Journal of Real-Time Image Processing (2021) 18:445–461
https://doi.org/10.1007/s11554-020-00974-z

ORIGINAL RESEARCH PAPER

GPU-accelerated uncapacitated facility location and semi-dense SymStereo pipelines for piecewise-planar-based 3D reconstruction

Carlos Graca[1] · Carolina Raposo[2] · Joao P. Barreto[2] · Urbano Nunes[2] · Gabriel Falcao[1]

Received: 22 March 2018 / Accepted: 6 April 2020 / Published online: 1 May 2020

Abstract
Planar 3D reconstruction presents advantages over point cloud representations. This work focuses on the acceleration of piecewise-planar-based 3D reconstruction, a StereoScan method. We identify the SymStereo (log$N$) and uncapacitated facility location (UFL) algorithms as the most computationally expensive tasks, consuming nearly 80 $\times$ of total runtime, when detecting planes in a single stereo pair on a sequential CPU pipeline. Consequently, these algorithms have been parallelized using single- and multi-GPU architectures to perform significantly faster than previous sequential approaches. Experimental results show that accelerated parallel implementations of SymStereo (log$N$) can process up to 56 frames per second, achieving a speedup of 38 $\times$ against the sequential C implementation (Intel Core i7-4790k). The parallel version of the message-passing algorithm (max-sum) for the UFL problem processes up to five matrices per second and outperforms the sequential C baseline for computing UFL by 38 $\times$.

Keywords Piecewise-planar reconstruction · SymStereo · Uncapacitated facility location · Parallel image processing · 3D reconstruction · High-resolution images · Multi-GPU systems

1 Introduction

3D reconstruction became an important topic due to applications such as Google Street View [10, 28, 40, 43, 50] or medical imaging [12, 16, 24, 46, 49], to name only two popular cases. Higher-resolution images require increased computational performance. The amount of data captured and the resolution of images acquired by modern cameras and devices are expected to continue to increase over the coming years. This increase in complexity affects the performance of 3D reconstruction algorithms, most of which compute disparity maps using photo-similarity between pairs of images [18, 41, 45, 47]. More recent work from Antunes et. al [4, 5, 36] uses dense photo-symmetry instead to extracting disparities.

Despite being dominated by plane surfaces in buildings, facades and streets, urban scenarios are still typically represented as clouds of points, which pose additional challenges in terms of required storage capacity, bandwidth and processing power.

Recently, piecewise-planar 3D reconstruction was introduced by Antunes, Barreto and Raposo [5, 37, 38] as an alternative to clouds of points, with obvious advantages in storage, transmission speed and rendering simplification. The methods used to detect plane surfaces from stereo matching involve solving the uncapacitated facility location (UFL) problem to identify the best plane candidates. In our case, this represents more than 75% of the execution time of the plane detection pipeline (see Table 2). Also, the approach followed in this work is based on the analysis of the energy of symmetry between stereo pairs of images

✉ Gabriel Falcao
gff@co.it.pt

Carlos Graca
cgraca@co.it.pt

Carolina Raposo
carolinaraposo@isr.uc.pt

Joao P. Barreto
jpbar@isr.uc.pt

Urbano Nunes
urbano@isr.uc.pt

1 University of Coimbra, Instituto de Telecomunicações, Department of Electrical and Computer Engineering, 3030-290 Coimbra, Portugal

2 University of Coimbra, Institute of Systems and Robotics, Department of Electrical and Computer Engineering, 3030-290 Coimbra, Portugal

[4], which adds additional computational complexity to the 3D reconstruction pipeline. Therefore, the acceleration of essayed procedures in this work must cope with such a variety of functional challenges that should all be considered together with the most suitable and powerful parallel programming framework and architecture.

Modern desktop CPUs incorporate a few cores (typically 4) optimized for sequential execution, while graphics processing units (GPUs) provide parallel computer architectures including thousands of smaller processing units and very high memory bandwidths [34]. GPU architectures are becoming increasingly efficient for dealing with compute-intensive workloads, offering high speedups when compared to execution on conventional CPUs, even using multiple CPU threads. To illustrate this, the considered CPU (Intel i7-4790k) has four cores running at 4.0 GHz and 25.6 GB/s of maximum memory bandwidth, while the fastest GPU considered (GTX TITAN X) has 3072 cores running at 1.075 GHz and a maximum memory bandwidth of 336.5 GB/s. The GPU provides a higher number of cores and superior memory bandwidth. To overcome the main obstacles regarding the development of the parallel algorithms to run on GPUs, we use the Compute Unified Device Architecture (CUDA) framework [34].

The contributions of this paper are:

- *SymStereo (CUDA)* Development of C/CUDA parallel kernels of a semi-dense version of the SymStereo (*logN*) algorithm [4] compatible with single- and multi-GPU assemblies. By varying the number of virtual cut planes from 45 to 15, our multi-GPU configuration E (see Table 4) is able to process between 24 and 56 frames per second using 1024×768 pixel images, achieving a speedup of 40 to 43× over a sequential C program on a 4 GHz CPU (see Table 7);
- *Parallel UFL (CUDA)* Development of C/CUDA parallel kernels of the message-passing algorithm (max-sum) for solving the UFL problem executing under both single- and multi-GPU assemblies. By varying the size of the input matrix from 34560×553 to 11520×180, our multi-GPU configuration E (see Table 4) is able to process between 0.68 and 4.76 matrices per second, achieving a speedup of up to 38× over a sequential C program on a 4 GHz CPU (see Table 7);
- *CUDA parallel code publicly available* Our source code is available at http://montecristo.co.it.pt/PPR_Rec [15].

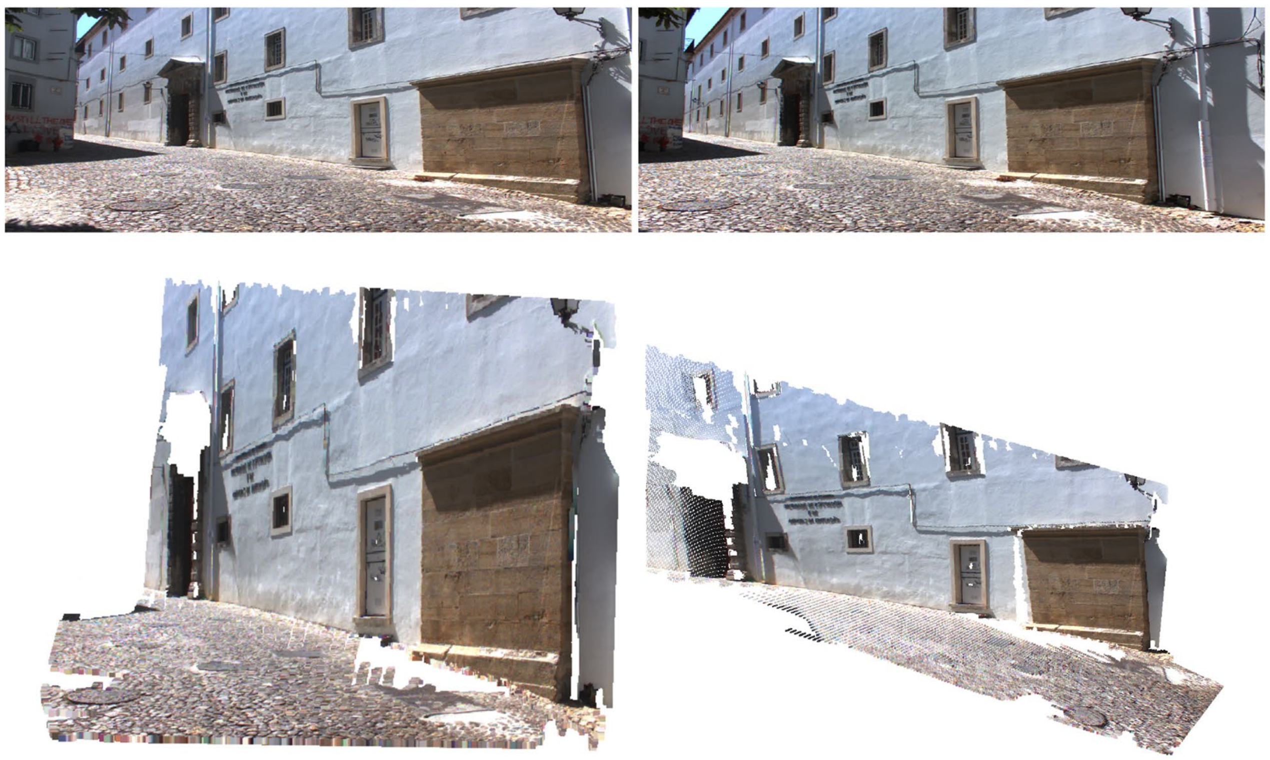
This paper is structured as follows. An overview of the UFL and SymStereo piecewise-planar reconstruction methods and the illustration of some early achievements are provided in Sect. 2. Related work is presented in Sect. 3. The GPU architecture, memory bandwidth optimization and parallel kernels configuration for the UFL and SymStereo algorithms are described in Sect. 4. In Sect. 5, we detail a multi-GPU balanced workload approach. Performance benchmarking with detailed time analyses for single- and multi-GPU configurations is presented in Sect. 6. Finally, we close the paper in Sect. 7.

2 Two-view piecewise-planar reconstruction

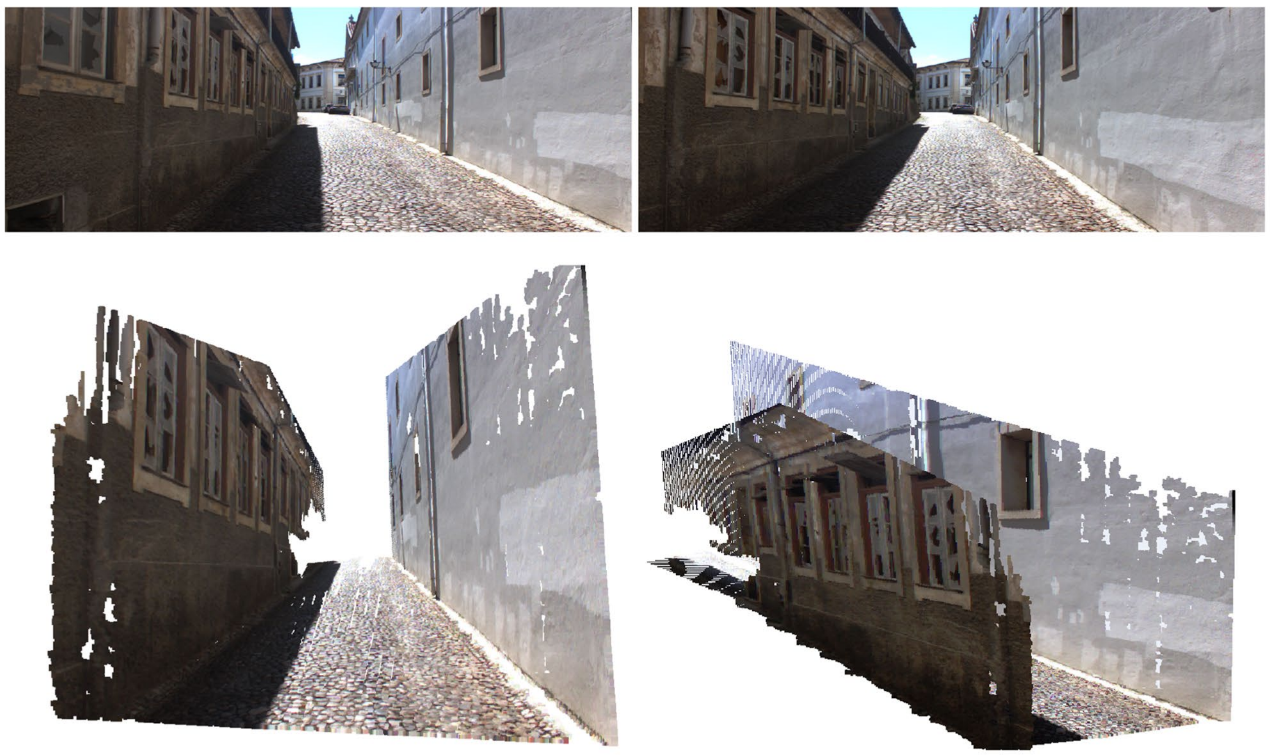
The strong planarity assumption has recently made piecewise-planar models popular for the reconstruction of man-made environments. In stereo reconstruction, it is useful for overcoming problems of poor texture and non-Lambertian reflections. Moreover, the obtained dense 3D models are perceptually pleasing and geometrically simple, making their rendering, storage and transmission computationally more efficient. Raposo et al. [37, 38] recently used a simplified version of the two-view semi-dense piecewise-planar reconstruction (PPR) method proposed by Antunes et. al [3, 5]. The method starts by using the SymStereo framework [4] to compute the energies associated with a sparse set of virtual cut planes that intersect the baseline of the stereo camera in its midpoint. The energies are used as input to a Hough transform to extract a set of line segments, which are the intersections of the virtual planes with the scene planes. The extracted hypotheses are used in a MRF formulation to improve the estimation of the line segments [6, 11, 42], which is solved by Delong et. al [8] using graph cut optimization (GCO). A plane hypothesis is then computed from a set of two lines. As a final step, the algorithm solves a UFL problem to select a subset of planes from the large set of input plane hypotheses that is likely to describe the observed scene. In Figs. 1 and 2, we can see two examples of 3D reconstruction results for this PPR method which receive stereo pairs as input.

We coded the semi-dense PPR algorithm in sequential C. The execution time of the complete algorithm is 1.92× faster than the one obtained using MATLAB (see Table 1).

Table 2 highlights the most computationally expensive step in the semi-dense PPR algorithm—the UFL algorithm—which is the best candidate for parallelization. The SymStereo framework is a recent and promising matching function for stereo pairs; thus, we believe that its acceleration towards real-time execution can represent an important evolution and a relevant contribution. Therefore, we focus on the parallelization of the SymStereo and UFL algorithms presented in the semi-dense PPR for single stereo pairs, which consume 80% of total runtime. In Sects. 2.1 and 2.2, the reader can find more details regarding the SymStereo and UFL algorithms.

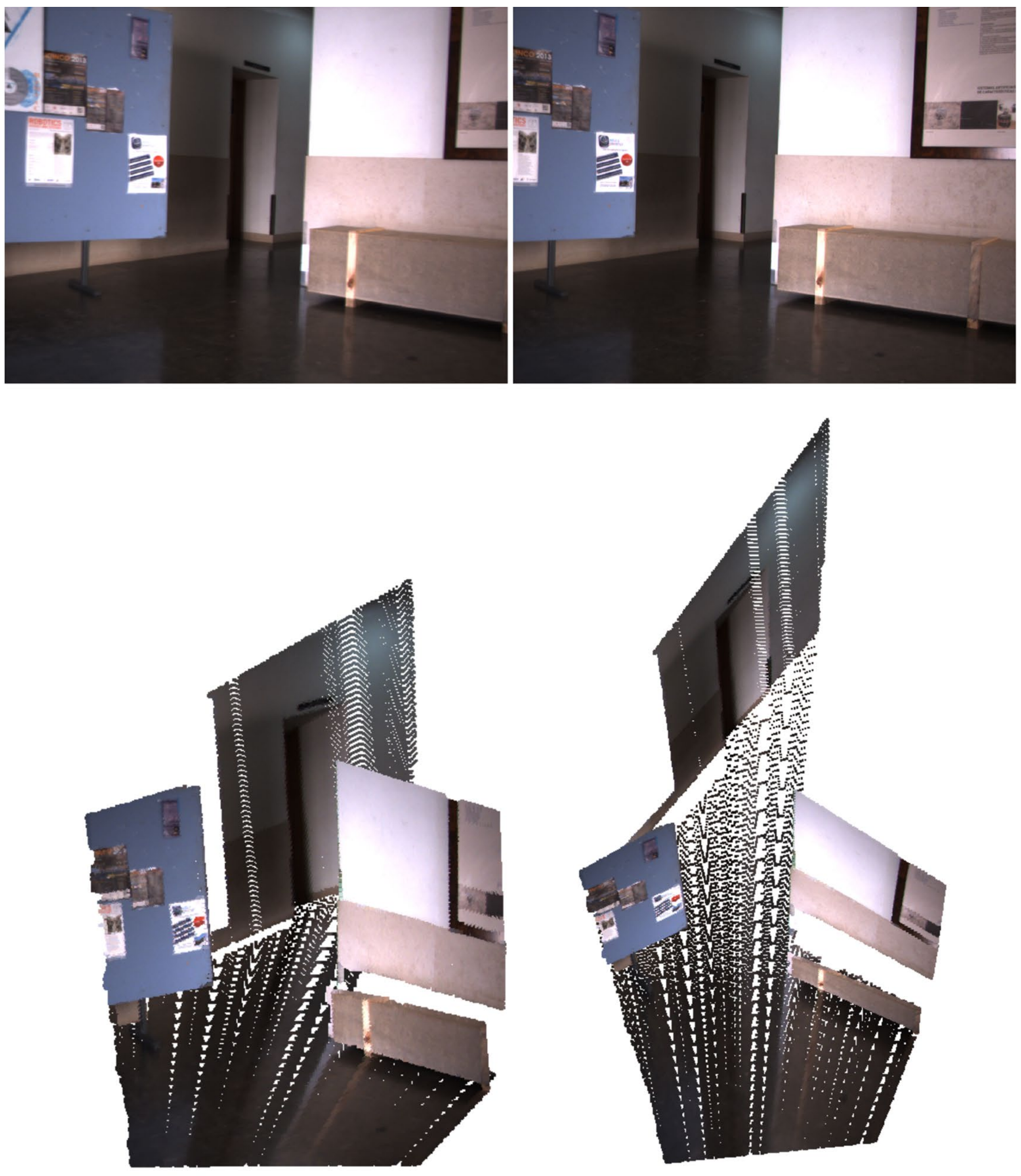


(a) 3D vision of reconstructed example 1.
On top, the stereo pair used as input for 3D reconstruction; below, two different perspectives of the 3D reconstruction results.



(b) 3D vision of reconstructed example 2.
On top, the stereo pair used as input for 3D reconstruction; below, two different perspectives of the 3D reconstruction results.

Fig. 1 3D reconstruction results computed by GPUs for a two-view semi-dense piecewise-planar reconstruction (PPR) method using a stereo pair as input. The original size of images is 1280×960 pixels, and SymStereo uses 25 virtual cut planes and ten LogGabor filtering scales



3D vision of reconstructed example 3.
On top, the stereo pair used as input for 3D reconstruction; below, two different perspectives of the 3D reconstruction results.

Fig. 2 Further 3D reconstruction results computed by GPUs for a two-view semi-dense piecewise-planar reconstruction (PPR) method using a 1024×768 pixel stereo pair as input. SymStereo used 25 virtual cut planes and ten LogGabor filtering scales. The dark areas represent the floor and shadows, while the white patches show the imperfections of the algorithm, mostly due to variations in luminosity and brightness, shadows and occlusions

2.1 SymStereo theory

Most stereo methods use matching costs that measure intensity differences between image regions centered on possible matches to determine whether two pixels match. Recently, Antunes and Barreto [4] measured symmetry instead of photo-similarity to associate pixels across views. They show that a virtual plane that intersects the baseline of the stereo camera allows the rendering of image signals that are symmetric or antisymmetric about the contour where the plane meets the scene. These symmetric and antisymmetric signals are obtained, respectively, by adding and subtracting the left

Table 1 Execution time in seconds (s) for the semi-dense PPR procedure. The tests were performed on urban images with 1024 × 768 pixels using 35 virtual cut planes and 15 LogGabor filtering scales

CPU Intel core i7-4790K	
Processing platform	Execution time (s)
MATLAB	82
C Sequential	42.8

Table 2 Execution time in seconds (s) for the semi-dense PPR procedure. The tests were performed running a sequential C program on a 4 GHz CPU using urban images with 1024 × 768 pixels, 35 virtual cut planes and 15 LogGabor filtering scales

CPU Intel core i7-4790K	
Function	Execution time (s)
UFL	31.9
Hough transform	5.6
Graph cut optimization	2.6
SymStereo	1.4
Simple arithmetic	1.3
Time SUM	42.8

image of the stereo pair $\mathtt{I}$ with $\hat{\mathtt{I}}$, which is the image that is obtained by warping the right image $\mathtt{I}'$ by the plane homography of the virtual cut plane. When the matching function resulting from these image signals is evaluated across all possible disparities and pixel locations, the so-called disparity space image (DSI) is obtained. In other words, the entire DSI domain can be fully covered by carefully choosing the set of virtual cut planes. The more virtual cut planes we use, the better the final 3D model accuracy.

We now present a formal interpretation of the relations between virtual cut planes and the DSI for rectified stereo images, i.e., images acquired by cameras related by a rigid transformation with rotation component $\mathsf{R} = \mathtt{I}_{3\times3}$, where $\mathtt{I}_{3\times3}$ is the three-by-three identity matrix, and translation component $\mathbf{t} = (b \quad 0 \quad 0)^\mathsf{T}$, where b is the baseline length.

Let $\boldsymbol{\Pi} = (\mathbf{n} \quad -h)^\mathsf{T}$, where $\mathbf{n} = (n_1 \quad n_2 \quad n_3)$ and h is the distance from the plane to the origin, be a particular virtual cut plane that induces a homography H that maps points $\mathbf{q}' \sim (q_1' \quad q_2' \quad 1)^\mathsf{T}$ on the right image to points $\mathbf{q} \sim (q_1 \quad q_2 \quad 1)^\mathsf{T}$ on the left image and is defined by

$$\mathsf{H} \sim \left(\mathtt{I}_{3\times3} + \frac{\mathbf{t}\mathbf{n}^\mathsf{T}}{h}\right)^{-1} \sim \begin{pmatrix} 1 + \frac{bn_1}{h-bn_1} & \frac{bn_2}{h-bn_1} & \frac{bn_3}{h-bn_1} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{1}$$

Using this homography, the relation between points on the left and right images becomes

$$q_1' = \left(1 - \frac{bn_1}{h}\right)q_1 - \frac{bn_2}{h}q_2 - \frac{bn_3}{h}, \tag{2}$$

because it is known that $q_2' = q_2$ for rectified images. It follows that the stereo disparity $d = q_1 - q_1'$ is

$$d = \frac{bn_1}{h}q_1 + \frac{bn_2}{h}q_2 + \frac{bn_3}{h}, \tag{3}$$

which specifies a 3D plane parametrized by (q_1, q_2, d). This means that the matching hypothesis implicitly defined by a virtual cut plane $\boldsymbol{\Pi}$ corresponds to a plane

$$\boldsymbol{\Phi} \sim \begin{pmatrix} \frac{bn_1}{h} & \frac{bn_2}{h} & -1 & \frac{bn_3}{h} \end{pmatrix}^\mathsf{T} \tag{4}$$

in the DSI domain. To obtain dense stereo matching, it is necessary to choose a set of virtual cut planes $\boldsymbol{\Pi}_i$ such that the corresponding planes $\boldsymbol{\Phi}_i$ fully cover the DSI space. Suppose each $\boldsymbol{\Pi}_i$ is a vertical plane that intersects the baseline in its midpoint, such that it induces a homography defined by

$$\mathsf{H}_i \sim \begin{pmatrix} -1 & 0 & 2\tan(\theta_i) \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{5}$$

where θ_i is the rotation angle around the vertical axis. If $\mathbf{q}$ and $\mathbf{q}'$ are in pixel coordinates, the homography mapping is $\mathbf{q} \sim \mathsf{K}\mathsf{H}_i\mathsf{K}^{-1}\mathbf{q}'$, where K is the matrix of intrinsic parameters consisting of a focal length f and a principal point (c_1, c_2). This homography mapping can be rewritten as

$$q_1 = \underbrace{2c_1 - q_1'}_{\text{flip}} + \underbrace{2f\tan(\theta_i)}_{\lambda_i}, \tag{6}$$

yielding a disparity $d = 2q_1 - 2c_1 - \lambda_i$. In this case, the DSI plane $\boldsymbol{\Phi}_i$ of Equation 4 takes the simplified form

$$\boldsymbol{\Phi}_i \sim \begin{pmatrix} 2 & 0 & -1 & -2c_1 - \lambda_i \end{pmatrix}^\mathsf{T}. \tag{7}$$

These results lead to two important observations: i) if λ_i takes consecutive integer values, the full range of disparities in the DSI domain can be covered and ii) the homography mapping in (6) allows the warping of images, which is required to compute the symmetric and antisymmetric signals, to be accomplished by simply flipping the original right image $\mathtt{I}'$ around the vertical axis passing through the principal point and then shifting the resulting image by λ_i pixels along the horizontal direction. Since this configuration of virtual cut planes yields such simplified computations, it is employed throughout the remainder of this paper.

Let $\mathtt{I}_s$ and $\mathtt{I}_a$ be the symmetric and antisymmetric image signals obtained from $\mathtt{I}_s = \mathtt{I} + \hat{\mathtt{I}}$ and $\mathtt{I}_a = \mathtt{I} - \hat{\mathtt{I}}$ for a particular virtual cut plane $\boldsymbol{\Pi}$. In order to locate the contour where $\boldsymbol{\Pi}$ meets the scene, it is necessary to quantify the symmetry and antisymmetry of $\mathtt{I}_s$ and $\mathtt{I}_a$ along the epipolar lines. This is done by applying a set of N log-Gabor wavelets with predefined scales for measuring the image signal symmetry and antisymmetry at every pixel location.

As explained in [4], the convolution of images $\mathtt{I}_s$ and $\mathtt{I}_a$ with the log-Gabor wavelets generates the symmetric and antisymmetric energies $\mathtt{E}_s$ and $\mathtt{E}_a$, respectively. A joint energy $\mathtt{E}$, which is the pixel-wise multiplication of $\mathtt{E}_s$ and $\mathtt{E}_a$, is defined to account for the fact that the individual energies contain several local maxima, precluding a correct detection of the relevant contour. This energy $\mathtt{E}$ is the output of the SymStereo pipeline.

Besides efficiently computing the warped image, $\hat{\mathtt{I}}$, which is a simple reflection and shift of $\mathtt{I}'$, another important advantage of using a vertical pencil of cut planes whose axis intersects the baseline in its midpoint is that it efficiently computes the convolution. Since it is a linear operator, the convolution of $\mathtt{I}_s = \mathtt{I} + \hat{\mathtt{I}}$ can be performed by first convolving $\mathtt{I}$ and $\hat{\mathtt{I}}$ and then adding the result. Also, since $\hat{\mathtt{I}}$ is a reflected and shifted version of $\mathtt{I}'$, it is only necessary to perform one convolution for the K virtual cut planes.

In the implementation of SymStereo, note that the log-Gabor filters are analog signals, so here we filter by taking a DFT of the rows of Is and Ia, multiplying by the log-Gabor kernels and then taking the IDFT.

Raposo et al used SymStereo [37, 38] to obtain a semi-dense stereo reconstruction by estimating depth along user-defined virtual cut planes. The intersection of two planes results in a line; thus, the line segments extracted from this energy correspond to the intersection of the virtual plane with a scene plane, allowing these scene planes to be reconstructed.

2.2 UFL theory

In generic terms, the UFL problem can be formulated as follows. Suppose a set of facilities $\boldsymbol{\pi}_j^0$ has to be opened to serve N customers $\mathbf{p}_i \in \mathscr{P}$ whose locations are known. Given a set $\mathscr{V}_0$ with M possible facility locations, the cost c_{ij}^0 of assigning facility $\boldsymbol{\pi}_j^0$ to customer $\mathbf{p}_i$ and the cost v_j^0 of opening the particular facility $\boldsymbol{\pi}_j^0$, the goal is to select a subset of $\mathscr{V}_0$ such that each customer is served by one facility, and the summed customer-facility and facility opening costs are minimized. This leads to an integer programming problem that can be formulated using unary indicator variables y_j^0 and binary indicator variables y_{ij}^0, with the objective of finding vector $\mathbf{x}^0 = \{x_{11}^0 \dots x_{ij}^0 \dots x_{NM}^0\}$ such that

$$\min \sum_{i=1}^{N}\sum_{j=1}^{M} c_{ij}^0 x_{ij}^0 + \sum_{j=1}^{M} v_j^0 y_j^0$$
$$\text{subject to} \begin{cases} x_{ij}^0, y_j^0 \in \{0,1\}, \forall_{i,j} \\ \sum_{j=1}^{M} x_{ij}^0 = 1, \forall_i \\ y_j^0 \geq x_{ij}^0, \forall_{i,j} \end{cases} \cdot \tag{8}$$

The second constraint ensures that each customer is assigned to exactly one facility, while the last constraint guarantees that each customer is only served by open facilities. UFL problems can be efficiently solved using a message-passing inference algorithm as proposed by Lazic et al. [23].

As described at the beginning of the present section, an initial set of plane hypotheses $\{\pi\}$ is generated using the energy that is output by the SymStereo framework. Afterward, and in order to obtain a semi-dense PPR and a set of planar surfaces that properly describes the scene, a discrete optimization problem is formulated as follows [5].

Consider that there is a virtual camera, called the cyclopean eye, whose center of projection is the midpoint of the baseline and whose height is the number of epipolar planes (equal to the number of rows in the stereo images). Each pixel $\mathbf{p}_{ir}$ of the cyclopean eye is associated with the back-projection ray that is the intersection of the epipolar plane r with the virtual cut plane Π_i. Thus, the width of the cyclopean eye is N, the number of cut planes. The objective of the discrete optimization is to assign to each pixel $\mathbf{p}_{ir}$ a plane hypothesis $\boldsymbol{\pi}_j$ from the initial set of hypotheses $\{\pi\}$, or the discard label $l_\emptyset$. This is a multi-model fitting problem that can be cast as an uncapacitated facility location (UFL) instance since the energy to be minimized only contains label and data terms. The data term D_{ir} for pixel $\mathbf{p}_{ir}$, corresponding to cost c_{ij} in Equation 8, is defined as

$$D_{ir}(l) = \begin{cases} \min(1 - \mathtt{E}_i(r, x_l), \tau) & \text{if } l \in \{\pi\} \\ \tau & \text{if } l = l_\emptyset \end{cases}, \tag{9}$$

where $\mathtt{E}_i$ is the joint energy associated with Π_i, τ is a constant and the coordinate x_l is the column defined by hypothesis l, corresponding to the intersection of the back-projection ray of pixel $\mathbf{p}_{ir}$ and the scene plane indexed by l. The role of the label term is to select as few plane hypotheses as possible to obtain a proper description of the 3D scene.

The output of this UFL stage is a semi-dense PPR of the scene and a small set of planar surfaces that can be used as input to a dense labeling scheme to generate dense PPRs as the ones shown in Figs. 1 and 2.

3 Related work

A common strategy to obtain dense PPR models is to start by reconstructing a sparse point cloud of the scene to generate plane hypotheses, which are then used to segment the input images into piecewise-planar regions. Pollefeys et al. [35] used the plane normals, obtained from sparse 3D point features, to guide the plane sweep stereo. Sinha et al. [42] presented a probabilistic framework for assigning plane hypotheses to pixels, within the constraints of planar structures provided by point cloud reconstruction, matching line segments and estimation of vanishing points. Gallup et al. [11] used a robust scheme to fit plane hypotheses to dense depth maps, which are then used to label the input images into planar regions.

Other works that produce dense PPRs and, unlike previous algorithms, work with monocular cameras have recently been proposed. In [39], Raposo et al. employed a new error metric that enables the efficient segmentation of affine correspondences into planes, providing plane hypotheses. The dense labeling of the input images is obtained using a standard Markov random field (MRF) approach. Kou et al. [21] and Bódis-Szomorú et al. [7] used superpixels to segment the input images, and while [7] used planarity assumptions to overcome the problems of SfM sparsity and textureless regions, [21] made use of optical flows stacked over multiple images and a factorization approach to directly obtain a dense piecewise-planar reconstruction of the scene.

In the last few years, GPU parallel processing techniques have been applied to diverse areas of image processing and computer vision. In 2008, a fast graph cuts implementation using CUDA was introduced by Vineet et al. [44]. A parallel implementation of *Canny* edge detection developed by Luo et al. and Ogawa et al. can be found in [25, 32]. Addressing a correlated topic regarding image distortion caused by the use of small lenses, an efficient solution for camera calibration and real-time image distortion correction has been proposed by Melo et al. for medical endoscopy [27]. But many other contributions using accelerators have been proposed in the field of computer vision [33, 48]. Among those, we find new accurate stereo matching systems using GPU architectures such as the ones proposed by Mei et al. and Zhang et al. [26, 51] and GPU-based 3D reconstruction. In 2008, an efficient 3D reconstruction method from video was developed by Pollefeys et al. to run under the CUDA framework, achieving real-time processing capabilities [35]. Real-time 3D reconstruction using visual hull computation running on GPUs was also proposed by Ladikos et al. in 2008 [22]. A symmetric dynamic programming stereo matching algorithm running under GPU was presented by Kalarot and Morris in 2010 [20]. In 2011, KinectFusion was presented by Isadi et al. for real-time 3D reconstruction using the Kinect RGB-D system and executed on GPU [19]. A full-body volumetric reconstruction of a person in a scene using a sensor network and the CUDA framework was developed in 2011 by Aliakbarpour et al. [2]. Alexiadis et al. developed a new parallel approach based on the generation of separate textured meshes from multiple RGB-D cameras to recover a full 3D model of moving humans in real time [1]. In 2015, a new magnetic resonance imaging (MRI) reconstruction algorithm via three-dimensional dual-dictionary learning using CUDA was reported by Li et al. [24]. A GPU optimization and refinement of slanted 3D reconstructions using dense stereo induced from symmetry were presented in 2016 by Ralha et al. [36]. Our dense stereo 3D reconstruction method uses all valid information in the stereo pair, but since urban scenarios are dominated by planes, we realized there is a high potential gain in representing such structures using planes instead of millions of pixels, thus minimizing pixel information and storing much less information in data centers, while reducing the network bandwidth necessary to visualize the 3D model. With this in mind, our current work is based on the detection of planes using a symmetry-based algorithm [4] to calculate the energy of the intersection of a subset of virtual cut planes (we used 15 to 45 cut planes) with the volume background. By extracting the cut planes energy, we find 3D line segments that better represent the scene and apply refinement techniques in order to define a 3D world plan to use in our final 3D model.

We designed and tested a multi-GPU system with multi-threading software that makes efficient use of all the available CUDA devices in the workstation and significantly reduces idle time resulting from task management and communication overheads. In previous parallel computing work, Graca et al. [13, Sect. 8.2] developed workload distribution based on single-GPU execution times, to determine how many images can be processed on each GPU at the same time, drastically reducing the GPU idle time. However, Graca et al. [13, Fig. 10] indicated that considerable idle time is spent between data transfers and instruction execution. Also, when the GPU completes the execution of a kernel, it always needs to wait before processing new data.

4 Parallelization

4.1 Computational complexity

The number of operations can be modeled as described next. The UFL receives as input one matrix with dimensions $M \times N$ and another with dimensions $M \times 1$. M is the product of the number of image lines and the number of cut planes

(*CuPla*), and N is the number of plane hypotheses + 1. At the output, it produces a matrix of dimensions $M \times N$. Assuming that the number of iterations processed is represented by N_{iters}, then the global number of arithmetic operations calculated can be described by:

Subtractions: $N_{\text{iters}} \times 8 \times M \times N$
Additions: $N_{\text{iters}} \times N \times (3 \times M + 1)$
Multiplications: $N_{\text{iters}} \times 2 \times M \times N$
Conditional operations: $N_{\text{iters}} \times 10 \times M \times N$

The performance of the algorithm depends on the number of image lines, cut planes, plane hypotheses (that directly relate to the number of iterations). Since the last two are much lower compared to image size, the algorithm presents computational complexity $O(n^3)$. On the limit, if we assume super-resolution images, the problem tends to complexity $O(n^2)$. Also, one matrix of dimension $M \times 1$ and four matrices of dimension $M \times N$ are allocated and initialized, but the time necessary to do so is negligible compared to the main loop.

A model to estimate the execution time T_{procTime} on the GPU is shown from (10) to (12). Here, Th represents the total number of threads running on the GPU, MP the number of multiprocessors of the GPU, SP the number of streaming cores per MP, OPs/iteration the number of cycles executed per iteration and f_{op} the GPU operating frequency. Each thread accesses global memory with latency L and performs a total of M_{access} memory accesses.

$$T_{\text{Arch}} = N_{\text{iters}} \frac{\frac{\text{Th}}{\text{MP}} \times (\frac{\text{OPs/iteration}}{\text{SP}}) + Th \times M_{\text{accesses}} \times L}{f_{\text{op}}} \tag{10}$$

and

$$T_{\text{Alg}} \sim \psi(\psi_1(M,N), \psi_2(CuPla), \psi_3(N_{\text{iters}})), \tag{11}$$

identify, respectively, the architectural and algorithmic parameters that influence performance, with ψ_n representing the cost function that minimizes execution time. Thus, global processing time can be modeled as

$$T_{\text{procTime}} \sim \Phi(T_{\text{Arch}}, T_{\text{Alg}}). \tag{12}$$

The throughput (in matrices per second) is thus obtained by

$$T = \frac{1}{T_{\text{host}\to\text{device}} + T_{\text{procTime}} + T_{\text{device}\to\text{host}}} \quad [MPS]. \tag{13}$$

$T_{\text{host}\to\text{device}}$ and $T_{\text{device}\to\text{host}}$ represent data transfer times between the host CPU and the GPU.

4.2 Apparatus

The pipeline of the methods described in Sects. 2.1 and 2.2 was first coded for reference in MATLAB and sequential C, and profiled on a system with an Intel Core i7-4790K CPU @ 4.0GHz, 32GB of RAM, running CentOS 7 with GNU / Linux kernel 2.6.32−573.7.1.el6. × 86_64. The C code was compiled using GCC-4.7.2.

In order to process more frames per second and improve the quality of the generated 3D volume, the SymStereo and UFL algorithms were parallelized to execute in the different single-GPU configurations, compiled using CUDA 7.5 [34]. The parallelization of the SymStereo and UFL procedures exploiting the single-instruction, multiple-thread (SIMT) principles makes these algorithms compatible with GPU-based computing. In this section, we also describe the GPUs used in benchmarking and their underlying architectures (see Table 3).

4.3 GPU architecture

Typically, parallel platforms (composed of one or more GPU devices) need to be managed by a host CPU that controls the entire processing pipeline by sending data, launching parallel kernels and collecting the computed data from the device(s) before terminating execution. These functions are supported by the CUDA parallel programming model [34], which allows the programmer to write transparent and scalable parallel C code. This enables good utilization of thread and data parallelism on the GPU.

Table 3 In this table, we show to the reader a comparison of the relevant specifications of different GPUs used in the experiments

GPU	CUDA cores	Normal/boost clock (MHz)	Memory size (MB)	Memory bandwidth (GB/sec)	Constant/shared memory per block (bytes)	Max [ower (Watts)
NVidia GTX 680	1536	1006 / 1058	2048	192.2	65536 / 49152	195
NVidia GTX TITAN	2688	837 / 876	6114	288.4	65536 / 49152	250
NVidia Tesla K40c	2880	745 / 875	12000	288	65536 / 49152	235
NVidia GTX TITAN X	3072	1000 / 1075	12000	336.5	65536 / 49152	250

As we can see, we run our tests using GPUs with significant performance differences. Such diversity in terms of architectures represents a non-ideal scenario and thus increases the validation of our new multi-GPU approach

Figure 6 shows a simplified overview of the GPU architecture. As shown, several multiprocessors contain a large number of stream processors (the number and management of stream processors and multiprocessors vary with the GPU model and architecture). In this case, the GTX TITAN X has 24 multiprocessors, each of which contains 128 stream processors, totaling 3072 CUDA cores running at 1.0/1.075GHz.

Another important consideration when building an GPU has several memory types, which have different impacts on the final throughput. Two of them, registers and shared memory, share the same die as the processor itself. Constant, texture and global memories are placed outside the GPU chip, but depending on device capabilities, they may be cached (see Chapter 9, Section 9.2 in [31] for detailed information). When a kernel runs, consecutive threads are grouped for execution in groups of 32 threads (a warp). When a branch (such as an 'if' or a 'case' statement) is present, the warp checks all possible paths of execution, resulting in additional clock cycles. If all threads follow a different path, execution is serialized. Thus, whenever possible, all branches should be eliminated. Another important consideration when building an efficient parallel software running on GPUs is the use of coalesced memory accesses when performing accesses to global memory. These memory accesses are extremely slow, and they can severely penalize the system's overall throughput. Thus, coalesced accesses should be employed whenever possible. They imply data accesses in global memory to be contiguously aligned so that all 32 threads in a warp can access the corresponding data element concurrently in the same clock cycle, with thread $T(x, y)$ accessing pixel $P(x, y)$, as depicted in Fig. 5.

4.4 Optimizing memory bandwidth

SymStereo relies intensively on filtering, which can be slow. In the CPU baseline, images are filtered faster using frequency-domain methods from the FFTW3.3.3 library [9]. We used methods from the cuFFT library [14, 30] for the LogGabor filters. Other processing-intensive tasks consist of calculating minima and maxima values from intermediate results and the computation of accumulated values for each column in a matrix. Functions from the cuBLAS library [17, 29] were used to compute the reductions.

The present work introduces efficient algorithms and optimizes the use of device memory, in the following order:

- *cuFFT* In Sect. 2.1, in order to perform the DTFT and IDTFT on the GPU, the optimized cuFFT library [30] is used. In this algorithm, shared memory blocks that provide higher memory bandwidth are used, and thus higher throughputs can be achieved;
- *cuBLAS* In Sect. 2.2, we compute an accumulation for each column in a matrix. Using shared memory, the optimized cuBLAS library from NVidia [29] can perform this computation with higher efficiency and throughput;
- *Find Max/Min values* In Sect. 2.2, we calculate maximum and minimum values. To compute these reductions, shared memory is used to make data accessible by all the threads in a block [17]. Thus, data transfers with global memory are minimized, enabling higher throughput;
- *Simple arithmetic operations* Many operations use global memory accesses instead of the faster shared memory, because in these cases there is no data reuse between GPU threads and it is more efficient to access global memory directly.
- *CUDA streams* To optimize memory data transfers and kernel executions, we use CUDA streams [34]. A stream is a sequence of commands that execute in order, and different streams can run concurrently. We created the following three streams for each device: (1) upload data; (2) execute kernel; and (3) download data.

4.5 SymStereo parallelization

To achieve superior performance in the SymStereo algorithm described in Sect. 2.1, some functions make use of shared memory blocks. However, as mentioned previously, other functions perform faster without using any shared memory. In the case of SymStereo, only cuFFT and the function that determines the maximum values use shared memory. All other functions perform slower if shared memory is used since the total number of transactions with global memory would be higher and slow in these cases. Also, no data are used repeatedly (see Simple Arithmetic Operations, Sect. 4.4). The results of the maximum are processed in two stages: the first uses GPU grids with a 256×256 block size; the second uses 1×256 grids. The remaining functions in the segmentation process only use global memory and GPU grids with a 256×4 block size as best-performing configurations. By observing Fig. 3, the reader should be able to understand the entire kernel execution pipeline for the SymStereo algorithm running on the GPU.

4.6 UFL parallelization

To accelerate the UFL algorithm described in Sect. 2.2, we use the shared memory blocks in the functions that compute the accumulator value for each column in a matrix (cuBLAS library [29]) and that determine the maximum values: find the maximum in all matrix rows and then find the global maximum from the resulting column. However, the other arithmetic functions use global memory instead. All other functions perform slower if shared memory is used since the total number of transactions with global memory would

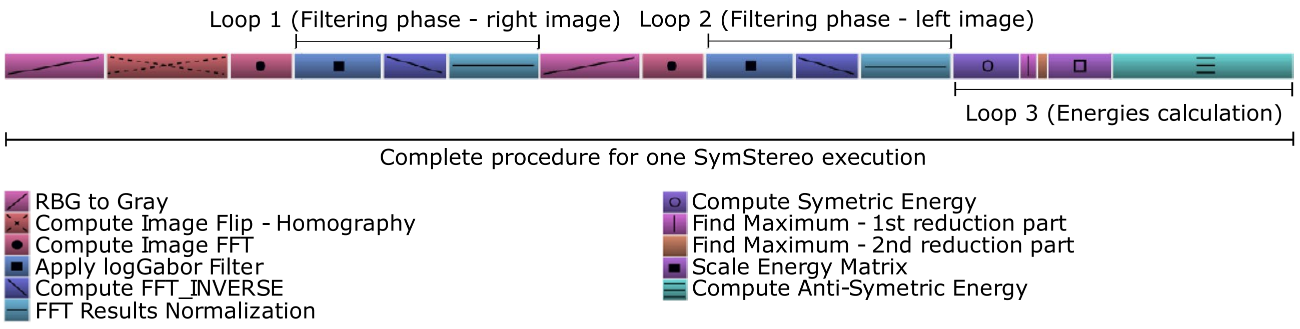


Fig. 3 Profiling the entire parallel kernel execution assigned for the SymStereo algorithm. This figure includes all functions that compose the SymStereo algorithm and a simple comparison of total execution times

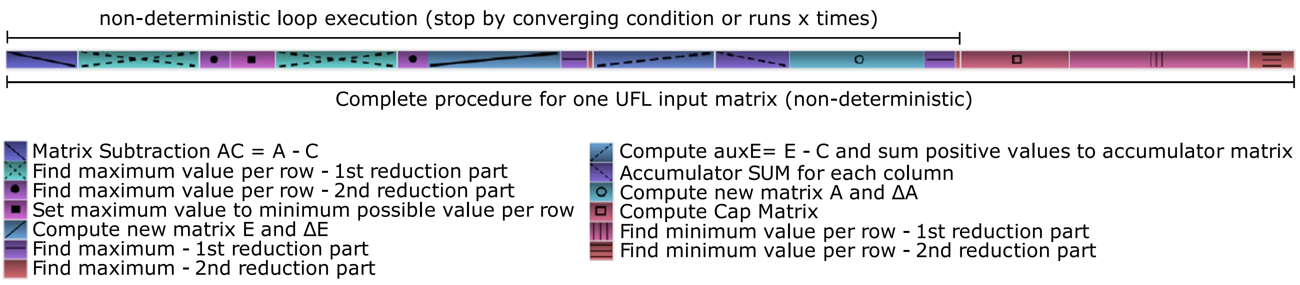


Fig. 4 Profiling the entire parallel kernel execution assigned for the UFL algorithm. This figure shows all functions that compose the UFL algorithm and a simple comparison of total execution times

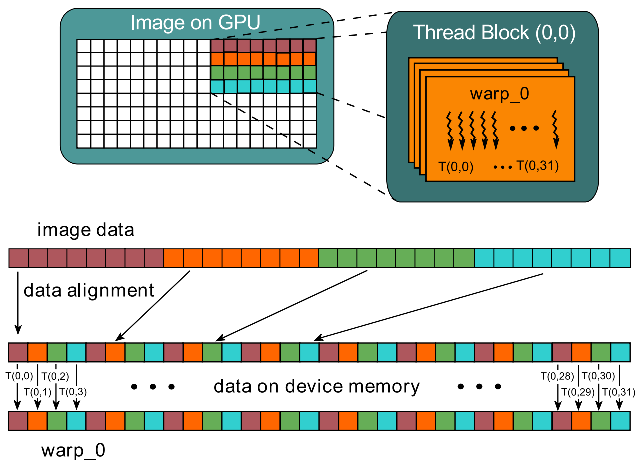


Fig. 5 Coalesced memory accesses illustrating a warp of 32 threads reading/writing the respective 32 data elements on a single clock cycle

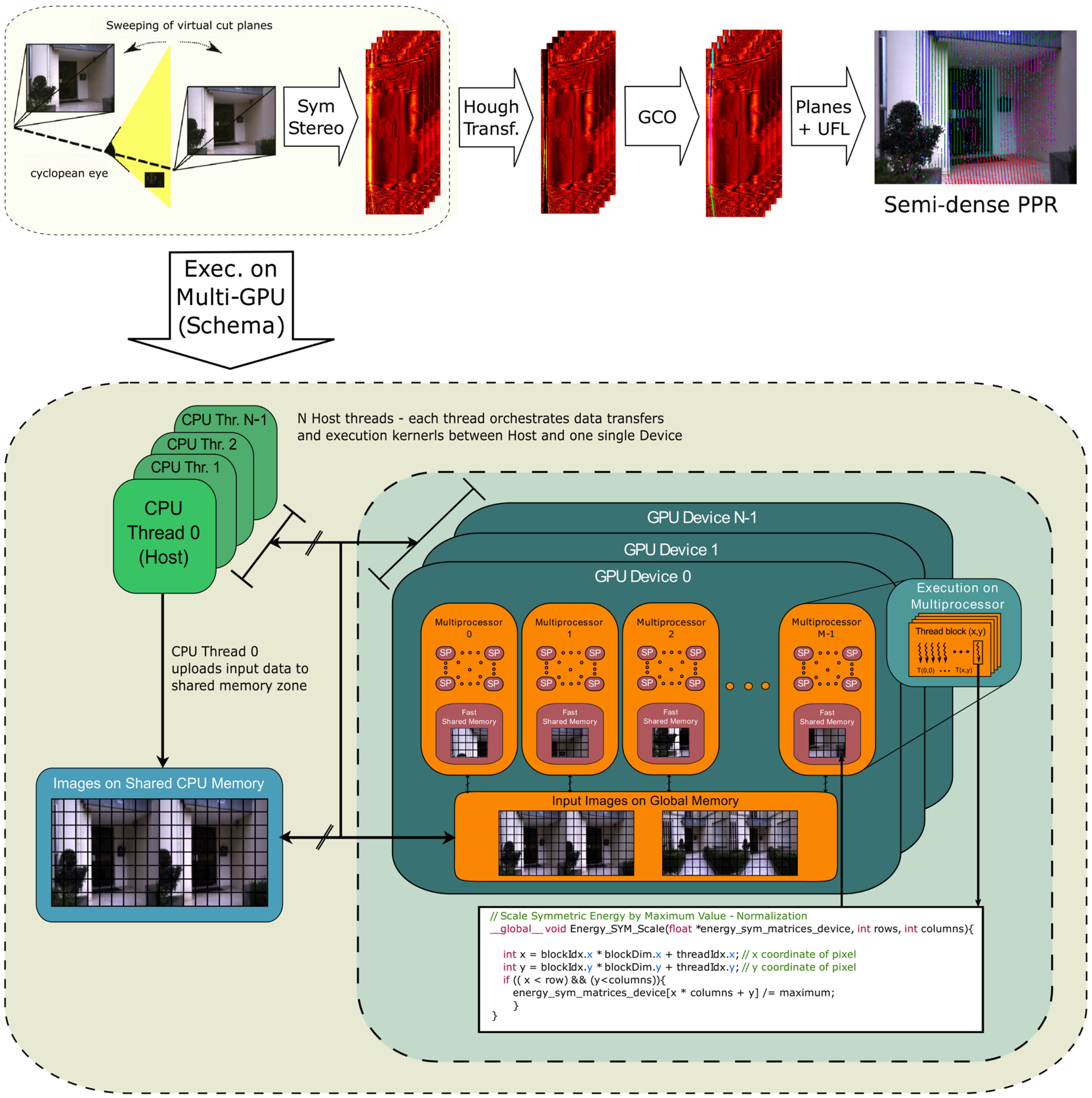
be higher in these cases. There is also no data reuse in this case (see Simple Arithmetic Operations in Sect. 4.4). The results of the maximum are processed in two stages: the first uses GPU grids with a 256×256 block size; the second uses 1×256. In the function that finds maximum value per row, we only use blocks with 128×1 threads, which is justified by the small size of shared memory. To compute the vector that stores the accumulator values for each column in a matrix, we use the *'cublasSgemv()'* function from the cuBLAS library. The remaining functions in the UFL pipeline only use global memory and GPU grids with 256×1 block sizes as best-performing configurations. Figure 4 shows the entire kernel execution pipeline for the UFL algorithm running on GPU.

5 Multi-GPU processor configuration

Here, a suitable multi-GPU framework speeds up the SymStereo and UFL algorithms of the 3D reconstruction framework. Table 4 shows all multi-GPU configurations tested, using an Intel Core i7-4790K @ 4.0GHz as the host CPU.

5.1 GPU workload balance

We propose a new workload distribution for concurrently running GPUs with distinct architectures (see Table 3). This new system comprises multi-threaded software running on the host CPU (see Fig. 6). One dedicated host thread is assigned to each GPU, and another host thread parses input data from disk or other I/O devices. A shared memory zone in the host is used to store input data and the necessary synchronization metadata. The first host thread (Thread


Sweeping of virtual cut planes
cyclopean eye
Sym Stereo
Hough Transf.
GCO
Planes + UFL
Semi-dense PPR
Exec. on Multi-GPU (Schema)
CPU Thr. N-1
CPU Thr. 2
CPU Thr. 1
CPU Thread 0 (Host)
N Host threads - each thread orchestrates data transfers and execution kernerls between Host and one single Device
GPU Device N-1
GPU Device 1
GPU Device 0
Execution on Multiprocessor
Thread block (x,y)
T(0,0) ... T(x,y)
Multiprocessor 0
Multiprocessor 1
Multiprocessor 2
Multiprocessor M-1
SP
Fast Shared Memory
CPU Thread 0 uploads input data to shared memory zone
Images on Shared CPU Memory
Input Images on Global Memory

```
// Scale Symmetric Energy by Maximum Value - Normalization
__global__ void Energy_SYM_Scale(float *energy_sym_matrices_device, int rows, int columns){

   int x = blockIdx.x * blockDim.x + threadIdx.x; // x coordinate of pixel
   int y = blockIdx.y * blockDim.y + threadIdx.y; // y coordinate of pixel
   if (( x < row) && (y<columns)){
      energy_sym_matrices_device[x * columns + y] /= maximum;
      }
 }
```

Fig. 6 The figure illustrates the different steps of the accelerated 3D reconstruction pipeline. For each stereo pair, a semi-dense PPR is computed as described in Sect. 2 using a multi-GPU workload distribution strategy. It illustrates how threaded blocks are processed simultaneously on GPU multiprocessors and how the same code segment is executed by multiple threads concurrently. Each thread processes a single pixel

0 in Fig. 6) reads the input data and stores it in a shared memory zone, setting a shared boolean variable to 1 (which indicates the arrival of input data). On the GPU device, all host threads that control GPU execution work concurrently to read the input data from the shared memory zone. If the boolean variable is set to 1, we can read data for processing (the first thread upcoming reads the data and sets the boolean variable to 0). Until the system has data to process, all GPUs work concurrently without significant idle times between executions or data transfers, as shown in Figs. 7 and 8.

With the adopted workload distribution, the execution workflow is different for distinct GPUs, reducing global

Table 4 Labeling for all multi-GPU configurations

Configuration	GPU hardware installed
A	GTX 680 / Tesla K40c
B	Tesla K40c / GTX TITAN
C	Tesla K40c / GTX TITAN X
D	GTX TITAN / GTX TITAN X
E	Tesla K40c / GTX TITAN / GTX TITAN X

execution times and minimizing previously existing idle times. The current solution can perform multiple tasks concurrently, such as running a kernel and transferring data between the host device and the GPU. This workload balancing can adapt to a variable number of GPUs. Thus, GPUs from different generations can operate together.

6 Performance evaluation

Here, we address the speedups obtained for SymStereo and UFL algorithms using single- and multi-GPU assemblies, by comparing against the sequential C versions running on an Intel Core i7-4790k CPU.

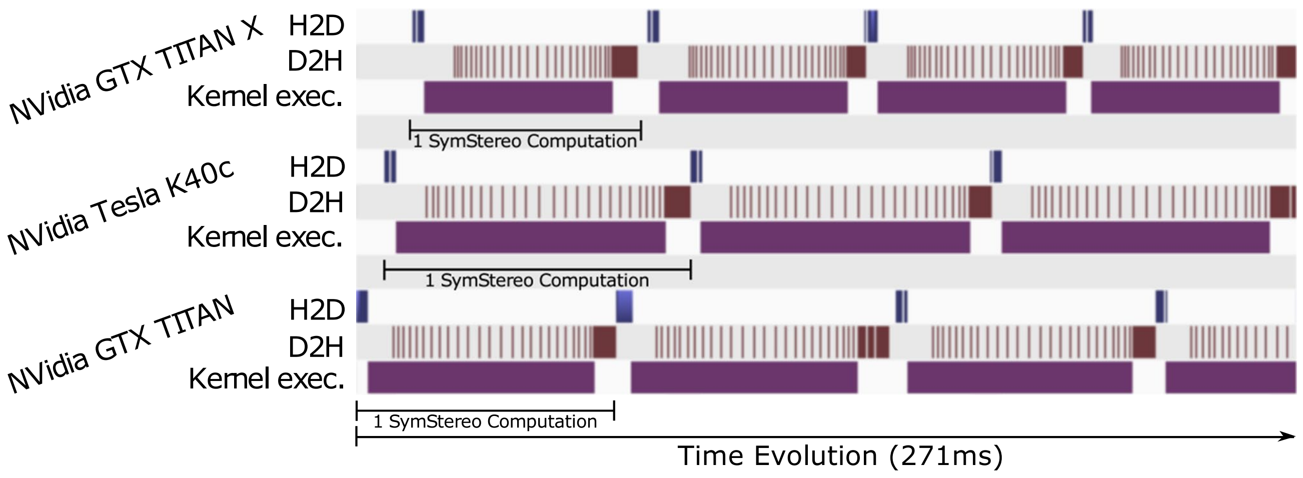


Fig. 7 Execution pipeline of SymStereo multi-GPU assemblies. In the figure, Configuration E (Tesla K40c / GTX TITAN / GTX TITAN X) considers the SymStereo procedure processing more than ten images using a time interval of approximately 271 ms. The tests were performed on urban images with 1024 × 768 pixels, 35 virtual cut planes and 15 LogGabor scales

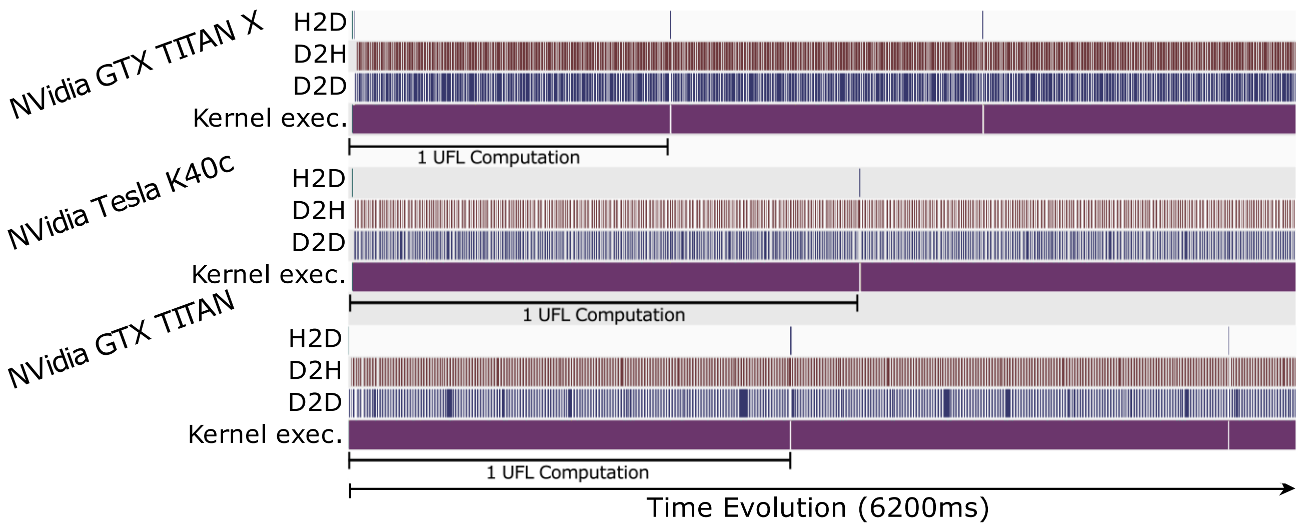


Fig. 8 Execution pipeline of UFL with workload distribution on multi-GPU assemblies. In the figure, Configuration E (Tesla K40c / GTX TITAN / GTX TITAN X) considers the UFL algorithm processing more than seven images using a time interval of approximately 6200 ms. The tests were performed using an input matrix with 26880 × 407

6.1 Single-GPU throughput and speedup evaluation

We first analyze the speedups achieved for the SymStereo and UFL algorithms using single-GPU assemblies and compare them with the sequential C versions running on an Intel Core i7-4790k CPU.

6.1.1 SymStereo

Table 5 shows the computation time and throughput in frames per second for the SymStereo algorithm, varying the number of virtual cut planes (we tested 15, 25, 35, 45) using 1024×768 images as stereo input. As displayed in this table, the faster GPU (GTX TITAN X) executes the entire procedure in 44.05ms for 15 virtual cut planes and 100.21ms for 45 virtual cut planes against 759ms and 1672ms produced by Intel Core i7-4790k.

Table 5 shows the single-GPU speedups. The best single-GPU assembly produces a speedup of 16.68 to 17.23× over execution on an Intel Core i7-4790k CPU.

6.1.2 UFL

Table 6 shows computation time and throughput (in matrices per second, or MPS) for the UFL algorithm. Input matrix dimensions 11520×180, 19200×271, 26880×407 and 34560×553 are considered. The fastest GPU (GTX TITAN X) executes the entire procedure in 0.5s for 11520×180

Table 5 Computation times, including Host/Device communication and data transfer, for single-GPU assemblies

Virtual cut planes	15		25		35		45	
Processing platform	Execution time (ms)	FPS	Execution time (ms)	FPS	Execution time (ms)	FPS	Execution time (ms)	FPS
Intel Core i7-4790K	759	1.32	1063	0.94	1365	0.73	1672	0.598
GTX 680	71.98	13.9	109.01	9.17	146.49	6.83	181.80	5.50
Tesla K40c	58.18	17.19	87.33	11.45	116.5	8.58	140.75	7.10
GTX TITAN	51.49	19.42	77.76	12.86	104.1	9.61	127.95	7.82
GTX TITAN X	44.05	22.70	63.75	15.69	83.58	11.96	100.21	9.98
	Speedup							
GTX 680	10.55 x		9.75 x		9.32 x		9.20 x	
Tesla K40c	13.05 x		12.17 x		11.71 x		11.88 x	
GTX TITAN	14.74 x		13.67 x		13.11 x		13.07 x	
GTX TITAN X	17.23 x		16.67 x		16.33 x		16.68 x	

Throughput is measured in frames per second. Speedup is calculated for the SymStereo procedure varying the number of virtual cut planes (15, 25, 35, 45). The tests were performed on urban images with 1024×768 pixels

Table 6 Computation times, including host/device communication and data transfer, for single-GPU assemblies

Input matrix dimensions	11520×180		19200×271		26880×407		34560×553	
Processing platform	Execution time (s)	MPS	Execution time (s)	MPS	Execution time (s)	MPS	Execution time (s)	MPS
Intel Core i7-4790K	6.13	0.16	15.2	0.066	31.9	0.031	55.5	0.018
GTX 680	0.95	1.05	2.30	0.43	4.70	0.21	7.10	0.14
Tesla K40c	0.70	1.42	1.60	0.63	3.30	0.30	5.20	0.19
GTX TITAN	0.60	1.67	1.40	0.71	2.90	0.34	4.70	0.21
GTX TITAN X	0.50	2.00	1.00	1.00	2.00	0.50	3.20	0.31
	Speedup							
GTX 680	6.4 ×		6.6 ×		6.7 ×		7.8 ×	
Tesla K40c	8.7 ×		9.5 ×		9.6 ×		10.6 ×	
GTX TITAN	10.2 ×		10.8 ×		11.0 ×		11.8 ×	
GTX TITAN X	12.2 ×		15.2 ×		15.9 ×		17.3 ×	

Throughput is measured in matrices per second and speedup for the UFL procedure. Input matrix dimensions 11520×180, 19200×271, 26880×407 and 34560×553 are considered input matrices and 3.2s for 34560×553 input matrices, compared with 6.13s and 55.5s for an Intel Core i7-4790k CPU.

Single-GPU speedups for UFL procedure are shown in Table 6. The best single-GPU assembly can solve the UFL problem between 12.2 and 17.3× faster than the Intel Core i7-4790k CPU.

6.2 Multi-GPU throughput and speedup evaluation

We present below the speedup results obtained by applying these parallelization techniques to our case studies, using multi-GPU systems.

6.2.1 SymStereo

Table 7 shows execution times and throughput for the SymStereo algorithm, for different virtual cut planes (15, 25, 35, 45) using the multi-GPU systems and 1024×768 images as stereo inputs. Configuration E (Tesla K40c / GTX TITAN / GTX TITAN X) executes the entire procedure in 17.62ms for 15 virtual cut planes and 41.68ms for 45 virtual cut planes, against 759ms and 1672ms produced by the Intel Core i7-4790k CPU. The multi-GPU speedups are also shown in Table 7. The best multi-GPU assembly produces a speedup between 43 and 40.1 when compared to the Intel Core i7-4790k CPU. With this new multi-GPU approach, we SymStereo algorithm execution is 2.5× faster than using the best single-GPU version (GTX TITAN X).

6.2.2 UFL

Table 8 shows execution times and throughput in matrices per second for the UFL algorithm, for distinct input matrix sizes (11520×180, 19200×271, 26880×407,

Table 7 Computation times, including host/device communication and data transfer, for multi-GPU assemblies

Virtual cut planes	15		25		35		45	
Configuration	Execution time (ms)	FPS	Execution time (ms)	FPS	Execution time (ms)	FPS	Execution time (ms)	FPS
A	30.45	32.84	46.51	21.50	62.95	15.89	76.96	12.99
B	27.36	36.55	38.60	25.91	52.07	19.21	63.60	15.72
C	24.97	40.05	35.42	28.23	47.03	21.26	56.96	17.56
D	23.61	42.36	33.98	29.42	45.69	21.89	55.12	18.14
E	17.62	56.75	25.29	39.54	34.34	29.12	41.68	23.99
	Speedup							
A	24.9 ×		22.8 ×		21.6 ×		21.7 ×	
B	27.7 ×		27.5 ×		26.2 ×		26.2 ×	
C	30.4 ×		30.0 ×		29.0 ×		29.3 ×	
D	32.1 ×		31.2 ×		29.8 ×		30.3 ×	
E	43.0 ×		42.0 ×		39.7 ×		40.1 ×	

Throughput is measured in frames per second and speedup for the SymStereo procedure varying the number of virtual cut planes (15, 25, 35, 45). The tests were performed on urban images with 1024×768 pixels

Table 8 Execution times including host/device communications and data transfers for multi-GPU assemblies, throughput measured in matrices per second and speedup for the UFL procedure for different matrix sizes (11520×180, 19200×271, 26880×407, 34560×553)

Input matrix dimensions	11520×180		19200×271		26880×407		34560×553	
Configuration	Execution time (s)	MPS	Execution time (s)	MPS	Execution time (s)	MPS	Execution time (s)	MPS
A	0.43	2.33	0.97	1.03	1.95	0.51	3.02	0.33
B	0.35	2.86	0.78	1.28	1.56	0.64	2.55	0.39
C	0.3	3.33	0.65	1.54	1.26	0.79	2.03	0.49
D	0.28	3.57	0.61	1.64	1.25	0.80	2.01	0.50
E	0.21	4.76	0.46	2.17	0.91	1.10	1.46	0.68
	Speedup							
A	14.3 ×		15.7 ×		16.4 ×		18.4 ×	
B	17.5 ×		19.5 ×		20.4 ×		21.8 ×	
C	20.4 ×		23.4 ×		25.3 ×		27.3 ×	
D	21.9 ×		24.9 ×		25.5 ×		27.6 ×	
E	29.2 ×		33.0 ×		35.1 ×		38.0 ×	

34560×553). Configuration E executes the entire procedure in 0.21s for a 11520×180 input matrix and 1.46s for a 34560×553 input matrix, against 6.13s and 55.5s produced by an Intel Core i7-4790k CPU.

Table 8 also shows the multi-GPU speedups for the UFL procedure. Thus, the best multi-GPU assembly achieves a speedup between 29.2 and 38× faster than Intel Core i7-4790k CPU. By comparing against the best single-GPU version (GTX TITAN X), we can solve the UFL problem up to 2.39× faster using the multi-GPU approach.

7 Conclusions and remarks

With the rapidly increasing performance of graphics processors, improved programming support and excellent price-to-performance ratio, GPUs have emerged as competitive parallel computing platforms for computationally expensive tasks in a wide range of image processing applications. Our semi-dense PPR algorithm is a computationally intensive part of the global 3D reconstruction pipeline, and it can be processed independently for each stereo pair. Consequently, we have identified and parallelized the semi-dense PPR functions. In Table 2, we can see the more expensive semi-dense PPR procedures (SymStereo and UFL) that require parallelization. Thus, we introduced single-GPU and multi-GPU versions of a framework for computing the SymStereo and UFL procedures, which can be helpful in many computer vision applications. Also, in the multi-GPU approach, the workload distribution adopts a generic method that can be used to run other applications without significant changes. To perform these steps efficiently, we have built parallel kernels that make appropriate use of the memory hierarchy, supported by parallel C code and the CUDA API. This approach allows multiple pixels of an image to be processed simultaneously. The multi-GPU framework can process multiple images concurrently, using different devices, thus producing the reported throughput. We show that by varying the number of cut planes from 45 to 15, the parallel implementation of SymStereo (*logN*) running in our best single-GPU platform (GTX TITAN X) processes between 9.98 and 22.70 frames per second, with corresponding speedups of 16.68 and 17.23× over a sequential C implementation running on CPU (Intel Core i7-4790k). The multi-GPU system with configuration E (Tesla K40c / GTX TITAN / GTX TITAN X) achieves a throughput ranging from 23.99 to 56.75 frames per second processing SymStereo, and a speedup from 40.1 up to 43.0× over the sequential C implementation. For SymStereo, we can conclude that the multi-GPU approach can be up to 2.5× faster than the best single-GPU version (GTX TITAN X).

In the message-passing algorithm for processing the UFL, with matrix dimensions ranging from 11520×180 to 34560×553, the best single-GPU configuration considered (GTX TITAN X) processes between 0.31 and 2.0 matrices per second with an associated speedup of 17.3× over the sequential C implementation. The multi-GPU system with configuration E achieves a throughput between 0.68 and 4.76 matrices per second for UFL, and a speedup of 38× as compared to the sequential C implementation. We conclude that the multi-GPU approach solves the UFL algorithm up to 2.39× faster than using the best single-GPU version (GTX TITAN X). Our workload distribution achieves superior performance when compared with the best multi-GPU workload distributions in our previous work [13]. With the speedups obtained in both algorithms, we can reduce the execution time of the entire 3D reconstruction pipeline, targeting a real-time system in the near future.

The parallel CUDA source code and datasets are available online [15].

Acknowledgements This work was partially supported by a Google Faculty Research Award from Google Inc. and also by the Portuguese Foundation for Science and Technology (FCT) under grant AMS-HMI12: RECI/EEIAUT/0181/2012. It was equally supported by Instituto de Telecomunicações and funded by FCT/MCTES through national funds and when applicable co-funded EU funds under the project UIDB/EEA/50008/2020. It was carried out at the Multimedia Signal Processing Laboratory, a GPU Research Center from the University of Coimbra, Portugal.

References

1. Alexiadis, D., Zarpalas, D., Daras, P.: Real-time, full 3-D reconstruction of moving foreground objects from multiple consumer depth cameras. IEEE Trans. Multimedia **15**(2), 339–358 (2013). https://doi.org/10.1109/TMM.2012.2229264
2. Aliakbarpour, H., Almeida, L., Menezes, P., Dias, J.: Multi-sensor 3D volumetric reconstruction using CUDA. 3D Research **2**(4), 6 (2011). https://doi.org/10.1007/3DRes.04(2011)6
3. Antunes, M., Barreto, J.P.: Semi-dense piecewise planar stereo reconstruction using SymStereo and PEARL. In: Second International Conference on3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT), pp. 230–237. IEEE (2012)
4. Antunes, M., Barreto, J.P.: SymStereo: stereo matching using induced symmetry. Int. J. Comput. Vis. **10**, 1–22 (2014)
5. Antunes, M., Barreto, J.P., Nunes, U.: Piecewise-planar reconstruction using two views. Image Vis. Comput. **46**, 47–63 (2016)
6. Antunes, M.G.: Stereo Reconstruction using Induced Symmetry and 3D scene priors. Ph.D thesis, http://www2.isr.uc.pt/michel/files/final.pdf (2014)
7. Bódis-Szomorú, A., Riemenschneider, H., Van Gool, L.: Fast, approximate piecewise-planar modeling based on sparse structure-from-motion and superpixels. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR, pp. 469–476 (2014)
8. Delong, A., Osokin, A., Isack, H.N., Boykov, Y.: Fast approximate energy minimization with label costs. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR, pp. 2173–2180. IEEE (2010)
9. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**(2), 216–231 (2005)

10. Frome, A., Cheung, G., Abdulkader, A., Zennaro, M., Wu, B., Bissacco, A., Adam, H., Neven, H., Vincent, L.: Large-scale privacy protection in Google Street View. In: IEEE International Conference on Computer Vision, CVPR, pp. 2373–2380 (2009). https://doi.org/10.1109/ICCV.2009.5459413
11. Gallup, D., Frahm, J.M., Pollefeys, M.: Piecewise planar and non-planar stereo for urban scene reconstruction. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR, pp. 1418–1425. IEEE (2010)
12. Gijsen, F.J., Schuurbiers, J.C., van de Giessen, A.G., Schaap, M., van der Steen, A.F., Wentzel, J.J.: 3D reconstruction techniques of human coronary bifurcations for shear stress computations. J. Biomech. **47**(1), 39–43 (2014). https://doi.org/10.1016/j.jbiomech.2013.10.021
13. Graca, C., Falcao, G., Figueiredo, I., Kumar, S.: Hybrid multi-GPU computing: accelerated kernels for segmentation and object detection with medical image processing applications. J. Real-Time Image Process. (2015). https://doi.org/10.1007/s11554-015-0517-3
14. Graca, C., Falcao, G., Kumar, S., Figueiredo, I.: Cooperative use of parallel processing with time or frequency-domain filtering for shape recognition. In: Proceedings of the 22nd European Signal Processing Conference, EUSIPCO, pp. 2085–2089 (2014)
15. Graca, C., Raposo, C., Barreto, J.P., Nunes, U., Falcao, G.: Urban-Scan Website: 3D modeling of urban scenes. http://montecristo.co.it.pt/PPR_Rec/ (2016)
16. Guenoun, B., Hajj, F.E., Biau, D., Anract, P., Courpied, J.P.: Reliability of a new method for evaluating femoral stem positioning after total hip arthroplasty based on stereoradiographic 3D reconstruction. J. Arthroplasty **30**(1), 141–144 (2015). https://doi.org/10.1016/j.arth.2014.07.033
17. Harris, M., et al.: Optimizing parallel reduction in CUDA. Nvidia Dev. Technol. **2**(4), 70 (2007)
18. Hirschmuller, H., Scharstein, D.: Evaluation of cost functions for stereo matching. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR, pp. 1–8. IEEE (2007)
19. Izadi, S., Kim, D., Hilliges, O., Molyneaux, D., Newcombe, R., Kohli, P., Shotton, J., Hodges, S., Freeman, D., Davison, A., Fitzgibbon, A.: KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera. In: Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, UIST '11, pp. 559–568. ACM, New York, NY, USA (2011). https://doi.org/10.1145/2047196.2047270
20. Kalarot, R., Morris, J.: Implementation of symmetric dynamic programming stereo matching algorithm using CUDA pp. 141–146 (2010)
21. Kou, W., Cheong, L.F., Zhou, Z.: Proximal robust factorization for piecewise planar reconstruction. Comput. Vis. Image Underst. **166**, 88–101 (2018)
22. Ladikos, A., Benhimane, S., Navab, N.: Efficient visual hull computation for real-time 3D reconstruction using CUDA. In: IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPRW, pp. 1–8 (2008). https://doi.org/10.1109/CVPRW.2008.4563098
23. Lazic, N., Frey, B.J., Aarabi, P.: Solving the uncapacitated facility location problem using message passing algorithms. In: International Conference on Artificial Intelligence and Statistics, pp. 429–436 (2010)
24. Li, J., Sun, J., Song, Y., Zhao, J.: Accelerating MRI reconstruction via three-dimensional dual-dictionary learning using CUDA. J. Supercomput. **71**(7), 2381–2396 (2015). https://doi.org/10.1007/s11227-015-1386-z
25. Luo, Y., Duraiswami, R.: Canny edge detection on Nvidia CUDA. In: IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPRW, pp. 1–8 (2008). https://doi.org/10.1109/CVPRW.2008.4563088
26. Mei, X., Sun, X., Zhou, M., Jiao, S., Wang, H., Zhang, X.: On building an accurate stereo matching system on graphics hardware. In: IEEE International Conference on Computer Vision Workshops, ICCV Workshops, pp. 467–474 (2011). https://doi.org/10.1109/ICCVW.2011.6130280
27. Melo, R., Barreto, J.P., Falcao, G.: A new solution for camera calibration and real-time image distortion correction in medical endoscopy - initial technical evaluation. IEEE Trans. Biomed. Eng. **59**(3), 634–644 (2012). https://doi.org/10.1109/TBME.2011.2177268
28. Micusik, B., Kosecka, J.: Piecewise planar city 3d modeling from street view panoramic sequences. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR, pp. 2906–2912 (2009). https://doi.org/10.1109/CVPR.2009.5206535
29. Nvidia, C.: cuBLAS. [Online]. https://developer.nvidia.com/cuBLAS (2015)
30. Nvidia, C.: cuFFT. [Online]. https://developer.nvidia.com/cuFFT (2015)
31. Nvidia, C.: CUDA C best practices guide. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#device-memory-spaces (2019)
32. Ogawa, K., Ito, Y., Nakano, K.: Efficient canny edge detection using a GPU, pp. 279–280 (2010). https://doi.org/10.1109/IC-NC.2010.13
33. Park, I.K., Singhal, N., Lee, M.H., Cho, S., Kim, C.: Design and Performance Evaluation of Image Processing Algorithms on GPUs. IEEE Trans. Parallel Distrib. Syst. **22**(1), 91–104 (2011). https://doi.org/10.1109/TPDS.2010.115
34. Podlozhnyuk, V., Harris, M., Young, E.: Nvidia CUDA C programming guide. Nvidia Corporation (2012)
35. Pollefeys, M., Nistér, D., Frahm, J.M., Akbarzadeh, A., Mordohai, P., Clipp, B., Engels, C., Gallup, D., Kim, S.J., Merrell, P., Salmi, C., Sinha, S., Talton, B., Wang, L., Yang, Q., Stewénius, H., Yang, R., Welch, G., Towles, H.: Detailed real-time urban 3D reconstruction from video. Int. J. Comput. Vis. **78**(2–3), 143–167 (2008). https://doi.org/10.1007/s11263-007-0086-4
36. Ralha, R., Falcao, G., Amaro, J., Mota, V., Antunes, M., Barreto, J., Nunes, U.: Parallel refinement of slanted 3D reconstruction using dense stereo induced from symmetry. Journal of Real-Time Image Processing pp. 1–19 (2016). https://doi.org/10.1007/s11554-016-0592-0
37. Raposo, C., Antunes, M., Barreto, J.: Piecewise-planar stereoscan:structure and motion from plane primitives. In: D. Fleet, T. Pajdla, B. Schiele, T. Tuytelaars (eds.) European Conference on Computer Vision, ECCV, *Lecture Notes in Computer Science*, vol. 8690, pp. 48–63. Springer International Publishing (2014). https://doi.org/10.1007/978-3-319-10605-2_4
38. Raposo, C., Antunes, M., Barreto, J.P.: Piecewise-Planar StereoScan: sequential structure and motion using plane primitives. In: IEEE Transactions on Pattern Analysis and Machine Intelligence (2017)
39. Raposo, C., Barreto, J.P.: πMatch: Monocular vSLAM and Piecewise Planar Reconstruction using Fast Plane Correspondences. In: European Conference on Computer Vision, ECCV, pp. 380–395. Springer (2016)
40. Salmen, J., Houben, S., Schlipsing, M.: Google street view images support the development of vision-based driver assistance systems. In: IEEE Intelligent Vehicles Symposium, IV, pp. 891–895. IEEE (2012)
41. Seitz, S.M., Curless, B., Diebel, J., Scharstein, D., Szeliski, R.: A comparison and evaluation of multi-view stereo reconstruction algorithms. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR, vol. 1, pp. 519–528. IEEE (2006)

42. Sinha, S.N., Steedly, D., Szeliski, R.: Piecewise planar stereo for image-based rendering. In: IEEE International Conference on Computer Vision Workshops, ICCV, pp. 1881–1888 (2009)
43. Torii, A., Havlena, M., Pajdla, T.: From Google street view to 3d city models. In: IEEE International Conference on Computer Vision Workshops, ICCV Workshops, pp. 2188–2195 (2009). https://doi.org/10.1109/ICCVW.2009.5457551
44. Vineet, V., Narayanan, P.: CUDA cuts: Fast graph cuts on the GPU. In: IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPRW, pp. 1–8 (2008). https://doi.org/10.1109/CVPRW.2008.4563095
45. Vogel, C., Roth, S., Schindler, K.: View-consistent 3d scene flow estimation over multiple frames. In: European Conference on Computer Vision, ECCV, pp. 263–278. Springer (2014)
46. Whitmarsh, T., Humbert, L., Barquero, L.M.D.R., Gregorio, S.D., Frangi, A.F.: 3D reconstruction of the lumbar vertebrae from anteroposterior and lateral dual-energy X-ray absorptiometry. Medical Image Analysis **17**(4), 475–487 (2013). https://doi.org/10.1016/j.media.2013.02.002. http://www.sciencedirect.com/science/article/pii/S1361841513000091
47. Woodford, O., Torr, P., Reid, I., Fitzgibbon, A.: Global stereo reconstruction under second-order smoothness priors. IEEE Trans. Pattern Anal. Mach. Intell. **31**(12), 2115–2128 (2009)
48. Yang, Z., Zhu, Y., Pu, Y.: Parallel image processing based on CUDA. Int. Conf. Comput. Sci. Softw. Eng. **3**, 198–201 (2008). https://doi.org/10.1109/CSSE.2008.1448
49. Yeom, E., Nam, K.H., Jin, C., Paeng, D.G., Lee, S.J.: 3D reconstruction of a carotid bifurcation from 2D transversal ultrasound images. Ultrasonics **54**(8), 2184–2192 (2014)
50. Zamir, A.R., Shah, M.: Accurate image localization based on google maps street view. In: European Conference on Computer Vision, ECCV, pp. 255–268. Springer (2010)
51. Zhang, K., Lu, J., Yang, Q., Lafruit, G., Lauwereins, R., Van Gool, L.: Real-time and accurate stereo: A scalable approach with bitwise fast voting on CUDA. IEEE Trans. Circuits Syst. Video Technol. **21**(7), 867–878 (2011). https://doi.org/10.1109/TCSVT.2011.2133150