GPU Implementation of Multi-View Stereo Algorithms

Master Thesis

GPU Implementation of Multi-View Stereo Algorithms

Luis Rafael Marval Perez

GPU Implementation of Multi-View Stereo Algorithms

by

Luis Rafael Marval Perez

Submitted in partial fulfilment of
the requirements for the degree of
Master of Science

Department of Computer and Mathematical Sciences
Graduate School of Information Sciences
Tohoku University
March 2012

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, the demand of high-accuracy 3D measurements is rapidly growing in a variety of computer vision applications such as human body measurement for medical treatment, component analysis for factory automation, 3D modeling of cultural heritage for digital archiving, etc. Existing 3D measurement techniques are classified into two major types — active and passive. Active measurement employs structure illumination (structure projection, phase shift, moire topography, etc.) or laser scanning, which is not desirable in many applications due to its cost and applicability. On the other hand, passive 3D measurement techniques based on stereo vision have the advantages of simplicity and applicability, since such techniques require simple instrumentation. However, poor reconstruction quality still remains as a major issue for passive 3D measurement. Addressing this problem, Multi-View Stereo (MVS) has been receiving much attention [3].

The idea behind MVS is to reconstruct a complete 3D object model from a collection of images taken from different camera viewpoints. The procedure of the MVS algorithm consists of 3 steps: (i) camera parameter estimation using Structure-from-Motion (SfM) [4], (ii) object reconstruction and (iii) rendering. SfM using SIFT (Scale-Invariant Feature Transform)-based feature point matching [5] is successfully applied to estimate camera parameters for 3D point reconstruction and the efficient software for SfM is also available in [6]. On the other hand, the processing time of object reconstruction is a problem due to dense correspondence matching among a lot of images.

So far, a number of MVS-based reconstruction algorithms have been developed, and the quality of 3D reconstruction result has been improving rapidly. The MVS-based reconstruction algorithms can be roughly categorized into four classes: (i) 3D volumetric approach, (ii) surface evolution-based approach, (iii) depth-map merging approach and

(iv) feature region propagation approach [7]. Using the approaches (i) and (ii), we can reconstruct a high-quality 3D object model compared with the approaches (iii) and (iv). Since the approaches (i) and (ii) are based on the complex optimization technique, these approaches take much time to reconstruct a 3D object model and also may fall into a local minima without setting the proper initial values and the constraint condition. So, these approaches are sometimes used as the post processing of the approaches (iii) and (iv). Using the approaches (iii) and (iv), there is no complex process to reconstruct a 3D object model. In the case of the approach (iii), it takes much time to obtain a dense and high-quality 3D model and needs to remove outliers. In the case of the approach (iv), it takes relatively shorter time compared with other approaches. However, the reconstruction result may be quasi-dense, since the reconstructed 3D model is based on the extracted feature points. Among the above approaches, the depth-map merging approach (iii) is suitable to reconstruct dense and accurate 3D object models except for its high computational cost.

Addressing this problem, this thesis proposes a GPU (Graphics Processing Unit) implementation of the depth-map merging algorithm. This thesis focuses on the reconstruction algorithm proposed by Goesele, et al. [8]. Goesele's algorithm consists of two steps: (i) reconstructing a depth map for each input view and (ii) merging the results into a mesh model. In the step (i), the depth maps are computed using a very simple but robust version of window-matching with a small number of neighboring views. So, we are able to compute the depth maps for each pixel of interest. In other words, the depth map computation in the Goesele's algorithm is suitable for parallel processing with GPU. The GPU is not only a powerful graphics engine, but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially outpaced its CPU counterpart. Nowadays, the effort in the General-Purpose computing on the Graphics Processing Unit (GPGPU) has positioned the GPU as a compelling alternative to traditional microprocessors in high-performance computer systems. In this thesis, we implement the Goesele's algorithm on GPU to develop the practical 3D measurement system which reconstructs a dense and accurate 3D object model from multi-view images. Through a set of experiments, we evaluate the performance of the GPU implementation compared with the CPU implementation.

The structure of this thesis is as follows: Chapter 2 presents fundamentals of multi view stereo algorithms included in the Goesele's algorithm. Chapter 3 proposes the GPU implementation of the Goesele's algorithm. Chapter 4 evaluates the GPU and CPU implementations and discusses the experimental results. Finally, Chapter 5 describes

conclusions of the thesis and the future prospects.

Chapter 2

Fundamentals of Multi-View Stereo Algorithms

In this chapter, we first give an overview of the fundamental components of MVS algorithms. Next, the Seitz [7] classification is explained with the representative cases in state-of-the-art algorithms. Then, the basis of stereo matching is explained to finally describe Goesele's algorithm.

2.1 Multi-View Stereo algorithms

The goal of MVS algorithms is to reconstruct a complete geometric model of an object or scene from a collection of pictures taken from known camera view points (see Fig. 2.1). The reliability of the model is not determined by the algorithm alone, the application defines different requirements and conditions such as density of views, initialization, range of resolutions and texture quality among others. Thus, existing techniques vary widely in range of operation or application, as well as in assumption and behavior. There are six properties that indicate such variations [7].

1- The scene representation: The stages of the reconstruction pipeline of each MVS algorithm determine the required representation(s). Main representations are voxels, level-sets, polygon meshes and depth maps. In voxels representation, a voxel value would tell the occupancy value of the coordinates in a regularly sample grid. Level-sets also uses a regular grid where the value of each coordinate correspond to a function that encode the distance to the closest surface. Polygon meshes represent a surface as a set of connected planar facets [7], they are desirable for final representation because they are suitable for visibility computation in the rendering step. Depth map representations avoid the need for re-

Fig. 2.1 Multi-View Stereo set up, requires multiple images of an object from different view angles

sampling or limited resolution as in the case of three dimensional grids, furthermore, depth map representation is convenient as previous step for meshing. Depth map representation is used in Goesele's algorithm.

2- Photo-consistency measure: This measure establishes how good the compatibility of the reconstructed surface with a set of images is, such compatibility is called photo-consistency. Such measures are crucial in the trade-off of the best feasible reconstruction and the robustness. When the measure is made in the scene space, the used representation (point, patch, volume or polygon) is projected on the input images, then, the measure evaluates the amount of mutual agreement between those projections. Examples of this measure could be the variance of the projected pixels or windows matching score as in the case of Goesele's algorithm. When the measure is on the image space, an estimated of the scene model is used to predict a given view point and then the prediction error is computed. In both cases, the reflectance properties of the scene can be considered and estimated jointly with the surface.

3- Initialization requirements: In addition to the calibrated set of images, MVS algorithms require some information of the geometric extent of the object or scene being reconstructed. In controlled environment such geometric extent is usually given, but for real world environments the algorithms have to include a step to compute it. Common geometric extents are a bounding box, silhouettes or visual hull approximations. The bounding box is particular difficult to estimate in uncontrolled environments. The silhouette estimation provides useful background foreground segmentation. The visual hull is the approximation to the object using the silhouette boundary restriction of the object as illustrated in Fig.

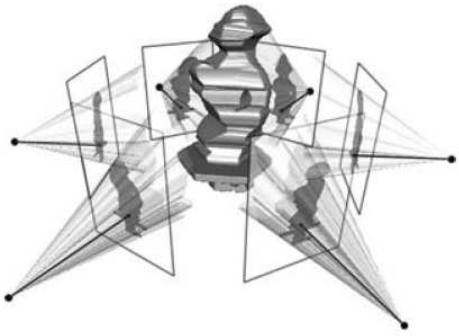

Fig. 2.2 Visual Hull is the surface generated by the projection of the silhouette on each image, taken from [1]

2.2. However, silhouette and visual hull estimations require apply segmentation which may fail to capture the object silhouette. Some algorithms depend on such information to ensure convergence or speed; other algorithms do not use such information but they constraint the range of disparity or depth values, thereby the scene geometry becomes constrained between two planes.

4- Visibility model: Specifies which views to compare with photo-consistency measurements. The traditional approach is to determine the visibility from shape approximations, thus, surface that evolve from a bounding box by carving away the volume can consistently approximated the surface. A similar approach is to compute the visibility from a rough estimation of the surface such as the visual hull. Another approach is to treat occlusion as outliers, this becomes very convenient when close views (neighboring views) are selected because images are more likely to have the same occlusion, this is the case of Goesele's algorithm.
5- Shape prior: A photo-consistency measure alone does not guarantee a precise geometry, particularly in low texture areas. Geometric characteristics can be improved if an adequate constraint in shape prior is used, this also leads to regularization. In binocular stereo, shape priors are helpful to ensure such characteristic, however, in MVS the constraints of multiple views are stronger (i.e. silhouette constraint), and thus, the importance of shape priors becomes lower. Techniques that minimize scene photo-consistency (i.e. variance) naturally seek a *minimal surface*; this feature allows level sets and volumetric min-cuts to converge from gross initial shape. The drawback is that surface's high-curvatures will be smoothed

or distorted, for example, some meshing algorithms incorporate terms that tend to prefer one surface prior such as plane, sphere or triangle. On the other hand, techniques based on voxels or surface carving, naturally seek a maximal surface. Since this algorithms remove voxel that are not photo consistent, the resulting scene is called the *photo-hull*. They are good at reconstructing high curvatures but in the lack of texture surface tends to bulge out. Another kind of approach is local priors; this is suitable for depth maps representation where the depth of neighboring pixel is expected to have close values. The drawback is a tendency towards fronto-parallel surfaces.
6- The reconstruction algorithm: The MVS [7] algorithms can coarsely be categorized into four classes:
First class: 3D volumetric approaches.
Second class: surface evolution techniques.
Third class: algorithms that compute and merge depth maps.
Fourth class: techniques that expand or grow a surface from featured points.

In the next section we describe the MVS categories using some representative cases on the state-of-the-art reconstruction algorithms.

2.2 Reconstruction algorithms classification

In the first class of algorithms, a simple approach is the voxel coloring where a cost function is computed across the volume and then, the voxel with cost over a threshold are reconstructed. Representative approaches define a volumetric Markov Random Field and use max flow and graph cuts to extract the optimal surface. In the work of Furukawa 2009 [9] a particular estimation of the visual hull is obtained and then the visual hull is carved using graph cuts, this optimization is only intended for main features and global photo-consistency, then an iterative refinement step takes care of the fine/local details. In the work of Hoang 2008 [10] an initial cloud of point is fused in a mesh by a s-t cut optimization, then a variational refinement is used to capture small details and ensure regularization and photo-consistency.

In the second class of algorithms, a surface is deformed iteratively in order to minimize a cost function. The surface can be represented as voxels, meshes or level-sets. Space carving methods and level-set methods start from a large volume and shrink inwards;

space carving methods remove inconsistent voxels and level-set methods minimize a set of differential equations formulated in the volume. However, level-sets and some space carving methods allow expansion for refinement or for avoidance of local minima. When meshes are used they evolve moving by inner and external forces that lead to a photo-consistent scene. The algorithm of Hernandez 2004 [11] excels among the state-of-the-art algorithms when dense sets of views are used [12]. This algorithm computes a visual hull and then, a good approximation of the surface is estimate through a depth map algorithm (very similar to the one of Goesele's algorithm), then a snake deformable model is used to evolve the surface by using silhouette and texture information(forces). In the depth map algorithm a speedup of 5 times faster is attained with a pyramid of three image resolution layers.

The algorithms in the third class are divided into two stages: first, depth-map are computed from local groups of pictures, and second, each depth map is merged into a global surface by the means of registration and/or regularization. This division is elegant because allows to choose approaches for each stage independently. In the first stage the simple approach is to use binocular stereo matching. Nevertheless, the algorithm of Goesele 2006 is representative because it extends the traditional binocular stereo matching by using a locality of neighboring view for every depth-map. The merging step is done with the Curless method [13] which is initially intended for range scan data, this method has good regularization and good scaling capability.

The algorithm on Bradley 2008 [14] excels among the state-of-the-art algorithms when sparse sets of views are used [12]. This algorithm enhances binocular stereo matching by considering three possible scaling between windows $(1/\sqrt{2}, 1, \sqrt{2})$. In wide-base line stereo, the scaling is particularly pronounced, this is the case of sparse MVS setups with few cameras. The resulting depth map from the binocular stereo, is initially filtered by rejecting outliers which values are different from its local neighborhood median, then, the point normal vectors are estimated from a vicinity of nearest neighbors. The surface reconstruction is done by down sampling, rejecting outliers that do not fit in the average plane, and finally, joining overlapping clusters in a meshing procedure.

In Campbell 2008 a depth map construction is done by taking into account multiple depth hypotheses for each pixel. The depth estimations are selected from the hypotheses by global optimization, so, to guarantee depth map smoothness and accuracy. A key feature is an unknown tag that implies that the depth cannot be found (indicating lack of texture or occlusion).

In the algorithm of Li 2010 [15], a rectified fast stereo matching with DAISY descriptors

is done up to pixel level. The resulting matches are used to assemble a matching chain across multiple views in the same stereo axis. Then, the matches that have not depth coherence along the views are rejected. Next, a patch model is used by estimating the normal vectors with an optimization of a photo consistence measure. This algorithm is accelerated using OpenMP on a multi-core CPU.

In the fourth class of algorithms, a sparse set of matched features are extracted and then a surface is grown from them. The algorithm in Furukawa 2007 [16] excels among the state-of-the-art algorithms when low texture object scene and sparse set of views are used [12]. It generates a dense collection of small-oriented rectangular patches from Harris corner and Difference of Gaussian features. Next, an iterative refining is made through surface expanding and filtering procedure. Resulting rectangular patches tightly cover the observed surface except in small texture-less and occluded regions. Then, the patches are merged in a triangular mesh that includes a photo-metric refinement. A key feature of this algorithm is the patch approach that alleviates the distortion of rectangle matching window derived from surface projection. The patch model and the scale windows model brings an homography relation that lead to the estimation of optimal point depth and normal [15].

The algorithm presented in Goesele 2007, targets the 3D reconstruction from Internet photo collections where a high number of occlusions is expected (crowed scenes). Then, occluded pixels are determined by a complex selection of neighboring views and validation process. This approach improves the completeness over Goesele's (2006) algorithm at the cost of complexity and inflexibility for more wide applications.

The algorithm of Song 2010 [1] includes use of stereo and silhouette information, the silhouette information is used as constraint of the depth map but also applied to recover the textureless and occluded areas. This algorithm calculates the depth for a low number of windows and then, it spans the depth value average from the windows with high correlation to the ones with low. As the number of windows is low, the depth map construction is faster than the one in Hernandez [11]. An oriented point cloud is conceived by filtering outliers, down-sampling and estimating surface normal from the generated depth map. A point cloud of silhouettes is generated taking vertices from the visual hull mesh. Finally, both point clouds are used for a Poisson surface reconstruction.

A new tendency of MVS algorithms is to focus on the problem of scalability for the use of high resolution pictures (Vu Hoang 2009 [10], Tola and Strecha 2011 [17]) and multiple resolutions (Mucke 2011 [18], Fuhrmann 2011 [19]).

The use of GPU for MVS is not new, we point out four antecedents of 3D reconstruction using GPU: First, Cornelis 2005 [20] implements a pixel based model with connectivity accelerated through shaders, although this case is more of a pure stereo than MVS reconstruction. Second, Zach 2008 [21] proposed a depth map merging procedure that, unlike the Curless 96 [13] procedure, enforces spatial coherence and smoothness. Zach 2008 [21] introduced a total variation regularization by the energy optimization of a variational formulation called $TV - L^2$, the computing of such optimization is accelerated with GPU using Cg and OpenGL. The third antecedent is Labatut 2005 which belongs to the second class, where level-sets algorithm is implemented on Cg and OpenGL: OpenGL features took care of occlusion and textures when rendering reprojection of the images, then, a cross- correlation based energy optimization is done within the level-set framework.

In the last 2 years, a broad variety of MVS related tools on GPU have become available, for example: segmentation with graph-cuts [22] and level-sets [23], sift and surf features matching [24], etc.

In the next sections a review on camera parameters and stereo matching is presented, then Goesele's algorithm is explained.

2.3 Pinhole camera model

A camera is an artifact that captures or maps the 3D world (object space) to a 2D image. This represents a lost of dimensionality given by the process known as the central projection, where the image plane is intersected by the rays coming from the center of projection (or camera center) to the 3D world structure.

In Fig. 2.3, the origin of Euclidean space is set to the center of projection, the image plane also called the focal plane is indicated; the normal to this plane and the center of camera define the principal axis. This axis intersects the image plane in its principal point.

A simplified mapping in the Z-Y plane is presented in Fig. 2.4, in this case the next triangular relation is clearly observed:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = \begin{bmatrix} fx_1/x_3 \\ fx_2/x_3 \end{bmatrix} \tag{2.1}$$

Where x_p, y_p are the image plane coordinates and $(x_1, x_2, x_3) = X_{real}$ are the real world coordinates of the projected point. It is convenient to use a 3-dimensional projective space $\mathbb{P}^3$ for the real world and a 2-dimensional projective space $\mathbb{P}^2$ for the image

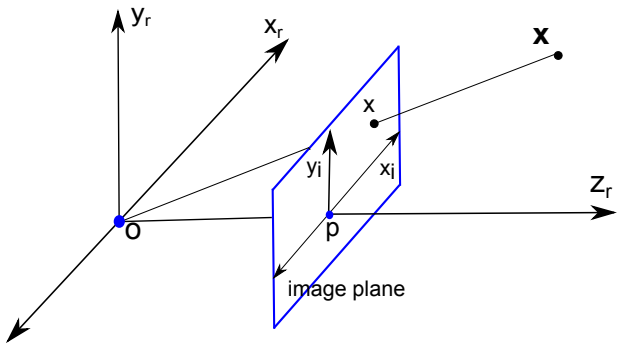

Fig. 2.3 Camera Projection, a point X is projected on the image plane in the intersection between the image plane and the ray from camera center to the point X

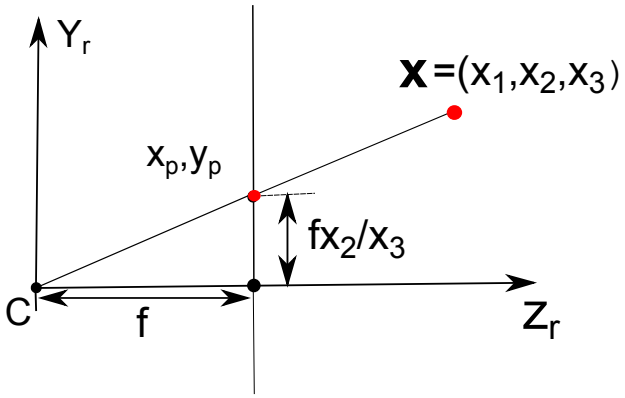

Fig. 2.4 Camera Projection simplified, observe how the real points scale in image plane according to depth and focus

space. Projective spacesdefine homogeneus vectors by adding an extra coordinate to their Euclidean counterparts, this coordinate express a degree of freedom for scaling (like the one when the focal plane is moved). Then, the central projection is just a map from $\mathbb{P}^3$ to $\mathbb{P}^2$ through a transformation given by a 3x4 matrix P, thus the lost of dimensionality is conceived as it is illustrated in the Eq. (2.2):

$$x = PX = P \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix} \tag{2.2}$$

Where x are the projective space coordinates in the image plane, for the simplified case would become:

$$\begin{bmatrix} x_p x_3 \\ y_p x_3 \\ x_3 \end{bmatrix} = \begin{bmatrix} f & & & 0 \\ & f & & 0 \\ & & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{real} \\ 1 \end{bmatrix} \tag{2.3}$$

When we work with digital images, the coordinates are referenced with pixel indexes with no relation to the projection process. In order to solve this practical issue the effect of offsets, axis pixel scalling and skew factor are represented in the projection matrix with (x_{offset}, y_{offset}), (m_x, m_y) and s, as it is illustrated in the Eq. (2.4):

$$\begin{aligned} x &= \begin{bmatrix} m_x f & s & x_{offset} & 0 \\ & m_y f & y_{offset} & 0 \\ & & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} k & 0 \end{bmatrix} \begin{bmatrix} x_{real} \\ 1 \end{bmatrix} \end{aligned} \tag{2.4}$$

So far, the geometric entities (projection center and image plane) of the camera properties have been presented, these correspond to the intrisic parameters of the camera and are resumed by the calibration matrix K. However, if the coordinate system is not fixed in the camera center, the extrinsic parameters are used to model the corresponding translation and rotation. The center of the camera in homogeneous coordinates is the vector C making the camera relative coordinate $X_{cam} = X - C$, the rotation is modeled with the matrix R which represents the direction for camera relative coordinates system $X_{cam} = R(X - C)$, then:

$$P = K \begin{bmatrix} R & -RC \end{bmatrix} \tag{2.5}$$

If we make $t = -RC$ then the projection matrix becomes:

$$P = K \begin{bmatrix} R & t \end{bmatrix} \tag{2.6}$$

In resume, the used notation for the camera (projection) matrix and its parameters is:
P : Projection matrix.
K : Calibration matrix (intrinsic parameters).
C : Camera center (extrinsic parameters).
R : Rotation matrix (extrinsic parameters).

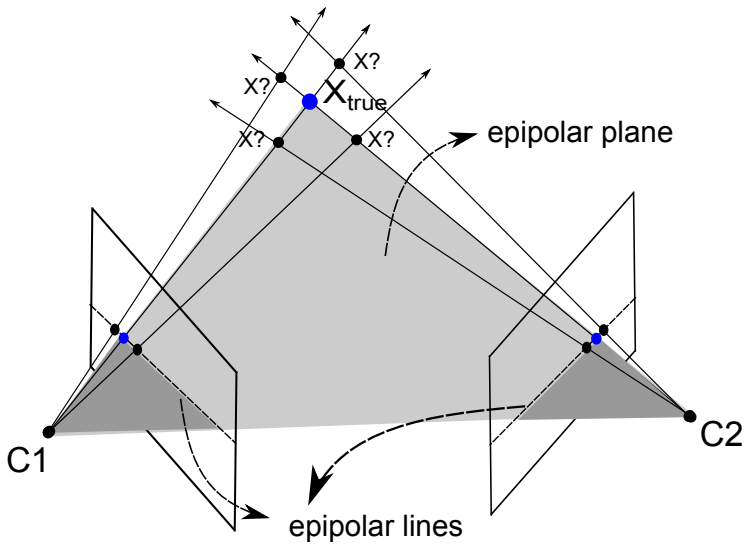

Fig. 2.5 Epipolar geometry, study the mapping properties between two camera identifying the epipolar plane and consequently the epipolar line

In binocular-stereo case, simplification is usually attained by setting the center and axis of real world coordinates to the center and axis of one camera. Then, the intrinsic and extrinsic parameters are solved through a rigorous calibration process. On the other hand when the system is multi-view, the center is commonly in some place of the scene and the parameters are obtained by careful setup or by the means of Structure from Motion.

The intrinsic and extrinsic parameters describe the constraints between each pair of views by the epipolar geometry: in Fig. 2.5, we can appreciate a given real world coordinate and the coordinates of the camera centers define the epipolar plane. This plane intersects the image planes in the epipolar lines; thus every location in an image plane can only correspond to an image coordinate that lies on the produced epipolar line. Notice that the real world coordinate X_{true} can be triangulated if its exact location in each view is known.

2.4 Registration and stereo matching

Image registration is the procedure made to find the mapping between two images; this mapping would be from $\mathbb{R}^2$ to $\mathbb{R}^2$. There are two kinds of registration: rigid and non-rigid. Rigid registration solves simple rotation and translation, while non-rigid registration solves nonlinear mapping usually by a local strategy. Local strategies are based on dense and

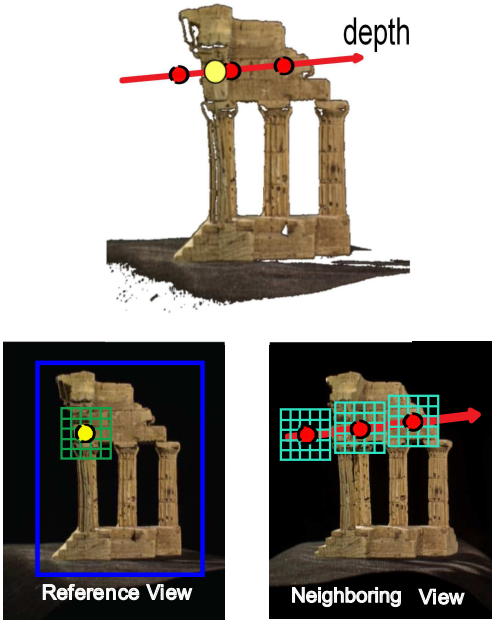

Fig. 2.6 Stereo matching, a window in the reference view is match against multiple windows along the epipolar line in the neighboring view

solid registration of small localities such as block matching, if the distortion is low between those localities.

In block matching, the blocks or windows centered on each intended pixel from a designated reference image are compared with blocks or windows centered in candidate coordinates from a designated target image. The best matches are selected according to a certain matching score.

For stereo image registration the epipolar geometry constraints eliminates one dimension of the problem ($\mathbb{R}^1$ to $\mathbb{R}^1$.), thus simplifying the task for a *Stereo Matching* approach. The distance between the pixels coordinates in the reference and in the target image is called pixel disparity. Then, in stereo matching the candidate coordinates in the target image lie on the epipolar line, such candidates are restricted to an interval of disparity (or depth) values, this is illustrated in Fig. 2.6.

In stereo matching and MVS the scene is usually assumed to have lambertian surface[1] : some example of matching score such as Sum of Absolute Differences (SAD), Sum of Squared Differences (SSD) and cross-correlation rely in this assumption. On the other hand, the score of Normalized Cross-Correlation (NCC) has performed well because it is

Fig. 2.7 Original and normalized windows, left original image, center original window, and right NCC window, NCC window is robust against illumination changes

robust against reflectance and illuminations changes [8, 11, 14, 25].

2.4.1 Normalized Cross-Correlation score

NCC score gives a similarity measure between two windows of size $m \times n$; let v_o and v_1 be the vectors which components are the $m \times n$ elements of the two windows, the Normalized Cross-Correlation is the inner product between the normalized zero mean version of v_0 and v_1. If $N = m \times n$ and $\bar{v}_0, \bar{v}_1$ are the windows averages then the NCC equation will be:

$$NCC_{(v_0,v_1)} = \frac{\sum_{i=0}^{N-1} \left[(v_0(i) - \bar{v}_0) \times (v_1(i) - \bar{v}_1)\right]}{\sqrt{\left[\sum_{i=0}^{N-1}(v_0(i) - \bar{v}_0)\right]^2 \times \left[\sum_{i=0}^{N-1}(v_1(i) - \bar{v}_1)\right]^2}} \tag{2.7}$$

where

$$\bar{v}_n = \frac{\sum_{i=0}^{N-1} v_n(i)}{N} \tag{2.8}$$

In Eq. (2.7) and Eq. (2.8), a window in the reference image is written v_0 and a window in the target image is written v_1. The effect of a normalized windows cancel local brightness and contrast variations, this is illustrated in Fig. 2.7 where a 9×9 original and NCC windows are presented.

In MVS context, a reference image is a reference view and a target image is a neighboring view.

[1]The brightness of the lambertian surface is the same regardless of the observer's angle of view

2.5 Goesele's algorithm

The Multi-view Stereo algorithm introduced by Goesele 2006 extends the traditional binocular-stereo matching scheme to a comparison scheme of one-to-multiple views. Therefore, each image of the input dataset is processed as a reference view and matched against a subset of neighboring views. The back projected ray of each pixel in the reference view is truncated by the bounding box; thus defines a range of possible discrete depth values.

In the algorithm, for each depth value d inside the range, the resulting 3D coordinate is projected on the neighboring view as follow:
- Calculates the 3d world coordinate X_{real} of the reference pixel x_{ref}.

$$\begin{bmatrix} x1 \\ y1 \\ z1 \end{bmatrix} = K_{ref}^{-1} x_{ref}$$

$$X_{real} = R_{ref}^{-1} \left[\begin{bmatrix} d \cdot x1/z1 \\ d \cdot y1/z1 \\ d \end{bmatrix} - t_{ref} \right] \tag{2.9}$$

- Projects the obtained 3D world coordinate in each near view image, $x_{near,i}$ for near view i.

$$\begin{bmatrix} x2 \\ y2 \\ z2 \end{bmatrix} = P_{near(i)} \begin{bmatrix} X_{real} \\ 1 \end{bmatrix}$$

$$x_{near,i} = \begin{bmatrix} x2/z2 \\ y2/z2 \\ 1 \end{bmatrix} \tag{2.10}$$

- Computes NCC correlation for the reference pixel with each of the near view obtained pixel coordinates.
- Compare the NCC values with a threshold, every value over the threshold is valid.
- Calculates the (joint) correlation value, this value is an average of the valid NCC scores or zero if the number of valid matches is less than 2.

Then the depth with higher correlation value is chosen, if this correlation is zero the pixel is discarded from the output. A confidence value is calculated for every recovered

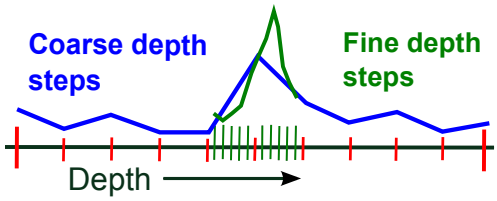

Fig. 2.8 Coarse to fine depth scan, a coarse depth step is used to find the locality of the maximum and then around it a fine step is used

depth. This confidence function increases with the number of valid views and it is used in the merging step.

Goesele proposed a coarse to fine depth-sweep approach: this approach uses a coarse depth step to find the locality of the maximum and a fine step in the locality to find a closer approximation to the maximum, as illustrated in Fig. 2.8. The drawback of this speed-up strategy is that it can lead to outliers if the coarse step is not small enough to find the best estimate.

In Goesele voting approach, one pixel is punished in its confidence measure if its correlation is lower than a threshold in some views, although it can be high in the others. The intention is to solve outliers expecting them to have less weight for the merging step.

However, for the case of uncluttered images, confidence can be low due to occlusions from the target structure itself, so, achieved high correlation values might be good enough, otherwise considerations in the neighboring view selection like the one of Goesele 2007 should be used.

The computation time of the depth map construction in Song 2010 [1] is lower than Goesele 2006 at cost of density of points (or patches). The speedup in Hernandez 2004 [11] with resolution pyramid has still long computation times and involves a lost of accuracy. Then, Goesele's algorithm persist as a representative case which has not accuracy to speedup trade-off.

2.6 Summary

In this chapter, we explained the fundamentals of Multi-View Stereo and presented an overview of each category of MVS algorithms with representative cases. Furthermore, we introduced the fundamentals of stereo matching and then we described the Goesele's

algorithm.

Chapter 3

Implementation of MVS Algorithms on Graphic Processor Units

In this chapter it is introduced an overview the evolution of GPU programming and the used GP-GPU platforms and our interest in them. Then, we explain the general guidelines of the algorithms implementation and, finally, the details of our implementation, its novelty and the implemented configurations are presented.

3.1 GPU Programming evolution

The video game industry has required graphic cards to evolve highly efficient for rendering complex 3D scenes at high frame rates, post-processing and, video encoding and decoding. The interaction of programmers and GPUs start with fixed functionalities of the graphic pipeline that are exposed on the graphics APIs known as OpenGL and DirectX. In the last decade, the programmers become allowed to incorporate in the graphic pipeline stage, custom functions called shaders[1] (Vertex Shaders, Pixel Shaders and recently Geometric Shaders). Early shader functions could only be written in assembly language but this was very inconvenient [26], therefore, two alternatives were developed OpenGL Shading Language *GLSL* and High-Level Shading Language *HLSL* for use with his OpenGL and Direct3D API respectively. Consequently, NVIDIA developed C for Graphics *Cg* that is compatible with both OpenGL and DirectX APIs.

The maturity of shader languages allows some scientific applications to get benefits from it by mapping their problem to the graphic pipeline. In consequence, applications

[1] In 2001, NVIDIA introduced in the GeForce 3, the first programmable vertex processor that executes special functions in the graphic pipeline for the shadowing. Then the term of shader functions or shadders was coined and become extended later to other stages in the graphic pipeline.

in computer vision (even MVS [20, 21, 27]) boost their speed because the suitability of the graphic engine, nevertheless, this is not the case in general parallel computing due to the lack of flexibility of the graphic pipeline structure.

3.2 Platforms for General Purpose computing on GPU

In order to address the graphic pipeline limitation, NVIDIA presented Compute Unified Device Architecture (CUDA) in 2006 as a first GPGPU alternative. Then, Khronos Group released the Open Computer Language API specifications (OpenCL) to be implemented not only for graphics processor but for any hardware (accelerator) that can expose its architecture potential in the OpenCL model. Although, OpenCL is not only for GPU, it has become the vendor independent alternative for GPGPU.

In consequence, the use of GPUs has being expanded to sophisticated image processing such as stereo vision, medical image registration, and pattern recognition. Also in no image rendering/processing such as signal processing, physics simulation, computational finance, computational biology, etc. [28].

In the case of CPU, traditional parallel platforms are: MPI (Message-Passing Interface) for independent Processing Elements connected in a network, it is based on function calls. OpenMP (Open Multi-Processing) that is for shared-memory architectures, it is based on compiler directives. Although OpenMP is suitable to be used for parallel processing on commodity multi-cores CPU, OpenMP model does not fit for GPU architectures.

3.3 GPGPU platforms models

The CUDA API expresses the CUDA programming model by two different interfaces for high level or low level interaction. The high level is given by the CUDA C API which extends C-language allowing the programmer to define C-functions called kernels that will be executed by the GPU devices. The low level is given by the CUDA driver API which main characteristic is that expose more control over the kernel compilation procedure.

The OpenCL API consists on the specification of C/C++ functions to be implemented by vendors. OpenCL API was designed in resemblance to CUDA driver API and therefore their programming models are quite similar, both are based in a scheme of *host - device(s)*

- compared with a CPU and co-processor.

The abstraction of these models for parallel programming is based on two hierarchical levels of parallelism: First, the level of fine-grained data or fine execution entities called thread in CUDA and work-item in OpenCL, each thread/work-item executes the kernel function. Second, the level of coarse-grained data or task entity called block of threads in CUDA and work group in OpenCL [28]. In the rest of this document we will use CUDA nomenclature, also GPU device and device will be used interchangeable.

Then, a parallel program is mapped between these two levels, breaking the problem in independent coarse tasks executed by the blocks. These tasks are broken up by threads which execute a function called kernel. The threads inside a block can cooperate efficiently between them through a shared cache and synchronization primitives.

Each of these blocks' execution is independent; they are scheduled in any order, concurrently or sequentially, allowing scalability of the whole task among the number of physical processors [28]. Therefore, program execution can take advantage of the number of processors that is very variable among the OpenCL devices. Also, the number of processors in a NVIDIA GPU varies with their cost and generation.

In addition especial features may be present, either for OpenCL devices or for NVIDIA's GPUs: in OpenCL interface individual features are exposed through extensions, in CUDA these features are indicated according to the computer capabilities index of the NVIDIA's GPUs. Both platforms provide scalability of these features by defining macros for preprocessing branches, so as to let programs take advantage of them, if such features are present.

In the evolution of video cards a two and three dimensional memory locality was developed in order to take advantage of the pixel to processing-core mapping. Therefore, the array of threads in a block was designed to be an array of 1, 2 or 3 dimensions leading to a memory access pattern known as coalesced. A coalesced access of memory occurs when the address locality and alignment meets certain criteria, taking advantage of the distribute bus of the main memory. The array of blocks is called grid and it is organized in 2-dimensional arrays, just for simplified the task partitions. In Fig. 3.1 this scheme is visualized.

The GPU device external DRAM memory is mapped as GPU own memory space, this is referred to the device memory, and it is not directly accessible from CPU. The host (CPU) is in charge of the device memory allocation and de-allocation, it is also in charge of the memory transfers between the host and the GPU device [28]. In addition, GPGPU

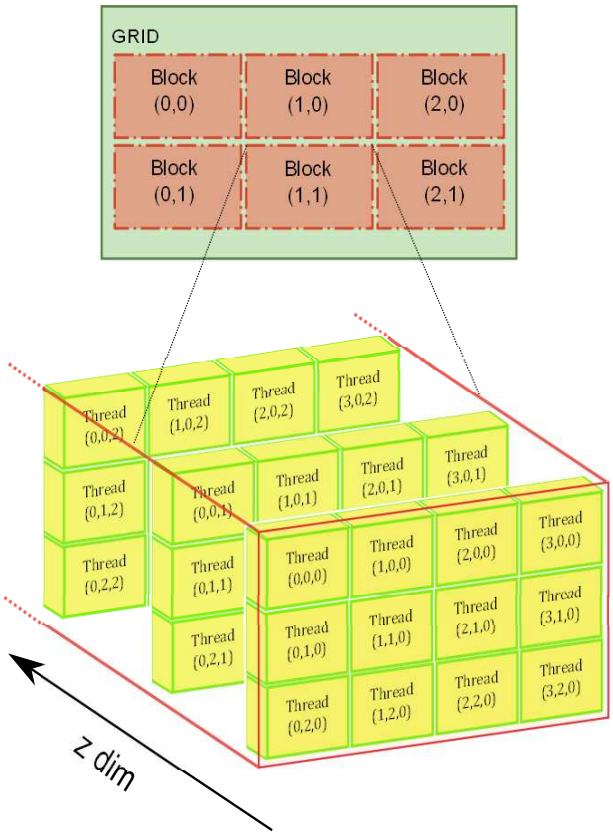

Fig. 3.1 CUDA block and thread arrays dimension and hierarchy

programming is alike an heterogeneous computing, because the CPU not only has to made the transfers of memory and request the executions, but also can perform some part of the task.

In NVIDIA devices, each thread has a local memory which physically consist of low latency registers and device memory; thread's variables typically reside in registers. Threads belonging to the same block can cooperate efficiently through shared memory (as shared L1-cache); this is expected to be a low-latency memory near each processor core. Every thread has access to a constant memory space, a texture memory space, and a global memory space (implemented in external DRAM) that are persistent across kernel launches by the same application. Constant memory is intended for broadcasting while texture memory is intended for fast extensive read access. In Fig. 3.2 you can appreciate a diagram of the model for CUDA.

Texture memory is cached and can perform especial addressing modes and interpolation.

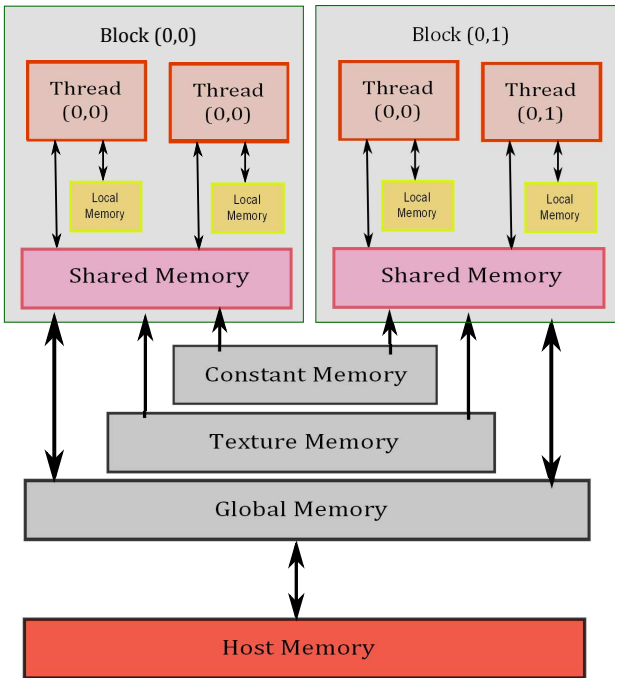

Fig. 3.2 CUDA memory scheme, threads have access to three scopes of memory: Local, Shared, Constant, Texture and Global

The texture can be bounded to especial allocation of memory prepared for 2 dimensional access (horizontal and vertical locality) or CUDA arrays that are opaque memory layouts optimized for fetching. In OpenCL the texture object equivalent is called sampler and specifies the addressing and interpolation, but only can be used with opaque memory layouts called Image objects thus, it cannot be used with 2 dimensional memory allocations. Another difference is that OpenCL allows writing in Image objects while CUDA does not in CUDA arrays.

CUDA model provides a scalable abstraction of physical architecture features, maximum performance can be attained if architectural features are considered.

In order to illustrate how these models are physically implemented, in the next section we expose the outlines of the CUDA architecture for computing and graphics models.

3.3.1 NVIDIA architecture

GPGPU computing on NVIDIA is done on CUDA architectures: TESLA and FERMI architectures which are based on scalable array of Streaming Multiprocessors (SM). In

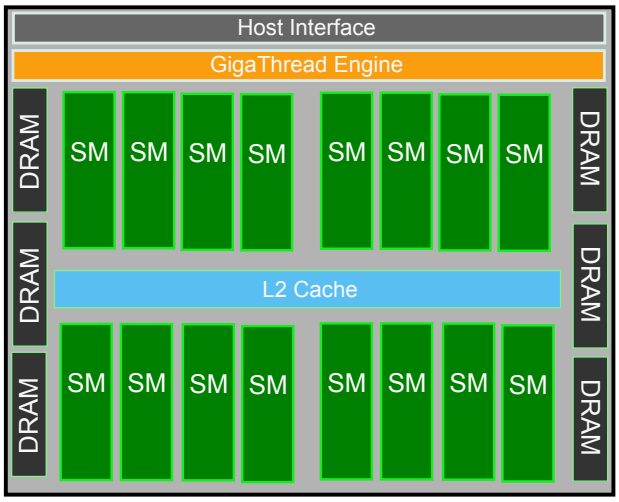

Fig. 3.3 Fermi GF100 architecture is based on Scalable Processors Array of Streaming Multiprocessors (SM)

Fig. 3.3 the case of Fermi GF100 is presented: this architecture consist in sixteen SM, an unified cache L2 and the device DRAM [29]. Each SM concurrently executes different kernels or shaders [30], as it is observed in the Fig. 3.4 a FERMI SM consist of thirty two CUDA cores also called Streaming processors [30], four Special Function Units (SFUs) for transcendental functions (sine, cosine, square root, etc.), sixteen load store units (LD/ST), a register array, a shared memory, caches, and four texture units [29].

CUDA architectures employ an architecture coined as Single Instruction Multi Thread (SIMT), this is an hybrid extension of traditional Single Instruction Multiple Data (SIMD) and Single Instruction Single Data (SISD) architectures, hence is not constrained by the SIMD width and can perform branching.

The SM SIMT unit creates, manages, schedules, and executes threads in group of 32-parallel threads called warp. The SM maps the warp threads to the SP cores, and each thread executes independently with its own instruction address and register state. In Fermi GF100, each SM features two warp scheduled and two instruction dispatch unit, allowing two warps to be executed at the same time; a SM manages a pool of 48 warp of 32 threads per warp allowing up to 1536 threads to be executed concurrently [30, 31].

A block execution is distributed along several warps of one SM. Every thread inside a warp start together and every warp executes one common instruction at a time. If threads diverge via conditional branch, the warp serially executes each branch path taken, disabling

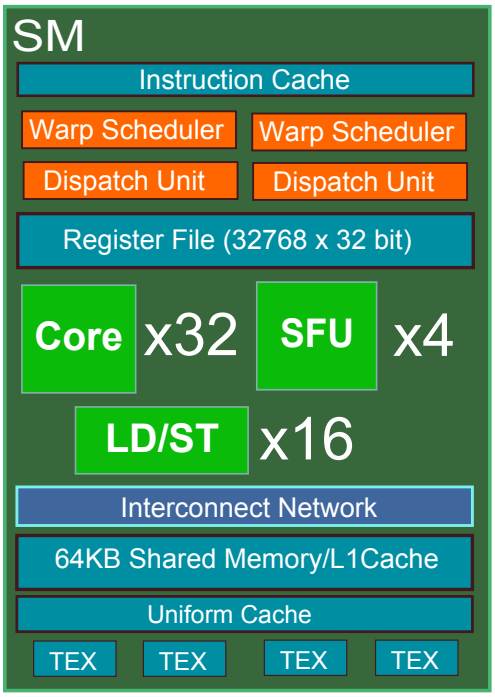

Fig. 3.4 Streaming Multiprocessor in Fermi architecture, each SM consist of thirty two CUDA cores, sixteen Load/Store units (LD/ST), four Special Function Units (SFU)

threads that are not on that path; the threads converge back to the same execution path when all paths are completed. Thus, full efficiency is attained when all threads of a warp agree in the execution path whether they disagree or not with others warp [30, 31].

The SM coalesced individual DRAM access of parallel threads in the same warp into fewer memory block accesses when the address falls in the same block and meets an alignment criteria [31]. As this access is slow, shared memory is used to store the data that will be accessed multiple times. Also NVIDIA GPUs usually provide atomic read-write memory instruction, for concurrent collaboration or reduction in the global and in the local scope [31].

3.4 Algorithm implementation

In resume, the implemented program receives as input an image data set and its camera parameters; during the execution the program writes a file in PLY format with the resulting depth map.

The host program is implemented in C++ and it includes an implementation of the algorithm in CPU with the alternative of OpenMP. OpenMP on CPU offers a fair reference in order to compare the multi-cores capabilities of nowadays CPUs with the many-cores capabilities of GPUs. For the CUDA implementation a wrapper is written in CUDA files (.cu) in order to isolate GPU memory management calls and kernel calls from the program logic. Functions defined in .cu files can be called using the " extern "C" " linker directive in C++. For the OpenCL implementation the wrapper is written in C++ files (.cpp). Kernel functions is written in header files, therefore, the isolation between program logic, platform calls and kernel functions allows software reuse.

The GPU memory capacity is limited so the whole process is divided in tasks with a fixed small numbers of reference views N_{Ref} (1 or 2) to be processed.

In our implementation we do not assume we have the bounding box instead we assume that we have depth ranges although they are calculated from the bounding box. Depth range is more practical for GPU implementation and more reliable for real practice. Not all the pixel of the reference image are used, we took a subset given by a bounding frame that enclose the projection of the bounding box.

In the initialization the host performs the following duties:
- To calculate which are the corresponding neighboring views for each reference view.
- To calculate the depth range that enclose the bounding box and the dimensions of the bounding frame for each reference view.
- To allocate CPU and GPU necessary memory spaces.

Memory assignment In the program we refer to the processing of a number of views loaded as a task. In order to execute one task, the host stores the corresponding view parameters in the device constant memory. In the global memory, the host stores an array of the images; a column of this array is conformed by a reference view followed for its neighboring views, this array can be observed in Fig. 3.5, so the number of columns correspond to the number of reference image N_{Ref} to be processed in that task. In CUDA this array is in a 2D memory spaces allocation and it is accessed through CUDA texture fetching. In OpenCL we used an Image object and it is accessed by a sampler object. For compatibility with Image object and the CUDA texture, four components for RGB color representation are used instead of three components.

The direct strategy of the kernel is store in the Local memory of the kernel: the reference view parameters $K^{-1}, R^{-1}, -t$, the neighboring views parameters P, the reference windows and the target windows.

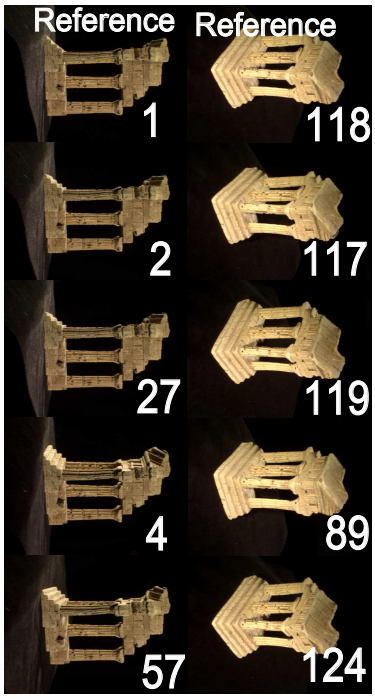

Fig. 3.5 Storage of images in GPU memory, an array row is conformed by the reference view images and the corresponding neighbor views in the rest of columns

When the host calls a kernel function the output is an array of the estimated depth, correlation, confidence values and 3D world coordinates. This output is an array with the size of the maximum bounding frame.

3.4.1 Task partition

Straight forward parallelization is done by pixel level partition: every thread takes care of a limited number of pixels selected by its index coordinates in the block. In this way, the block directly covers a range of pixels where neighboring pixels correspond to neighboring threads. After each re-projection procedure on every neighboring view the locality will prevail so the texture fetching is more likely to be coalesced. The first block dimension index $blockDim.x$ breaks the image in horizontal sections that are given to each block. The second dimension $blockDim.y$ indicates the number of reference pictures in memory.

When a block process a range of pixels, it continues with a next range of pixels inside the horizontal section, this serves as a thread recycling factor. There is not enough processing

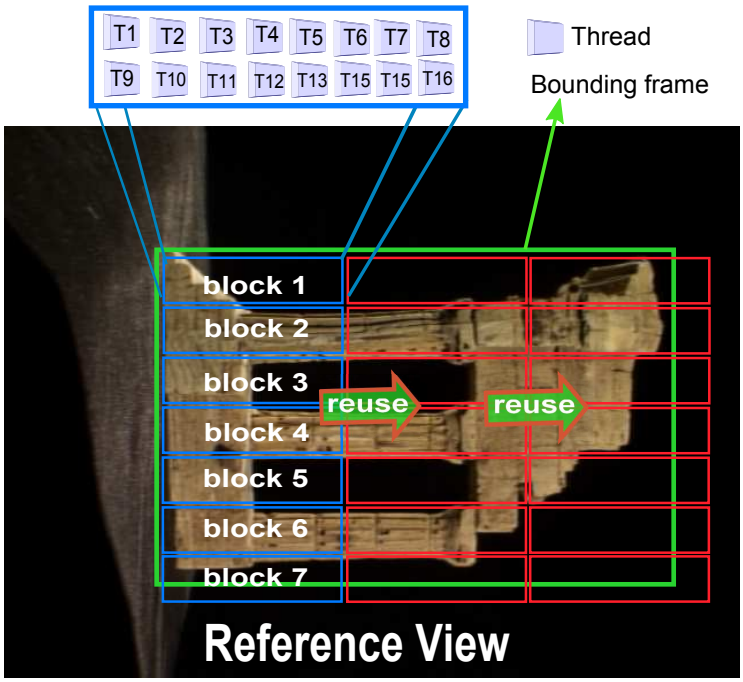

Fig. 3.6 Pixel to thread mapping and reuse. In this example each thread in every block process 3 pixels

capacity to run at the same time a thread per pixel of the whole bounding frame, then we can make a tradeoff between the serial processing for a kernel and the overhead caused by the thread allocation, scheduling and inherent initialization of the kernel - algorithm - function. In Fig. 3.6 an illustration with grid dimension $(1, 7)$ is shown, observe that the thread is reuse 3 times to scan the whole section, then grid of block will have a size of $ceil(height/(blockdimy), N_{Ref})$.

3.5 Kernel function implementation

The flow chart of the kernel function is presented in Fig. 3.7. The initialization consists in loading view parameters from the constant memory to the Local memory; and pre-calculating different indexes related to thread id. The execution consists of three hierarchical loops: pixel reuse loop, depth loop, neighboring view loop. The reference NCC window is loaded in the Local memory inside the pixel loop. Each neighboring windows is loaded in the Local memory inside the near view loop, once loaded the NCC is calculated

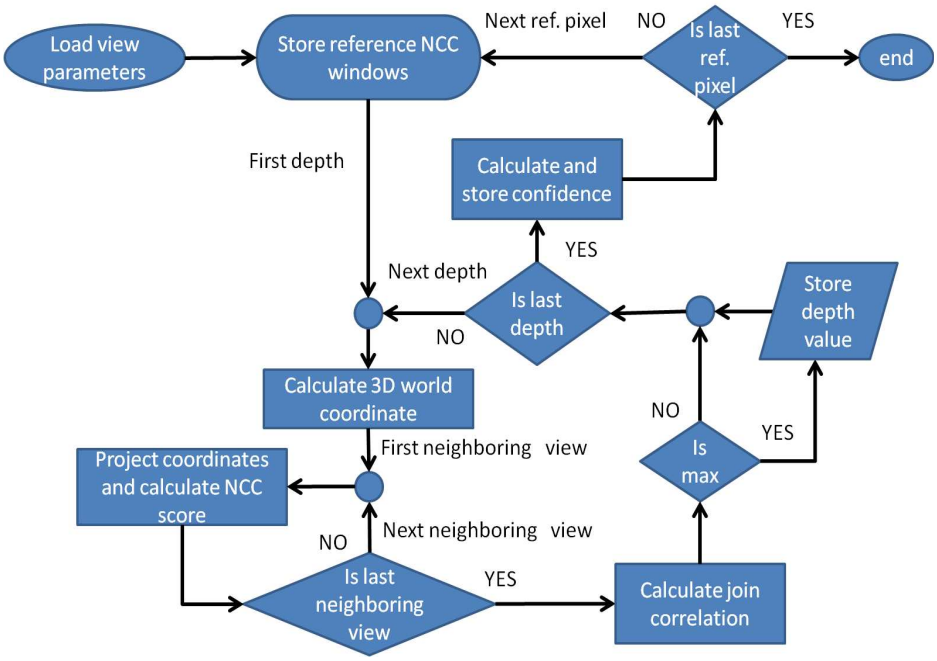

Fig. 3.7 Flow chart of kernel function, consist in three hierarchical loops: pixel reuse loop, depth loop, near view loop

and stored.

Recalling the NCC equation, it can be appreciated that the denominator in the expression is the square root of the product of two scalar values, this division can be done as last operation and not one per element.

$$NCC_{(v_0,v_n)} = \frac{\left[\sum_{i=0}^{N-1}\left[(v_0(i)-\bar{v}_0)\times(v_n(i)-\bar{v}_n)\right]\right]}{\sqrt{\left[\sum_{i=0}^{N-1}(v_0(i)-\bar{v}_0)^2\right]\times\left[\sum_{i=0}^{N-1}(v_n(i)-\bar{v}_n)^2\right]}} \tag{3.1}$$

In Eq. (3.1) is computed serially for each neighboring windows, thus only the storage for a windows is needed for all the neighboring windows.

Therefore, a straightforward implementation of Eq. (3.1) is characterized by the storage of two windows, this approach will be called implementation type 1. Windows storage requires a high amount of Local memory which can be implemented either in register or device memory. The amount of Local memory used will impact the performance due two resulting limitations: first, decrease device occupancy because registers are a

limited resource that is distributed among the threads executed concurrently in a SM. And second, slowdown the memory access by increasing the used amount of device memory used. Addressing this issue, we propose the usage of summation expressions: -For the denominator terms in Eq. (3.1) can be converted as follow:

$$\begin{aligned}&\sum_{i=0}^{N-1}(v_n(i)-\bar{v}_n)^2\\&\quad=\left[\sum_{i=0}^{N-1}v_n(i)^2\right]-2\Big(\sum_{i=0}^{N-1}v_n(i)\bar{v}_n\Big)+\left[\sum_{i=0}^{N-1}\bar{v}_n^2\right]\end{aligned}\tag{3.2}$$

Then, is reduced to:

$$\begin{aligned}\sum_{i=0}^{N-1}(v_n(i)-\bar{v}_n)^2&=\Big[\sum_{i=0}^{N-1}v_n(i)^2\Big]-N\bar{v}_n^2\\&=\sum_{i=0}^{N-1}v_n(i)^2-\frac{\Big[\sum_{i=0}^{N-1}v_n(i)\Big]^2}{N}\end{aligned}\tag{3.3}$$

In similar way, the numerator in equation (3.1) is reduced to:

$$\begin{aligned}&\sum_{i=0}^{N-1}\big[(v_0(i)-\bar{v}_0)\times(v_n(i)-\bar{v}_n)\big]\\&\qquad=\sum_{i=0}^{N-1}\big[(v_0(i)-\bar{v}_0)v_n(i)\big]\end{aligned}\tag{3.4}$$

Hence

$$\sum_{i=0}^{N-1}(v_0(i)-\bar{v}_0)\bar{v}_n=0\tag{3.5}$$

Then, the score equation becomes:

$$\begin{aligned}&NCC_{(v_0,v_n)}=\\&\frac{\sum_{i=0}^{N-1}\big[(v_0(i)-\bar{v}_0)v_n(i)\big]}{\sqrt{\big[\sum_{i=0}^{N-1}(v_0(i)-\bar{v}_0)^2\big]\times\big[\sum_{i=0}^{N-1}v_n(i)^2-\frac{\left[\sum_{i=0}^{N-1}v_n(i)\right]^2}{N}\big]}}\end{aligned}\tag{3.6}$$

This will be called implementation type 2 in the rest of this document. Type 2 requires the storage of the reference window and the terms related to the neighboring windows are stored in the three summations presented in Eq. (3.7):

$$\sum_{i=0}^{N-1}v_n(i)^2;\sum_{i=0}^{N-1}v_n(i);\sum_{i=0}^{N-1}\big(v_0(i)v_n(i)\big)\tag{3.7}$$

In type 2, the NCC score of each neighboring view windows is still computed serially and the memory access pattern is not affected.

Further reduction of the required memory can be attained by converting the denominator of Eq. (3.6) as follow:

$$\begin{aligned}\sum_{i=0}^{N-1}\big[(v_0(i)-\bar{v}_0)v_n(i)\big] &= \sum_{i=0}^{N-1}\big[(v_0(i)v_n(i)-\bar{v}_0v_n(i)\big] \\ &= \sum_{i=0}^{N-1} v_0(i)v_n(i) - N\bar{v}_0\bar{v}_n\end{aligned} \tag{3.8}$$

Then, the score equation becomes:

$$NCC_{(v_0,v_n)} = \frac{\sum_{i=0}^{N-1} v_0(i)v_n(i) - N\bar{v}_0\bar{v}_n}{\sqrt{\big[\sum_{i=0}^{N-1} v_0(i)^2 - \frac{\left[\sum_{i=0}^{N-1} v_0(i)\right]^2}{N}\big] \times \big[\sum_{i=0}^{N-1} v_n(i)^2 - \frac{\left[\sum_{i=0}^{N-1} v_n(i)\right]^2}{N}\big]}} \tag{3.9}$$

Thus, we need two summations for the only reference view terms:

$$\sum_{i=0}^{N-1} v_o(i)^2; \sum_{i=0}^{N-1} v_o(i)$$

For each window of the neighboring views, we need three summations in Eq. (3.10):

$$\sum_{i=0}^{N-1} v_n(i)^2; \sum_{i=0}^{N-1} v_n(i); \sum_{i=0}^{N-1} \big(v_0(i)v_n(i)\big) \tag{3.10}$$

The drawback of this approach is that the reference window has to be read for each depth and the memory access is not consecutive on each view. Therefore, the efficiency of the texture fetching is decreased. This approach will be called the implementation type 3 in the rest of this document. As color images are used multiplication between pixel values is solved as an inner product. Consequently, the summations of order one are of three components (color images) and the others are scalar values. Therefore, if a windows of 5x5 and 4 neighboring views is used: the window storage needed for type 1 is (5x5)x3x2 = 150 scalars values, for type 2 is (5x5)x3+(1+3+1) = 80 scalars values and for type 3 is (1+3)+(1+3+1)x4=30 scalars values. The implementations type 2 Eq. (3.6) and type 3 Eq. (3.9) are novelty of this thesis as it will be appreciated in the chapter 4.

3.5.1 Reusable kernel function

We took advantage of include directives in order to maintain an unified kernel function file for every implementation. Then, the kernel function is written in a C header file (.h) with several preprocessing branches for each implementation.

In OpenCL, the kernel file is passed as a string to the runtime compiler, then, the macro definitions are added to this string before the compilation. Some macros were defined in order to have a translation point between CUDA and OpenCL. For example:
-To translate the thread/work-item index id from threadIdx structure to get_local_id functions.
-To translate the built-in type constructor of vectors, from make_type(value) in CUDA, to the C casting form (type)(value) in OpenCL.
-To translate space qualifiers.

This is illustrated in the next code fragment used as a common header where UCL_CUDADR and UCL_OPENCL macros switch the direction of the translation.

Listing 3.1 CUDA-OpenCL mapping

```
#ifdef UCL_CUDADR
#define thrx              threadIdx.x
#define thry              threadIdx.y
#define thrz              threadIdx.z
#define GLOBAL_ID_X (blockIdx.x*blockDim.x + thrx)
#define GLOBAL_ID_Y (blockIdx.y*blockDim.y + thry)
#define GLOBAL_ID_Z (blockIdx.z*blockDim.z + thrz)
#define __private
#define __constant
#endif

#ifdef UCL_OPENCL
#define __device__
#define __global__        __kernel
#define __shared__        __local
#define thrx              get_local_id(0)
#define thry              get_local_id(1)
#define thrz              get_local_id(2)
#define GLOBAL_ID_X       get_global_id(0)
#define GLOBAL_ID_Y       get_global_id(1)
#define GLOBAL_ID_Z       get_global_id(2)
#define make_float3       (float3)
#define make_float4       (float4)
```

```
#define make_uchar3      (uchar3)
#define make_uchar4      (uchar4)
#endif
```

3.6 Summary

In this chapter, we presented an overview of the evolution of GPU programming, also we explained the characteristics of CUDA and OpenCL platforms. Then, we introduced general guidelines of the algorithm implementation, and finally, the details of the GPU memory assignment and proposed implementations were explained.

Chapter 4

Performance Evaluation of MVS Implementations

In this chapter, the designed experiments are briefly explained. Then, the computation times of the different implementations are presented. Next, we discuss about the obtained speedup with each platforms and implementations. Finally, we present the resulting depth maps.

4.1 Experiment description

The next experiments are done over a set of 6 reference images with 4 neighboring views from the standard middelbury data set dense temple which is illustrated in Fig. 4.1. This dataset consist of 312 calibrated images taken in a semi-spherical fashion, the resolution is 640×480 pixels. At this resolution, a pixel in the image spans roughly 0.25mm on the surface of the object which dimensions are 10cm×16cm×8cm [7].

The selected six pictures cover a good portion of the temple, their indexes are 1, 12, 54, 118, 160 and 283. Only the computation times of the kernel function were considered, initialization or post processing that in on our case writing the output to the hard disk, was not considered. The CPU times are very high and they do not have significant variations in different executions but the GPU computing times vary close 5%, which can be unreliable for speedup determination, and so, the GPU computing times are determined as the average of 5 executions.

We used the parameters of the Goesele's paper [8]: Color images, coarse depth step used Δd =2.5mm, fine depth step $\Delta d/10 = 0.25$mm.

The hardware resource used in the experiments is described as follow:

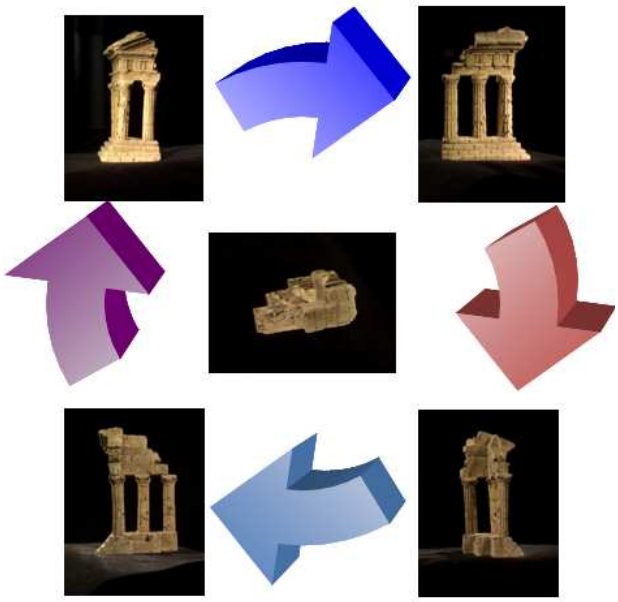

Fig. 4.1 Data sets benchmark for MVS, temple data set, in dense version cover 312 view angles

CPU-Host: we used an Intel core i7 CPU 975 3.33 GHz with a 3GB of RAM and with 4 physical cores. The used operating system is Microsoft Windows 7 Professional Service Pack 1, 32-bit.

NVIDIA video card: we used a GeForce GTX 580 with a main memory of 1.5 GB of RAM, GPU clock rate of 1.544 GHz and memory bandwidth of 192.4 GB/sec. CUDA computer capability 2.0 which corresponds to NVIDIA Fermi architecture that has 16 Streamming Multiprocessors (SM) each with 32 CUDA cores giving a total of 512 CUDA cores. The number of register per SM is 32K register.

AMD video card: we used an AMD Radeon HD 6970 of AMD cayman series. It has a main memory of 2 Giga Bytes, GPU clock rate of 0.88 GHz and memory bandwidth of 176.4 GB/sec. The GPU has 24 SIMD engines, each of these SIMD engines includes 16 thread processors giving a total of 348 thread processors. Each thread processor has four ALUs, ginving a total os 1536 ALUs. The number of registers per SIMD engine is 16K register.

Table 4.1 Computing times per implementation and platform

Platform	Implementation	Kernel execution time(sec)
CPU	Type 1	10190.8209
CPU	Type 2	9026.8614
CPU	Type 3	10847.3911
CPU+ OpenMP	Type 1	2356.6144
CPU+ OpenMP	Type 2	2090.8091
CPU+ OpenMP	Type 3	2490.2887
CUDA	Type 1	5.7703
CUDA	Type 2	4.0575
CUDA	Type 3	6.339
OpenCL NVIDIA	Type 1	6.5059
OpenCL NVIDIA	Type 2	4.1426
OpenCL NVIDIA	Type 3	3.2203
OpenCL AMD	Type 3	1.5146

4.2 Computation times

In the table 4.1 the resulting kernel execution times are presented. The computation times for CPU are used as a first reference, the best CPU implementation was the type 2 with an average time of 25 min per image. However, a direct CPU implementation is unfair for commodity multi core CPUs, then, OpenMP is used in order to take advantage of the multi-core capability. OpenMP results in speedups around 4.3 times faster for each implementation, such speedups correspond directly to the number of cores.

For GPU executions, the number of reference images processed by the kernels in a kernel execution is two. In this way the kernel's loading-time is reduced, and the number of required threads will be high enough to insure that GPU is busy.

4.2.1 NVIDIA execution times

In Fig. 4.2, a plot of the NVIDIA's execution times is presented. It is appreciable that type two represent an improvement over type 1, for type 1 and type 2 CUDA has slightly

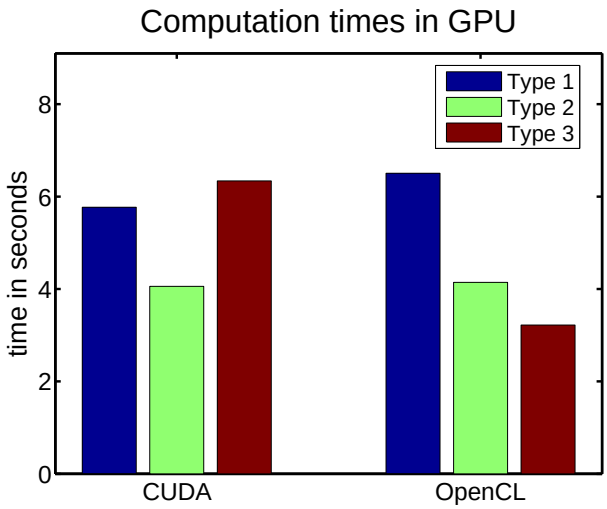

Fig. 4.2 The NVIDIA execution times of the algorithm implementations for a set of six images

time improvement over OpenCL. It is interesting that type 3 in OpenCL gives the lowest time but in CUDA type 3 is particularly slow. that in OpenCL NVIDIA has a better use for the accumulator versions 2 than CUDA.

Although, type 3 has lowest needs of Local memory, the balance of register and device memory used is up to the compiler. Optimizations performed during compilation phase can prioritize the computing over device memory access. This is a logical consideration assuming a high efficiency of the cache L2 (for device memory) in Fermi architectures. However, the implemented algorithm main bottleneck is in memory access. The NVIDIA's visual profiler reveals that CUDA reads 605 Giga bytes in cache L2 while OpenCL only reads 2.31 Giga bytes.

The best NVIDIA execution average times are 0.67 sec for type 2 in CUDA and 0.53 sec for type3 in OpenCL.

4.2.2 AMD execution time

For the AMD card implementation type 1 and type 2 could not be properly executed and so, only the execution time of implementation type 3 is presented.

The execution time in AMD OpenCL was 1.51 sec, that is less than half the time of OpenCL NVIDA and the average time per image is 0.25 seconds. The AMD card

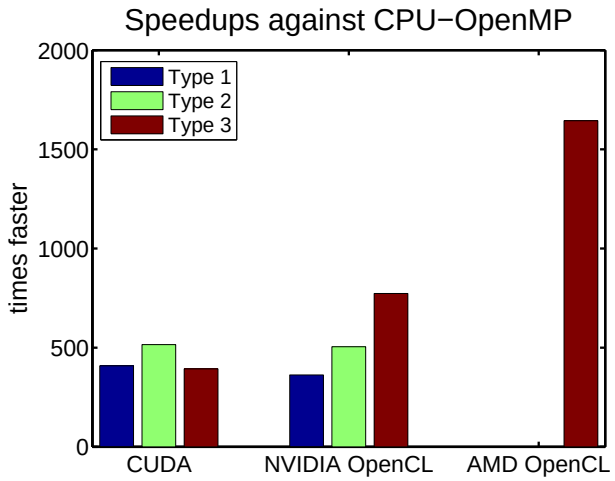

Fig. 4.3 The attained speed ups against CPU+OpenMP implementations

performance improvement is associated to the coarse granularity of the processing elements on each GPU. For the implementation type 2 and type 3 in NVIDIA OpenCL the occupancy indicated in the profile is 25%, the occupancy is limited by the number of registers (32K registers of 32 bits per SM) resulting in only 4 CUDA cores used on each SM. On the other hand, in the AMD card each SIMD engine has 8K registers of 128 bits, this allows higher occupancy of SIMD engines, moreover, the number of SIMD engines is 24 in the AMD GPU while the number of SM is 16 in the NVIDIA GPU. In concequence the efficiency of the AMD GPU is higher than NVIDIA at both levels (SIMD/SM, CUDA-core/Thread-processor).

4.3 Discussions

The use of OpenMP directives was very straight forward in terms of prototyping, so OpenMP is the fair reference for commodity multi core CPUs. Then the "fair" speedups are obtained by implementation-wise and are presented in Fig. 4.3.

In CUDA a speedup up to 515 times faster is attained with type 2. In NVIDIA OpenCL the speedup reach 773 times faster with type 3. In OpenCL AMD the speedup of 1644 times faster. In every platform, even for CPU and CPU+OpenMP, type 2 shows outperform the straight-forward implementation, on the other hand, type 3 is suitable for GPU architectures but it depends on the platform to be efficiently implemented.

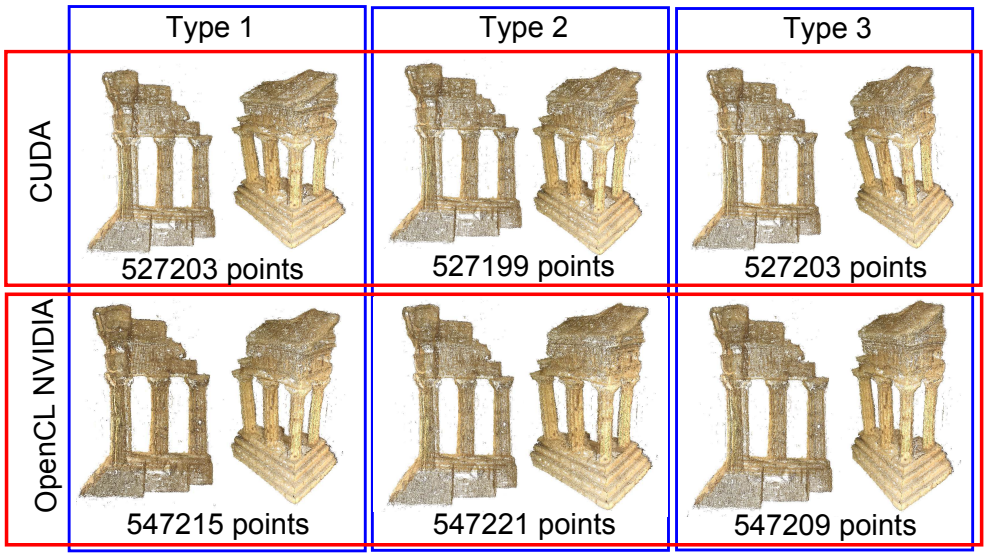

Fig. 4.4 Two views of each resulting depth map after confidence filtering. Six reference images were enough to recover a consistent object shape

The considerations taken in type 2 and 3 are typical from signal processing systems, it is not strange that the same ideas apply to GPUs because they share several characteristic addressed in signal processing systems. Then, massive parallel computing on GPU is a convergence point that exposes either high programmability for flexibility or architecture dependencies for efficiency.

The Goesele's algorithm implementation on GPU exploits several capabilities such as high bandwidth, two dimensional memory locality and coarse parallelism. Unlike CPU which constrained the algorithm's applicability, GPU offers new possibilities of applications allowing further improvement and flexibility.

4.3.1 Resulting depth map

The raw output of the algorithm is very noisy *per se*. This noise is mainly caused by the resulting outliers, the confidence value obtained from the algorithm can be used to filter outliers regardless it is intended for clean these outliers during the merging process. In Fig. 4.4, the resulting depth map processing six images for CUDA and NVIDIA OpenCL platform is presented.

The reconstructions are consistent with the temple shape. There is not an appreciable

difference among them. However, the resulting number of pixels of each reconstruction shows two numerical effects related to the finite representation of floating values:
-The difference between the shapes of the equations type 1 Eq. (3.1), type 2 Eq. (3.6) and type 3 Eq. (3.9) causes slightly differences on the computation result.
-CUDA and OpenCL have different configurations for floating point operation by default.

In addition Fig. 4.5 presents the depth map filtered from a full set reconstruction.

Fig. 4.5 Depth map for whole temple set, five angles of the reconstructed depth map using the original algorithm over all the 312 images of the data set. Rendered with MeshLab [2]

4.4 Summary

In this chapter, the computation time over several configurations and platform were presented. Then, a discussion about the attained speedups is presented allowing us to state the high efficiency of GPU implementations against CPU implementations. Finally, the resulting depth maps of temple for the different configurations were presented.

Chapter 5

Conclusions

This chapter summarizes the main points discussed in previous chapters and then it concludes this thesis with future work plans.

High-accuracy 3D reconstruction is a task spanning on several fields of applications, and crucial complement of 3D measurement in computer vision. 3D reconstruction with Multi-View Stereo is flexible and requires simple instrumentation. Particularly, Multi-View Stereo algorithms based on depth maps can reconstruct dense and accurate 3D object models. However, these algorithms applicability is limited by high computational cost; such is the case of the algorithm introduced by Goesele in 2006 which is representative depth map among the-state-of-the-art algorithms. Addressing the long computation time, GPU implementation of the Goesele algorithm is proposed in this thesis. The GPU is not only a powerful graphics engine, but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially outpaced its CPU counterpart.

In chapter 2, the fundamentals of MVS were described. It was presented the different characteristics of MVS algorithms. Also, the four MVS algorithms categories were examined by pointing out some representative cases. Then, the bases of windows matching were briefly explained and followed by a description of Goesele's algorithm.

In chapter 3, an overview of GPU evolution and platforms, CUDA and OpenCL, were presented. Then, the details of the algorithm implementation were described. In addition, besides the straightforward implementation of the Normalized Cross-Correlation (NCC) equation, we proposed two implementations where different key considerations about GPU cores memory are taken into account.

Futhermore chapter 4, the evaluation of the GPU implementations were presented: The algorithm was implemented on CUDA and OpenCL platforms that offer the advantage

of maturity and vendor free-royalty respectively. A CPU implementation was used as reference; also OpenMP is used to consider the multi-core capabilities of commodities CPUs. Then, the execution times were obtained using six reference images of a standard dataset. The resulting times of kernel execution for CPU, CUDA and OpenCL in each configuration were discussed. This experimental evaluation shows paramount speedups of 773 times faster for NVIDIA OpenCL and 515 times faster for CUDA using our proposed implementations. Previously, Goesele's algorithm could not be properly applied due to long computation times. Furthermore, this thesis proved that a GPU implementation is very efficient reducing the computation time to less than a second per image, as a result it makes the algorithm available for several applications.

The resulting depth map from the set of six images was presented for each implementation. They are consistent with the object shape and they did not present major difference among each implementation. However, two numerical effects related to the finite representation of floating values are found in the number of points in the depth maps. These effects are attributable to the different used forms of the NCC equations and to the difference between the default configurations for floating point operations in CUDA and OpenCL.

Future work includes the extension of the algorithm using Phase Only Correlation (POC) instead of NCC. POC has performed very well in binocular stereo accomplishing an accuracy of 1/100 pixel disparity. In the POC matching procedure an average of POC functions is attained around the intended pixel. Therefore, it is propose to use an NCC like approach to assembly this average in order to take advantage of the nonlinearity properties of NCC matching and the POC accuracy.

References

[1] P. Song, X. Wu, and M.Y. Wang, "Volumetric stereo and silhouette fusion for image-based modeling," Vis. Comput., vol.26, pp.1435–1450, December 2010.

[2] P. Cignoni, M. Corsini, and G. Ranzuglia, "Meshlab: an open-source 3d mesh processing system," ERCIM News, no.73, pp.45–46, April 2008.

[3] R. Szeliski, Computer Vision : Algorithms and Applications, 1 ed., Springer, New York, NY, USA, 2011.

[4] R. Hartley and A. Zisserman, Multiple View Geometry in Computer Vision, 2 ed., Cambridge University Press, New York, NY, USA, 2003.

[5] N. Snavely, S.M. Seitz, and R. Szeliski, "Photo tourism: Exploring photo collections in 3d," ACM TRANSACTIONS ON GRAPHICS, pp.835–846, Press, 2006.

[6] N. Snavely, S. Seitz, and R. Szeliski, "Modeling the World from Internet Photo Collections," International Journal of Computer Vision, vol.80, no.2, pp.189–210, Nov. 2008.

[7] S.M. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski, "A comparison and evaluation of multi-view stereo reconstruction algorithms," 2006.

[8] M. Goesele, B. Curless, and S.M. Seitz, "Multi-view stereo revisited," CVPR, 2006.

[9] Y. Furukawa and J. Ponce, "Carved visual hulls for image-based modeling," Int. J. Comput. Vision, vol.81, pp.53–67, January 2009.

[10] H. Vu, R. Keriven, P. Labatut, and J.P. Pons, "Towards high-resolution large-scale multi-view stereo," Tech. Rep. 08-41, Nov 2008.

[11] C.H. Andez, C. Hern, E. Esteban, and F. Schmitt, "Silhouette and stereo fusion for 3d object modeling," Computer Vision and Image Understanding, vol.96, pp.367–392, 2004.

[12] Steve Seitz, Brian Curless, James Diebel, Daniel Scharstein, Richard Szeliski , "Middlebury multi-view stereo evaluation," 01 2012.

[13] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," SIGGRAPH, pp.303–312, 1996.

[14] D. Bradley, T. Boubekeur, and W. Heidrich, "Accurate multi-view reconstruction using robust binocular stereo and surface meshing," CVPR, 2008.

[15] J. Li, E. Li, Y. Chen, L. Xu, and Y. Zhang, "Bundled depth-map merging for multi-view stereo," Computer Vision and Pattern Recognition, IEEE Computer Society Conference on, vol.0, pp.2769–2776, 2010.

[16] Y. Furukawa and J. Ponce, "Accurate, dense, and robust Multi-View stereopsis," Computer Vision and Pattern Recognition, 2007. CVPR '07., pp.1–8, June 2007.

[17] P.F. Engin Tola, Christoph Strecha, "Machine vision and applications," 2011.

[18] P. Mücke, R. Klowsky, and M. Goesele, "Surface reconstruction from multi-resolution sample points," VMV, pp.105–112, 2011.

[19] S. Fuhrmann and M. Goesele, "Fusion of depth maps with multiple scales," ACM Trans. Graph., vol.30, pp.148:1–148:8, Dec. 2011.

[20] N. Cornelis and L.V. Gool, "Real-time connectivity constrained depth map computation using programmable graphics hardware," Computer Vision and Pattern Recognition, IEEE Computer Society Conference on, vol.1, pp.1099–1104, 2005.

[21] C. Zach, "Fast and high quality fusion of depth maps," 2008.

[22] V. Vineet and P.J. Narayanan, "CUDA cuts: Fast graph cuts on the GPU," vol.0, pp.1–8, June 2008.

[23] H. Mostofi, J. Schnabel, and V. Grau, "Fast level set segmentation of biomedical images using graphics processing units," 2009.

[24] N. Cornelis and L. Van Gool, "Fast scale invariant feature detection and matching on programmable graphics hardware," Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on, pp.1 –8, june 2008.

[25] M. Goesele, "Multi-view stereo for community photo collections," 2007.

[26] Nvidia Corporation, "The Cg Tutorial," 2003.

[27] P. Labatut, R. Keriven, E. Nationale, and P. Chaussees, "A gpu implementation of level set multi-view stereo," in 'International Conference on Computational Science - Workshop General Purpose Computation on Graphics Hardware GPGPU: Methods, Algorithms and Applications', LNCS, pp.212–219, 2005.

[28] Nvidia Corporation, "CUDA programming guide," 2010.

[29] C.M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi gf100 gpu architecture," IEEE Micro, vol.31, pp.50–59, March 2011.

[30] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," IEEE Micro, vol.28, pp.39–55, March 2008.

[31] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," Queue, vol.6, pp.40–53, March 2008.

Acknowledgment

This study was carried out in the Computer Structures Laboratory, Department of Computer and Mathematical Sciences, Graduate School of Information Sciences, Tohoku University.

First, I am extremely grateful to my advisor Prof. Takafumi Aoki for his unfailing enthusiasm, guidance and crucial support throughout my studies in Japan and for his unconditional assistance during my recovery stage.

I am glad to acknowledge Prof. Michitaka Kameyama and Prof. Ayumi Shinohara, members of the thesis examination committee, for their valuable comments and assertive suggestions during the examination of this thesis.

Similarly, I would like to express my heartfelt gratitude to Assoc. Prof. Naofumi Homma for his support and contributions throughout my learning carrier in Japan. As well, I appreciate Dr. Koichi Ito for his thoroughly guidance and meticulous suggestions during my research.

I also acknowledge Toru Takahashi, my senior in the laboratory, for his directions at the beginning of this research. I would like to thank my friendly colleagues in the Aoki Laboratory who have made my daily life more interesting and comfortable. Specially, I thanks to Mamoru Miura and Shuji Sakai for our daily fruitful discussions.

Last but not least, I want to say thank you to my family, my parents and my sisters, for their unconditional love, support, encouragement and advice during my study in Japan.

I am very thankful to the Japanese Minister of Education, Culture, Sports, Science and Technology for the scholarship granted. Without this opportunity given, I would not be able to achieve this goal.

February 10, 2012