GPU-accelerated Nearest Neighbor Search for 3D Registration

Deyuan Qiu[1], Stefan May[2], and Andreas Nüchter[3]

[1] University of Applied Sciences Bonn-Rhein-Sieg**,
Sankt Augustin, Germany
dqiu2s@smail.inf.h-brs.de
[2] INRIA, Sophia-Antipolis, France
stefan_may@arcor.de
[3] Jacobs University Bremen, Germany
andreas@nuechti.de

Abstract. Nearest Neighbor Search (NNS) is employed by many computer vision algorithms. The computational complexity is large and constitutes a challenge for real-time capability. The basic problem is in rapidly processing a huge amount of data, which is often addressed by means of highly sophisticated search methods and parallelism. We show that NNS based vision algorithms like the Iterative Closest Points algorithm (ICP) can achieve real-time capability while preserving compact size and moderate energy consumption as it is needed in robotics and many other domains. The approach exploits the concept of general purpose computation on graphics processing units (GPGPU) and is compared to parallel processing on CPU. We apply this approach to the 3D scan registration problem, for which a speed-up factor of 88 compared to a sequential CPU implementation is reported.

Key words: NNS, GPGPU, ICP, 3D registration, SIMD, MIMD

1 Introduction

Nearest Neighbor Search (NNS) algorithms aim to optimize the process of finding closest points in two datasets with respect to a distance measure. It is a commonly employed geometrical algorithm in computer vision. In the context of 3D vision, NNS is used frequently in 3D point cloud registration. The registration of large data sets, such as acquired by means of modern 3D sensors, is computationally expensive.

In recent years, growing interest has been attracted by GPUs (Graphics Processing Units) on account of their immense computational power assembled in compact design. Increasing programmability enables general purpose computation and yields a powerful massively parallel processing alternative to conventional multi-computer or multi-processor systems. Moreover, costs of commodity graphic cards are lower when measured in cost per FLOPS.

** This work was supported by B-IT foundation, Applied Sciences Institute, a cooperation between Fraunhofer IAIS and University of Applied Sciences Bonn-Rhein-Sieg.

Being of a recursive nature, traditional NNS is difficult to be implemented on GPUs. We take advantage of Arya's priority search algorithm [1], to fit NNS in the SIMD (Single Instruction Multiple Data) model, so that it is possible to be accelerated by use of a GPU. *GPU-NNS*, the proposed algorithm, is implemented using *CUDA*, nVidia's parallel computing architecture [10]. Additional approaches are also applied to accelerate the NNS process, such as single-element priority queue and array-based k-d tree (see 3.1). GPU-NNS is used to implement a 3D registration algorithm: *GPU-ICP* (see 3.2). As a standard scan matching algorithm, ICP is widely used for 3D data processing [6]. The analysis on the standard sequential CPU-based ICP algorithm shows that the most time-consuming part of ICP is NNS. In the proposed GPU-ICP implementation, not only NNS, but also the remaining stages are accelerated by GPU. The experiment shows that GPU-ICP performs 88 times faster than a kd-tree based sequential CPU ICP algorithm, which runs on a Intel Core 2 Duo E6600 CPU.

The following sections are structured as follows: Section 2 outlines work related to GPU-based NNS. Section 3 explains our GPU-NNS and GPU-ICP implementation. Section 4 analyzes the results of two experiments: registering two 3D scans and indoor 3D mapping. Section 5 concludes with an outlook on future work.

2 Related Work

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 have been a couple of implementations available. Purcell et al. used cells to represent photon locations [11]. By calculating the distances between the photons in the 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 [11]. Bustos et al. stored indices and distance information as quaternions in RGBA channels of a texture buffer. They used three fragment programs to calculate Manhattan distances and to minimize them by reduction [2]. Rozen et al. adopted a bucket sort primitive to search nearest neighbors [12]. Van Kooten et al. introduced a projection based nearest neighbor search method for the particle repulsion problem [14], 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, showing that it is multiple times faster than CPU k-d tree approach in high dimensional situations [4].

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 with a different manner 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 [3]. Horn et al. extended Foley's work by restarting the search at half of the tree but not from the root [5]. Singh presented an SIMD photon mapping framework, using stack based k-d tree traverse to search k nearest neighbors [13]. Lieberman et al. used quad-tree for the similarity joint algorithm [8]. Zhou et al. implemented k-d tree based NNS on GPU for both ray tracing and photon mapping, using the CPU as a coordinator [15]. He 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.

3 Massively Parallel Nearest Neighbor Search

The NNS problem can be stated as follows: Given a point set S and a query point q in metric space M, the problem is to optimize the process of finding the point $p \in S$, which has the smallest Euclidean distance to q. Our massively parallel nearest neighbor search algorithm, GPU-NNS, is implemented on the CUDA architecture. Instead of using the brute force and linear search method, we use a space partitioning structure, k-d tree, to improve the search process. In the following, the terms *host* and *device* refer CPU and GPU respectively.

3.1 GPU-NNS

Next we describe our GPU-NNS procedure which features three steps.

Array-based k-d Tree. Before the search procedure, a piece of page-locked memory is allocated on the host side. A left-balanced k-d tree is built for S by splitting the space always on 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 [7]. Since the device memory cannot be dynamically allocated, the array-based k-d tree is downloaded to the device before NNS stage. It is worth mentioning that in order to satisfy the coalescing of CUDA global memory, a Structure of Array (SoA) is used.

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 by the registers in the GPU. The priority search algorithm iteratively executes the follow three steps: (1) Extract the element with the minimal distance to the query point from the queue. (2) Expand the extracted node, insert the higher node in the queue and then expand the lower node. The step is repeated till the leaf node. (3) 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.
assign n threads, where n = number of query points.
assign arrays pair[n] and distance[n] in GPU memory for results.
for every query point **in parallel**, **do**
  assign the query point to a thread
  allocate a dynamic list for the thread
  construct dynamic queue q
  Initialize q with the root node
  do **priority search**, find: *pair* with shortest distance d
  if $d <$ distance threshold **then**
    pair[threadID] = *pair*
    distance[threadID] = d
  else
    pair[threadID] = non-pair flag
    distance[threadID] = 0
  end if
end for

Single-element Priority Queue. Registers are scarce resources in GPU hardware, i.e., the more threads are launched at a time, the less registers can be assigned to each. If the space is unevenly partitioned (e.g., in a point cloud of real scene), a k-d tree based NNS routine performs frequently backtracking to find the real nearest neighbor. In this case, a long queue has to be maintained for each search thread. Such long queues are either not affordable by GPU registers or not desirable in terms of a long search time. In our approach, early termination is achieved by fixing the length of the queues. The insertion of nodes is ignored when the queues are filled. It is observed from experiments that single-element queues also result in valid search. With single-element queues, the process needs fewer steps, especially extractions to find the minimum. A 3D registration experiment in section 4 shows that a GPU-NNS with a single-element queue performs valid matching while consuming less time. In [9] a series of evaluations is presented stating that approximation does not significantly deteriorate the quality of 3D registration using approximate nearest neighbors in an ICP framework.

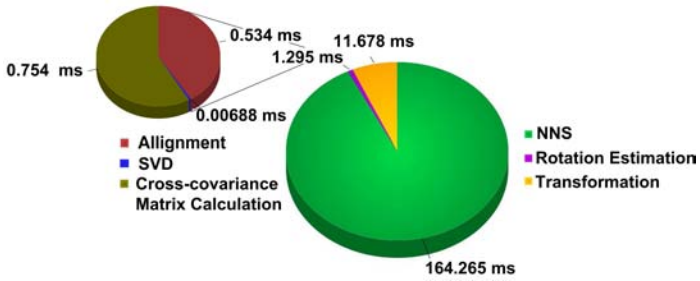


Fig. 1: The pie chart depicts the timing of different stages of one ICP iteration with two point clouds of 68229 points. In the standard sequential CPU implementation, the tree is constructed using the sliding midpoint splitting rule of ANN library, with a bucket size of 1. The tree is searched by an approximate nearest neighbor search method, with maximal visited points of 50. Matrix calculation, i.e., the singular value decomposition (SVD) for the closed form minimization of the ICP error function is implemented by NewMat library.

3.2 GPU-based 3D Registration

The ICP algorithm is a standard algorithm of 3D registration [6]. The algorithm iterates over three steps: (1) Find point correspondences by the nearest neighbor criteria. (2) Estimate the transformation using a mean square error function. (3) Transform the points. Terminate if an error criterion is fulfilled, otherwise go back to (1). Fig. 1 shows the runtime of different stages of a standard sequential CPU-based ICP algorithm in which NNS consumes the majority of the time. The proposed approach accelerates not only the NNS stage but also the remaining parts of the ICP algorithm, namely the transformation, alignment and covariance matrix calculation. Fig. 2 illustrates the coordination between the host and the device of the GPU-ICP algorithm.

In our implementation, a stepwise reduction of the search radius is applied to refine the iterations. The model point cloud is stored in the texture buffer, not global memory, since they are cached in multiprocessors. Matrix operations are performed by employing the CUBLAS library (CUDA implementation of basic linear algebra methods). Counting the number of valid pairs is implemented with the compact method of CUDPP (CUDA Data Parallel Primitives Library). The block size is configured to be 192 or 256 to get the optimal compromise of enough active threads and enough registers per multi-processor.

4 Experiments and Results

We analyze the results of the GPU-ICP algorithm and compare its performance with that of the OpenMP-based ICP algorithm run on an Intel® Core^{TM}2 Duo 6600 CPU. The GPU-ICP algorithm is performed on a nVidia GeForce GTX280® GPU.

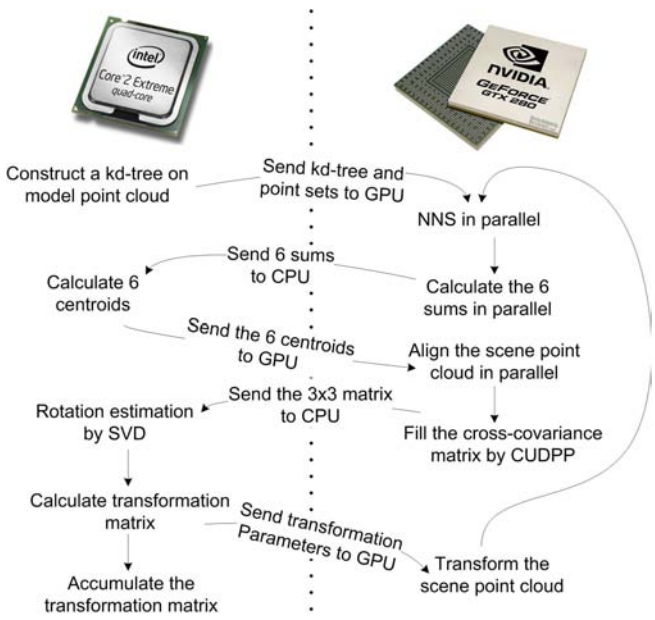
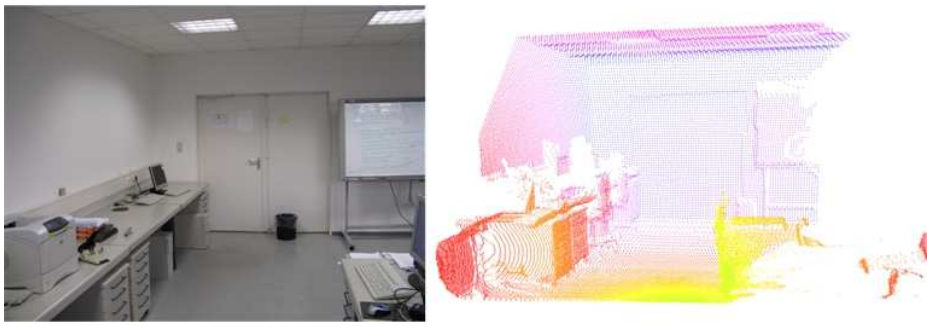


Fig. 2: The coordination between CPU and GPU in the GPU-ICP algorithm. Notice that other than constructing the k-d tree, CPU does only negligible work. The data transfer over the PCIe is minimized. Pictures of the chips are taken from manufacturer websites.

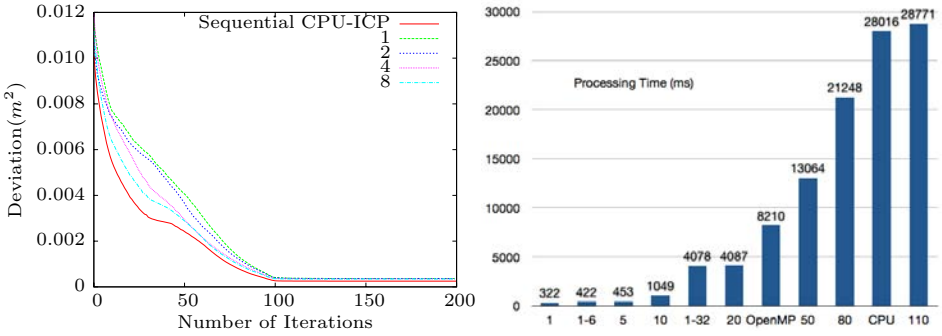
Experiment 1 – Registering two 3D Scans. The 3D scan registration experiment is based on two partially overlapping 3D point clouds captured by a SICK laser scanner in an office-like indoor environment as shown in Fig. 3. Both of the scans have 68229 points. Other than the OpenMP-based ICP algorithm, the performance of GPU-ICP is also compared with a sequential CPU-based ICP (the same implementation as the one of Fig. 1), which uses a single CPU and sets the maximal visited points to 50 as an early termination condition.

The convergence test on GPU-ICP with different queue lengths is presented in Fig. 4 (a). The convergence, which represents the matching quality, is measured by variation. Variation is defined by root mean squared error of the distances between nearest neighbors. The root operation is eliminated in implementation. It is observed that although the executions with different queue lengths perform at different convergence speeds at the beginning, they all converge to a similar variation limit, so as the one with a single-element queue. Figure 4(b) shows the time consumed by different queue length configurations, as well as the two CPU implementations. When registering 68229 points, GPU-ICP with single-element queues performs more than 88 times faster than the standard sequential CPU implementation. Fig. 5 compares the matching results between GPU-ICP and CPU based ICP. Notice that the purple is the blend of the red and the blue when point clouds are aligned. GPU-ICP with single-element priority queues performs similar matching results in shorter time. Fig. 6 compares the speed of the three



(a) (b)

Fig. 3: The experimental scene, where the point clouds are captured. (a) An office-like indoor environment. (b) The point cloud captured in the environment, displayed in fake depth color.



(a) Convergence test. (b) Speed test.

Fig. 4: 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 88 times faster than the sequential CPU-based ICP.

implementations: standard CPU-based ICP, OpenMP-based ICP and GPU-ICP. When registering 68229 points, OpenMP multi-threading on a dual core machine achieves a speedup of around 3 compared to the sequential execution, while a speedup of 88 is achieved by GPU implementation. It is observed that with more points, the speedup ratio of GPU-ICP to CPU implementations is improved. Data (68229×2 points which amount to around 7 MB) are transfered to the GPU only once in an ICP process, which take less than 3 milliseconds if the GPU is connected by PCIe 2.0.

The same experiment is also tested on a contemporary system, which has an Intel® Core^{TM} i7-965 Extreme Edition CPU and a GeForce GTX 295 GPU. All experimental settings are the same as above. The results are in Table 1.

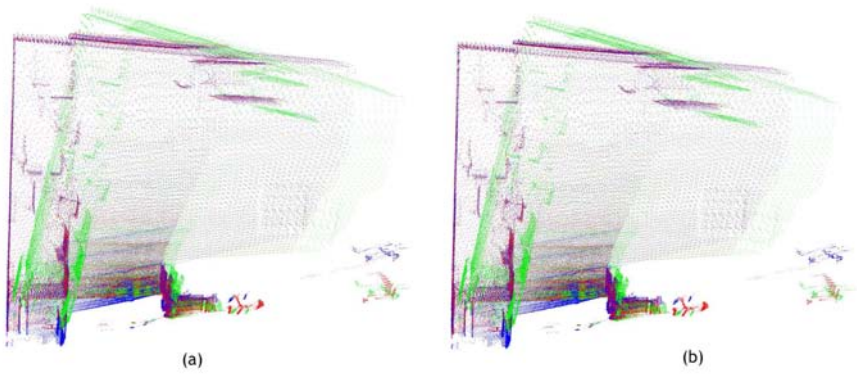
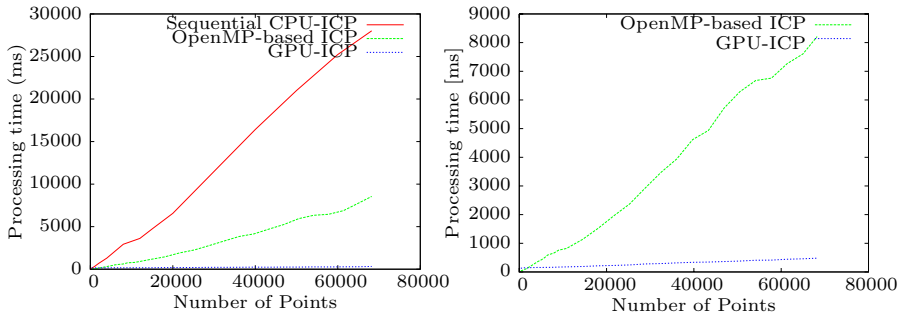


Fig. 5: Top view of the matched point clouds. Model point cloud is in red, scene in green, and transformed scene in blue. (a) Matching result of standard sequential CPU-ICP. (b) Matching result of GPU-ICP with single-element priority queue.



(a) Without kd-tree construction (b) Including kd-tree construction

Fig. 6: The runtime comparison among sequential CPU implementation, OpenMP-based implementation and GPU-ICP.

Table 1: Testing results of three ICP methods on a contemporary system.

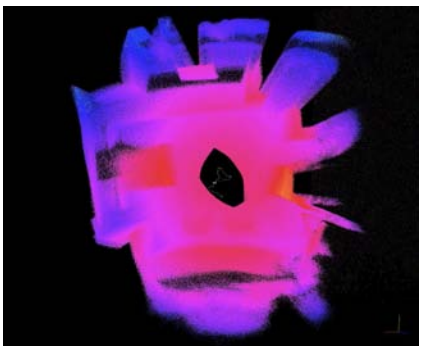
Sequential CPU-ICP	OpenMP-based ICP	GPU-ICP
13140 ms	2800 ms	260 ms

Experiment 2 – Indoor 3D Mapping. The proposed GPU-ICP algorithm is applied to an indoor mapping task. 180 frames are captured by an SR3000 Swiss Ranger Camera®, with resolution of 25344 points[4]. The mapping environment and the result of GPU-ICP are shown in Fig. 7 and the mapping performance is listed in Table 2. Total time includes file operation, data transfer and matching process. 1-2-3-4-5-6 means the queue length increases during the iterations from

[4] The data are taken from `http://www.robotic.de/242/`.

1 to 6. Since the number of points in every frame is less than that of *Experiment 1*, the speedup ratio of GPU-ICP is somewhat less (see Fig. 6b).

(a) Experimental environment.



(b) Mapping result of GPU-ICP.

Fig. 7: The mapping result is attained with increasing queue length from 1 to 6 and 130 iterations. Notice that the result is produced by the pure ICP process, without any preprocessing or refinement.

Table 2: The runtime of the mapping experiments.

Queue Length	Number of Iterations	Total Time (s)	ICP Time (s)
1	20	55.57	11.58
1	30	78.95	19.51
1	60	99.08	40.86
1	130	107.79	50.55
1-2-3-4-5-6	130	123.05	65.80
OpenMP-based ICP	130	463.78	454.30

5 Conclusion and Future Work

We developed a GPU-accelerated NNS approach that uses the k-d tree based priority search method. The performance is enhanced significantly by the massive parallelism of the GPU SIMD architecture. The GPU-NNS is applied in a 3D registration algorithm, GPU-ICP. When registering 68229 points, results show that a performance increase of up to 88 times can be obtained using a modern commercial video card. The resulting approach is meanwhile around 26 times faster than the MIMD-based ICP algorithm running on a dual-core CPU. The acceleration ratio of GPU-ICP to CPU implementations is greater when registering larger point clouds.

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

approach. In addition Nüchter et al. show that a larger bucket size improves NNS performance [9]. GPU-NNS could also increase the points in every leaf node. As for the GPU-ICP algorithm, more extensions can be implemented to make it a fully-fledged 3D registration algorithm, such as global relaxation, frustum culling and point reduction.

References

1. S. Arya and D. M.Mount. Algorithms for Fast Vector Quantization. In *Proc. Data Compression Conference*, pages 381–390. IEEE Computer Society Press, 1993.
2. B. Bustos, O. Deussen, S. Hiller, and D. Keim. A Graphics Hardware Accelerated Algorithm for Nearest Neighbor Search. In *Proc. of the 6th Int. Conf. on Computational Science*, pages 196–199, May 2006.
3. T. Foley and J. Sugerman. KD-Tree Acceleration Structures for a GPU Raytracer. In *Graphics Hardware*, pages 15–22, July 2005.
4. V. Garcia, E. Debreuve, and M. Barlaud. Fast k Nearest Neighbor Search using GPU. In *Proc. Comp. Vision and Pattern Recognition Workshops (CVPRW)*, pages 1–6, June 2008.
5. D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-D Tree GPU Raytracing. In *Proc. Symp. on Interactive 3D graphics and games*, pages 167–174, April 2007.
6. P. J.Besl and N. D.McKay. A Method for Registration of 3-D Shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, February 1992.
7. H. W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. AK Peters, July 2001.
8. M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A Fast Similarity Join Algorithm Using Graphics Processing Units. In *Proc. of the 24th IEEE International Conference on Data Engineering*, pages 1111–1120, May 2008.
9. A. Nüchter, K. Lingemann, J. Hertzberg, and H. Surmann. 6D SLAM with Approximate Data Association. In *Proc. of the 12th IEEE International Conference on Advanced Robotics (ICAR)*, pages 242–249, July 2005.
10. nVidia. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. nVidia, version 2.0 edition, June 2008.
11. T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon Mapping on Programmable Graphics Hardware. In M. Doggett, W. Heidrich, W. Mark, and A. Schillin, editors, *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*, 2003.
12. T. Rozen, K. Boryczko, and W. Alda. GPU bucket sort algorithm with applications to nearest-neighbour search. In *Journal of the 16th Int. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision*, February 2008.
13. S. Singh and P. Faloutsos. SIMD Packet Techniques for Photon Mapping. In *Proc. of the IEEE/EG Symposium on Interactive Ray Tracing*, pages 87–94, September 2007.
14. K. van Kooten, G. van den bergen, and A. Telea. *GPU Gems 3*, chapter 7, pages 123–148. Addison Wesley Professional, August 2007.
15. 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.