This CVPR2013 Workshop paper is the Open Access version, provided by the Computer Vision Foundation.
The authoritative version of this paper is available in IEEE Xplore.

GPU-SHOT: parallel optimization for real-time 3D local description

Daniele Palossi[a,b,*] Federico Tombari[b,†] Samuele Salti[b,†]
Martino Ruggiero[a,†] Luigi Di Stefano[b,†] Luca Benini[a,†]
[a]DEI, University of Bologna [b]DISI - University of Bologna
Bologna, Italy
*daniele.palossi@gmail.com †name.surname@unibo.it

Abstract

The fields of 3D computer vision, 3D robotic perception and photogrammetry rely more and more heavily on matching 3D local descriptors, computed on a small neighborhood of a point cloud or a mesh, to carry out tasks such as point cloud registration, 3D object recognition and pose estimation in clutter, SLAM, 3D object retrieval. One major drawback of these applications is currently the high computational cost of processing 3D point clouds, with the 3D descriptor computation representing one of the main bottlenecks. In this paper we explore the optimization for parallel architectures of the recently proposed SHOT descriptor [22] and of its extension to RGB-D data [23]. Even though some steps of the original algorithm are not directly suitable for parallel optimization, we are able to obtain notable speed-ups with respect to the CPU implementation. We also show an application of our optimization to 3D object recognition in clutter, where the proposed parallel implementation allows for real-time 3D local description.

1. Introduction

3D local descriptors are becoming a standard tool for several tasks related to disciplines such as computer vision, robotic perception, photogrammetry, computer graphics, given their ability to provide a repeatable, albeit compact, representation of a 3D surface. The main tasks that currently exploit such ability are point cloud registration and surface alignment, 3D object recognition and pose estimation in clutter, 6D SLAM, 3D object retrieval, the majority of which is also being pushed by the current advent of low-cost, real-time 3D sensors such as the Microsoft Kinect and the Asus Xtion. Given the importance of these tasks in many applications, several 3D local descriptors have been recently proposed in literature [6, 11, 15, 18, 28, 20, 22].

3D local descriptors are usually deployed in a matching framework with the aim of determining point-to-point correspondences between common parts of two different surfaces. They are computed on a set of points from both surfaces and matched via fast indexing schemes [3]. This set of points is usually obtained by means of a specific interest point detector or by means of uniform sampling of the point cloud. A common aspect among most 3D descriptors is the definition of a *support*, *i.e.* a local neighborhood around the point being described, usually determined by a sphere centered on the point. The support is used not only to determine the points which will be included in the description, but often also to compute a Local Reference Frame (LRF) [6, 22, 15, 18, 28].

One of the main disadvantages is using 3D local descriptors is the computational burden associated with most proposals. Indeed, the local description stage represents often the main bottleneck of application pipelines: this limits the applicability of this powerful tool, especially in those applications having real-time constraints. To tackle this problem, in this paper we explore the optimization of 3D local descriptors by means of general purpose GPU programming. We propose a GPU optimization aimed at achieving real-time processing on typical 3D datasets of a state of the art 3D descriptor, SHOT [22]. To this aim, we divide the SHOT computation (see Fig. 1) in several elementary steps, analyze their computing time requirements and propose a GPU optimization of every one of them, which will be outlined in more details in Section 5. As for the experimental evaluation (Sec. 6), we compare the proposed GPU optimization with the original CPU implementation to measure the obtained speed-up, and we also show some results of the proposed GPU-SHOT algorithm within a 3D object recognition pipeline to demonstrate the practical usefulness of our proposal.

2. Related work

The development of computer technology has recently led to an unprecedented performance increase of Graphical Processing Units (GPUs). Modern GPUs integrate hun-

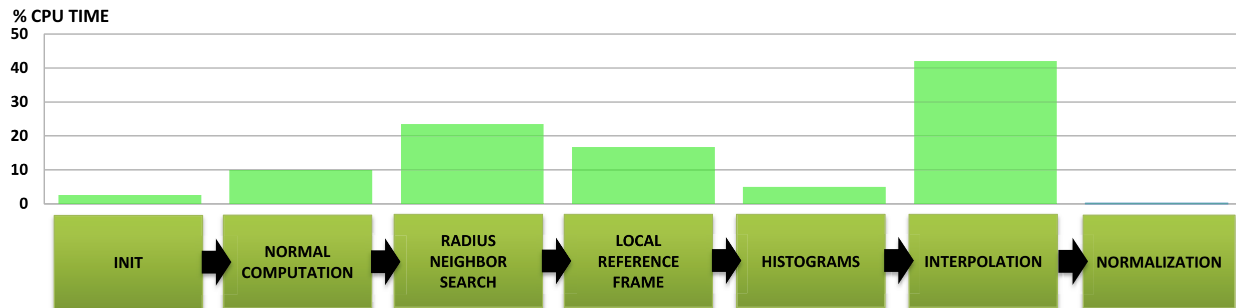

Figure 1: The elementary steps involved in the computation of the SHOT descriptor on each 3D point together with their computational weight (in % of the total CPU execution time) estimated by our profiling.

dreds of processors on the same device, communicating through low-latency and high bandwidth on-chip networks and memory hierarchies. Additionally, such scalable computation power and flexibility is delivered at a rather low cost by commodity GPU hardware. Besides hardware performance improvement, the programmability of GPUs has also been significantly increased in the last years [12]. Several GPU-accelerated image processing and computer vision libraries are already available in literature [7, 1, 9, 2]. Usually, they offer an OpenCV-like programming interface to easily port existing OpenCV applications, while taking advantage of the high level of parallelism and computing power available on recent GPUs. GPU acceleration has been applied to several computer vision tasks, such as segmentation [25], feature processing [4, 27], stereo imaging and vision [29, 30], machine learning and data processing [17, 8, 31], particle filtering [14], optical flow [16], edge detection [13].

However, only a few 3D descriptors have been already ported on GPUs. The Fast Point Feature Histograms [18] and Point Pair Features [5], for example, have parallelized versions already available in the PCL library [19]. However, almost all the mentioned GPU-accelerated computer vision algorithms use the CUDA programming interface, which strongly limits the portability and flexibility of the implementation. Though CUDA provides a general-purpose model for data parallelism as well as low-level access to hardware, only OpenCL provides an open, industry-standard framework which is supported by nearly all processor manufacturers including AMD, Intel, and NVIDIA, as well as others that serve the mobile and embedded computing markets. As a result, applications developed in OpenCL are theoretically portable across a wide variety of GPUs and CPUs. This scenario motivated our idea of porting and optimizing the 3D SHOT local descriptor [22] on GPU-based architecture leveraging an OpenCL implementation.

3. The SHOT descriptor

The SHOT descriptor [22] encodes a signature of histograms of topological traits. A 3D spherical grid of radius r, made out of 32 sectors, is centered at the 3D point being described and oriented according to a unique Local Reference Frame (LRF) which is invariant with respect to rotations and translations. The computation of the LRF is based on the EigenValue Decomposition of the distance-weighted covariance matrix of the points within the 3D grid. For each of the 32 spherical grid sectors a one-dimensional histogram with $b_S + 1$ bins is computed by accumulating the cosine of the angle between the normal at the keypoint and the normal of each of the points belonging to the spherical grid sector. The final descriptor is then formed by orderly juxtaposing all histograms together according to the canonical orientation provided by the LRF, its size being equal to $(b_S + 1) \cdot 32$ elements. To better deal with quantization effects, quadrilinear interpolation is applied to each accumulated element. Finally, to improve robustness with respect to point density variations, the descriptor is normalized to unit length. Fig. 1 reports the several steps involved in the computation of the SHOT descriptor. For more details concerning the SHOT descriptor we refer the reader to [22]. If color information is available at each 3D point (*e.g.* when using RGB-D sensors), an additional set of histograms can be computed [23], where the L_1 norm between the Lab triplet of the keypoint and that of each point of the spherical grid sector is accumulated in each histogram, quantized into $b_C + 1$ bins. The two sets of histograms, *i.e.* those associated to shape and those related to color, are joined together to form the final *color-SHOT* descriptor [23].

4. The Fermi GPU Architecture and OpenCL

The Fermi-based GPU used in this work is a Nvidia Tesla C2075, a two-level shared memory parallel machine comprising 448 SPs (Stream Processsors) organized in 16 SMs (Streaming Multiprocessors). SMs manage the execu-

tion of programs. All instructions are executed in a SIMD fashion, where one instruction is applied to all threads. This execution method is called SIMT (Single Instruction Multiple Threads). All threads in the same group execute the same instruction or remain idle (different threads can perform branching and other forms of independent work). One of the key architectural innovations that greatly improved both the programmability and performance of GPU applications is on-chip shared memory. In the Fermi architecture, each SM has 64 KB of on-chip memory that can be configured as 48 KB of shared memory with 16 KB of L1 cache or as 16 KB of shared memory with 48 KB of L1 cache. Fermi features also a 768 KB unified L2 cache which provides efficient data sharing across the GPU.

OpenCL (Open Compute Language) is an open industry standard for programming a heterogeneous collection of CPUs, GPUs and other discrete computing devices organized into a single platform. OpenCL programming involves running code on two different platforms: a host system that relies on one or more CPUs to perform calculations, and a OpenCL-enabled NVIDIA GPU (the device). The device works as a coprocessor to the host, so a part of the application is executed on the host and the rest, typically calculation intensive, on the device. The OpenCL framework fits the concepts of SP and SM, respectively, with WI (Work-Item) and WG (Work-Group).

5. GPU optimization

To carry out a GPU optimization of the SHOT descriptor, we first performed an accurate profiling, aimed at determining which elementary operations involved in the descriptor computation are computationally more relevant, as well as those that are more suited to parallelization. Fig. 1 reports the average % over the total measured CPU execution time associated to each elementary step of the SHOT descriptor. The three steps taking the longest time to complete are the search of the neighbors within the spherical support, the computation of the LRFs, and the quadrilinear interpolation.

It is worth pointing out that a preliminary task to be carried out previously to the descriptor computation concerns the setup of the OpenCL structures. In particular, the OpenCL program is created together with all kernel instances on which it will be executed. In addition, also the data structures (buffers) aimed at data transferring between CPU and GPU have to be allocated. However, this overhead has to be paid only once, at application start-up. We will now analyze all elementary step related to the SHOT descriptor, and describe the proposed specific optimization.

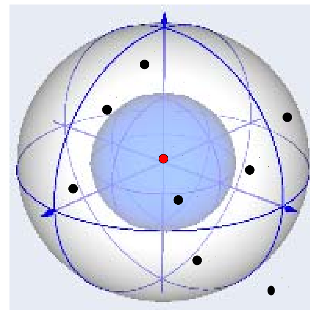

Figure 2: Spherical support built around the given feature point used during Radius Neighbor Search.

5.1. Normal computation

The normal computation kernel takes as input a 3D mesh. The computation of the normal vectors for each point of the mesh can be divided into two main stages. The first stage computes the normal vector for each polygon of the mesh, while the second one calculates the normal vector for each point. The normal vector at each point is obtained by weighting, according to the polygon areas, the polygonal normals on all polygons adjacent to each point which were calculated in the first step. Clearly, the computational time taken by the entire task is proportional to the number of points in the input cloud, however it is usually less than 10% of the overall description time even considering meshes with huge number of points (Fig. 1).

Our OpenCL implementation of the normal computation reflects the described steps since this task is split into two distinct OpenCL kernels: the first computes the polygonal normals, while the second one computes the normal at each point. In the first kernel, one OpenCL work-item has been allocated for each polygon normal calculation. The second one exploits one OpenCL work-item for the computation of the normal for each point in the mesh. This implementation choice has been mainly dictated by the lack of synchronization barrier between different work-groups in the OpenCL API. This becomes particular relevant in this case because, considering the second step, it is not guaranteed that all the polygonal normals are being computed by the work-items in the same work-group. Considering the data structures deployed in this step of the algorithm, all the information is stored in the global memory of the GPU.

5.2. Radius Neighbor Search

Given each Feature Point (FP), the radius neighbor search step determines the index of all points within the spherical support centered at the given FP (neighbors points) (see Fig. 2).

The OpenCL kernel implementing the radius neighbor search exploits a number of work-groups equal to the num-

ber of FPs. All points in the cloud, stored in the global memory, are used by every work-group. Considering each work-group, the computation is equally partitioned among the work-items considering the indexes of the points within the cloud. Each work-item scans the subset of the points whose index is a multiple of its local *id*. This way, it is unlikely that a single work-item will find a big amount of neighbours, because typically indexes of neighboring points are numerically close to the index of the central point (FP), and we obtain a more balanced parallel execution.

In order to perform the point search, two private memory buffers are allocated in the private memory and assigned to each work-item. These buffers are used to store temporary results of each work-item (*i.e*. index and distance of the found neighboring points). Additionally, each work-item updates a variable stored in the shared memory with its total number of found neighbors. This information is then used during the last step of the kernel, where all work-items move in parallel their results from the private to the global memory. This also avoids using further barrier synchronization or the delegation of all writes to one work-item only.

A last optimization is obtained by reducing the effective number of points examined by the radius search task. As previously said, the index of a point within the spherical support is typically close to the index of the FP. We can thus reduce the number of visited points, using only those closest to the FP's index (we take a sub-group both before and after the FP's index). Using such optimization, a performance improvement proportional to the size of the cloud has been obtained.

5.3. Local Reference Frame

This step computes a Local Reference Frame (LRF) for the current FP, first by computing the 3x3 covariance matrix of the spherical support centered on the FP, then by computing the EigenValue Decomposition of the matrix and disambiguating the sign of the three eigenvalues. Thanks to its iterative procedure, the covariance matrix computation is a good candidate for a parallel porting. The input to this substep is represented by the list of the neighboring points obtained in the previous radius search step, since each neighbor will contribute to build up the covariance matrix.

The covariance matrix computation kernel is executed by a number of work-groups equal to six times the number of the FPs, since each work-group computes one of the six elements of each covariance matrix (exploiting the symmetric property of the covariance matrix). Every work-item within the work-group uses two local memory buffers for temporary partial data. The final 3x3 matrix, which is stored in the global memory shared by all work-groups, is generated by reducing the temporary buffers exploiting Warp Synchronization [26]. This technique avoids to perform the final sum of the partial results with only one work-item, thus maximizing performance of the parallel execution.

After the covariance matrix computation, the LRF is computed by taking two eigenvectors of the matrix as two reference axes and performing sign disambiguation of each axis by selecting the hemisphere on which the majority of neighboring points falls [22]. We remind here that, according to the available implementation, in case the density of the two hemispheres is equivalent, a different disambiguation procedure is applied based only on 5 neighboring points. The third reference axis is then computed as the cross product of the other two.

A number of work-groups equal to the number of FP is used for this second substep. The OpenCL implementation presents two particular design choices: the eigenvector computation is assigned to a single work-item for each work-group; the second disambiguation (which is rarely needed and represents anyway a light computational task) is performed on the host side. Two structures are allocated in local memory to store the temporary results of the first disambiguation procedure applied for the two axes. These are then reduced with the same warp synchronization technique used for the covariance matrix computation.

5.4. Histogram Computation

In this stage, a spherical grid of 32 sectors (see Fig. 2) defines 32 volumes in the 3D space, each one used to compute a histogram. Every point falling within each volume contributes to the histogram associated to that volume. The specific contribution brought in by each point is related to its normal (in case only the 3D shape information is deployed) or also to the color triplet associated with the point (in case the color SHOT descriptor is computed, exploiting RGB-D data). We will refer to these two sources of information, respectively, as shape channel and color channel. From a memory footprint viewpoint, there is an important difference between the two channels, since the number of bins associated to a color SHOT histogram is usually three times that associated to a shape-only SHOT histogram. In any case, the task allocates a work-group for each FP and uses the maximum number of work-items allowed by the GPU used. The results are stored in global memory via coalesced accesses, thus exploiting the high bandwidth provided by the memory interface.

5.5. Quadrilinear Interpolation

Following the computation of the histograms associated to the SHOT descriptor, there's an interpolation stage where the contribution brought in by each point within the spherical support is distributed to neighboring histograms along the 4 domains represented by the azimuth, elevation and radial spatial directions, and binning domain of each histogram (for more details about this stage we refer the reader

Figure 3: The three clouds used for the experiments in Subsection 6.1: from left to right, Happy Buddha, Dragon, Mario.

to [22]).

This is the heaviest step in term of computational complexity of the original SHOT implementation. The algorithm has to: find the 3D grid sub-sectors involved in the redistribution (three interpolations), compute the interpolation weights, add each weight to the correct histogram and finally compute and add the interpolation weight between adjacent bins of the current histogram (fourth interpolation). The OpenCL implementation of this step is based on two mutually-exclusive kernels, i.e. either single-channel (shape-only SHOT) or double-channel (the color SHOT descriptor) interpolation. The allocated work-groups are as many as the FPs, with a number of work-items equal to the warp size. This configuration is due to the amount of private memory used to store temporary data. A higher number of work-items will indeed generate issues with the size of the available memory. Every work-item executes the interpolation of a sub-set of the neighborhood. The final write operation to global memory is coalescent: all work-items write consecutive locations of the memory and shift a position after every operation.

5.6. Normalization

The final step applies L_2 normalization to all elements of the descriptor. In case the color SHOT descriptor is computed, two separate normalization procedures are applied, one on the shape histograms, another one on the color histograms.

The OpenCL parallelization uses a number of work-groups equal to the number of FPs. The L_2 norms are obtained cooperatively by all group-items storing the partial results in an area of local memory sized as the work-group elements. When every item has finished its task, a reduction is performed with the Warp Synchronization technique. The final normalization is computed by one work-item for each element of the descriptor. The final results is stored in global memory with coalescent accesses.

Table 1: GPU configuration used for the experiments in Subsection 6.1

Model Name	Tesla C2075
# of CUDA Cores	448
Frequency of CUDA Cores	1.15 Ghz
Dedicated Memory	6 GB GDDR5
Memory Speed	1.5 Ghz
Memory Interface	384-bit
Memory Bandwidth	144 GB/sec

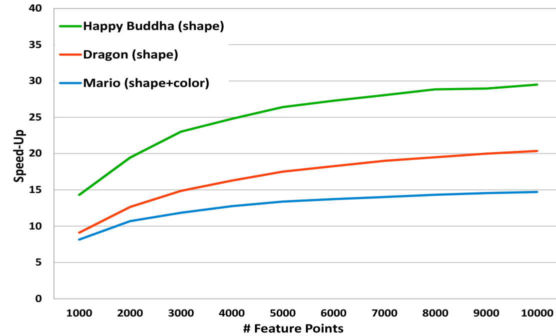

Figure 4: Exp. 1: speed-up yielded by the proposed GPU optimization over the CPU implementation encompassing all stages of the SHOT descriptor.

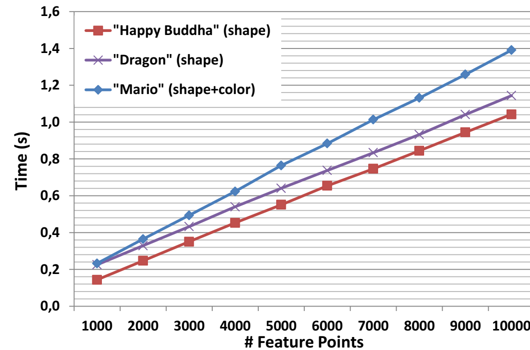

Figure 5: Exp. 1: execution times of the proposed GPU optimization on the three point clouds.

6. Experimental Results

We tested our parallel implementation against the sequential implementation of SHOT made available by its authors[1]. We performed experiments to measure the speed-up brought in by the proposed parallel implementation, and experiments that show how the parallel implementation enables real-time description for the task of object detection in RGB-D data.

[1] http://vision.deis.unibo.it/SHOT/

6.1. GPU vs. CPU evaluation

In this subsection, we present the evaluation of the proposed GPU-based parallel implementation against the original CPU-based sequential implementation of the SHOT descriptor. We provide quantitative results by measuring the total speed-up, computed as the ratio between the CPU measured execution time and the GPU one. The specific GPU configuration employed is reported in Table 1, which is compared with an Intel Xeon 2.13GHz Quadcore CPU equipped with 8192 Kb of cache and 20 GB of system memory. We propose four different experiments, which are based on 2 point clouds (Happy Buddha, Dragon, composed of, respectively, 32328 and 100250 points) taken from the Stanford 3D Scanning Repository[2] and 1 point cloud (Mario, 90190 points) taken from the Bologna dataset [22]. The three point clouds are shown in Fig. 3. On the first two clouds we compute the only-shape SHOT descriptor, whereas on the third one we compute the color extension of the SHOT descriptor [23]. The color SHOT descriptor, which computes histograms also for the RGB channels, is computationally more demanding.

In Experiment 1, we have measured the total speed-up obtained by the proposed GPU optimization over the serial CPU implementation. Results are shown in Fig. 4 for different numbers of feature points on each point cloud, ranging from 1000 to 10000. These results demonstrate the effectiveness of the proposed optimization, with speed-ups of at least 7-8 with 1000 feature points and ranging up to at least 15 for 10000 feature points. Although the RGB-D version seems to benefit less from the parallel optimization, this should instead be ascribed to the smaller size of the Mario point cloud with respect to the other two. Additionally, Fig. 5 reports the measured execution times of GPU-SHOT on the three evaluated point clouds, showing that the time to compute standard sizes of descriptor sets (i.e. between 1000-5000 descriptors) is always greatly below $1s$, even in the case of color SHOT.

In Experiment 2, instead, we analyzed the GPU vs. CPU speed-up for each single elementary step associated with the SHOT descriptor. Results are reported in Fig. 6 and are averaged over 1000 descriptors computed on the Mario point cloud. The elementary steps that mostly benefit from the GPU implementation are the radius search, the histogram binning and the interpolation step. Notably, two of these three steps, *i.e.* radius search and interpolation, were two of the three most computationally intensive steps of the original CPU implementation (see Fig. 1).

In Experiment 3 we analyzed how the GPU time concerning each SHOT elementary step varies on different point clouds and by extracting different amounts of descriptors (*i.e.* 1000 and 10000). In this experiment we evaluate two point clouds: Happy Buddha (shape channel) and Mario (shape and color channels). Results are shown in Fig. 7. As it can be seen from the pie charts, a higher number of points being described reduces the relative computational weight of keypoint-independent stages such as initialization and normal computation which are prominent when computing a small number of descriptors, while on the other hand it proportionally increases the relative burden associated to the keypoint-dependent computational stages. In particular, LRF computation, radius search and interpolation become the most relevant stages with 10000 feature points.

[2] http://graphics.stanford.edu/data/3Dscanrep/

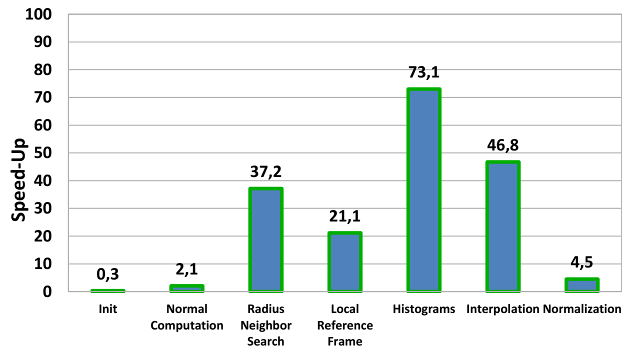

Figure 6: Exp. 2: GPU vs. CPU speed-ups relative to each SHOT elementary step averaged on 1000 descriptors computed on the Mario point cloud.

Finally, Experiment 4 aims at comparing the GPU and CPU SHOT implementations not in terms of efficiency but in terms of accuracy for the goal of descriptor matching. Hence, a different version of each of the three evaluated point clouds has been created by randomly rotating them and by adding on each point white Gaussian noise with σ equal to 0.3 times the cloud resolution. Each cloud pair has been then matched by computing the Euclidean distance between SHOT descriptor sets computed on each cloud. Fig. 8 reports the *Precision-vs.-Recall* curves yielded by descriptor matching on each cloud pair. The GPU and CPU versions are basically equivalent in all tested cases, the GPU version exhibiting even slightly improved performance with respect to the serial implementation.

6.2. 3D object recognition in clutter with GPU-SHOT

As an additional experiment, we have plugged in the proposed GPU optimization of the SHOT descriptor in a recently proposed pipeline for RGB-D object recognition in clutter [24]. The pipeline exploits both depth and color information coming from a RGB-D sensor and handles multiple instances of the same model simultaneously present in the scene. The model library is represented by a set of different RGB-D views taken around the models. As for the

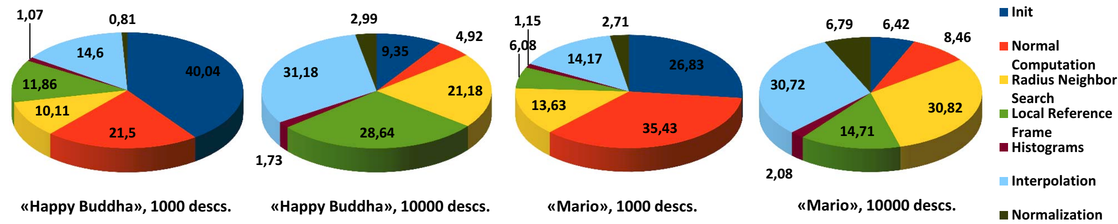

Figure 7: Exp. 3: percentages of total GPU time spent along the several SHOT elementary steps for different clouds (Happy Buddha, Mario) and different numbers of computed descriptors (1000, 10000).

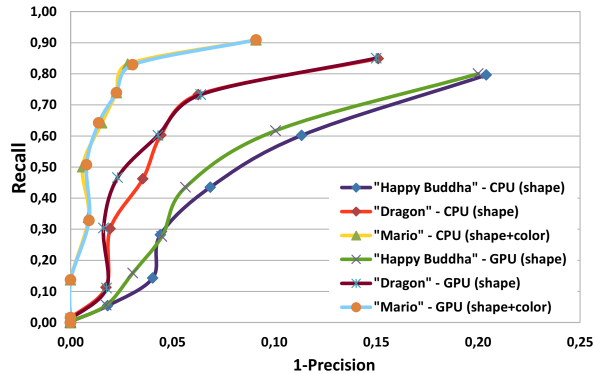

Figure 8: Exp. 4: Precision-Recall curves yielded by matching descriptors over rotated and noisy versions of the evaluated point clouds using the GPU and the CPU SHOT implementations.

first step, a set of keypoints are extracted from the scene and each model view by means of the SURF detector applied on the color frame; then, they are described by means of the color-enhanced version of SHOT [23]. After the description stage, point-to-point correspondences between the scene keypoints and the views in the model library are attained by means of descriptor matching based on *kd-trees*. Next, a 3D Hough Voting stage [21] followed by RANSAC is applied to cluster together subsets of geometrically coherent keypoints, discarding outliers. For each model, the view with the highest number of remaining correspondences is selected as the *best view*: if this number is higher than a threshold (set to 5 in our experiment), the 6-Degree-Of-Freedom (6DOF) pose aligning the best model view to the scene is computed via Absolute Orientation [10] on the validated correspondences.

The proposed optimization did not result in a loss of accuracy of the pipeline, whose good detection performance are retained even when deploying GPU-SHOT. Qualitative samples of the recognition capabilities of the pipeline when used in conjunction with the proposed parallel implementation of SHOT are provided in Fig. 9. GPU-SHOT allows for real-time feature description within such experiment: on average it describes 1500 SURF keypoints in 180 ms, *i.e.* at 5.5 frame/s. The overall pipeline still takes some seconds to go through all steps, mainly because of detected feature re-projection from 2D to 3D data, which in turn involves a search for a point nearest neighbor, and the matching of the scene descriptors with the model library. Both steps call for the relative GPU optimizations, to obtain a full real-time pipeline.

7. Conclusion and Future Works

This paper has proposed GPU-SHOT, the GPU optimization of the various stages building up the SHOT descriptor. The proposed optimization yields notable speed-ups with respect to the sequential CPU version, pushing the use of this descriptor in scenarios based on tight computational constraints. In addition, several steps analyzed and optimized by the proposed implementation are common to other state-of-the-art proposals, *i.e.* normal computation, radius neighbor search and covariance matrix computation. Hence, the proposed optimization could be further deployed to inspire and enable the parallel optimization of other 3D descriptors.

References

[1] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan. GpuCV: an opensource GPU-accelerated framework for image processing and computer vision. In *ACM Multimedia*, pages 1089–1092, 2008. 2

[2] P. Babenko and M. Shah. MinGPU: a minimum GPU library for computer vision. *Journal of Real-Time Image Processing*, 3(4):255–268, 2008. 2

[3] J. Beis and D. Lowe. Shape indexing using approximate nearest-neighbour search in high dimensional spaces. In *Proc. of Comp. Vision and Pattern Recog. (CVPR)*, pages 1000–1006, 1997. 1

[4] N. Cornelis and L. Van Gool. Fast scale invariant feature detection and matching on programmable graphics hardware. In *Proc. of Comp. Vision and Pattern Recog. Workshops (CVPRW)*, pages 1–8, 2008. 2

[5] B. Drost and S. Ilic. 3D object detection and localization using multimodal point pair features. In *3DIMPVT*, pages 9–16, 2012. 2

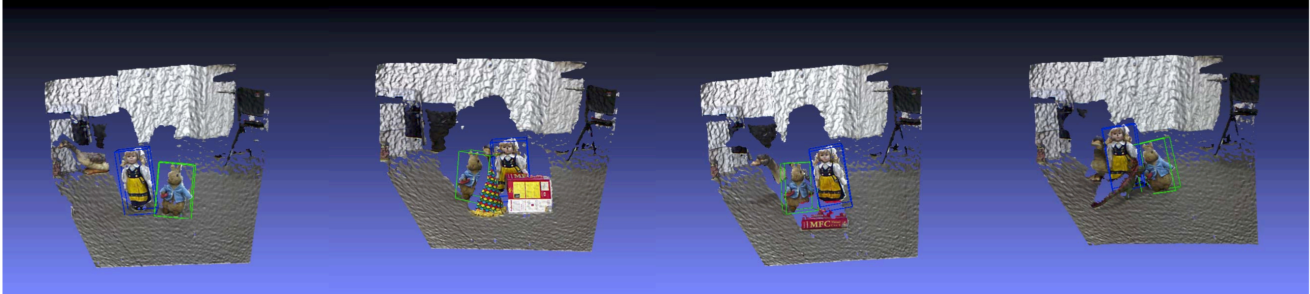

Figure 9: 3D object recognition in clutter with GPU-SHOT: the two models (*bunny*, green bounding box, and *doll*, blue bounding box) are correctly recognized and localized despite the presence of clutter and occlusions

[6] A. Frome, D. Huber, R. Kolluri, T. Bülow, and J. Malik. Recognizing objects in range data using regional point descriptors. In *Proc. of the European Conference on Computer Vision (ECCV)*, volume 3, pages 224–237, 2004. 1

[7] J. Fung and S. Mann. OpenVIDIA: parallel GPU computer vision. In *ACM Multimedia*, pages 849–852, 2005. 2

[8] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using GPU. In *Proc. of Comp. Vision and Pattern Recog. Workshops (CVPRW)*, pages 1–6, 2008. 2

[9] GPU4Vision project. http://gpu4vision.icg.tugraz.at. Accessed: 2013-04-29. 2

[10] B. Horn. Closed-form solution of absolute orientation using unit quaternions. *J. Optical Soc. of America A*, 4(4), 1987. 7

[11] A. Johnson and M. Hebert. Using spin images for efficient object recognition in cluttered 3D scenes. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21(5):433–449, 1999. 1

[12] Khronos OpenCL Working Group. The OpenCL Specification, version 1.0.29. http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf. Accessed: 2013-04-29. 2

[13] Y. Luo and R. Duraiswami. Canny edge detection on NVIDIA CUDA. In *Proc. of Comp. Vision and Pattern Recog. Workshops (CVPRW)*, 2008. 2

[14] O. Mateo Lozano and K. Otsuka. Real-time visual tracker by stream processing. *Journal of Signal Processing Systems*, 57:285–295, 2009. 2

[15] A. S. Mian, M. Bennamoun, and R. A. Owens. On the repeatability and quality of keypoints for local feature-based 3D object retrieval from cluttered scenes. *Int. J. Comput. Vision*, 89(2-3):348–361, 2010. 1

[16] Y. Mizukami and K. Tadamura. Optical flow computation on compute unified device architecture. In *Image Analysis and Processing (ICIAP). 14th International Conference on*, pages 179–184, Sept. 2

[17] N. Pinto, D. Doukhan, J. J. DiCarlo, and D. D. Cox. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLoS Comput Biol*, 5(11):e1000579, 2009. 2

[18] R. Rusu, N. Blodow, and M. Beetz. Fast point feature histograms (FPFH) for 3D registration. In *Proc. of the Int. Conf. on Robotics and Automation (ICRA)*, 2009. 1, 2

[19] R. Rusu and S. Cousins. 3D is here: Point Cloud Library (PCL). In *Proc. Int. Conf. Robot. Autom. (ICRA)*, 2011. 2

[20] J. Sun, M. Ovsjanikov, and L. Guibas. A concise and provably informative multi-scale signature based on heat diffusion. In *Proc. Symp. Geom. Proc.*, 2009. 1

[21] F. Tombari and L. Di Stefano. Object recognition in 3D scenes with occlusions and clutter by Hough voting. In *Proc. Pacific-rim Symposium on Image and Video Technology (PSIVT)*, 2010. 7

[22] F. Tombari, S. Salti, and L. Di Stefano. Unique signatures of histograms for local surface description. In *Proc. of the European Conf. on Comp. Vision (ECCV)*, 2010. 1, 2, 4, 5, 6

[23] F. Tombari, S. Salti, and L. Di Stefano. A combined intensity-shape descriptor for texture-enhanced 3D feature matching. In *Proc. of the Int. Conf. on Image Processing (ICIP)*, 2011. 1, 2, 6, 7

[24] F. Tombari, S. Salti, and L. Di Stefano. RGB-D object recognition and localization with clutter and occlusions. In *Proc. RGB-D Workshop on 3D Perception in Robotics in conj. with euRobotics Forum*, 2011. 6

[25] V. Vineet and P. J. Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *Proc. of Comp. Vision and Pattern Recog. Workshops (CVPRW)*, volume 0, Los Alamitos, CA, USA, 2008. IEEE. 2

[26] OpenCL programming for the CUDA architecture. http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingOverview.pdf. 4

[27] C. Zach, D. Gallup, and J. M. Frahm. Fast gain-adaptive KLT tracking on the GPU. In *Proc. of Comp. Vision and Pattern Recog. Workshops (CVPRW)*, 2008. 2

[28] A. Zaharescu, E. Boyer, and R. Horaud. Keypoints and local descriptors of scalar functions on 2D manifolds. *Int. J. Comput. Vision*, 100(1):78–98, 2012. 1

[29] K. Zhang, J. Lu, G. Lafruit, R. Lauwereins, and L. Van Gool. Real-time accurate stereo with bitwise fast voting on CUDA. In *Proc. of Int. Conf. on Comp. Vision Workshops (ICCVW)*, pages 794–800, 2009. 2

[30] Y. Zhao and G. Taubin. Real-time stereo on GPGPU using progressive multi-resolution adaptive windows. *Image Vision Comput.*, 29(6):420–432, 2011. 2

[31] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time KD-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 papers*, pages 126:1–126:11, 2008. 2