmar Lang from the Active Vision Group, University of Koblenz-Landau, Germany using a Velodyne HDL-64E Laser range finder with a field of view 360° × 28.8°. Each scan contains roughly 102,000 points.

Registration of two 3D Scans: This experiment is conducted to compare the performance of the GPU-ICP algorithm, the OPENMP-based ICP algorithm [15] and the sequential CPU-based ICP implementation, which uses a single CPU. The convergence test of GPU-ICP with different queue lengths is presented in Fig. 5 (1). It is observed that although the executions with different queue lengths yield different convergence speeds at the beginning, they all converge to a similar variation limit, so as the one with a single-element queue. Figure 5 (2) shows the time consumed by different queue length configurations, as well as the two CPU implementations. When registering 68,229 points, GPU-ICP with single-element queues performs more than 25 times faster than the standard sequential CPU implementation. Fig. 5 compares the speed of the three implementations: standard CPU-based ICP, OPENMP-based ICP and GPU-ICP. When registering 68,229 points, OPENMP multi-threading on a quad core machine achieves a speed-up of around 3 compared to the sequential execution, while a speed-up of 25 is achieved by GPU implementation. It is observed that with more points, the speed-up ratio of GPU-ICP to CPU implementations is improved. Data are transferred to the GPU only once per ICP registration process (here: $2 \times 68{,}229$ points amounting to $\approx 7\,\text{MB}$), which take less than 3 msec in case the GPU is connected by PCIe 2.0.

The data sets *Hannover1* and *Koblenz* were recorded using continuous 3D scanning. By using this scanning methodology a very large number of scans can be recorded. Fig. 6 shows the timing results and two views of the map obtained by registration of the Koblenz data set. Similar results were achieved with the *Hannover1* data sets. The overall speed-up is mainly attributed to the acceleration of the ICP algorithm. The high number of points in every scan is advantageous for our GPU-accelerated ICP. It turned out, that one can obtain only a speed-up factor of about 5 for the GraphSLAM part as given in Fig. 3. This is due to fact that in every GraphSLAM iteration all trees must be transferred to the GPU in order to execute the NNS for all graph links.

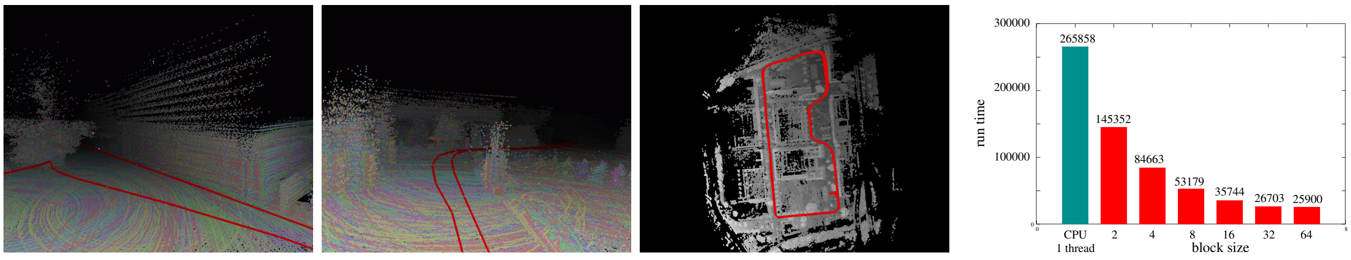


Fig. 6. Left two images: 3D views of the point cloud acquired by a Velodyne scanner. Different scans have different colors. Middle: Bird's-eye view. Right: Run time comparison for the Koblenz data set.

VII. Discussion, Conclusions, and Future Work

We developed a GPU-accelerated NNS approach and used this as part of our implementation for the SLAM front-end. More precisely, we employ a k-d tree based priority search method for computing closest points. The performance is enhanced significantly by the massive parallelism of the GPU SIMD architecture. It is higher by one order of magnitude using a modern commodity video card. The resulting approach is still faster than the MIMD-based approaches running on a state-of-the-art quad core CPU.

Needless to say a lot of work remains to be done. Our current approach suffers from the fact, that in every iteration of GraphSLAM, where we conduct a NNS, compute covariances and execute a SLAM back-end, the k-d trees must be time-consumingly transferred to GPU memory. In future work, we will concentrate on solving this memory bounded GraphSLAM optimization problem, where we only store a certain part of the scans in the limited GPU memory (here 896 MB).

With the rapid development of GPU technology, the k-d tree construction stage will be migrated to GPU, possibly using a breadth first, dynamic list based approach.

Acknowledgements

The GPGPU based registration software is available in the *3DTK – The 3D Toolkit*. All contributers are gratefully acknowledged. This work was partially supported by B-IT foundation, Applied Sciences Institute, a cooperation between Fraunhofer IAIS and University of Applied Sciences Bonn-Rhein-Sieg and has been partially funded by the grants AUTOSCAN (BMWi KF2470003DF0) and VeloSLAM (NSFC 41050110437).

References

[1] S. Arya and D. M. Mount. Algorithms for Fast Vector Quantization. In *Proc. Data Compression Conf.*, IEEE Comp. Society Press, 1993.

[2] P. Besl and N. McKay. A method for Registration of 3-D Shapes. *IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI)*, 14(2), 1992.

[3] D. Borrmann, J. Elseberg, K. Lingemann, A. Nüchter, and J. Hertzberg. Globally consistent 3d mapping with scan matching. *Journal Robotics and Autonomous Systems (JRAS)*, 56(2), 2008.

[4] B. Bustos, O. Deussen, S. Hiller, and D. Keim. A Graphics Hardware Accelerated Algorithm for Nearest Neighbor Search. In *Proc. of the 6th Intl. Conf. on Computational Science*, pages 196–199, 2006.

[5] S.-I. Choi, S.-Y. Park, J. Kim, and Y.-W. Park. Multi-view Range Image Registration using CUDA. In *The 23rd Intl. Technical Conf. on Circuits/Systems, Computers and Communications*, 2008.

[6] T. A. Davis. Algorithm 849: A concise sparse Cholesky factorization package. *ACM Transaction of Mathematical Software*, 31(4), 2005.

[7] T. Foley and J. Sugerman. KD-Tree Acceleration Structures for a GPU Raytracer. In *Graphics Hardware*, pages 15–22, July 2005.

[8] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transaction on Mathematical Software*, 3(3):209–226, September 1977.

[9] V. Garcia, E. Debreuve, and M. Barlaud. Fast k Nearest Neighbor Search using GPU. In *Proceedings of the Comp. Vision and Pattern Recognition Workshop (CVPRW)*, pages 1–6, June 2008.

[10] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard. A tutorial on graph-based SLAM. *IEEE Transactions on Intelligent Transportation Systems Magazine*, 2:31–43, 2010.

[11] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-D Tree GPU Raytracing. In *Proceedings of the Symposium on Interactive 3D graphics and games*, pages 167–174, April 2007.

[12] S. Kaneko, T. Kondo, and A. Miyamoto. Robust Matching of 3D Contours Using Iterative Closest Point Algorithm Improved by M-estimation. In *Pattern Recognition*, vol. 36, pp. 2041–2047, 2003.

[13] Y. Kitaaki, H. Okuda, H. Kage, and K. Sumi. High speed 3-D registration using GPU. In *SICE Ann. Conf. 2008*, 2008.

[14] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A Fast Similarity Join Algorithm Using Graphics Processing Units. In *Proc. of the 24th IEEE Intl. Conf. on Data Engineering*, 2008.

[15] A. Nüchter. Parallelization of Scan Matching for Robotic 3D Mapping. In *Proceedings of the 3rd European Conference on Mobile Robots (ECMR '07)*, Freiburg, Germany, September 2007.

[16] A. Nüchter, J. Elseberg, P. Schneider, and D. Paulus. Study of Parameterizations for the Rigid Body Transformations of The Scan Registration Problem. *Journal Computer Vision and Image Understanding (CVIU)*, 114(8):963–980, August 2010.

[17] A. Nüchter, K. Lingemann, J. Hertzberg, H. Surmann, K. Pervölz, M. Hennig, K. R. Tiruchinapalli, R. Worst, and T. Christaller. Mapping of Rescue Environments with Kurt3D. In *Proc. of the IEEE Intl. SSRR*, pages 158–163, Kobe, Japan, June 2005.

[18] nVidia. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. nVidia, version 2.0 edition, June 2008.

[19] I. K. Park, M. Germann, M. D. Breitenstein, and H.-P. Pfister. Fast and automatic object pose estimation for range images on the gpu. In *Machine Vision and Applications*. Springer-Verlag, August 2009.

[20] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon Mapping on Programmable Graphics Hardware. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*, 2003.

[21] D. Qiu, S. May, and A. Nüchter. GPU-accelerated Nearest Neighbor Search for 3D Registration. In *Proc. of the 7th Intl. Conf. on Computer Vision Systems (ICVS '09)*, 2009.

[22] T. Rozen, K. Boryczko, and W. Alda. GPU bucket sort algorithm with applications to nearest-neighbour search. In *Journal of the 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, February 2008.

[23] S. Singh and P. Faloutsos. SIMD Packet Techniques for Photon Mapping. In *Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing*, pages 87–94, September 2007.

[24] S. Thrun. Robotic mapping: A survey. In G. Lakemeyer and B. Nebel, editors, *Exploring Artificial Intelligence in the New Millenium*. Morgan Kaufmann, 2002.

[25] K. van Kooten, G. van den bergen, and A. Telea. *GPU Gems 3*, chapter 7, pages 123–148. Addison Wesley Professional, August 2007.

[26] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-Time KD-Tree Construction on Graphics Hardware. In *SIGGRAPH Asia 2008*, page 10, April 2008.