Real-time 3D semi-local surface patch extraction using GPGPU: Application to 3D object recognition

Sergio Orts-Escolano Vicente Morell Jose Garcia-Rodriguez Miguel Cazorla
Robert B. Fisher

Abstract

Feature vectors can be anything from simple surface normals to more complex feature descriptors. Feature extraction is important in order to solve various computer vision problems: e.g. registration, object recognition and scene understanding. Most of these techniques cannot be computed online due to their complexity and the context where they are applied. Therefore computing these features in real-time for many points in the scene is impossible. In this work a hardware-based implementation of 3D feature extraction and 3D object recognition is proposed in order to accelerate these methods and therefore the entire pipeline of RGBD based computer vision systems where such features are typically used. The use of a GPU as a General Purpose processor (GPGPU) can achieve considerable speed-ups compared with a CPU implementation. In this work advantageous results are obtained using the GPU to accelerate the computation of a 3D descriptor based on the calculation of 3D semi-local surface patches of partial views. This allows descriptor computation at several points of a scene in real-time. Benefits of the accelerated descriptor have been demonstrated in object recognition tasks. Source code will be made publicly available as contribution to the Open Source Point Cloud Library (PCL).

The final publication is available at
`http://www.springer.com/alert/urltracking.do?id=L41fe935Mdd48a5Sb0d354a`

1 Introduction

In recent years, the number of applications concerned with 3D data processing has increased considerably due to the emergence of cheap 3D sensors capable of providing a real time data stream. The Kinect device[1] and other range cameras as the Asus Xtion or the Carmine from PrimeSense are examples of these devices. Besides providing 3D information, these devices can also provide color information of the observed scene. The availability of real-time 3D streams has provided a key resource for solving many challenging problems as full DoF (Degrees of Freedom) tracking of a hand [1], 3D reconstruction and interaction [2] and dense visual SLAM [3]. Additionally, as these systems are becoming more complex, huge computational resources are demanded, especially when hard real-time constraints are required.

With the advent of the GPU as a General Purpose Graphic Processing Unit (GPGPU) some methods related to 3D data processing have been implemented on GPUs in order to accelerate them. Examples of these can be found in the calculation of feature descriptors and keypoint extraction on the GPU. In [4], the GPU performs curvature estimation of 3D meshes in real time. The work presented in [5], provides a parallel implementation using the GPU and the CUDA language from NVIDIA to accelerate the computation of Histograms of Oriented Gradient (HoG) features. In [6], a Point Feature Histograms (PFH) GPU implementation is proposed allowing its computation in real-time on large point clouds. A real-time GPU-based method for patches extraction is presented in [7]. These surfaces patches with associated uncertainties are extracted by means of Kinect cameras. Despite the methods mentioned above, which demonstrate the feasibility of the GPU for 3D feature extraction, there are still comparatively few methods implemented with respect to all those currently prevalent in the state of the art. Most 3D Local Shape Descriptors (LSDs) have not been implemented yet on the GPU. It can also be noted that the integration of these methods in complete systems, that require real-time constraints, is still very low. Kinect Fusion [2] has been one of the first works where the GPU has been used as the main core processor, allowing the reconstruction of 3D scenes in real-time.

Other motivations for this work include the existing gap of 3D object recognition solutions based on models that support real-time constraints. Until now, most of the proposed works that supported real-time constraints are view-based. For example in [8], a local feature descriptor for RGB-D images was proposed. This descriptor combines color and depth information into one representation. However, 3D information possibilities are still negligible, it depends mostly on textures and illumination of the specific scene. In [9], a combined descriptor formed by 3D geometrical shape and texture information is used to identify objects and its pose in real-time. The proposed system is accelerated using GPU achieving real-time processing. However, 3D information is only

[1] Kinect for XBox 360: http://www.xbox.com/kinect Microsoft

used to extend shape information and considerably relies on texture information, making it sensitive to scene illumination conditions.

Model-based approaches are less sensitive to illumination, shadows, and occlusions of the scene, allowing more robust object recognition systems. They also improve pose estimation. However, they also have some drawbacks: first, model-based descriptors usually require noise-free and dense data, which is not common in range cameras. Moreover, these kind of systems require a large number of descriptors to be calculated and also their correspondences in the database have to be found at runtime. Consequently, most of them have not been processed in real-time.

In the following paragraphs a brief review of some LSDs is presented.

One of the local shape descriptors that first appeared was Spin images [10]. It is a regional point descriptor used to characterize the shape properties of a 3D object with respect to a single oriented point. This descriptor constructs 2D histograms using the tangent plane and the surface normal passing through the reference point. For all the neighbouring points within a certain distance, the distance to the tangent plane and the perpendicular distance to the normal line are computed. Since it is 2D and not 3D it does lose some discriminative power and leads to many ambiguous matches. Another descriptor that was proposed in the beginning of nineties (also sensitive to noise and occlusions) is the SPLASH [11] representation.

The Point Signature LSD proposed in [12] is defined as a 1D array based on the distance profile of the intersection of a sphere with the object from the tangent plane. Other model-based descriptors are Spherical Spin Images [13] and Surface Signatures [14].

In [15], the Pairwise Geometric Histograms (PGHs) are proposed for object recognition tasks. PGHs are similar to 2D histogramas constructed by Spin Images but PGHs are constructed using the relative angle between the normal of the reference facet and the normal of neighboring facets. Moreover, it also adds to the histogram the perpendicular distance between facets.

Local Surface Patches (LSPs) [16] are another local shape descriptor constructed as 2D histograms. This is constructed using the angle between the surface normal of neighbouring points and the surface normal of the reference point. Moreover, it adds the *Shape Index*, which is defined as the curvature of the surface.

Among most recent works is the Variable Dimensional Local Shape Descriptor (VDLSD) [17]. VDLSD is based on the extraction of invariant features extracted from the Principal Component Space of the local neighborhood around that point. Eigenvalue decomposition is used to associate an orthonormal frame $(\vec{i}, \vec{j}, \vec{k})$, and the three Eigenvalues. Using these vectors and scalars a variety of histograms are constructed. These contain various levels of geometric information. As much descriptive information is available, this research finds the subset that best defines each model that is to be later recognized. Moreover, points where the VDLSD is calculated are selected using a random criteria which is inefficient due to the amount of points that need to be computed.

Although many of the presented model-based works are capable of dealing with background clutter and noise, their computational complexity is often very high. Another different approach from those presented above and based on global point cloud information instead in local shape is Potential Well Space Embedding (PWSE) [18]. The PWSE is based on the ICP algorithm and it uses local minima to embed the pose of an object into a small set of feature vectors. Although this method runs in real-time, it still requires significant preprocessing steps and a good previous segmentation.

A more extensive review of free-form 3D object representation and recognition techniques can be found in [19] and [20].

The reviewed 3D model-based descriptors are the most common techniques applied to free-form object recognition using LSDs, but in this work we are going to focus on a 3D tensor-based representation proposed by [21]. It is also based on local shape information and it has demonstrated good performance in free-form object recognition even under significant noise and background clutter. Furthermore, this work also presented a novel hash technique to organize descriptors in an efficient way.

Although the 3D tensor-based representation has also a high computational cost, in contrast with other methods it is highly parallelizable. This makes possible that most of its processing steps can take advantage of modern GPU architectures to accelerate its computation. This work goes a step further in the application of massively parallel processor architectures, such as the GPU, to 3D data processing tasks. It will help running complex 3D vision systems under real-time constraints.

The 3D data used in this work has been obtained mainly as RGB-Depth maps provided by the Kinect sensor. Advantageous results have been obtained using the GPU for accelerating the extraction of a feature descriptor based on the calculation of 3D tensors. This descriptor is computationally expensive to compute on the CPU and also requires some preprocessing steps for its computation: normals estimation, surface triangulation and keypoint detection. Since these preprocessing steps are common for a large number of applications, their acceleration and integration in the pipeline of a GPU architecture becomes essential to make progress in 3D data real-time processing.

The paper is structured as follows: in Section 2 the proposed descriptor to be accelerated is presented. In the following Subsections 2.1 and 2.2 the GPU implementation of pre-processing steps are detailed. In Section 2.3

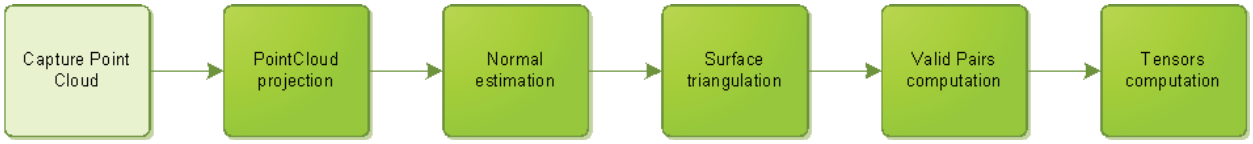

Figure 1: General system overview. Steps coloured in dark are computed on the GPU.

a GPU-based tensor extraction implementation is described. Next, in Section 3, performance results are shown for different steps and hardware configurations. Finally, Section 4 presents a real object recognition application with time constraints to validate our implementation, followed by our main conclusions and future work.

2 GPU-based tensor extraction algorithm

The feature descriptor proposed for implementation on the GPU is based on the descriptor introduced in [22]. This descriptor is based on the calculation of semi-local surface patches obtained by a range camera. It has been used successfully for different applications like global registration and 3D object recognition, where other descriptors like Spin Images [10] or Geometric Histograms [23] obtained worse results. Additionally, in [21] it is demonstrated how this descriptor can be successfully applied in object recognition problems with high levels of occlusion. The main problem of this descriptor is its high computational cost, which is prohibitive for running in a conventional CPU under real-time constraints.

This descriptor extracts a semi-local model of the scene, computing semi-local features that assist the local object recognition even under conditions of occlusion. This feature is referenced in the following sections as a tensor. A tensor is defined by the surface of the model that is intersected by each voxel on a centered grid. This set of values defines a third order tensor.

To compute the required descriptor some preprocessing steps are necessary. These steps have also been implemented on the GPU in order to accelerate the entire pipeline. Therefore it has been necessary to implement on the GPU the following processes: depth map and color map transformation to a coloured point cloud, noise removal, normal estimation and surface reconstruction. A general system overview for semi-local surface patch extraction is shown in Figure 1. Figure 2 shows the different steps required prior to the extraction of the tensor and their 3D visualization after each step. As we can appreciate, most steps are computed on the GPU taking advantage of parallel computing power of the GPU and avoiding transfers between CPU and GPU after each step. Pseudo-code of the entire GPU-based tensor extraction algorithm is presented in Algorithm 1. Moreover, pseudo-code snippets for all GPU-based preprocessing steps are presented in next sections.

input : A depth map M_d of size 640×480
output: A set of 3D tensors $T = \{t_0, t_1, t_2, ..., t_N\}$ that describe the input data

Depth map is transferred to the GPU memory;
cudaMemCopyHostToDevice(d_M_d, M_d);
$d\_M_{filtered} \leftarrow$ gpuBilateralFiltering(d_M_d);
$d\_P_{xyz} \leftarrow$ gpuPointCloudProjection($d\_M_{filtered}$);
$d\_N_{xyz} \leftarrow$ gpuNormalEstimation(d_P_{xyz});
$d\_Tri \leftarrow$ gpuSurfaceTriangulation(d_P_{xyz}, d_N_{xyz});
$d\_V_{pairs} \leftarrow$ gpuValidPairsComp(d_P_{xyz}, d_N_{xyz});
cudaMemCopyDeviceToHost(V_{pairs}, d_V_{pairs});
for $i \leftarrow 0$ **to** $|V_{pairs}|$ **do**
    tensors are computed in parallel at thread level and tensor level;
    AsyncGpuTensorComp(v_i,d_P_{xyz},d_N_{xyz},d_Tri, d_T);
end
cudaMemCopyDeviceToHost(T, d_T);

Algorithm 1: Pseudo-code of the GPU-based 3D tensor extraction algorithm. $d\_$ prefix means that variable is allocated in the GPU memory.

2.1 RGB-D processing on the GPU

In this work we focus on the processing of 3D information provided by the Kinect sensor. This processing is performed on the GPU with the aim of achieving a real-time implementation.

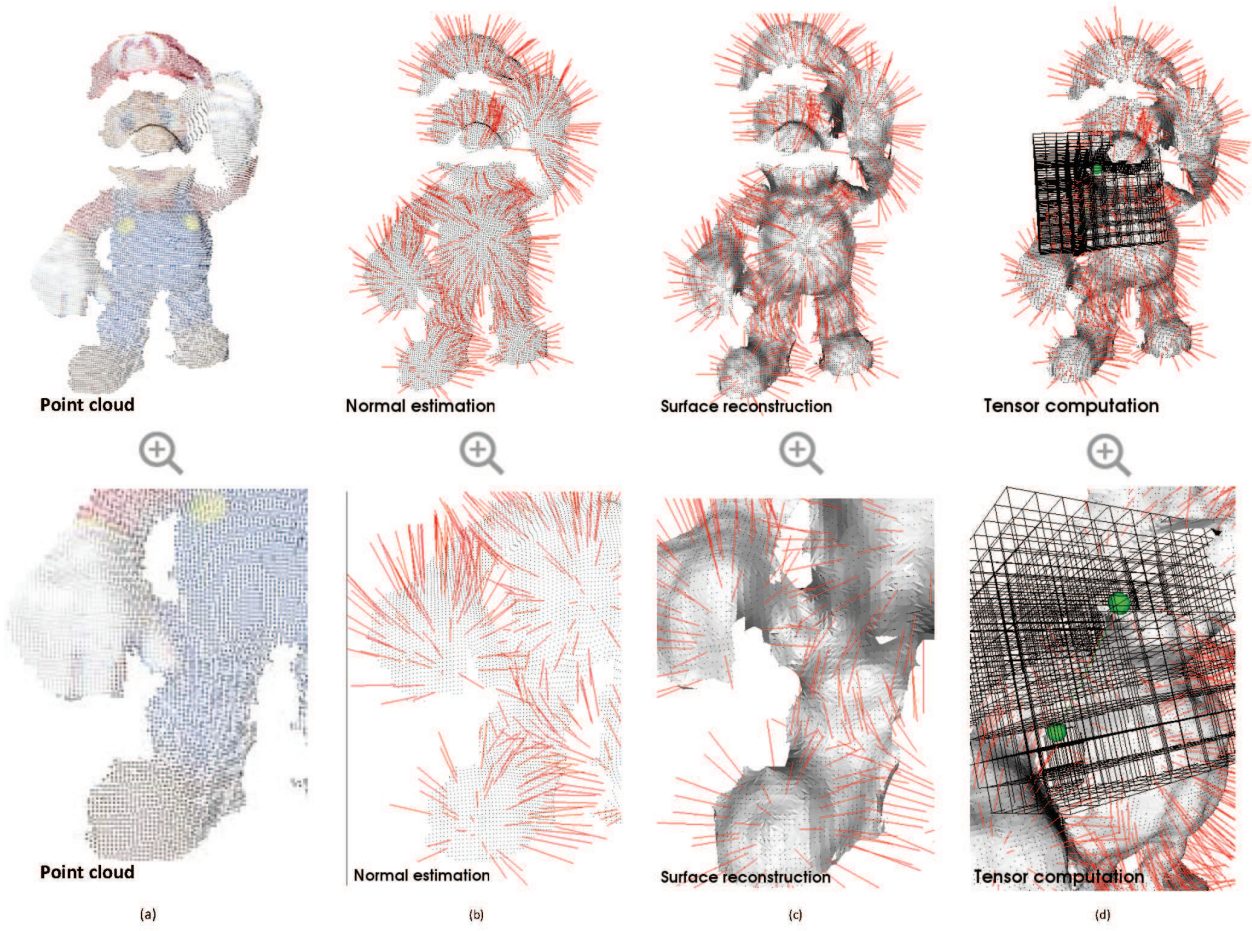

Figure 2: (a) Point cloud obtained after transforming depth and color maps provided by the Kinect sensor. (b) Normal estimation. (c) Surface reconstruction. (d) Feature descriptor extraction: 3D tensor computed over a partial view

The overall goal is the implementation of systems that offer interaction with the user. The Kinect sensor provides a RGB map with color information M_c and a disparity map M_d. The first step to carry out on the GPU with the aim of accelerating future steps is the projection of the depth and color information in a three-dimensional space, where the depth and colour information is aligned, allowing the production of a coloured point cloud that represents the scene, Figure 3.

The relationship between a disparity map provided by the Kinect sensor and a normalized disparity map is given by $d = 1/8 \cdot (d_{off} - kd)$, where d is the normalized disparity, kd is the disparity provided by the Kinect and d_{off} is a particular offset of a Kinect sensor. Calibration values can be obtained in the calibration step [24]. In this way the relationship between depth and a disparity map is given by the following equation:

$$z = \frac{b \cdot f}{1/8 \cdot (d_{off} - kd)} \tag{1}$$

where b is the baseline between the infrared camera and the RGB camera (in meters), and f is the focal distance of the cameras (in pixels). Once the depth map M_d is obtained calculating the depth z for all points, the projection of each point in 3D space is given by:

$$\begin{aligned} p_x &= z \cdot (x - x_c) \cdot 1/f_x \\ p_y &= z \cdot (y - y_c) \cdot 1/f_y \\ p_z &= z \end{aligned} \tag{2}$$

where $p \in R^3$, x and y are the row and the column of the projected pixel, x_c and y_c are the distances (in pixels) to the map centre and f_x and f_y are the focal distances of the Kinect sensor obtained during the calibration [24].

input : A depth map M_d of size 640×480
output: Projected point cloud P_{xyz} into 3D space

```
1  __global__ void;
2  gpuPointCloudProjectionKernel( Md );
3  {
4      This kernel is executed creating one thread for each pixel in parallel;
5      int u = threadIdx.x + blockIdx.x * blockDim.x;
6      int v = threadIdx.y + blockIdx.y * blockDim.y;
7      float z = Md [v][u] / 1000.f Depth is stored in millimetres;
8      float px = z * (u - cx) * fx_inv;
9      float py = z * (v - cy) * fy_inv;
10     float pz = z;
11 }
```

Algorithm 2: Pseudo-code of the GPU-based point cloud projection algorithm.

This transformation can be computed independently for each pixel of the map, so it fits perfectly on massively parallel architectures such as the GPU, accelerating processing time related to the CPU implementation. As this transformation is often followed by other processing steps, it is not necessary to copy data back to the CPU memory and we therefore avoid the latency caused by these transfers by storing the projected 3D points on the GPU memory. Pseudo-code of the kernel executed onto the GPU is shown in Algorithm 2. In section 3.1 we show the acceleration factor and time of execution obtained by the GPU implementation. All these methods are developed in C++. GPU programming is done using the CUDA language created by NVIDIA [25]. Finally, 3D data management (data structures) and their visualization is done using the PCL[2] library.

2.1.1 Noise removal: Bilateral filtering

In structured light imaging a predefined light pattern is projected onto an object and simultaneously observed by a camera. The appearance of the light pattern in a certain region of the camera image varies with the camera-object distance. This effect is used to generate a distance image of the acquired scene. The predefined light patterns can be e.g. gray codes, sine waves, or speckle patterns. Speckle patterns are used in popular structured light (infrared) cameras like the Microsoft Kinect. This method of obtaining 3D information from the scene presents problems when the surfaces have a high level of specularity (reflection of the incident light) making it impossible for the sensor to obtain depth information about some surfaces [27]. The same problem

[2]The Point Cloud Library (or PCL) is a large scale, open project [26] for 2D/3D image and point cloud processing.

Figure 3: Left: Depth map. Center: RGB Map. Right: Projected point cloud.

occurs in the case of objects that are very far away from the sensor. Therefore, if we want to extract coherent information about the observed surfaces it is necessary to minimize this observation error. In previous works, simple filters such as the mean or the median have been used as they correct the error and run in real-time. As the computing power of the GPU can be applied in this step, this allows the application of more complex filters that are able to reduce the depth map error without removing important information, such as edge information. An example of these filters, the Bilateral filter [28], is able to remove noise of the image whilst preserving edge information. This filter was used originally in color and grey scale images to reduce the noise while keeping edge information, but we can also use it to reduce the noise on depth maps obtained from 3D sensors like the Kinect. A Bilateral filter is a combination of a domain kernel, which gives priority to pixels that are close to the target pixel in the image plane, with a range kernel, which gives priority to the pixels which have similar values as the target pixel. This filter is often useful when it is necessary to preserve edge information because of the range kernel advantages. The new value of a filtered pixel is given by:

$$P_f = \frac{1}{K_p} \sum_{q \in \omega} V_q f(||p - q||) g(||V_p - V_q||) \tag{3}$$

where K_p is a normalization factor, ω is the neighbourhood of the target pixel p, V_q is the neighbouring pixel value and P_f is the filtered value of pixel p. This equation also contains the domain kernel and the range kernel: $f(||p - q||)$, $g(||V_p - V_q||)$. Often, f and g are Gaussian functions with standard deviation σ_s and σ_r.

Section 2.1.2 shows how the estimation of the normal vectors is improved after applying bilateral filtering, providing more stable normal vectors by removing original noise presented on the depth map. Most 3D features extracted from the scene are based on the curvature of the geometry, which is calculated using information from normal vectors at each point in the scene, therefore obtaining more stable normal vectors leads to more accurate scene knowledge.

The calculation of filtered values at each pixel of the image can be computed independently and therefore is well suited for parallel architectures like the GPU. In [29] and [30] GPU implementations able to run in real time were proposed. The runtime is considerably improved, allowing filtering in real time depth maps generated by the sensor. In Section 3 GPU and CPU runtimes and speed-ups for our implementation on different graphics boards are presented.

2.1.2 Normal estimation

Estimation of normal vectors on a geometric surface has been widely used in many application areas such as computer graphics: generating realistic illumination of the surfaces and in computer vision as geometric features of the observed environment: keypoints with high curvature (corner or edge points).

Given a geometric surface it is possible to estimate the direction of the normal vector at a point obtaining the outward facing vector of the surface. However, we still have a point cloud without information about the surfaces that compose it, therefore we approach the normal vector estimation of a point efficiently by using its neighbourhood to calculate the normal vector. Here we focus on the estimation of the plane that best fits the neighbourhood of points using least squares. In this way the point normal vector is calculated as the plane normal vector.

The search for this plane is reduced to the calculation of eigenvalues and eigenvectors, Principal Component Analysis (PCA) of the covariance matrix created using the neighbourhood of the point on which we want to know its normal vector. The orientation of the normal vector is easily calculated because we know the point of view of the scene, in this case the Kinect position, so that all normal vectors must be facing consistently toward the point of view satisfying the following equation:

$$\vec{n_i} \cdot (v_p - p_i) > 0 \tag{4}$$

where $\vec{n_i}$ is the calculated normal vector, v_p is the point of view, and p_i is the target point. In cases where this constraint is not satisfied, it is necessary to reverse the sign of the calculated normal vector.

Once we have the organized point cloud stored in the memory of the GPU, the normal estimation process using PCA can be performed efficiently on the GPU. The normal vector calculation is performed on the GPU independently at each point of the scene, considerably accelerating the runtime. Pseudo-code of the GPU-based normal estimation algorithm is shown in Algorithm 3. Moreover, thanks to the previous noise removal using bilateral filtering, normal vectors obtained are much more stable than normal vectors computed directly from the original depth map that does not take into account the borders and corner points of the scene. In Figure 6 we can see this effect.

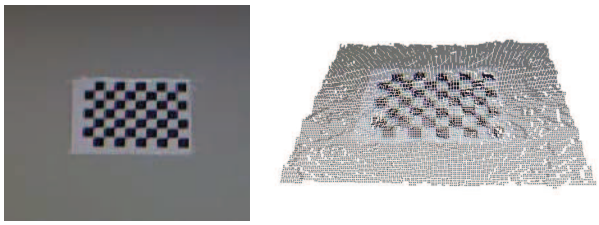

Figure 4: Left: RGB map of the captured plane. Right: Projected Point cloud of the captured plane. Corners of the chessboard has been used for estimating normal information at these points and compare that normals with the ground truth data (plane normal, 90 degrees).

	Mean error	**RMS**
Raw plane 1.5m	11.00	12.36
Bilateral plane 1.5m	4.84	5.53
Raw plane 3.5m	27.85	31.42
Bilateral plane 3.5m	13.89	16.56

Table 1: Angle errors between the normals of the points and the normal of the ground truth plane (90 degrees). All angle errors are in degrees.

In order to obtain a quantitative measurement of the improvement on the normal estimation step, a simple experiment was performed. We captured a perpendicular wall from two different distances, 1.5 and 3.5 meters (Figure 4). Using the points of the wall, a plane that fits best to these points is extracted. To estimate that plane we used a RANSAC method that gets the plane model with more inliers of the plane. Once we have the normal of the plane, we computed the angle between the estimated normals and the plane normal. In order to measure the quality of the normal estimation we show the arithmetic mean and the Root Mean Square (RMS) of the angle (in degrees).

Results in table 1 show that the angle error between the normals of the points and the normal of the plane is two times bigger without using the bilateral filter. We can also see the improvement in Figure 5 where the normal map of the scene is visually more regular applying bilateral filtering before normal estimation.

input : A projected point cloud d_P_{xyz}
output: Point cloud of normals d_N_{xyz}

```
1  __global__ void;
2  gpuNormalEstimationKernel( P_xyz, k );
3  {
4      This kernel is executed creating one thread for each point in parallel;
5      int u = threadIdx.x + blockIdx.x * blockDim.x;
6      int v = threadIdx.y + blockIdx.y * blockDim.y;
7      Compute Covariance matrix centered at point p using k neighbours;
8      d_N_xyz[u][v] = compCovarianceMat(u,v,k,N);
9      d_N_xyz[u][v] = checkOrientation();
10 }
```

Algorithm 3: Pseudo-code of the GPU-based normal estimation algorithm

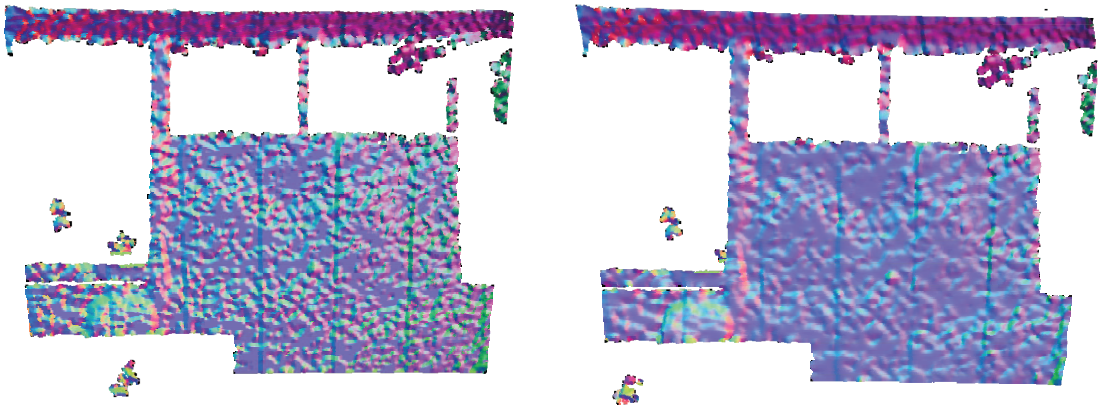

Figure 5: Left: Normal map from a raw plane captured using 3D range sensor. Right: Normal map from a filtered plane captured using 3D range sensor (bilateral filtering).

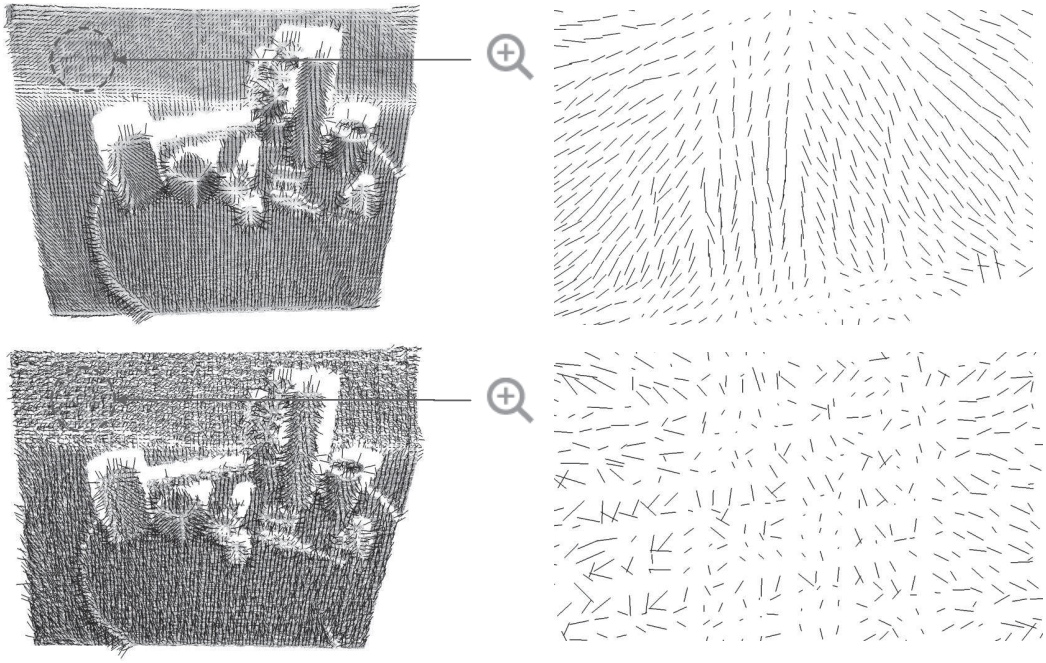

Figure 6: Bottom row. Normal estimation using the original map. Top row. Normal estimation using the filtered map (bilateral filtering). It can be observed that the normal estimation is improved resulting in more stable normal directions. This effect can be visually observed on plane surfaces where the normals estimated using a noisy map are much less stable than normals computed over a filtered map.

2.2 Surface triangulation on the GPU

In [31] an efficient method to triangulate organised point clouds is presented. In this section we present an accelerated and robust implementation of this method. In the original work a triangulation method for 3D points, obtained from range cameras or structured light, is proposed. Using sensors such as the Kinect, 3D points can be accessed using their matrix organization using x for the row and y for the column. In this way, a 3D point $p_{x,y}$ can be accessed using a 2D indexing system. Using this representation it is possible to obtain the scene surface from the point cloud captured by the sensor. The method assumes that the viewpoint is known and in this way it is possible to calculate the angle formed by the viewpoint vector v_p and the target point $p_{x,y}$ and the vector formed by the target point $p_{x,y}$ and its neighbour points $p_{x+1,y}$ or $p_{x,y+1}$. If points fall into a common line of sight with the viewpoint from where the measurements are taken, one of the underlying surfaces

occludes the other. If all checks are passed the triangle $(p_{x,y}, p_{x+1,y}, p_{x,y+1})$ is added to the mesh, otherwise a hole arises in the final reconstructed mesh. Moreover, if the sensor cannot acquire a valid depth measurement for a certain pixel that triangle is also rejected creating a hole. Figure 7 visually shows the proposed condition for point triangulation.

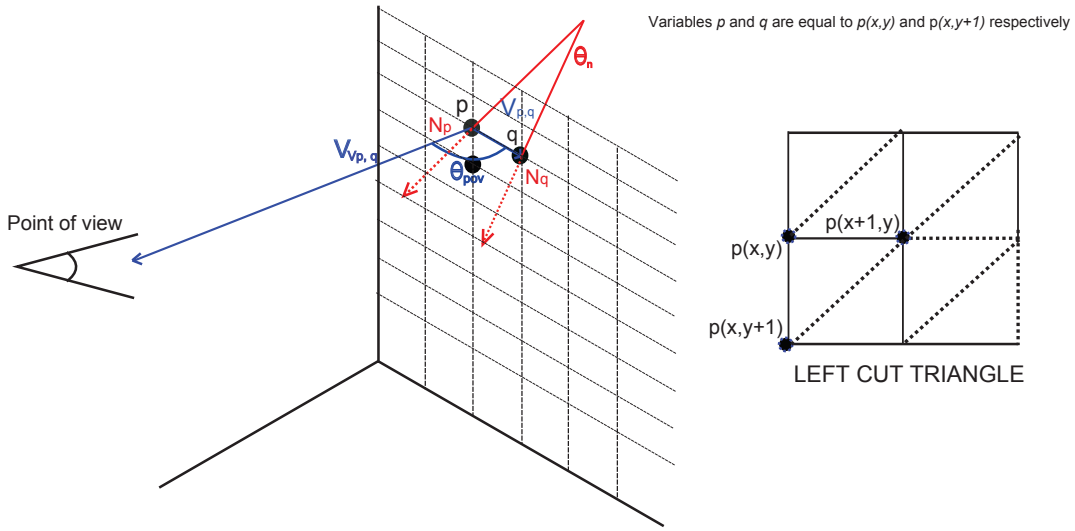

Figure 7: Left: Point triangulation condition. This images shows how the condition for creating an edge establishes that the angle θ_{pov} formed by vectors $v_{v_p,p}$ and $v_{p,q}$ must be within an established threshold $\epsilon_{\theta_{pov}}$. This threshold assures that points are not occluded among themselves. The Euclidean distance between p and q also must be smaller than an established threshold T_d, dynamically calculated according to mesh resolution and its standard deviation. Right: Triangles are established by left cut checking constraints between points.

Our proposed method is more robust than the original method, as the normal information at each point of the scene is used as an additional condition for meshing the point cloud. As the triangulation of the points can be done independently, the algorithm has been ported to the GPU, where each GPU thread tests the point we are targeting to form a triangle with its neighbourhood. As a result, we obtain a vector with all the triangles. Pseudo-code of the GPU-based surface triangulation algorithm is shown in Algorithm 4.

Invalid triangles are created on points that do not satisfy the proposed constraint to keep the organization of the point cloud. Finally, the condition to create an edge between two points is formulated as follows:

$$\begin{aligned} edge_{valid} \quad &= (|v_{v_p,p} \cdot v_{p,q}| \leq \cos \epsilon_{\theta_{pov}}) \wedge \\ &\quad (||p - q||^2 \leq T_d) \wedge \\ &\quad (|n_p \cdot n_q| \leq \cos \epsilon_{\theta_n}) \end{aligned} \tag{5}$$

where $\epsilon_{\theta_{pov}}$ is the angle existing between two points and the point of view establishing whether or not these points are occluded. This angle value is computed based on the visual analysis shown in Figure 7 and also based on results provided in [31]. The maximum distance between two points is T_d. This distance is obtained in real-time based on point cloud resolution. For that, the average distance between the targeted point and its neighbourhood k is calculated. Next, based on the mean of these average distances and standard deviation, threshold T_d is given by: $T_d = \overline{d_k} + \sigma_d$ where $\overline{d_k}$ is the mean distance and σ_d is the standard deviation. Finally, ϵ_{θ_n} is the established threshold for the maximum angle between two normal vectors. This is calculated in the same way as T_d, obtaining an angle threshold.

The proposed method allows us to obtain fast approximate meshing of the input point cloud. This fast meshing method is therefore used in Section 2.3 for computing the proposed 3D semi-local surface patch descriptor in real-time, as it requires a surface representation of the scene. The proposed accelerated meshing method takes advantage of the knowledge about the point of view position and also takes advantage of having already calculated normal vectors on GPU memory for every point of the scene. The GPU implementation achieves run times considerably lower compared to the CPU. The GPU implementation achieves processing frame rates close to 30 fps for 640 by 480 depth maps while the CPU implementation achieves a frame rate close to 6 fps. Figure 8 shows a point cloud mesh obtained using the proposed method.

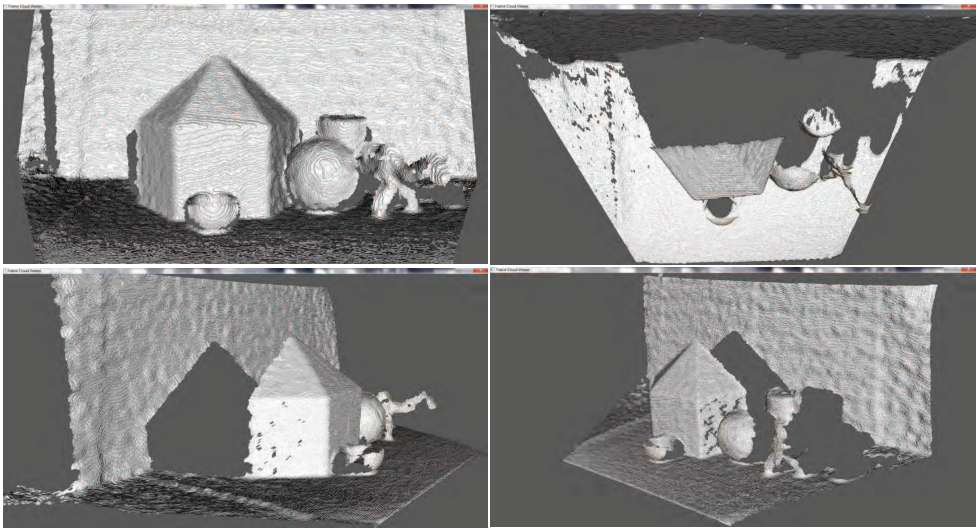

Figure 8: Point cloud meshing using the proposed method. Note that some holes and gaps still exist in the approximate surface reconstruction due to the noisy information obtained from the Kinect sensor.

input : A projected point cloud d_P_{xyz}
input : A point cloud of normals d_N_{xyz}
output: List of triangles d_Tri

```
1  __global__ void;
2  gpuTriangulationKernel( d_P_xyz, d_N_xyz );
3  {
4      This kernel is executed creating one thread for each point in parallel;
5      int u = threadIdx.x + blockIdx.x * blockDim.x;
6      int v = threadIdx.y + blockIdx.y * blockDim.y;
7      check constraints with neighbour points;
8      if (isValidTriangle (i, index_down, index_right));
9          addTriangle (d_Tri);
10     if (isValidTriangle (index_right, index_down, index_down_right)) ;
11         addTriangle (d_Tri);
12 }
```

Algorithm 4: Pseudo-code of the GPU-based surface triangulation algorithm.

2.3 Tensor computation on the GPU

Once point cloud normal information and surface triangulation are obtained, pairs of points along with their normals are selected to define local 3D coordinate bases for tensor computation. To avoid the C_2^n combinatorial explosion of the points, a distance constraint is used on their pairing. This distance constraint allows the pairing between only those points that are within a previously specified distance. The distance constraint also ensures that the vertices that are paired are far enough apart so that the calculation of the coordinate bases is not sensitive to noise but close enough to maximize their chances of being inside the same surface. The maximum and minimum distances between points are based on point cloud resolution, being $d_{min} = pcl_{res} * 5$ and $d_{max} = pcl_{res} * 14$. pcl_{res} is calculated for each point cloud captured by the Kinect sensor in real time, allowing the movement of the sensor. In addition to this distance constraint, an angle constraint θ_d is defined between valid pairs of points, so that points with approximately equal normals are not paired (since their cross product will result in zero). This mutual angle must be higher than 15^o degrees allowing the use of the mean value of these normals as an axis for the coordinate bases. In Figure 9 shows how valid pairs are selected considering mutual angle and mutual distance constraints. Moreover, each point is paired with only its three closest neighbours, limiting the number of possible pairs to $3n$ per view. In practice, due to the constraints this number is lower than $3n$.

Pair point calculation is accelerated using as many threads as points in the point cloud. In this way each

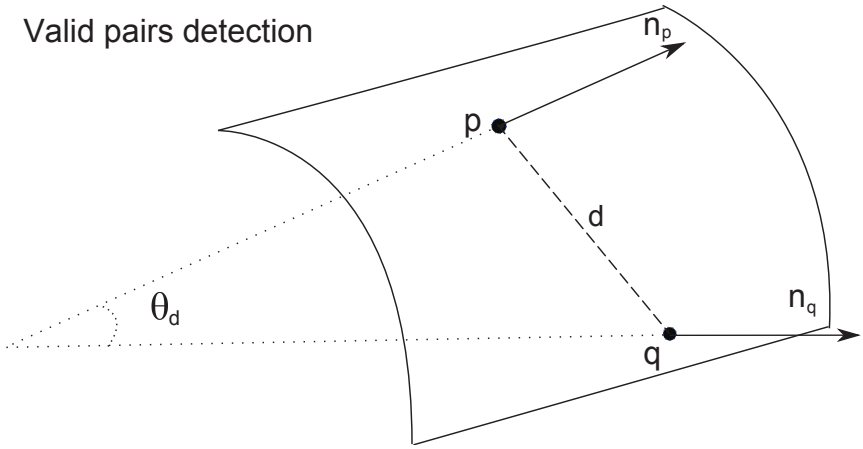

Figure 9: Detecting a valid pair using mutual angle θ_d and mutual distance d constraints. The midpoint of the pair will be used later to define a 3D grid for tensor computation.

GPU thread checks its corresponding point pair with its neighbours. Moreover, the matrix organization of the point cloud is used for improving this search. In this way, each thread of the GPU performs the search of valid pairs only in a defined window around the targeted point. The size of this window is based on the maximum distance constraint d_{max} and the point cloud resolution pcl_{res}: $windows_{radius} = d_{max}/pcl_{res}$ giving the radius of the windows in pixels. As d_{max} is defined as a constant based on the pcl_{res} (millimeters), the $windows_{radius}$ can be simplified as the constant 14.

In Section 3.1 a runtime comparison is presented, the CPU implementation applies the same technique for search acceleration.

Once a valid list of point pairs is obtained, a local 3D basis is defined for each valid pair in the following manner: the center of the line joining the two vertices defines the origin of the new 3D basis. The average of the two normals defines the z-axis. The cross product of the two normals defines the x-axis and finally the cross product of the z-axis with the x-axis defines the y-axis. This 3D basis is used to define a 3D grid centered at its origin. This step is also computed in parallel on the GPU for each valid pair of points.

For the grid computation, which will define the feature descriptor, it is necessary to define two more parameters. The first one is the number of voxels that compose the grid n_{voxels} and the size of each of these voxels $voxel_{size}$. Modifying the number of voxels and so the size of the grid causes the obtained descriptor to contain local, semi-local or global information of the scene. In the experiments done in [21] it is demonstrated how for the object recognition task, a size of $10 \times 10 \times 10$ grid allows the extraction of a descriptor with semi-local information of the object allowing identification even under a high level of occlusion. The size of the voxel $voxel_{size}$ is defined dynamically according to the point cloud resolution. Once the grid is defined, the surface area of the mesh surface intersecting each voxel of the grid is stored in a third order tensor. This tensor is a local surface descriptor which corresponds to a semi-local representation of the object where the pair of points are lying. Sutherland Hodgman's polygon clipping algorithm [32] is used for calculating area intersections between polygons and voxels. In this way an entry is made at the corresponding element position in the tensor. Since more than one triangulated facet can intersect a single voxel, the calculated area of intersection is added to the area already present in that voxel as a result of intersection with another triangulated facet. To avoid checking all triangles that compose the scene, a growing approach is used, which starts by checking the triangles that lie in the pair of points selected and growing along its neighbourhood until all the checked triangles are not intersected with the corresponding voxel. This approach is used in both CPU and GPU versions, allowing a fair runtime comparison. Finally extracted tensors are compressed by squeezing out the zero elements and retaining the non-zero values and their index positions in the tensor. These compressed tensors together with their respective coordinate basis and the mutual angle between their normals are called a tensor representation of the view.

The computation of each tensor is considerably accelerated using the GPU because there is no dependency between the calculation of the intersected area in each voxel of the grid. Therefore, $Dim_x \times Dim_y \times Dim_z$ threads are executed on the GPU organized as a three dimensional grid. Each thread calculates the intersected area between the mesh and its corresponding voxel, storing the calculated area in the position accessed by its indexes. See Figure 10. Due to the 3D index organization that the CUDA framework provides, the calculation of corresponding indexes is greatly accelerated. Sutherland Hodgman's polygon clipping algorithm is also executed by each thread in parallel. Pseudo-code of the GPU-based 3D tensor computation algorithm is shown in Algorithm 5. Additionally, there is also no dependency between the computation of different tensors, thereby

the computation of different tensors is overlapped occupying all the available resources on the GPU. Performance results are shown in Section 3.1.

input : A projected point cloud d_P_{xyz}
input : A valid pair of points d_N_{xyz}
input : List of triangles d_Tri
output: 3D tensor t_i

```
1  __global__ void;
2  gpuTensorCompKernel( v_i,d_P_xyz,d_N_xyz,d_Tri );
3  {
4      This kernel is executed creating one thread for each bin of the grid in parallel;
5      int x = threadIdx.x + blockIdx.x * blockDim.x ;
6      int y = threadIdx.y + blockIdx.y * blockDim.y ;
7      int z = threadIdx.z + blockIdx.z * blockDim.z ;
8      binLimits = computeBinLimits(cloud,tri) ;
9      d_Neigh_Tri = compIndexNeighTriangles() ;
10     calculate area that clip with the corresponding bin;
11     for i ← 0 to |d_Neigh_Tri| do
12 |       area += clipTriangle(cloud,tri,binLimits);
13     end
14     t_i[x][y][z] = area ;
15 }
```

Algorithm 5: Pseudo-code of the GPU-based 3D tensor computation algorithm

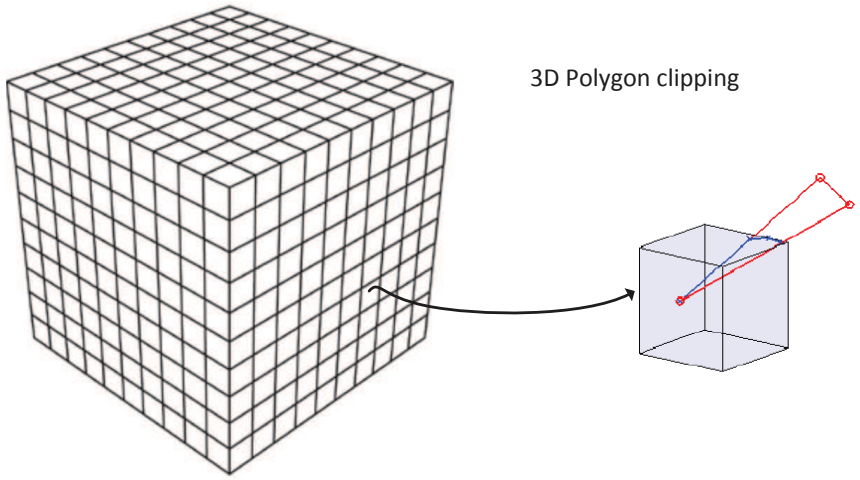

Figure 10: Launching $Dim_x \times Dim_y \times Dim_z$ threads in parallel where each GPU thread represents a voxel of the grid. Each thread with indexes i, j, k calculates the area of intersection between the mesh and its corresponding voxel using Sutherland Hodgman's polygon clipping algorithm. Taking advantage of thread indexes, the calculated area is stored in a flattened vector.

2.4 Offline learning and matching

Since computed 3D tensors can be used in many different applications it is necessary to find a way to store them so they can be retrieved in an efficient way. All extracted tensors from different partial views of an object are stored with their coordinate basis allowing the use of this information for grouping all tensors with similar angles between their normals. In this way an efficient matching is possible for different applications such as partial view registration and object recognition. This collection of tensors is stored during a training phase creating a hash table for efficient retrieval during the test phase. The hash table is efficiently constructed from the computed tensors of multiple views and it is filled up using the angle θ_d of the tensors as the hash key.

This hashing technique appears to be an efficient and an appropriate way to perform tensor matching due to its lower computational cost compared to linear matching techniques. This makes the matching method less sensitive to the number of views or models in the database.

In contrast to the implementation originally presented in [21] and in order to integrate the matching process on the GPU pipeline, the hash table is stored in the GPU memory performing the tensors matching in parallel on the GPU and considerably accelerating its performance. In the online matching step, once a tensor is extracted from the scene, there are launched as many threads as tensors are stored in the corresponding bin. This allows to check in parallel all tensors stored in a bin and therefore to accelerate the matching process. Further discussion and experimental results are presented in Section 4.

3 Experimental results

GPU versions of the proposed method described in this document has been tested on a desktop machine with an Intel Core i3 540 3.07Ghz and different CUDA capable devices. GPU implementations were first developed on a laptop machine equipped with an Intel Core i5 3210M 2.5 Ghz and a CUDA compatible GPU. Table 2 shows different models that have been used and their main features. We used different models ranging from the integrated GPU on a laptop to a more advanced model, demonstrating that the GPU implementations can be executed on different GPUs and that they can obtain good execution times on different graphic boards with different number of cores.

Device Model	CUDA cores	Global Mem	Bandwidth Mem	GPU Clock	Memory Clock
Quadro 2k	192	1 GB	41.6 GB/s	625 Mhz	1300 Mhz
GeForce GTX 480	480	1.5 GB	177.4 GB/s	607 Mhz	1215 Mhz
GeForce GT630M	96	1 GB	32 GB/s	625 Mhz	1000 Mhz

Table 2: CUDA capable devices used in experiments

GPUs are ideally suited to executing data-parallel algorithms. Data-parallel algorithms execute identical units of work (programs) over large sets of data. The algorithms can be parallelized for efficiency when the work units are independent and are able to run on small divisions on the data. One critical aspect of designing parallel algorithms is identifying the units of work and determining how they will interact via communication and synchronization. A second critical aspect is analyzing the data access patterns of the programs and ensuring data locality to the processing units. It is also necessary to consider the program execution pipeline in order to avoid unnecessary data transfers,

These three critical aspects have been satisfied by our GPU implementations because every step was decomposed as an independent execution unit. This is possible since there are no dependencies during their computation. Moreover, all the computed data on the GPU is not transferred back to the CPU until the entire pipeline is completed, thus avoiding expensive memory transfers. Finally, threads access memory using data-patterns in order to ensure locality to the processing units. Most steps implemented on the GPU use a 2D map of threads accessing memory in a coalesced way. For the tensor computation a 3D grid of threads is used.

3.1 Performance

The performance obtained by the GPU implementation allow us to compute the proposed methods under real-time constraints. In Table 3 we can see the different steps that have been accelerated using the GPU and their different runtime and the speed-ups achieved for the different graphics boards. The obtained acceleration is relative to a CPU implementation of the proposed method. In general the best performance was obtained with the graphics board with the largest number of CUDA cores (GTX480) and the largest memory bandwidth.

These results demonstrate how the proposed methods are suitable for massively parallel architectures such as the GPU, where each thread processes one of the points of the scene. Another interesting aspect of the results shown in Table 3 is that GPU implementations allows us to compute operations that are prohibitively slow on the CPU in real-time such as normal estimation, noise filtering, valid pairs detection or surface triangulation. Moreover, in table 3 it is shown how the entire computation of 200 tensors in the GPU is performed in less than 0.5 seconds for the faster device achieving a 93x performance boost related to the CPU implementation and allowing the computation of the descriptor at different points of a scene in real-time.

Another remarkable aspect of the performance obtained for the overall system is that tensor computation is not only parallelized at thread level, it is also parallelized at task level computing simultaneously different tensors. As tensor computation is not dependent, it can be parallelized using different CUDA streams on the GPU. This technology allows executing in parallel as many kernels as possible in different queues and therefore allows to exploit available resources on the GPU [33]. We decided to exploit the possibility of launching multiple kernels concurrently using CUDA streams, overlapping the paradigm of task parallelism with that of

Step	GT630M	GTX480	Q2k	CPU	GT630M	GT480	Q2k
Bilateral filtering of depth map	11ms	**5ms**	8ms	1008 ms	91.63x	**201.6x**	126x
Point cloud projection	2ms	**1ms**	1ms	50ms	25x	**50x**	50x
Normal estimation	9ms	**1ms**	8ms	190ms	21.11x	**190x**	23.75x
Compute surface triangulation	5ms	**2ms**	4ms	121ms	24.25x	**40.3x**	30.25x
Compute cloud resolution	7ms	**4ms**	6ms	330ms	47.14x	**82.5x**	55x
Compute valid pairs	71ms	**9ms**	35ms	4479ms	63x	**497x**	127.97x
Compute third order tensor	6ms	3ms	4ms	130 ms	31.6x	**43.33x**	32.5x
Total GPU time for extracting 200 tensors	854 ms	**490ms**	687ms	45887ms	53.72x	**93.64x**	66.79x

Table 3: Runtime comparison and speed-up obtained for proposed methods using different graphics boards. The fastest run times were achieved by the graphics board NVIDIA GTX480. Runtimes are averaged over 50 runs. Data transfers between GPU and CPU memories are included in the total GPU runtime.

data parallelism. In order to analyse and confirm stream parallel execution we profiled the algorithm using the NVIDIA Visual Profiler [34], which allows to visually appreciate how stream computation is performed along the time and also multiprocessors occupancy on the GPU. In Figure 12 shows the algorithm computation timeline using and not using streams to overlap computations. Runtime execution and multiprocessors occupancy is greatly improved thanks to concurrent kernel execution using streams. Runtime is improved by a speed-up factor of 5x overlapping tensors computation with many kernels enqueued in different streams and launched concurrently.

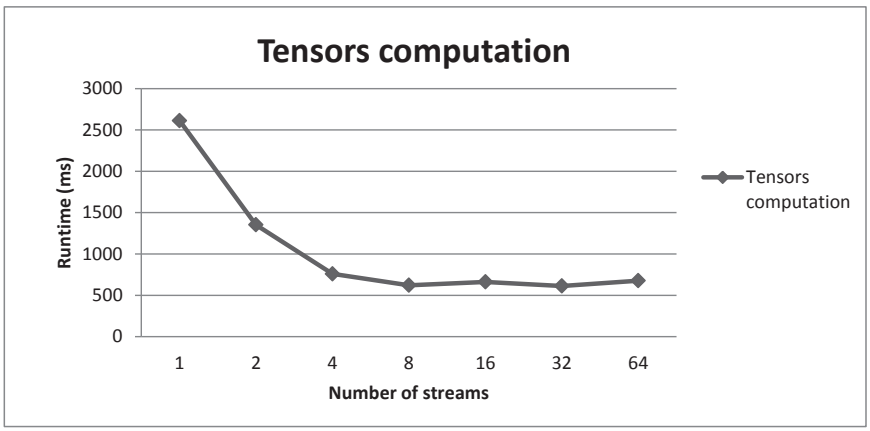

Figure 11: Tensors computation runtime using different numbers of streams. Number of tensors is fixed to 200 and the device used is the NVIDIA GTX 480.

We fixed the number of streams to 16 after testing different numbers of streams. Figure 11 shows how the runtime is improved as the number of streams is increased obtaining maximum performance and occupancy on the GPU using values larger than 4. Indeed, in Figure 12 it is shown how the maximum number of tensors that are calculated simultaneously is 8 without taking into consideration the maximum number of streams specified. This occurs due to the occupancy of all the available resources by the kernels running concurrently.

Finally, in Figure 13 an experiment computing different number of tensors is performed and the speed-up compared to the CPU version is presented. From Figure 13 we can conclude that the speed-up obtained by the GPU version is increased as the number of tensors is also increased achieving a larger speed-up factor. Computing times obtained using the CPU version are prohibitive for time-constrained applications.

4 Robot Vision: 3D object recognition

In this section, we show an application where the use of the accelerated semi-local surface feature extraction process allows to detect and recognize objects under cluttered conditions in real-time. The main goal of this application is the recognition of objects under real-time constraints in order to integrate the proposed algorithm in mobile robotics. Our method is designed to use only depth information because of the need for robots to work under bad or no illumination conditions.

To validate the proposal we tested the proposed feature in a similar application as was done in the original work [21] where the semi-local surface features are successfully used to recognize objects in cluttered scenes. For our experiments we have captured data from a Kinect sensor and tested the accelerated feature with some

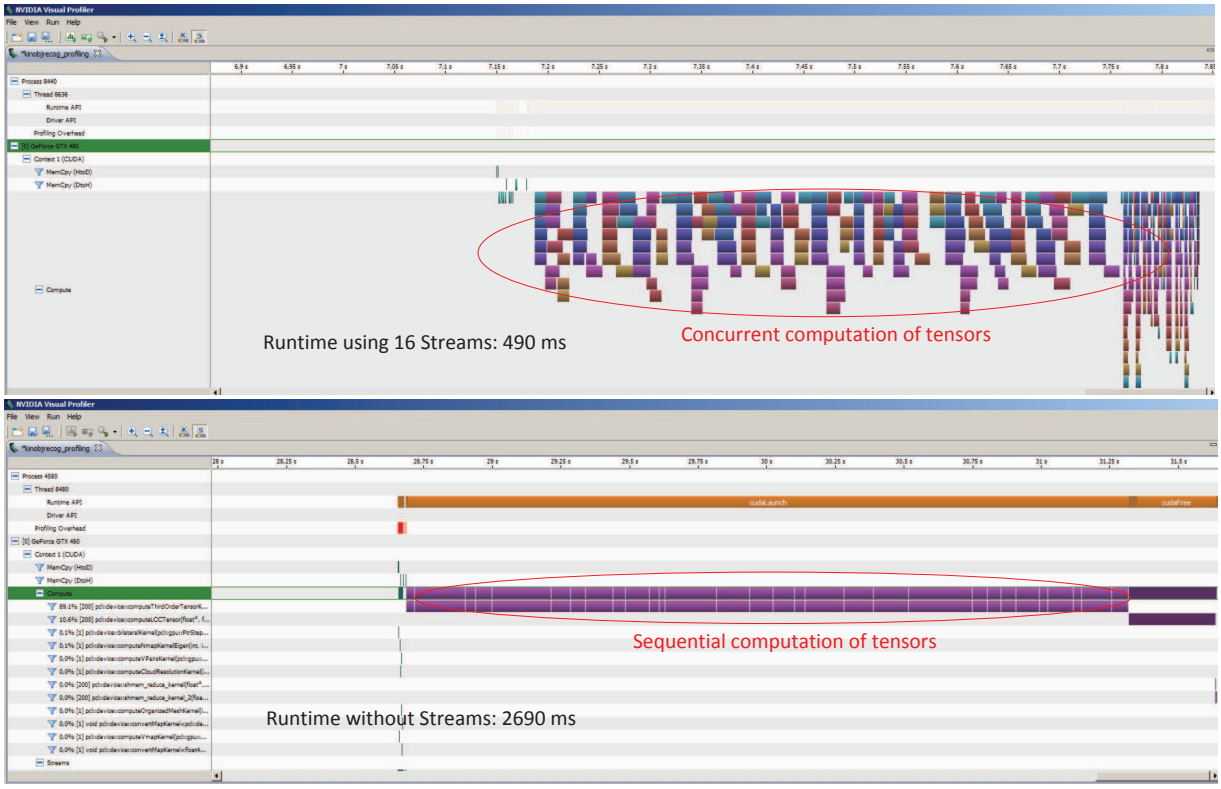

Figure 12: Profiling computation of tensors using streams (Top) and without streams (Bottom). On the top of the figure (using streams) it can be seen how kernels run simultaneously occupying all available resources on the GPU.

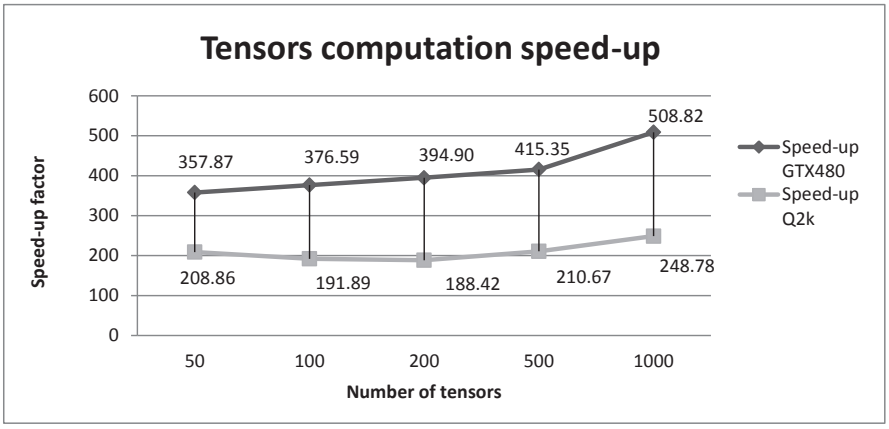

Figure 13: Achieved speed-ups compared to sequential CPU version and computational times for computing various number of tensors without considering the rest of steps (projection, bilateral filtering, normal estimation, etc.)

cluttered scenes. To do that, first a small library of models is constructed offline, storing all extracted tensors in an efficient way using a hash table. Afterwards, online object recognition is performed using cluttered scenes. Although the accelerated feature is tested using 3D data obtained from the Kinect sensor, this method is developed for managing 3D point sets collected by any kind of sensor and could be extended to other datasets.

We created a toy dataset to validate our proposal since the main goal of this work is to achieve real-time performance and integrate 3D data processing onto the GPU. Further analysis on recognition rates and feature parameters are already presented in the original work [21]. In addition, a deeper analysis on parametrization will be carried out in future works as this topic is out of the scope of the current work.

4.1 Offline learning

To recognize objects using our real-time tensor extraction algorithm, first a model library is built extracting tensors from different views of free-form objects. Each partial view is represented with tensors and they are stored in an efficient way for being used after in an online recognition phase. In Figure 14, some partial views of the models used to build the library are presented. Moreover, tensors extracted for some of the views are shown in Figure 14. For each of these views the process explained in Section 2.1 is computed, obtaining as a result a set of tensors that describe each partial view. Tensors are stored in an efficient way using a hash table as was explained in Section 2.4.

4.2 Online recognition

Once the model library is built and loaded in the GPU memory, the application is ready to start recognizing objects from a scene captured in real-time. Therefore, the input to our application is a point cloud of a scene. The point cloud is processed following the pipeline presented in Section 2.1. Once tensors from the scene are computed these are matched against the model library previously stored on the GPU memory. The matching process, as was introduced in previous section, is performed in parallel for the calculated entry using the angle as a key for the hash map. For the calculated entry are launched as many threads as tensors are stored in that bounded bin, computing in parallel a correlation coefficient originally presented in [21]. The correlation coefficient measures the similarity between the scene tensors and possible candidates stored in the model library. The correlation coefficient in the overlapped area between two tensors is calculated as follows:

$$C_c = \frac{n_q \sum_{i=1}^{n_q} p_i q_i - \sum_{i=1}^{n_q} p_i \sum_{i=1}^{n_q} q_i}{\sqrt{n_q \sum_{i=1}^{n_q} {p_i}^2 - \left(\sum_{i=1}^{n_q} p_i\right)^2} \sqrt{n_q \sum_{i=1}^{n_q} {q_i}^2 - \left(\sum_{i=1}^{n_q} q_i\right)^2}} \tag{6}$$

where p_i and $q_i (i = 1...n_q)$ are the respective elements of the model tensor T_m and scene tensor T_s in their region of overlap. Matchings whose $C_c < t_c$ are discarded ($t_c = 0.45$ based on results presented in [21]). The remaining tensors are considered as possible correspondences.

Once all correlation coefficients have been calculated in parallel for a scene tensor T_s and considered as possible correspondences, maximum correlation coefficient value is found and considered as the true correspondence. The reduction operation [35] to obtain the maximum value in parallel is also performed on the GPU pipeline using traditional divide and conquer approach to find the maximum value.

4.3 3D object recognition results

We performed out different experiments where the accuracy and the performance of the proposed GPU implementation are studied. The algorithm is initially validated on scenes with a single object. Then, a more complex study on noise scenes with multiple objects and occlusions is performed.

4.3.1 Recognition on scenes with a single object

In this first experiment, single views of each object model are used for testing descriptor recognition accuracy. Tested scenes only contained the object, so there are no occlusions caused by other objects, only self-occlusions. The experiment was performed on different views of each object. Different views from arbitrary viewing directions were selected. The confusion matrix showing the performance and recognition rates for extracted tensors is shown in Figure 15. The total rate of tensor recognition obtained for the experiment was 84% (TP). The rate of False Positives (FP) was 16% and False Negatives (FN) 0%. It is important to notice how some tensors extracted from tortoise views were wrongly assigned to the apple model and vice versa. This was caused by the lack of information of some tortoise views from these viewing directions and the similar geometric shape they have. It is also important to notice how observations from the top of two objects may be visually similar as occurs with the front view of the pyramid and the speaker. Therefore, some tensors extracted from the speaker views where wrongly classified as the pyramid and cube model.

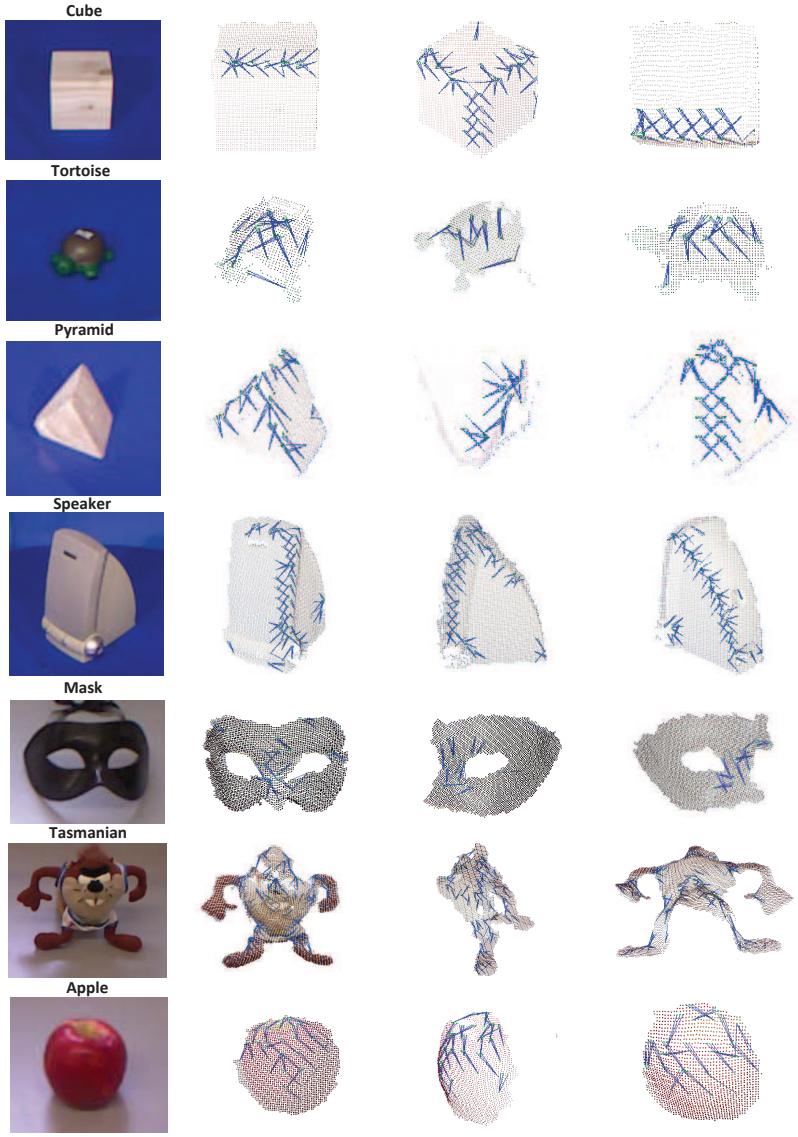

Figure 14: Model library consisted of 7 real models. Each object consists of several partial views. For every partial view of a model, tensors are computed (blue lines) describing the model by extracting 3D surface patches (tensors).

4.3.2 Recognition on scenes with multiple objects and occlusions

In this experiment, we used objects from the previously constructed dataset and other non-stored objects. There were multiple objects occluding each other and causing clutter in the scene. Constructed scenes were used as input data for the proposed GPU implementation in order to perform object recognition. Moreover, no prior information was provided to the algorithm regard models placed in the scene. In Figure 16 (Top) it is shown a first experiment with occlusions where three different objects (2 pyramids and a tortoise) are occluded by two objects non-stored in the database. In this experiment different percentage of occlusions are considered. We defined occlusion according to the next formula as it was done in a similar way in the original work:

$$occlusion = 1 - \frac{\text{object points visible in the scene}}{\text{total object points}} \tag{7}$$

From all the extracted tensors from the scene presented in Figure 16 (Top) the recognition rate (TP) achieved was 85 percent, with the only mis-classifications occuring with objects not in the database. They were classified as the cube and speaker model as the flat sides of the observation resembles the cube model. All extracted tensors from the occluded objects were successfully classified. In addition, this experiment shows how the proposed implementation can differentiate between multiple instances of the same object, while other

		tortoise	speaker	pyramid	cube	mask	tasmanian	apple
tortoise	v1	97.62	0.00	0.00	0.00	0.00	2.38	0.00
	v2	90.00	8.00	0.00	0.00	0.00	0.00	2.00
	v3	83.33	3.33	0.00	3.33	0.00	6.67	3.33
	v4	77.94	0.00	0.00	2.94	0.00	2.94	16.18
	v5	92.98	0.00	1.75	0.00	0.00	0.00	5.26
	v6	92.16	0.00	0.00	3.92	0.00	0.00	3.92
speaker	v1	0.00	60.00	17.78	6.67	0.00	2.22	13.33
	v2	0.00	96.70	2.20	1.10	0.00	0.00	0.00
	v3	0.00	73.33	6.67	6.67	0.00	6.67	6.67
	v4	0.00	94.69	0.00	0.00	0.97	2.42	1.93
	v5	1.89	76.42	9.43	2.83	0.00	5.66	3.77
	v6	2.05	94.87	0.51	0.00	0.00	2.05	0.51
pyramid	v1	9.43	13.21	64.15	11.32	0.00	1.89	0.00
	v2	0.00	11.11	85.71	1.59	0.00	0.00	1.59
	v3	0.00	4.69	84.38	9.38	0.00	0.00	1.56
	v4	0.00	3.77	92.45	0.00	0.00	0.00	3.77
	v5	0.00	11.94	80.60	4.48	0.00	2.99	0.00
	v6	6.06	27.27	60.61	0.00	0.00	6.06	0.00
cube	v1	0.00	7.06	0.00	92.94	0.00	0.00	0.00
	v2	0.00	14.29	3.57	82.14	0.00	0.00	0.00
	v3	0.00	5.80	0.00	92.03	0.00	0.72	1.45
	v4	0.00	1.53	0.00	96.95	0.00	0.00	1.53
	v5	0.00	13.73	13.73	58.82	0.00	0.00	13.73
mask	v1	0.00	2.53	0.00	0.00	92.41	5.06	0.00
	v2	3.77	3.77	0.00	0.00	86.79	3.77	1.89
	v3	7.69	0.00	0.00	0.00	89.74	0.00	2.56
	v4	2.17	8.70	2.17	0.00	86.96	0.00	0.00
	v5	9.68	4.84	0.00	0.00	77.42	3.23	4.84
	v6	7.32	2.44	7.32	0.00	75.61	2.44	4.88
tasmanian	v1	0.44	0.00	3.07	0.00	1.75	90.79	3.95
	v2	1.64	0.00	1.97	2.30	3.93	90.16	0.00
	v3	1.40	0.00	1.40	0.00	2.33	90.70	4.19
	v4	2.83	0.00	8.49	0.00	0.00	88.68	0.00
	v5	0.39	0.00	2.36	0.00	1.18	95.28	0.79
	v6	1.71	0.85	3.85	0.00	0.00	93.59	0.00
apple	v1	8.62	0.00	0.00	0.00	1.72	1.72	87.93
	v2	25.81	0.00	0.00	0.00	1.61	3.23	69.35
	v3	17.86	0.00	3.57	1.19	9.52	0.00	67.86
	v4	3.13	3.13	9.38	0.00	3.13	0.00	81.25
	v5	8.00	4.00	6.00	0.00	8.00	2.00	72.00

Figure 15: Confusion matrix for extracted tensors from different views of each object model contained in the dataset. The averaged recognition rate (TP) is 84%, wrong matches (FP) 16% and false negatives (FN) 0%. Cells numbers indicate the number of times (percentage) that an extracted tensor was sucessfully classified to the corresponding model. Presented results show that method provides high discrimination capability.

techniques [10] are only able to detect single instances in the scene. In Figure 16 (Bottom) a scene with different levels of occlusion is presented. The recognition rate achieved for this scene was 76%. As the level of occlusion is increased some objects such as the pyramid which has a level of occlusion close to 82% was wrongly classified, but the rest of objects in the scene were successfully classified. Moreover, some objects present in the scene but non-stored in the library such as the paper box was classified as the cube model. After testing the GPU implementation with a total of 8 scenes (Figure 17) with different levels of occlusion the average recognition rate achieved was 82%. We noticed that the capability to recognize objects with different level of occlusion is related to the part of the object that is visible and if this part of the object has areas with high discriminant features. Moreover, the area of overlap for the correlation coefficient between two tensors was established to 45%. If higher levels of occlusions are demanded this factor could be set to a lower value allowing the detection of tensors with higher levels of occlusions, but this would considerably affect the recognition rate, classifying a

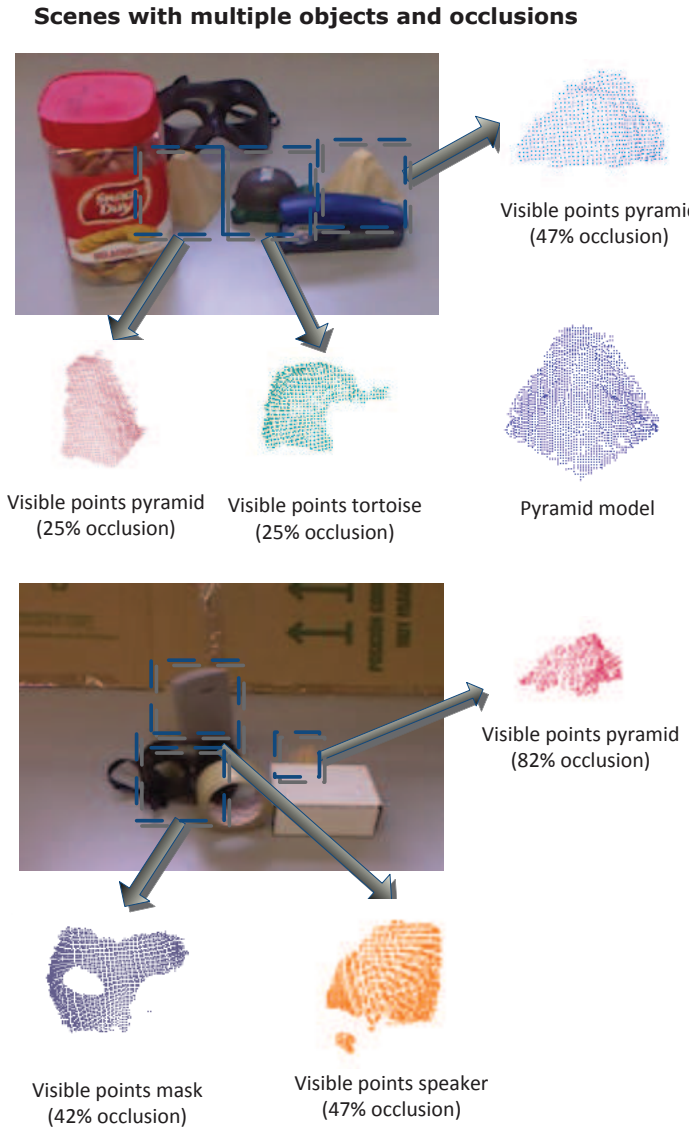

Figure 16: Object recognition is performed on scenes with different level of occlusion. The models are occluded by objects stored and non-stored in the library.

higher number of tensors in the wrong class.

Finally, Figure 18 shows two arbitrary scenes computed using the proposed method. Tensors are calculated randomly over the scenes and matched tensors are labelled with the closest model in the library. Multiple labels are shown in Figure 18 as all tensors present in the scene are evaluated and matched against the library model. Voting strategies within clusters may be performed in order to further accelerate the object recognition process. In addition since the main goal of this work is to achieve real-time geometric feature extraction, the final hypothesis verification step has not been implemented yet on the GPU, so it remains as a future work. This is the reason why multiple labels are drawn.

The number of tensors evaluated over the scene is 200 as experiments have demonstrated that evaluating over this number in most of cases achieves the recognition of all objects in the scene. A similar study was made in the original work [21]. Some wrong labelling appears in Figure 18 (Top) for the speaker as the partial view of the scene does not have enough geometric information to find a correspondence in the database. However, in the Scene 2 (Bottom) as the partial view contains more geometric information of the speaker, it is correctly recognized. For other objects as the tortoise, cube and pyramid, as similar views of the objects are present in the database and partial view of the scene has enough geometric information, the algorithm does correctly find

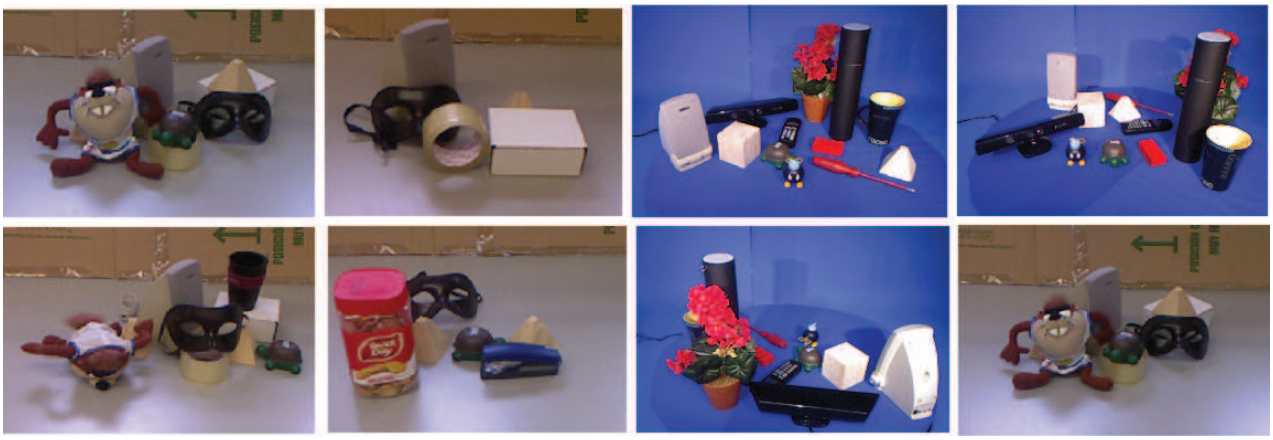

Figure 17: Scenes with multiple object and occlusions. These scenes were used in experiments presented in Section 4.3.2 achieving a recognition rate of 82%. Scenes presented a different level of clutter and occlusions caused by stored and non-stored objects.

tensors that match the model stored in the library. Future hypothesis verification and feature grouping steps will improve the classification accuracy of the object recognition task.

4.3.3 Performance

In this section, some experiments related to the performance of the parallel matching performed on the GPU are presented, comparing performances obtained by the GPU and CPU versions.

Model library size (tensors)	2000	4000	16000	64000
Runtime CPU (ms)	215	398	1589	6414
Runtime GTX480 (ms)	75	140	534	2130
Speed-up GTX480	2.87x	2.84x	2.98x	3.01x

Table 4: Runtime comparison and speed-up obtained for matching process. As the size of the model library is increased the speed-up achieved is slightly larger. Runtimes are averaged over 50 runs.

Table 4 shows how the matching process is computed faster on the GPU. The speed-up achieved is close to 3x but the most important observation is that computing matching process on the GPU also avoids transferring the data back to the CPU side after computing tensors on the GPU, which obtains an important acceleration factor compared to the CPU as was shown in Section 3.1. In this experiment, matching process is tested using different sizes of the model library, ranging from 4 to 64 objects. The number of objects was simulated by the real model library comprised of 4 objects. It is assumed that every object is described extracting tensors from 6 partial views obtaining an average of 150 tensors per model.

Regarding GPU memory limitations, since modern GPUs have large global memories, we can find in the market consumer GPUs equipped with a 6 GB RAM memory, so storage of the model library for the matching process is not a problem. Moreover, the tensor representation is compressed by squeezing out the zero elements and retaining the non-zero values and their index positions in the tensor. Even without compressing tensors, the overall storage of 120 models using an average of 6 views per model will results in less than 40 MB of storage space.

Finally, total computation for the GPU took around 800ms using the NVIDIA GTX480, managing 3D object recognition in real-time and therefore enabling its integration in mobile robotics. In the original Matlab single-threaded implementation it took an average time of 2 minutes to recognize and segment a single object in a cluttered scene. We believe that this GPU implementation is an important step towards the computation of 3D geometric descriptors in real-time.

5 Conclusions

The highlights of this paper are as follows:

- Our primary concern is the integration of 3D data processing algorithms in complex computer vision systems. Experiments have demonstrated that the GPGPU paradigm allows to considerably accelerate

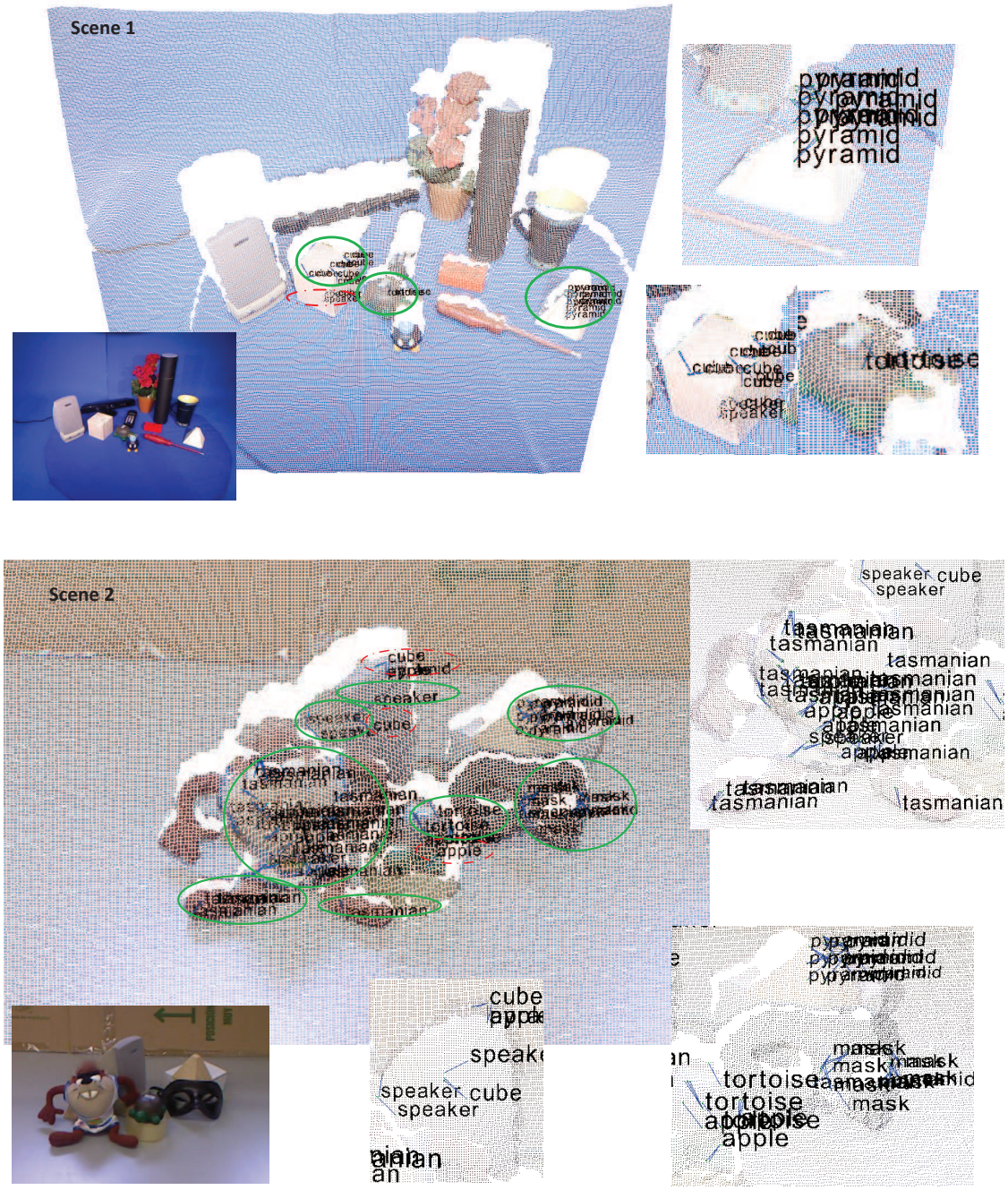

Figure 18: 3D object recognition in cluttered scenes. Different partial views of two scenes are shown. Multiple labels are shown as all computed tensors are evaluated and matched against the library model.

algorithms compared to CPU implementations and to run these in real-time.

- Within the 3D data algorithms used in the proposed pipeline, some progress has been made towards a faster and more robust normal estimation and point cloud triangulation algorithms, obtaining GPU implementations that run at 30 fps.
- Advantages are obtained by the use of the GPU to accelerate the computation of a 3D descriptor based on the calculation of 3D semi-local surface patches of partial views, thus allowing descriptor computation at several points of a scene in real-time.
- The matching process has also been accelerated onto the GPU, taking advantage of the GPU pipeline and achieving a speed-up factor of 3x compared the CPU implementation.

- We have implemented a prototype of the proposed pipeline and it has been tested with a real application obtaining satisfactory results in terms of accuracy and performance. We show that implemented prototype took around 800 ms with a GPU implementation to perform 3D object recognition of the entire scene.

Further work will include adding other processing steps to the GPU pipeline: hypothesis verification using ICP techniques on the GPU and using multi-GPU computation to improve performance and to manage computation of tensors and their matching on different devices.

Acknowledgements

This work was partially funded by the Valencian Government BEFPI/2012/056, and by the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC). Experiments were made possible with a generous donation of hardware from NVDIA.

References

[1] Oikonomidis, I., Kyriazis, N., Argyros, A.A.: Full DoF tracking of a hand interacting with an object by modeling occlusions and physical constraints. In: Proceedings of the 2011 International Conference on Computer Vision. ICCV11, Washington, DC, USA, IEEE Computer Society (2011) 2088–2095

[2] Izadi, S., Newcombe, R.A., Kim, D., Hilliges, O., Molyneaux, D., Hodges, S., Kohli, P., Shotton, J., Davison, A.J., Fitzgibbon, A.W.: Kinectfusion: real-time dynamic 3d surface reconstruction and interaction. In: SIGGRAPH Talks. (2011)

[3] Kerl, C., Sturm, J., Cremers, D.: Dense visual slam for RGB-D cameras. In: Proc. of the Int. Conf. on Intelligent Robot Systems (IROS). (2013)

[4] Griffin, W., Wang, Y., Berrios, D., Olano, M.: Real-time gpu surface curvature estimation on deforming meshes and volumetric data sets. IEEE Transactions on Visualization and Computer Graphics **18**(10) (2012) 1603–1613

[5] Prisacariu, V., Reid, I.: fastHOG - a real-time GPU implementation of HOG. Technical Report 2310/09, Department of Engineering Science, Oxford University (2009)

[6] Himmelsbach, M., Luettel, T., Wuensche, H.J.: Real-time object classification in 3d point clouds using point feature histograms. In: Proceedings of the 2009 IEEE/RSJ Int. conference on Intelligent robots and systems. IROS'09, Piscataway, NJ, USA, IEEE Press (2009) 994–1000

[7] Olesen, S., Lyder, S., Kraft, D., Krüger, N., Jessen, J.: Real-time extraction of surface patches with associated uncertainties by means of kinect cameras. Journal of Real-Time Image Processing (2012) 1–14

[8] Blum, M., Springenberg, J.T., Wülfing, J., Riedmiller, M.: A learned feature descriptor for object recognition in rgb-d data. In: ICRA. (2012) 1298–1303

[9] Lee, W., Park, N., Woo, W.: Depth-assisted real-time 3D object detection for augmented reality. ICAT'11 **2** (2011) 126–132

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

[11] Stein, F., Medioni, G.: Structural indexing: efficient 3-d object recognition. Pattern Analysis and Machine Intelligence, IEEE Transactions on **14**(2) (1992) 125–145

[12] Chua, C.S., Jarvis, R.: Point signatures: A new representation for 3d object recognition. International Journal of Computer Vision **25**(1) (1997) 63–85

[13] Ruiz-Correa, S., Shapiro, L., Melia, M.: A new signature-based method for efficient 3-d object recognition. In: Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on. Volume 1. (2001) I–769–I–776 vol.1

[14] Yamany, S.M., Farag, A.A.: Surface signatures: an orientation independent free-form surface representation scheme for the purpose of objects registration and matching. Pattern Analysis and Machine Intelligence, IEEE Transactions on **24**(8) (2002) 1105–1120

[15] Ashbrook, A., Fisher, R.: Finding surface correspondence for object recognition and registration. In: Proc. 5th Eur. Conf. on Computer Vision. Volume II., Freiburg, Germany (1998) 674–680

[16] Chen, H., Bhanu, B.: 3d free-form object recognition in range images using local surface patches. Pattern Recogn. Lett. **28**(10) (2007) 1252–1262

[17] Taati, B., Greenspan, M.: Local shape descriptor selection for object recognition in range data. Comput. Vis. Image Underst. **115**(5) (2011) 681–694

[18] Shang, L., Greenspan, M.A.: Real-time object recognition in sparse range images using error surface embedding. International Journal of Computer Vision **89**(2-3) (2010) 211–228

[19] Campbell, R.J., Flynn, P.J.: A survey of free-form object representation and recognition techniques. Computer Vision and Image Understanding **81** (2001) 166–210

[20] Mian, A.S., Bennamoun, M., Owens, R.A.: Automatic correspondence for 3d modeling: an extensive review. International Journal of Shape Modeling **11**(2) (2005) 253–291

[21] Mian, A.S., Bennamoun, M., Owens, R.: Three-dimensional model-based object recognition and segmentation in cluttered scenes. IEEE Trans. Pattern Anal. Mach. Intell. **28**(10) (2006) 1584–1601

[22] Mian, A.S., Bennamoun, M., Owens, R.A.: A novel representation and feature matching algorithm for automatic pairwise registration of range images. Int. J. Comput. Vision **66**(1) (2006) 19–40

[23] Hetzel, G., Leibe, B., Levi, P., Schiele, B.: 3d object recognition from range images using local feature histograms. In: CVPR (2), IEEE Computer Society (2001) 394–399

[24] Khoshelham, K., Elberink, S.O.: Accuracy and resolution of kinect depth data for indoor mapping applications. Sensors **12**(2) (2012) 1437–1454

[25] NVIDIA: CUDA Programming Guide 5.0. (2012)

[26] Rusu, R.B., Cousins, S.: 3D is here: Point Cloud Library (PCL). In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Shanghai, China (2011)

[27] Zhang, Z.: Microsoft kinect sensor and its effect. MultiMedia, IEEE **19**(2) (2012) 4 –10

[28] Tomasi, C., Manduchi, R.: Bilateral filtering for gray and color images. In: Proceedings of the Sixth International Conference on Computer Vision. ICCV '98, Washington, DC, USA, IEEE Computer Society (1998) 839–846

[29] Chan, D., Buisman, H., Theobalt, C., Thrun, S.: A Noise-Aware Filter for Real-Time Depth Upsampling. In: Workshop on Multi-camera and Multi-modal Sensor Fusion Algorithms and Applications - M2SFA2 2008, Marseille, France (2008)

[30] Wasza, J., Bauer, S., Hornegger, J.: Real-time preprocessing for dense 3-d range imaging on the GPU: Defect interpolation, bilateral temporal averaging and guided filtering. In: ICCV Workshops. (2011) 1221–1227

[31] Holz, D., Behnke, S.: Fast range image segmentation and smoothing using approximate surface reconstruction and region growing. In: Proceedings of the 12th International Conference on Intelligent Autonomous Systems (IAS), Jeju Island, Korea (2012)

[32] Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F.: Computer graphics: principles and practice (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1990)

[33] Wang, L., Huang, M., El-Ghazawi, T.A.: Exploiting concurrent kernel execution on graphic processing units. In Smari, W.W., McIntire, J.P., eds.: HPCS, IEEE (2011) 24–32

[34] NVIDIA: Visual profiler (2012)

[35] Harris, M.: Optimizing parallel reduction in CUDA. NVIDIA Dev. Technology (2008)