Free-viewpoint Immersive Networked Experience

# D4.3 Study of the feasibility of parallelizing view synthesis algorithms using high performance computing on multiple GPUs

| | |
|---|---|
| Project ref. no. | ICT-FP7-248020 |
| Project acronym | FINE |
| Start date of project (dur.) | 1 April, 2010 (36 months) |
| Document due Date : | 30 September, 2011 |
| Actual date of delivery | 30 September, 2011 |
| Leader of this deliverable | MPRO |
| Reply to | xarmangue@mediapro.es |
| Document status | Final version for internal review |

| Version | Date | Description |
|---|---|---|
| 0.1 | 04/07/11 | Initial version |
| 1.0 | 16/09/11 | Final version for internal review |
| 1.1 | 29/09/11 | Final version |
| | | |

**Deliverable Identification Sheet**

| | |
|---|---|
| **Project ref. no.** | ICT-FP7-248020 |
| **Project acronym** | FINE |
| **Project full title** | Free-viewpoint Immersive Networked Experience |
| **Document name** | Study of the feasibility of parallelizing view synthesis algorithms using high performance computing on multiple GPUs |
| **Security (distribution level)** | CO |
| **Contractual date of delivery** | Month 18, 30.09.2011 |
| **Actual date of delivery** | Month 18, 30.09.2011 |
| **Deliverable number** | D4.3 |
| **Deliverable name** | Study of the feasibility of parallelizing view synthesis algorithms using high performance computing on multiple GPUs |
| **Type** | Prototype |
| **Status & version** | Final version |
| **Number of pages** | 42 |
| **WP / Task responsible** | WP4 / MPRO |
| **Other contributors** | BM |
| **Author(s)** | Xavier Armangué, Antonio Baeza |
| **EC Project Officer** | Ertrit Puka |
| **Abstract** | In this document is reported the work done in the WP4 regarding parallelization of view synthesis algorithms. |
| **Keywords** | Virtual view synthesis, parallelization, GPU |
| **Sent to peer reviewer** | *Date 20/09/2011 to David Pujals (RETE)* |
| **Peer review completed** | *Date 27/09/2011* |
| **Circulated to partners** | Via plone |
| **Mgt. Board approval** | To be approved at the next SB meeting |

## Table of contents

## 1 Public Executive Summary

This document reports the work done in WP4T2 on parallelization of view synthesis algorithms using high performance computing. This task is aimed at computing depth and 3D reconstruction for virtual view synthesis. In a previous report of this task (Baeza et al., 2011) focus was made on the exploration of the capabilities and potential application of the various methods analyzed. This document continues the previous work with a study about how to improve the performance of these techniques to obtain faster and more robust results.

This report addresses the following project objectives:

- To generate a free-viewpoint video representation of a 3D scene at real-time performance from the captured data (O2)
- To develop image-based algorithms for photorealistic rendering of 3D characters synchronized with live-action video feeds (O5)

Virtual view synthesis aims to generate synthetic views, corresponding to arbitrary viewpoints, using as input real images captured with cameras whose positions and calibration are known. In this work we focus on the case of live sport events, and in particular to the application to soccer games.

The process of generating virtual views from multiple cameras involves several independent or nearly independent parts. The quality and speed of the results provided by each part determines the final characteristics of the synthetic images.

In this report we analyze the feasibility of parallelizing some of the algorithms involved in virtual view synthesis using Graphic Processing Units (GPUs) and multi-GPU systems. We have focused the analysis on segmentation, depth computation and 3D reconstruction, which are core parts of the novel view synthesis algorithm, and are representative for the analysis.

A careful analysis of single-GPU implementations of the aforementioned algorithms has been performed. The analysis shows the better performance of the GPU implementation with respect to single threaded CPU code. It also points out the weakest points, where performance can be improved. Solutions to these points are proposed.

In order to extend the conclusions of the single-GPU case to multiple GPUs, a more qualitative analysis has been performed. Simple algorithms have been used to show that the performance scales properly when using multiple GPUs.

## 2 Introduction

The purpose of the document is to report the work done in WP4T2 on parallelization of view synthesis algorithms using high performance computing. This task is aimed at computing depth and 3D reconstruction for virtual view synthesis. In a previous report of this task (Baeza et al., 2011) focus was made on the exploration of the capabilities and potential application of the various methods analyzed. This document continues the previous work with a study about how to improve the performance of these techniques to obtain faster and more robust results.

This report addresses the following project objectives:

- To generate a free-viewpoint video representation of a 3D scene at real-time performance from the captured data (O2)
- To develop image-based algorithms for photorealistic rendering of 3D characters synchronized with live-action video feeds (O5)

Virtual view synthesis aims to generate synthetic views, corresponding to arbitrary viewpoints, using as input real images captured with cameras whose positions and calibration are known. In this work we focus on the case of live sport events, and in particular to the application to soccer games.

The process of generating virtual views involves many sub-problems that are research problems in themselves. The following are the main sub-problems:

- Background learning. As a part of the segmentation process, a model for the background is computed.
- Background subtraction. The segmentation masks for the players are obtained as a difference between the actual images and the background model.
- Depth computation. Depth is computed for the pixels of each camera corresponding to the segmentation masks.
- 3D reconstruction. A volumetric 3D reconstruction is obtained by depth merging.
- Generation of the virtual image. With the aid of the computed depths and/or the 3D reconstruction, a novel view is generated.

All of these sub-problems, and especially depth computation and 3D reconstruction are difficult inverse problems. Most of the current solutions are computationally very expensive. For practical, real-time applications fast parallel implementations are required. A study of the feasibility of parallelizing novel view synthesis algorithms implies a careful study of each of these sub-problems.

Parallel algorithms, in contrast with sequential algorithms, are those pieces of code designed to be executed by more than one processing unit, operating concurrently on pieces of the input data. Today’s computers are endowed with many-core processors and GPUs, which allow for the execution of parallel algorithms on low cost dedicated servers or even in desktop workstations. As the requirements for real-time or near real-time computations on image processing are becoming more and more demanding, the design of parallel algorithms becomes a must. In particular, GPUs are precisely designed for the parallel execution of operations on images and 3D volumes, making them particularly well suited for the implementation of parallel view synthesis algorithms. Further, multi-GPU machines are nowadays available at a very reasonable cost, in systems equipped with many-core processors. For the best performance, an adequate scheduling of the target computations to the various processing units, GPUs and CPUs, as well as a overlapping of computations with data transfers is necessary.

In this report we analyze the feasibility of parallelizing some of the above algorithms using Graphic Processing Units (GPUs) and multi-GPU systems. We have focused the study on the algorithms for background learning and subtraction, depth computation and 3D reconstruction, as they are representative for the analysis. These modules constitute the core part of the process of generating virtual views from multiple images. A block diagram of the complete process is shown in Figure 1.

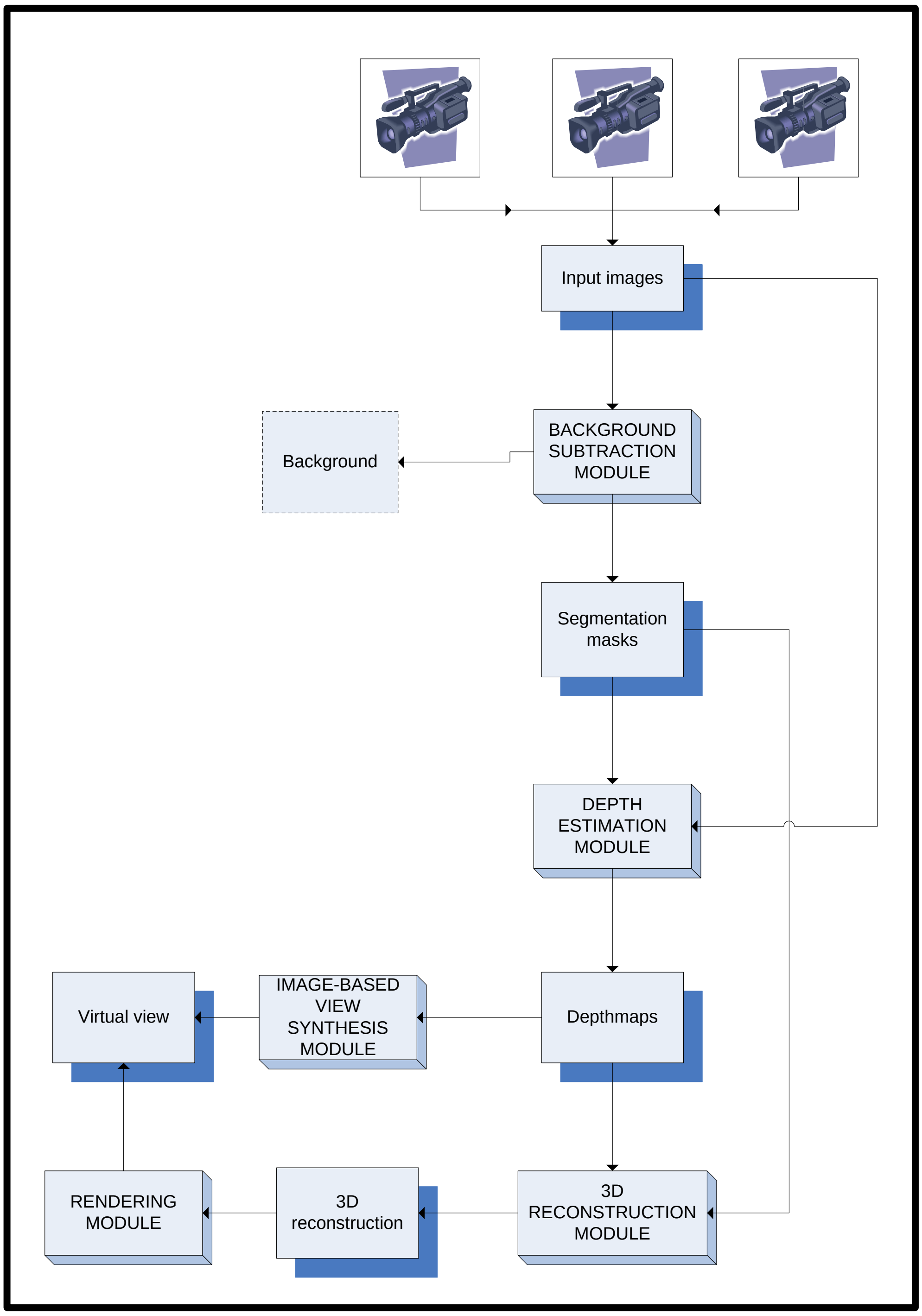


**Figure 1: Block diagram of the generation of virtual views from multiple images.**

The next section presents a review of these three algorithms. Then, Section 4 details the study of the feasibility of the parallelization GPUs. After that, conclusions of this report are presented in Section 5. Section 6 contains all references of this work. Finally document ends with 3 appendices with samples of code used to test the performance of parallelization.

## 3 Review of some algorithms involved in virtual view synthesis

In this Section we review some important algorithms involved in the generation of novel views, which will be used in the next section to study their parallelization on GPUs. We focus on algorithms for background subtraction, depth computation and 3D reconstruction. Although other algorithms could also be subject of parallelization, the analysis of the aforementioned cases covers the scope of this work.

### *3.1 Background subtraction / segmentation*

Background subtraction (BGS) is a process to segment out the foreground objects from the background of a video, and is a fundamental task in many applications of computer vision. The simplest case happens when the camera is static, and the background is often defined as the pixels that remain relatively constant along frames and whose colour is similar to its neighbours.

In FINE we need to segment moving players from video sequences for various purposes. The extracted masks will be used to constraint the 3D reconstruction of the players as well as to reduce the number of pixels processed to compute depth and 3D reconstruction. It is, therefore, a pre-processing step and should be as fast as possible in order to reduce the total delay of the process.

In real-life scenarios, the video can be affected by various factors: illumination changes, moving objects in the background, moving camera, etc. In order to take into account those issues, numerous BGS methods have been developed by the community, which are often divided into two groups: *parametric* models and *non-parametric* models. Parametric models, such as

- Approximated median filtering (AMF) (McFarlane & Schofield, 1995),
- Running Gaussian average (Wren et al., 1997), or
- Gaussian mixture model (GMM) (Stauffer & Grimson, 1999),

maintain a single background model that is updated with each new video frame. Non-parametric models, such as

- Eigenbackgrounds (Oliver et al, 2000),
- Kernel density estimator (KDE) (Elgammal et al., 2000)
- Median filtering (Calderara et al., 2006).

store several frames and estimate a background based on statistical proprieties of these fixed frames.

There are some surveys and comparative studies that examine a wide-range of BGS methods (Piccardi, 2004), (Benezeth et al., 2008), (Radke et al., 2005) (Parks & Fels, 2008). In those studies, it can be seen that no BGS method consistently outperforms any other; the choice of BGS algorithms thus depends on the specific context.

For images with players in a football field, parametric methods are considered to be good enough. A pixel in a new image is regarded as background if its new value is well described by the density function of the pixel. The Gaussian mixture model (GMM) approach proposed by (Stauffer & Grimson, 1999) is often considered as the exemplary version of GMM for BGS. The OpenCV library[1] contains several methods of background subtraction including Stauffer and Grimson's method. Figure 2 shows an example of background subtraction using the OpenCV library.

[1] OpenCV, Open Source Computer Vision. http://opencv.willowgarage.com/wiki/

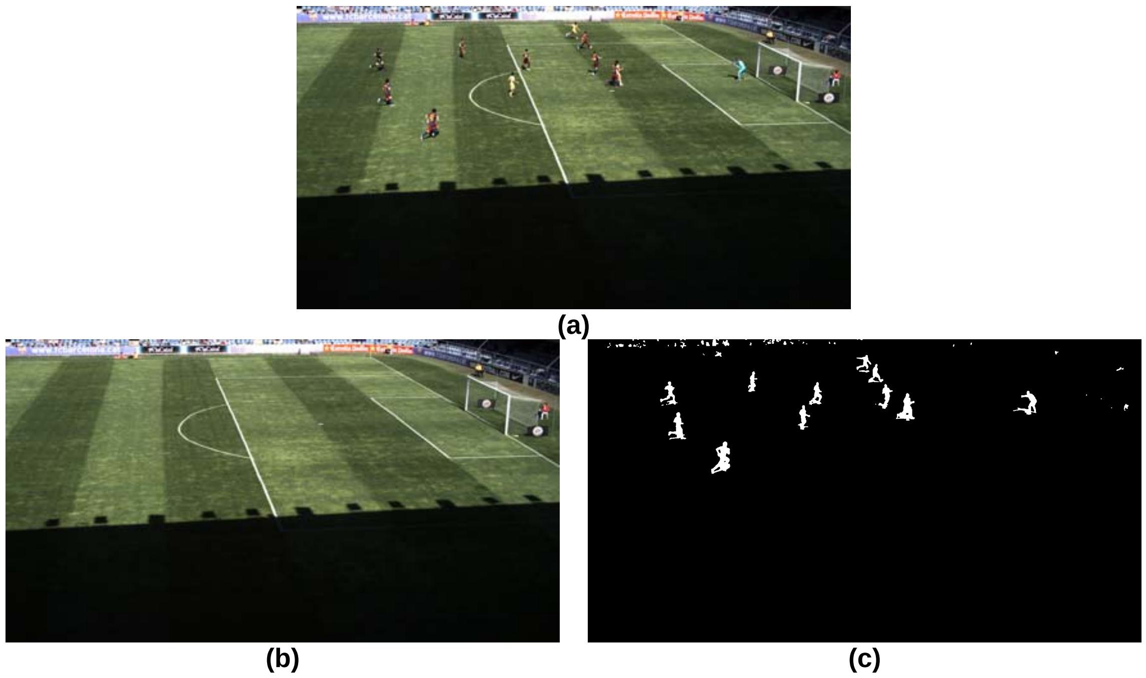


**Figure 2: Example of Background subtraction: a) Input Image; b) Background image; and c) Foreground image.**

## *3.2 Depth computation*

Depth computation aims to estimate a depth value for each image pixel, i.e., a distance to the camera, for the object whose projection in the camera is the given pixel. Numerous methods for solving this problem have been developed (Scharstein & Szeliski, 2002). Among the most common methods we can mention local winner-takes-all methods, like the plane-sweep algorithm (Collins, 1996), approaches based on graph cuts optimization (Boykov et al., 2001), (Kolmogorov & Zabih, 2001), belief propagation (Sun et al, 2003), (Yang et al., 2009) and variational methods (Pock et al., 2008).

In this Section a review of the main facts about a variational method for depth computation from images is given. In (Baeza et al., 2011) two approaches for depth computation were presented. One of them is based on graph cuts minimization, and the other is based on a variational formulation of the energy. Multi-label depth computation by graph cuts minimization (Papadakis & Caselles, 2010) is not parallelizable "as is", because of the inter-dependence between nodes in the graph. Some modified algorithms have been developed to address this limitation, mainly by dividing the graph into independent subgraphs and later merging the results to achieve a global minimum (Vineet & Narayanan, 2008, Liu & Sun, 2010). The results, however, only outperform the sequential algorithm in some cases, with relatively modest speedups. We thus focus on the feasibility of parallelizing the variational method. Let us briefly review this algorithm. We refer to (Baeza et al., 2011) for further details.

Consider a set of $K$ images $I_k, 1 \le k \le K$, obtained by means of calibrated cameras $C_k$. To compute the depth of the scene as seen from one of those cameras, denoted by $I_{ref}$, define a set of planes $\Pi_\lambda$ parallel to the camera plane $\Omega$ of $C_{ref}$. The problem is formulated as the minimization of the functional

$$J(u) = \int_\Omega \left|\nabla u(x,y)\right| dxdy + \int_\Omega \rho(x,y,u(x,y))dxdy, \quad (1)$$

where $u(x,y):\Omega \to [a,b]$ is the depth of pixel $(x,y)$, and the function $\rho(x,y,u(x,y))$ represents the cost of assigning a depth $u(x,y)$ to the pixel $(x,y)$. The depth range $[a,b]$ is computed as the minimum and maximum distances to the reference camera of a predefined 3D bounding box that contains the objects of interest.

In this work the data term $\int_\Omega \rho(x,y,u(x,y))dxdy$ is computed by photo-consistency, using the colour difference between each pixel $(x,y)$ and its reprojections at depth $u(x,y)$ onto some auxiliary input images. An illustration of how the photo-consistency influences the algorithm is provided in Figure 3. Three target depths, $D_1, D_2$ and $D_3$ are considered for the pixels of the central image. For each pixel $P=(x,y)$ its colour is compared with the colour of its reprojections at the considered depths on a pair of reference images. The algorithm will prefer to select the correct depth $D_2$, because its reprojections $Q_2$ and $R_2$ have the same colour as $P$, while the reprojections of the target depths $D_1$ and $D_3$ fall outside the ball, in a white area, in the auxiliary images.

The regularization term $\int_\Omega |\nabla u(x,y)| dxdy$ helps discarding bad reprojections that are photo-consistent by chance. The algorithm will thus look for the depth that minimizes the data term while having a depth similar to its neighbours.

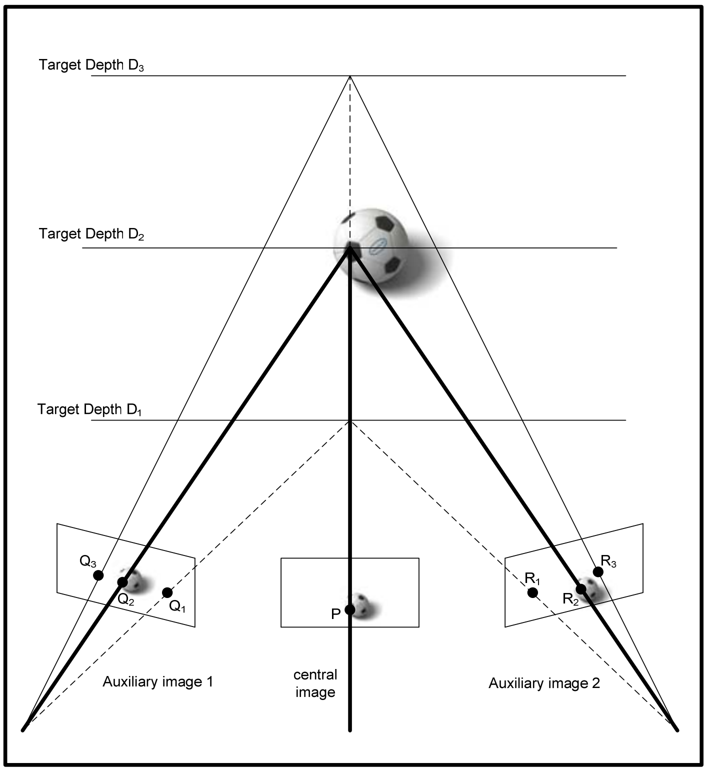


**Figure 3: Illustration of the depth estimation algorithm**

Following (Pock et al., 2008), a convexification of the functional (1) is performed. Let $\phi(x,y,s) = H(u(x,y)-s)$, where $H$ is the Heaviside function $H(r)=1$ if $r \geq 0$, $H(r)=0$ otherwise. The functional (1) can be written in terms of $\phi$ as

$$F(\phi) = \int_\Omega \int_a^b \left\{ |\nabla_{x,y}\phi(x,y,s)| + \rho(x,y,u(x,y))|\phi_s(x,y,s)| \right\} dsdxdy. \qquad (2)$$

The auxiliary function $\phi$ has to verify

$$0 \le \phi(x,y,s) \le 1, \phi_s \le 0. \quad (3)$$

For the solution of (2) an iterative primal-dual algorithm is considered. The minimization is performed by a gradient descent method with proximal point (Rockafellar, 1976). In this context, the narrow band technique described in (Baeza et al., 2010) is incorporated to improve the performance of the algorithm, by reducing the range of possible depth values considered for each pixel at a given iteration. If this range is given by the interval $[a^l(x,y), b^l(x,y)]$, then the functional to be minimized is given by

$$F^l(\phi) = \int_\Omega \int_{a^l}^{b^l} \left\{ \left|\nabla_{x,y}\phi(x,y,s)\right| + \rho(x,y,s)\, \phi_s(x,y,s) \right\} dsdxdy. \qquad (4)$$

As the solution $\phi^l$ of (4) allows variations of the depth only inside the band given by $[a^l(x,y), b^l(x,y)]$, the band is recomputed after a given number of iterations.

Assume that an initial estimation $\phi^0$ is available (in practice we compute it with the plane sweep algorithm of Collins (Collins, 1996)). Then, if $\phi^l$ is the solution corresponding to iteration $l$, the update of $\phi^l$ is performed by taking the initial values $\phi_0 = \phi^l$ and $Z_0 = 0$ and iterating

$$\begin{aligned} Z_{m+1} &= Z_m + \tau_Z \nabla\phi_m, \\ \phi_{m+1} &= \phi_m + \tau_\phi \operatorname{div} Z_{m+1}. \end{aligned} \qquad (5)$$

In (5) $Z$ are the dual variables, characterized by

$$Z = (z_1, z_2, z_3) : z_1^2(x,y,s) + z_2^2(x,y,s) \le 1, | z_3(x,y,s) | \le \rho(x,y,s), \qquad (6)$$

and $\tau_\phi, \tau_Z$ are the advance steps corresponding to the proximal point algorithm, which are fixed to the value $\frac{1}{\sqrt{3}}$. The value for $\phi^{l+1}$ is obtained by binarization, with respect to a predefined threshold, of the solution of ). Appropriate values of $[a^{l+1}(x,y), b^{l+1}(x,y)]$ are computed and the process is repeated to obtain $\phi^{l+2}$, until convergence.

We observe that the essence of the algorithm is a finite difference iteration, where computations are done independently for every point in a discrete grid. Section 4.1.2 presents the results of running the iteration in parallel for all the points.

## *3.3 3D reconstruction*

If depthmaps for each camera in the system are available, a 3D reconstruction of the scene can be computed by depth merging. The idea is to find a surface that, when projected into the camera planes, best fits the observed depthmaps. Due to errors in the computation of the depthmaps and in the camera calibration, it is in general not possible to find a surface that matches the depthmaps with no error. Therefore, the approach is to define a cost function to be minimized, in the spirit of (Curless & Levoy, 1996) and (Zach et al., 2007). The cost considered in this work was described in (Baeza et al., 2011) and is composed by a data term –measuring the difference between the depth computed at the image pixels and the depth of the 3D surface, as observed by the corresponding camera— and a smoothness term defined through the total variation of the occupancy function that defines the surface. More precisely, if $u : \Omega \subset \mathbb{R}^3 \to [-1,1]$ represents the surface, in the sense that $u = -1$ means voxels outside the surfaces and $u = 1$ means voxels inside the surfaces, then the zero level set of $u$ is the sought surface. Consider the functional

$$J(u) = \int_\Omega |\nabla u(x)| dx + \frac{1}{2\theta} \sum_{i=1}^{N} \int_\Omega w_i(x)\left(u(x) - f_i(x)\right)^2 dx, \qquad (7)$$

where $\theta$ is a positive number, $N$ is the number of cameras, $f_i : \Omega \to [0,1]$ is a truncated distance function –so that $f_i(x) = 1$ for voxels visible from camera $i$, $f_i(x) = -1$ for occluded voxels and $-1 < f_i(x) < 1$ for voxels near the surface— and $w_i : \Omega \to [0,1]$ are weighting functions, that take the value $0$ for occluded voxels that are far from the surface and a value between $0$ and $1$ for the rest. More precisely, the functions $f_i$ are defined as follows (see Figure 4): if $x$ is a voxel center, whose distance to the $i-th$ camera plane is $r_i(x)$, let $p = \Pi_i(x)$ be the its projection in camera $i$, and let $d_i(p)$ be the observed depth of pixel $p$. Define:

$$f_i(x) = \begin{cases} 1 & \text{if } r_i < d_i - \delta/2, \\ \text{-1} & \text{if } r_i > d_i + \delta/2, \\ (2/\delta)(d_i - r_i) & \text{if } d_i - \delta/2 \le r_i \le d_i + \delta/2, \end{cases}$$

where $\delta$ is a positive constant that determines the *thickness* of the surface. The function $f_i$ assigns the value 1 to the voxels visible from camera $i$ and the value -1 to the occluded ones. A value varying linearly between -1 and 1 is assigned to the voxels that are within a distance $\delta$ of the observed depth.
On the other hand, the weights $w_i$ are defined by

$$w_i(x) = \begin{cases} 1 & \text{if } r_i > d_i + \eta, \\ 0 & \text{otherwise}, \end{cases}$$

where $\eta > \delta$ is a constant. The values assigned by $w_i$ to the voxels can be interpreted as the confidence on the values assigned by $f_i$. The zero level set of $u$, given by $\{x \in \Omega : u(x) = 0\}$, corresponds to the sought surface. Each term of the form

$$\int_\Omega w_i(x)\left(u(x) - f_i(x)\right)^2 dx$$

Measures the mean square error committed when taking a surface $u$, with respect to the observed values, for the $i-th$ camera. Finally, the term

$$\int_\Omega |\nabla u(x)| dx$$

is a regularization term that leads to smooth, visually pleasant surfaces.

Note that solving (7) is equivalent to solving

$$J(u) = \int_\Omega |\nabla u(x)|\, dx + \frac{1}{2\theta} \int_\Omega w(x)\left(u(x) - f(x)\right)^2 dx, \qquad (8)$$

where

$$w(x) = \sum_{i=1}^{N} w_i(x), \qquad f(x) = \frac{\sum_{i=1}^{N} w_i(x) f_i(x)}{w(x)}. \qquad (9)$$

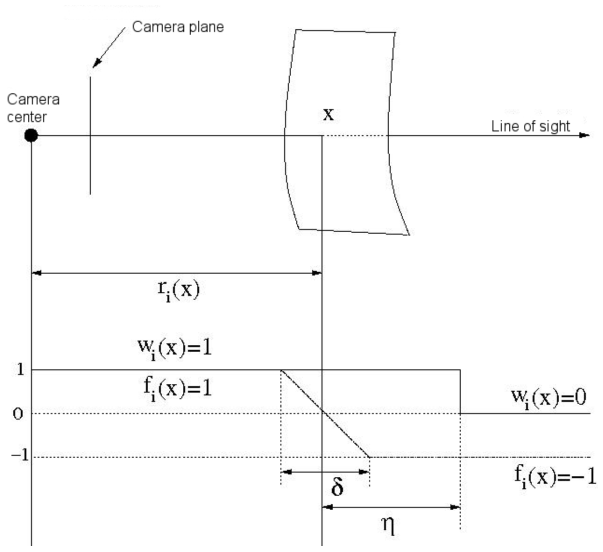


**Figure 4: Construction of weighted truncated distances from depths**

The minimization of (8) is done via a primal-dual formulation, in a fashion similar to the case of depth estimation. Given dual functions $p(x) = \{p_1(x), p_2(x), p_3(x)\}$ with $\| p \| \leq 1$, the problem to solve is

$$\max_{\|p\|\leq 1} \min_{u} \left\{ \int_{\Omega} u(x) \cdot \text{div} p(x)\, dx + \frac{1}{2\theta} \int_{\Omega} w(x) \left( u(x) - f(x) \right)^2 dx \right\}. \qquad (10)$$

The alternate solution with respect to $u$ and $p$ of the associated Euler-Lagrange equations, using the proximal point method leads to the iteration

$$\begin{aligned} p^{k+1}(x) &= \pi\left( p^k(x) - \tau_p \nabla u^k(x) \right), \\ u^{k+1}(x) &= \frac{\theta \tau_u}{\tau_u w(x) + \theta} \left( \frac{1}{\tau_u} u^k(x) + \frac{w(x)}{\theta} f(x) - \text{div} p^{k+1}(x) \right), \end{aligned} \qquad (11)$$

where $\pi(p) = \frac{p}{max\{1, \|p\|\}}$.

As in the previous section, we observe that the algorithm runs finite differences iterations that can be executed in parallel for all the points of the grid. Section 4.1.3 presents the results of such a parallelisation.

This section has presented the main three algorithms of the view synthesis process, whose parallelisation is evaluated in the following section.

## 4 Parallelization of view synthesis algorithms

Parallel algorithms, in contrast with sequential algorithms, are those pieces of code designed to be executed by more than one processing unit, operating concurrently on pieces of the input data. Today's computers are endowed with many-core processors and GPUs, which allow for the execution of parallel algorithms on low cost dedicated servers or even in desktop workstations. As the requirements for real-time or near real-time computations on image processing are becoming more and more demanding, the design of parallel algorithms becomes a must. In particular, GPUs are precisely designed for the parallel execution of operations on images and 3D volumes, making them particularly well suited for the implementation of parallel view synthesis algorithms. Further, multi-GPU machines are nowadays available at a very reasonable cost, in systems equipped with many-core processors. It is the goal of this Section to explore the feasibility of parallelizing view synthesis algorithms using GPUs and multi-GPU/multi core systems.

## *4.1 GPU view synthesis algorithms*

We will test the parallel implementation of the algorithms presented in Section 3 using the CUDA[2] (Compute Unified Device Architecture) language. CUDA is a set of tools developed by nVidia to allow for the implementation of programs in nVidia's GPUs. CUDA is an extension of the C programming language, though wrappers for other languages exist.

Programs written in CUDA are typically composed of subroutines, called kernels, which are executed in parallel in the GPU multiprocessors. This model uses a structure defined by a grid of thread blocks. A kernel call is supplied by parameters indicating the grid and block organization, with the property that a block is executed in a single GPU multiprocessor. The way how grid and blocks are defined can affect therefore the final performance.

Graphics cards supplied by nVidia are classified according to its compute capabilities. At the moment of writing this report the highest compute capability is tagged as 2.1, being the previous versions tagged as 1.0, 1.1, 1.2, 1.3 and 2.0. There are some limitations in the dimensions of the grid and blocks depending on the compute capability. Up to version 1.3, the grid has to be bi-dimensional, while the blocks can be 3D for all versions. Compute capabilities 2.x, started with the GT400 series in April 2010, allows for 3D grids of 3D blocks. On the other hand, the maximum number of threads per blocks, is limited to 512 (compute capability 1.x) or 1024 (compute capability 2.x).

For the grayscale conversion experiment presented in this Section we have used an nVidia GeForce GTX 285 graphics card, which has 1GB of main memory. The system used is composed by two GTX 285 graphics cards, on a 8-core (dual Quad Core) Intel Xeon E5520 @ 2.27 GHz processor, able to run 8 threads simultaneously, and 24 GB of RAM memory. The same system has been used in the multi-GPU experiments of Section 4.2.

For algorithms operating on images, where the algorithm computes a value or a set of values for each pixel (e.g. disparity, depth or optical flow) typically CUDA algorithms are based on the definition of a 2D block set, of dimension $B \times B$ so that $B^2 \leq 512$, being a common choice $B = 16$. 3D blocks can of course be used, but the limitation of having a 2D grid (up to compute capability 1.3) implies a computational overhead for handling of the relationships between 3D blocks and 2D pixels. We will adopt this approach only when the characteristics of the algorithm induces a penalization for using 2D blocks, as is the case of the narrow band algorithm described in Sections 3.2 and 4.1.2.

A typical structure for a CUDA program doing some per-pixel computation is compared to its sequential counterpart in Figure 5. A basic fact about GPU computing is that the variables operated by the algorithm have to be uploaded to GPU memory. This induces a computational cost that is better compensated as the computational cost of the overall algorithm increases. For some simple algorithms, the memory transfer cost can be higher than the cost of the algorithm itself, making inefficient the use of CUDA for those purposes. On the other hand, the amount of memory available in the GPU is limited when compared to the RAM memory.

If a program is running over a video sequence, processing frames sequentially, some of the operations described above can be executed concurrently, in a way such that processing of a frame can be started before the computations corresponding to the previous frame have ended, provided that those operations are independent. In particular the following possibilities are worth to be mentioned:

[2] http://developer.nvidia.com/category/zone/cuda-zone

- When a GPU kernel is launched, the program control returns to the CPU immediately. CPU and GPU computations can be thus run in parallel. In particular, disk operations and network data transfers can be made while the GPU is computing. In the FINE 3D video server architecture (Latour et al., 2011a) most data transfers are carried through the video server network. Details of the API to store and delivery into video server platform are presented in deliverable (Latour et al. 2011b).
- Data transfers between the GPU and the CPU can be done while the GPU is executing a kernel. Also, the PCI express bus allows for transferring data from the CPU to the GPU and from the GPU to the CPU simultaneously. When processing a video sequence, the code to be executed by the GPU for each frame can be overlapped with the transfer to the GPU of the data corresponding to the next frame.
- CUDA can be used in conjunction with parallel CPU programming interfaces, like OpenMP[3] and Message Passing Interface[4] (MPI), and with CPU threading. This allows for a second level of parallelism, where different CPU cores and graphics cards are executing different programs in parallel.

All those asynchronous operations can be controlled through barriers wherever needed.

As an illustration, consider an algorithm for converting an image, initially stored in a file, into gray scale. CUDA and CPU codes that perform this operation are presented in Appendix 1 and Appendix 2, respectively. The CImg library[5] is used to read and write the images. The codes have the structure presented in Figure 5. Note that this is a very unfavourable case for the GPU code, because of the small amount of operations to be done for each pixel (3 products and 2 sums). Measurements of the execution times of the different parts of the algorithms, applied to a FullHD colour image with floating point values give the results of Table 1, which clearly shows that, in this case, the input/output operations and data transfers between CPU and GPU represent almost all the execution time (99.83%).

Avoiding disk read and write operations, the memory movement between CPU and GPU represents a 97.32% of the execution time, for a 2.68% of the kernel execution (see Table 3). Note that, if FullHD images are to be handled in real time, at a frame rate of 25 fps, then each image has to be processed in at most 40 ms. The memory handling for our example represents a 30% of this time.

Compared to the sequential algorithm, the results in Table 1 to Table 4 indicate that the CUDA algorithm is not faster than the CPU version. Note that if we do not consider the I/O operations and the memory overload, the CUDA kernel is almost 40 times faster than its CPU counterpart. This highlights the importance of overlapping data transfers with GPU computation. On the other hand, the CUDA kernel performs a conditional check –necessary to verify that the thread coordinates represent a valid pixel— that is not necessary in the CPU code.

[3] http://openmp.org/wp/
[4] http://www.mpi-forum.org/
[5] Available at http://cimg.sourceforge.net/

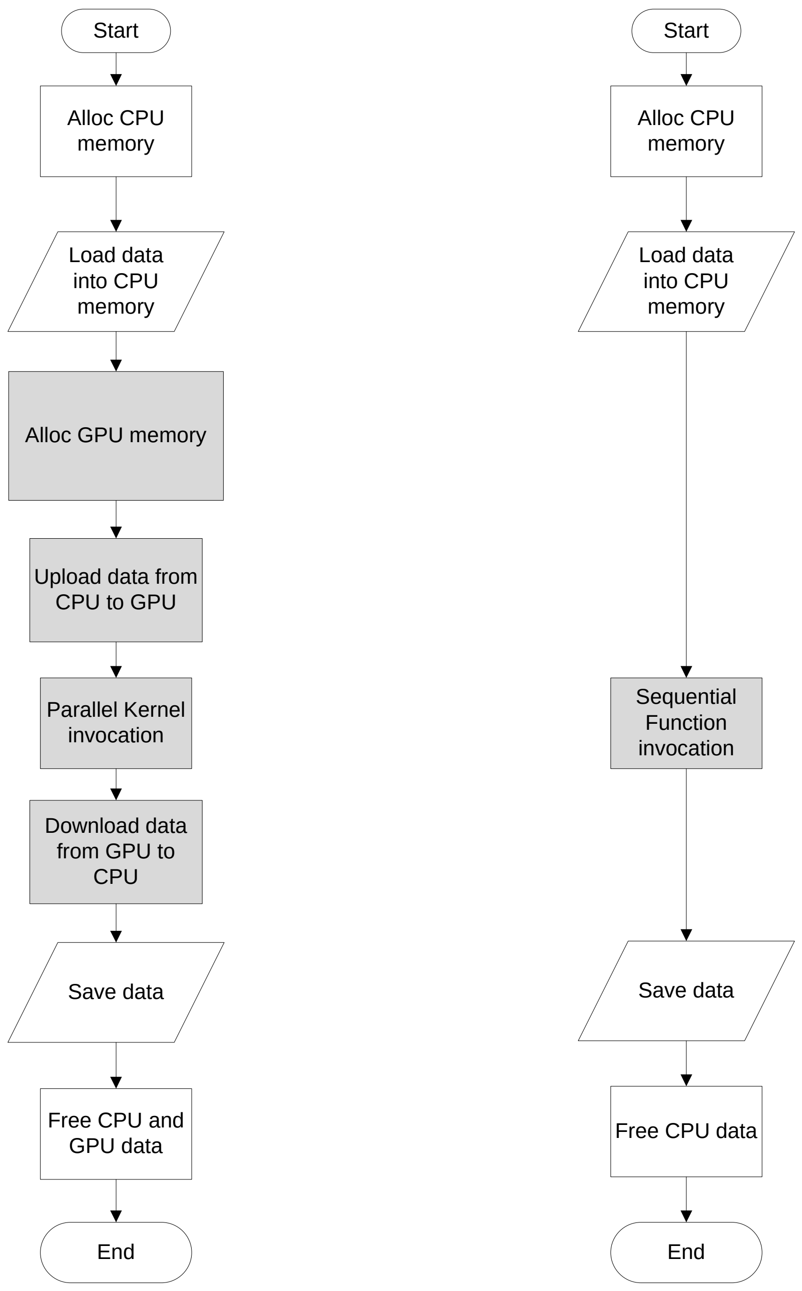

Start
Alloc CPU memory
Load data into CPU memory
Alloc GPU memory
Upload data from CPU to GPU
Parallel Kernel invocation
Download data from GPU to CPU
Save data
Free CPU and GPU data
End
Start
Alloc CPU memory
Load data into CPU memory
Sequential Function invocation
Save data
Free CPU data
End

**Figure 5: General structure of a simple CUDA (left) and sequential (right) program.**

| Process | Execution time (ms) | Percentage of time |
|---|---|---|
| Read image from disk to CPU | 105 (FullHD colour tiff image) | 53.21% |
| Upload data from CPU to GPU | 6.6 (24300 Kb) | 3.34% |
| Parallel kernel execution | 0.33 | 0.17% |
| Download data from GPU to CPU | 5.4 (8100 Kb) | 2.74% |
| Save image to disk | 80 (FullHD grayscale tiff image) | 40.54% |
| Total execution time | 197.33 | 100.00% |

**Table 1: Execution times for a sample GPU algorithm**

| Process | Execution time (ms) | Percentage of time |
|---|---|---|
| Read image from disk to CPU | 105 (FullHD colour tiff image) | 53.03% |
| Sequential function execution | 13 | 6.57% |
| Save image to disk | 80 (FullHD grayscale tiff image) | 40.40% |
| Total execution time | 198 | 100.00% |

**Table 2: Execution times for a sample CPU algorithm**

| Process | Execution time (ms) | Percentage of time |
|---|---|---|
| Upload data from CPU to GPU | 6.6 (24300 Kb) | 53.53% |
| Parallel kernel execution | 0.33 | 2.68% |
| Download data from GPU to CPU | 5.4 (8100 Kb) | 43.79% |
| Total execution time | 12.33 | 100.00% |

**Table 3: Execution times for a sample GPU algorithm, avoiding I/O operations**

| Process | Execution time (ms) | Percentage of time |
|---|---|---|
| Sequential function execution | 13 | 100.00% |
| Total execution time | 13 | 100.00% |

**Table 4: Execution times for a sample CPU algorithm, avoiding I/O operations**

### 4.1.1 Background subtraction / segmentation

The implementation of the Background Subtraction method of OpenCV obtains good results (see Figure 2) but at the price of a high the computational cost. Processing FullHD images is very expensive. This algorithm is a CPU implementation, and on a computer with a processor Intel i7 processor needs 200ms per frame, which is far too large to process images at 25 fps (40 ms per frame).

Some authors have tried to improve the performance by modifying the original method. For example, Zivkovic and van der Heijden (Zivkovic, 2004), (Zivkovic & van der Heijden, 2006) proposed an approach for BGS in which a mixture of Gaussians for the underlying distribution for each pixel's colour values is maintained. For each new frame, the mean and covariance of each component in the mixture is updated to reflect the change (if any) of the pixel values. If the new value is far enough from the mixture (the Mahalanobis distance from the RGB value to the component's means are larger than, for instance, three times of the standard deviation), the pixel will be considered as the foreground. Beyond this basic version, Zivkovic and van der Heijden used some additional control parameters and modified the model updating equations to not only decrease the processing time for each frame, but also gain better segmentation.

In 2010, a new version of the previous algorithm was implemented in GPU using CUDA (Pham et al., 2010). Each pixel is processed independently from each other; this makes GMM an embarrassingly parallel algorithm, which can be parallelized by devoting a thread for each pixel. In addition to launching *n* threads in parallel, this implementation exploits a three-level optimization:

- Pinned memory: It makes memory transfers concurrently and helps to increase bandwidth between host and device memory (NVIDIA, 2010).
- Memory coalescing: Input image is converted into four channels and internal elements are inflated to 4 bytes for each element. With this configuration CUDA can access faster to aligned continuous region in global memory (NVIDIA, 2010).
- Asynchronous execution: Using CUDA streams and pinned memory this method is able to interleave CPU code execution with kernel launches and memory transfers as described in Figure 6.

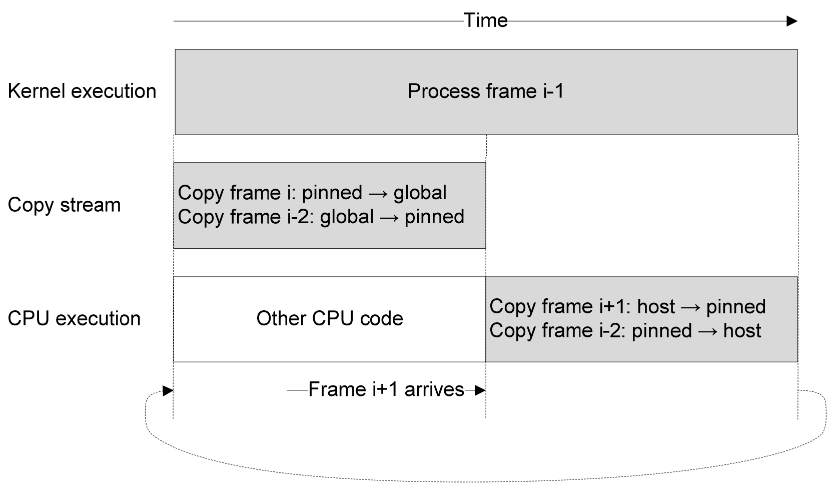

Time
Kernel execution
Process frame i-1
Copy stream
Copy frame i: pinned → global
Copy frame i-2: global → pinned
CPU execution
Other CPU code
Copy frame i+1: host → pinned
Copy frame i-2: pinned → host
Frame i+1 arrives

**Figure 6: Timeline from asynchronous execution (Pham et al., 2010)**

In order to test the background subtraction method, in this Section a FullHD resolution image sequence corresponding to a live football match is used. An example of images is shown in Figure 7 with a sample result. In terms of accuracy, the results are almost the same as those obtained with the CPU implementation (see Figure 2) but with a considerably smaller computation cost.

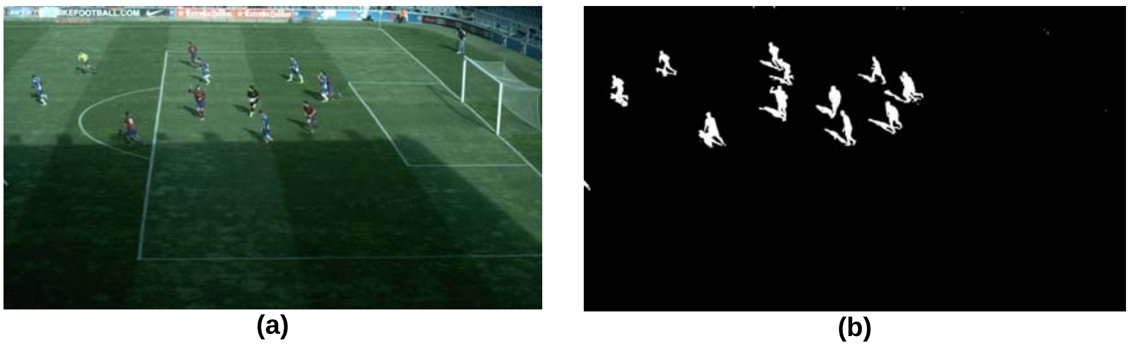


**Figure 7: Example of Background subtraction with CUDA: a) Input Image; and b) Foreground image.**

With the asynchronous execution of the GPU implementation, as shown in the Figure 6, it is difficult to measure the computational cost of each of involved parts. Therefore to measure the computational cost we calculate the average time of the whole algorithm applied to an image in a sequence of 500 images. In order to see how it affected the execution time with different computers, we tested the algorithm on different hardware configurations. Table 5 shows results obtained in 4 different hardware configurations.

| | Hardware configuration | GPU time (ms) | CPU time (ms) | Speedup |
|---|---|---|---|---|
| 1 | Intel Core2 Duo E6400 / nVidia GeForce 9800 GT | 24.3 | 265.1 | 10.90 |
| 2 | Intel Core2 Quad Q6600 / nVidia GeForce 9800 GT | 15.8 | 169.8 | 10.74 |
| 3 | Intel Core i7-920 / nVidia GeForce GTX 260 | 8.5 | 115.9 | 13.63 |
| 4 | Intel Core i7-920 / nVidia Quadro 5800 | 10.8 | 132.9 | 12.30 |
| 5 | Intel Dual Xeon Quad E5620 / nVidia Quadro 6000 | 11.6 | 121.0 | 10.43 |

**Table 5: Execution times and performance comparison of the GPU and CPU algorithms for Background Subtraction.**

These experiments show that our GPU implementation provides an average speedup of about 12x with respect to the CPU version. In most of the tests, the frame rate is always higher than 50 frames per seconds (FPS) on FullHD video format.

### 4.1.2 Depth Computation

Summarizing the algorithm in Section 3.2, the actions required for the estimation of the depth in an image $I_{ref}$ are the following:

- Load images, segmentation masks and other data from disk
- Compute homographies between camera pairs and between cameras and depth planes
- Compute the initial estimation $\phi^0$ using plane sweep
- For l = 1 ... L-1
    - Compute the cost $\rho(x,y,s)$ for s belonging to the current band
    - Perform M iterations of algorithm (5) by doing the following steps:
        - Perform an iteration of the dual variables for all pixels (first equation in (5))
        - Project the dual variables into the feasible set (i.e., ensure (6))
        - Perform an iteration of the primal variables for all pixels (second equation in (5))
        - Project the primal variables into the feasible set (i.e., ensure (3))
    - Binarize the solution $\phi_M$ to obtain $\phi^{l+1}$
    - Recompute the narrow band
- Compute the final depth estimation $u$ from $\phi^L$
- Save the computed approximation to disk

The flow diagram for the algorithm is shown in Figure 8. For testing purposes the algorithm has been implemented as a set of CUDA kernels corresponding to the processes in gray, while the processes in white correspond to input/output operations, data transfers and procedures implemented as sequential CPU code. The computation of the binarisation of $\phi_M$ and the "cut" $u = cut(\phi^L)$ involve the choice of the optimal cut based on the computation of the energy for several possible cuts. The energy computation requires a sum over all pixels of the individual pixel-wise energy values. Although it can be implemented as a parallel reduction operation, it is a sequential process in our implementation. These processes appear in dashed gray in the flowchart.

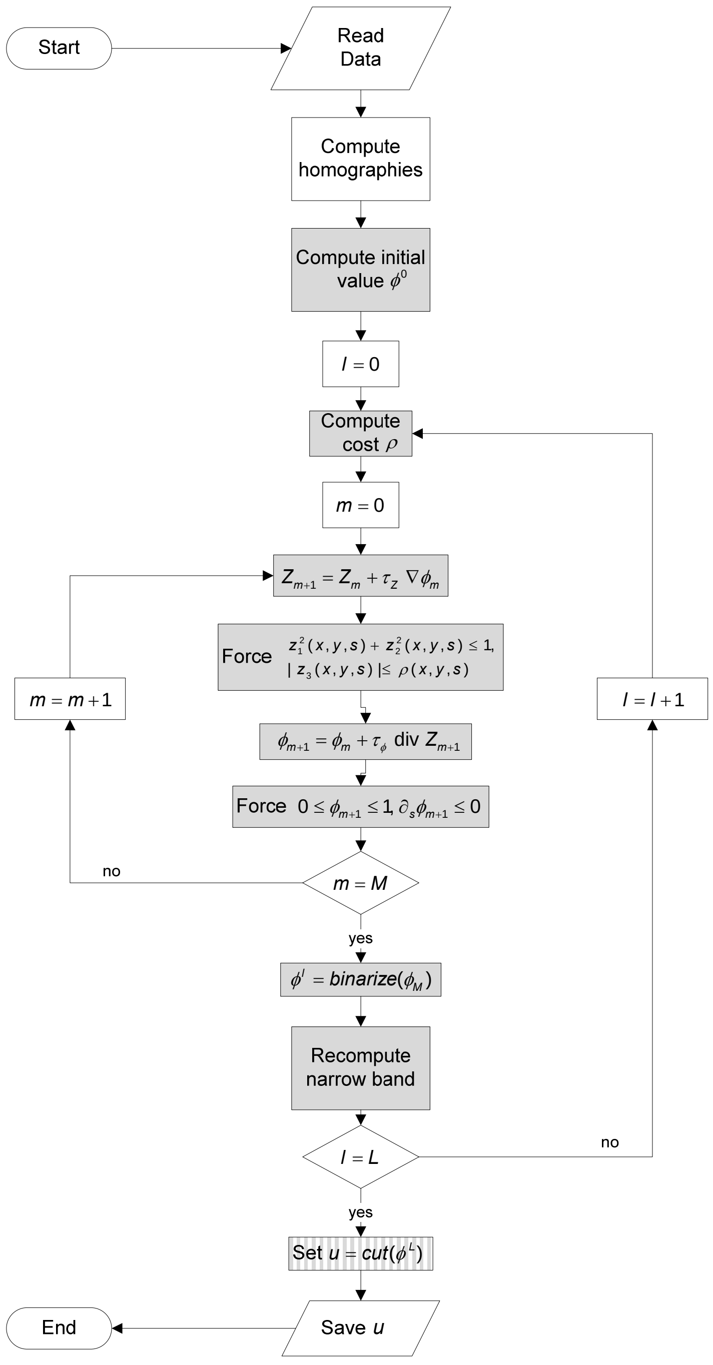
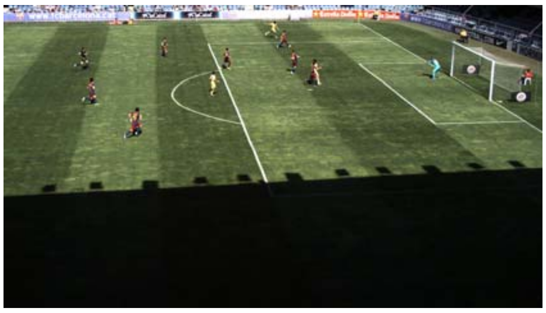
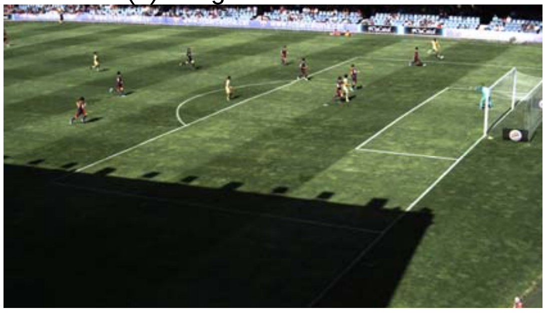

Start
Read Data
Compute homographies
Compute initial value $\phi^0$
$l = 0$
Compute cost $\rho$
$m = 0$
$Z_{m+1} = Z_m + \tau_Z \nabla \phi_m$
Force $z_1^2(x,y,s) + z_2^2(x,y,s) \leq 1$, $|z_3(x,y,s)| \leq \rho(x,y,s)$
$\phi_{m+1} = \phi_m + \tau_\phi \text{ div } Z_{m+1}$
Force $0 \leq \phi_{m+1} \leq 1, \partial_s \phi_{m+1} \leq 0$
$m = m+1$
no
$m = M$
yes
$\phi^l = binarize(\phi_M)$
Recompute narrow band
$l = l+1$
no
$l = L$
yes
Set $u = cut(\phi^L)$
Save $u$
End

**Figure 8: Flowchart for the depth computation algorithm**

We have tested the CUDA implementation of the algorithm using FullHD colour images captured in a live football match. The results in this Section correspond to the computation of the depth of a FullHD image which is part of a dataset composed by 7 images. This dataset corresponds to a time instant of a set of video sequences that was recorded in May, 2011 in Barcelona during a live football match, using seven cameras placed along a corner of the pitch. Sample images are shown in Figure 6, and a sample result is shown in Figure 10.


(a) Image from camera 1

(b) Image from camera 2


(c) Image from camera 5

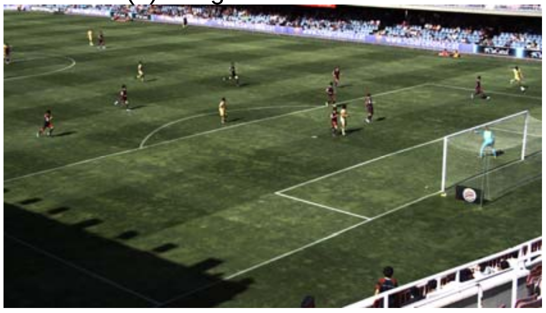
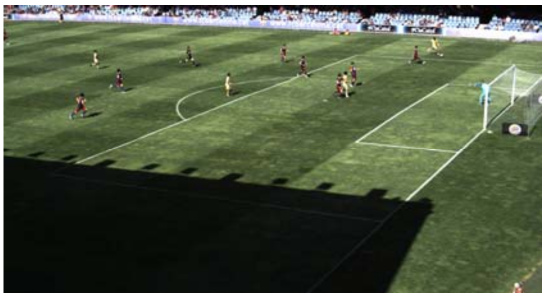
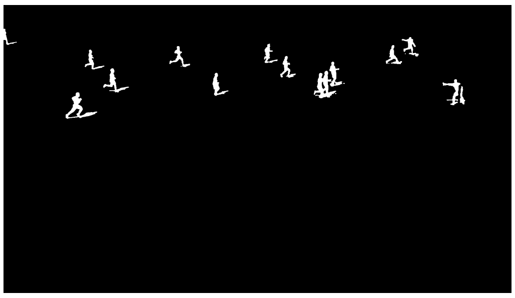

(d) Image from camera 7

**Figure 9: Four input images for a 7-image dataset**


(a) Input image


(b) segmentation mask

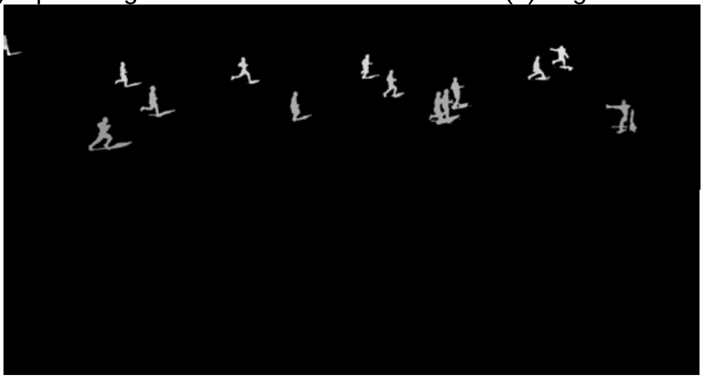

(c) Depthmap

**Figure 10: Sample depthmap (c) computed for the input image (a) on the segmentation mask (b). Darker gray values correspond to smaller depths.**

The GPU used for the experiments in this section is an nVidia GeForce GTX 280 card, with 1GB of main memory. The CPU is an Intel Core2 Quad Q9300 @ 2.50 GHz. processor, and the system has 3 GB of RAM memory.

Before analyzing the performance of the implementation, let us remark that the implementation of the algorithm that is used for the tests is not an optimized implementation, but one designed for testing. The goal is to test how the cost of the different parts of the algorithm compares to a CPU implementation, so as to give an insight of the potential capabilities of a fully optimized production code. In particular, the testing code has the following limitations:

- A part of the code (initialization, binarization, etc.) is not CUDA code, but CPU code.
- The kernels are launched for all pixels in the image, regardless if they belong to the segmentation mask or not. This causes a lot of overhead since most kernel blocks contain no pixels inside the segmentation mask.
- The discretization in the label space is done so as to cover the whole 3D space defined by the initial bounding box. A discretization with a much smaller number of labels could be used if an approximate localization of the players is known in advance.
- No special memory optimizations have been introduced.

Table 6 shows execution times for a run of the algorithm, which produces the output shown in Figure 10. The setup of the execution is as follows: a bounding box of $40 \times 40 \times 8$ m$^3$, defined by the 3D points (21.80, -15.00, -0.5) and (61.80, 25.00, 7.5) determines the part of the pitch to be processed. The origin of the 3D space is located in the centre of the pitch and the coordinates are given in meters. The bounding box is discretized using 81 depth values, distributed uniformly between the minimum and maximum depth of the bounding box as seen by camera 5, which corresponds to the depth range [27.66, 82.66]. This gives a resolution of 0.68 meters.

For testing purposes the values $M = 30$ and $L = 10$ have been fixed, so that 300 primal-dual iterations are performed overall. In practice, the number of iterations is not fixed, and a convergence check is performed periodically to decide when to stop, but for the objectives of this work this test has been avoided. The indicated number of iterations is given for guidance and depends on the problem. The size of the narrow band has been fixed to 3, which means that at least 7 values are checked for each pixel. More values are checked if neighbouring pixels present depth jumps, so as to avoid pixels to get stuck in local minima.

| | **Process** | **Time (ms)** |
|---|---|---|
| 1 | Read data from disk | 960 |
| 2 | Initialization (homography computation, initial estimation (mostly CPU code) | 1053.4 |
| 3 | Allocate GPU memory and CPU to GPU data transfer | 122.48 |
| 4 | Cost update | 73.3 |
| 5 | Primal iteration (per iteration) | 16.94 |
| 6 | Dual iteration (per iteration) | 32.82 |
| 7 | Binarization and narrow band recomputation | 125.46 |
| 8 | Computation of $u$ from $\phi^L$ (incl. GPU to CPU data transfer) | 30 |
| 10 | Save data to disk | 320 |
| | Total time (10 outer iterations, 30 inner iterations) | 19401.48 |

**Table 6: Execution times for the sample depth computation code**

In this example, 19.4 seconds are required to compute the depth of an image, including all processes. Avoiding input/output operations, memory transfers and initialization, which can be overlapped with the GPU code of the previous frame, the total time is reduced to about 17 second per frame. Table 7 shows a breakdown of the computational time corresponding to the main operations involved in the algorithm, namely the inner primal-dual iterations and the outer loop where the narrow band is recomputed.

| | Process | Time (ms) Per execution | Time (ms) Per inner iteration | Time (ms) Per outer iteration. (30 inner it.) | Total time (ms) (10 outer it.) |
|---|---|---|---|---|---|
| | Inner primal-dual iteration | 49.76 | 49.76 | 1492.8 | 14928.0 |
| | Outer iteration overhead | 198.76 | - | 198.76 | 1987.6 |
| | **Total** | **-** | **49.76** | **1691.6** | **16916** |

**Table 7: Execution times for the internal and external iterations in depth computation. The outer iteration overhead is the sums of the costs for the cost function update, binarization and narrow band recomputation.**

In order to give more insight on the performance of this implementation and its potential improvements, a comparison of the timings of the primal-dual part with a CPU version of the same algorithm is shown in Table 8 for the same example indicated above.

| | Process | GPU time (ms) | CPU time (ms) | Speedup |
|---|---|---|---|---|
| 1 | Primal iteration | 16.94 | 30.41 | 1.80 |
| 2 | Dual iteration | 32.82 | 85.67 | 2.61 |
| 3 | Primal-dual iteration | 49.76 | 116.08 | 2.33 |
| 4 | 300 Primal-dual iterations | 14928 | 34828 | 2.33 |

**Table 8: Execution times for the primal-dual part of the depth computation algorithm and comparison with a CPU code.**

The speedup obtained by the GPU algorithm is rather poor in this case. Reasons for this behaviour are diverse. Among the most evident are the following:

- *Divergence*. The fact that different codes have to be executed for pixels inside and outside the segmentation mask causes the code to fall into massive divergence. Divergence happens wherever different threads in the same warp execute different code, typically when a conditional statement appears and not all threads in the warp take the same path. In this situation, the code corresponding to the different paths is executed sequentially (all threads execute the code corresponding to all paths).
- *No use of shared memory, non-optimal use of device memory overall*.
- *Execution of kernels corresponding to zones of the image where no pixels are in the segmentation mask*. In the launch of the various kernels implemented, the image is divided in blocks, which correspond to CUDA blocks, and the kernel is launched for all blocks. In the example used in this Section, only a 1.16% of the pixels in the image are inside the segmentation mask. Translated into blocks, there is a 3.03% of blocks that contain pixels in the segmentation mask (for blocks with size $16 \times 16$). The rest of the blocks are launched and no operations are done on them, other than checking that all pixels are outside the mask. This represents a small GPU cost per block, but since the number of blocks for which this happens is high, it can represent a significant loss of performance. Furthermore, a bad load balancing between Streaming Multiprocessors (SMs) is produced. To study this effect, the depth estimation code acting on several segmentation masks with variable number of pixels, artificially created for test purposes, has been run. Figure 11 displays the execution times of a primal-dual iteration with respect to the percentage of blocks that contain pixels in the segmentation mask. It can

be clearly observed that, as the percentage of blocks with segmented pixels increase, the performance improves, because of a more balanced workload between SMs and because of the reduction of the overhead caused by the launch of useless blocks. When almost all the image is segmented, the performance degrades a little bit probably because the memory bandwidth is too small for the amount of data to be transferred. As a note, the per-iteration cost of launching the code on an empty segmentation mask is approximately 1.6 ms.

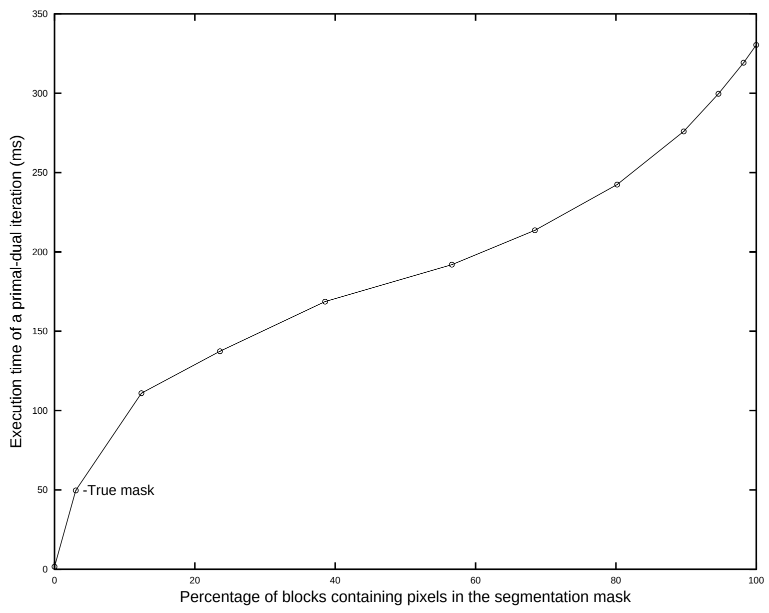


**Figure 11: Execution times of a primal-dual iteration for various segmentation masks**

A helpful information for the depth estimation problem is the approximate position of the players' feet and the referee in the pitch. It is the goal of WP4T4 to improve Tracab's real time tracking system[6], which can provide the position of the players on the 2D groundplane for every frame. From this data, a small bounding box can be placed around each player. If the approximate player position in the 2D pitch is given by coordinates $(P_x^k, P_y^k)$ we define its corresponding bounding box $B^k$ as the box defined by the points $(P_x^k - L, P_y^k - L, -1)$ and $(P_x^k + L, P_y^k + L, 3)$. The value $L = 2$ has been taken in the tests presented in this Section.

A process to merge overlapping bounding boxes is applied to ensure that each bounding box, which now may be fitted to one or more players, can be processed independently from each other. The idea is to define a certain number of bounding boxes that contain the players in the region of interest, and that can be processed independently from each other.

The approach considered in this work consists on the following steps:

1. Identify the position of each player in the pitch.
2. Define a bounding box around the given players. A box can contain one or more players.
3. Merge into bigger boxes the ones that overlap, so that non-interacting problems can be defined on each box.
4. Solve the sub-problems corresponding to each box. The ordinary depth estimation algorithm is applied to each bounding box separately

[6] http://www.tracab.com

5. Merge the partial results of each box into a global solution for the initial bounding box. All the computed depths are merged into a single depthmap.

This approach immediately produces two benefits:

- A reduction on the number of labels required to compute the depth of the player with a given accuracy. The ordinary way for discretizing the depth space is to define a global bounding box covering the entire 3D region seen by the camera setup. If smaller bounding boxes can be placed around the players' feet position, the same resolution can be obtained for each player separately, with an enormous reduction of the problem dimensionality.
- A reduction in the size of the images to be processed. For each player, only the image crop corresponding to the projection of its bounding box in the image plane needs to be processed. As each player or group of players can be processed independently from each other, the data transfers can be
    - split between multiple GPUs if the system has more than one GPU,
    - overlapped with the computation of the depth of another player, if the system has a single GPU.

  Further, a part of the workload (for example the process for one of the players) can be done by the CPU concurrently with the GPU.

In the experiment used throughout this Section, the global bounding box includes the 12 persons (11 players plus the referee) visible in the image of Figure 9 (a). The global bounding box is shown in Figure 12(a), depicted around a 3D reconstruction of the players, computed as explained in Sections 3.3 and 4.1.3.

After the application of the process described above, 10 small boxes are placed around the players and the referee. Nine of them contain a single person, and one of them contains three, as can be seen in Figure 12(b). A top view of the same 3D scene is depicted in Figure 12 (c). In this example the definition of the bounding boxes is conservative, in the sense that they are relatively big with respect of the size of the players. The image crops that result from the projections of one of the bounding boxes considered into the image planes of four of the cameras are shown in Figure 13.

For simplicity a discretization in 11 depth labels has been used in this experiment for all boxes.

Comparing the amount of data processed with the experiment where the global bounding box is considered, the overall number of pixels processed represents a 31.07% of the global case (a data reduction factor of 3.22). The maximum number of labels has been reduced by a factor of 7.3 (from 81 to 11) and in practice, the average (among all pixels in all bounding boxes) number of labels considered in the second case is a 22.68% of the average number of labels considered in the first case, which represents a reduction factor of 4.4. Combining these two factors, the total amount of data to be processed has been reduced in a factor of 14.2.

From the processed pixels, a 5.51% are pixels that fall inside the segmentation mask. The overall percentage of blocks that contain pixels in the segmentation mask is now the 14.04%. This represents also a reduction on the number of blocks unnecessarily launched.

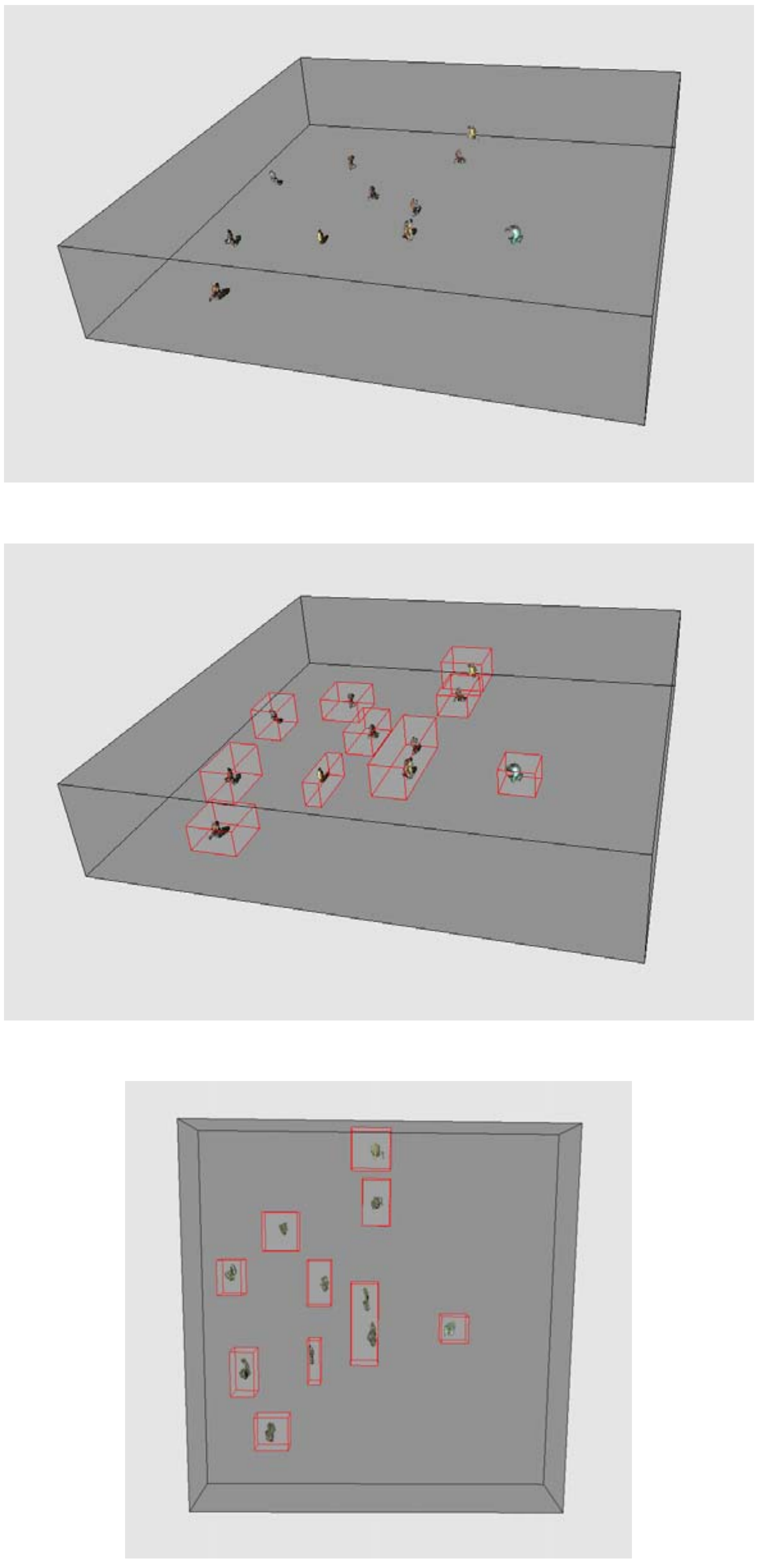


**Figure 12: Illustration of the global bounding box (top). Small boxes adapted to the players (middle). Top view of the middle figure (bottom).**

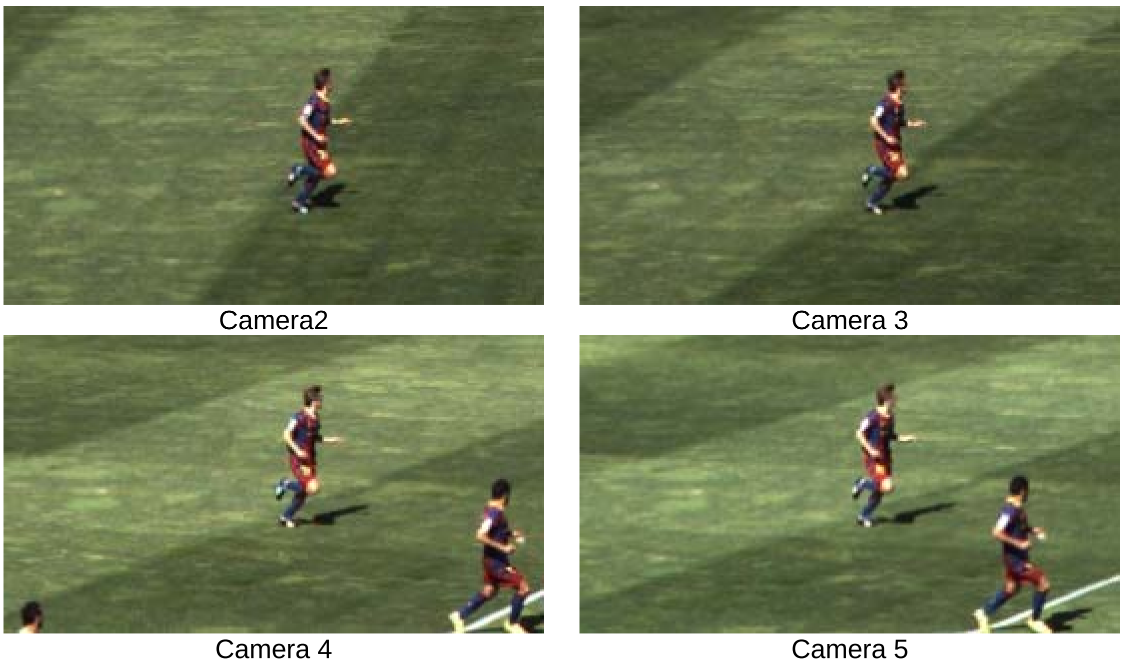


**Figure 13: Cropped images corresponding to the projection of a bounding box on the image planes of four cameras.**

The running times for the main parts of the GPU depth estimation algorithm applied to the small boxes are shown in Table 9. Compared with the case where a single bounding box is used (Table 7) a speedup of 4.61 is obtained. Even if the data has been reduced by a factor of 14.2, the resulting smaller speedup is explained by the following facts:

- Each bounding box needs its own kernel launches and data transfers, with the result of an increased latency. A possible solution to mitigate this would be again to pack the data into bigger sets and process them in single kernel launches, maintaining data structures that manage the relationship between data and bounding boxes, so as to reduce latencies in the data transfers and kernel launches.
- As the amount of data to be processed per-bounding box is small, so is the number of blocks per SM, which ultimately results in a bad load balancing between SMs.
- Finally, as the number of unnecessary block launches has been reduced (by a factor of 4.63), and as the projections of different bounding boxes on the image planes can overlap, the average per-block cost increases, because more blocks contain pixels where the actual computation has to be performed. More precisely, the total number of CUDA blocks that contain segmented pixels is now a 51.42% higher than in the global case (374 against 247 blocks).

| | **Process** | **GPU time (ms)** |
|---|---|---|
| | Primal-dual iteration | 9.95 |
| | 30 primal-dual iterations | 298.52 |
| | Outer iteration overhead | 68.14 |
| | Overall process (30 inner iterations, 10 outer iterations) | 3666.4 |

**Table 9: Execution times for the depth estimation problem when using adapted boxes**

### 4.1.3 3D reconstruction from depths

Figure 14 shows a flowchart for the 3D reconstruction algorithm described in Section 3.3. A CUDA implementation has been developed to test its efficiency compared to a CPU code implementing the same algorithm. For this test we have used the dataset described in Section 4.1.2.

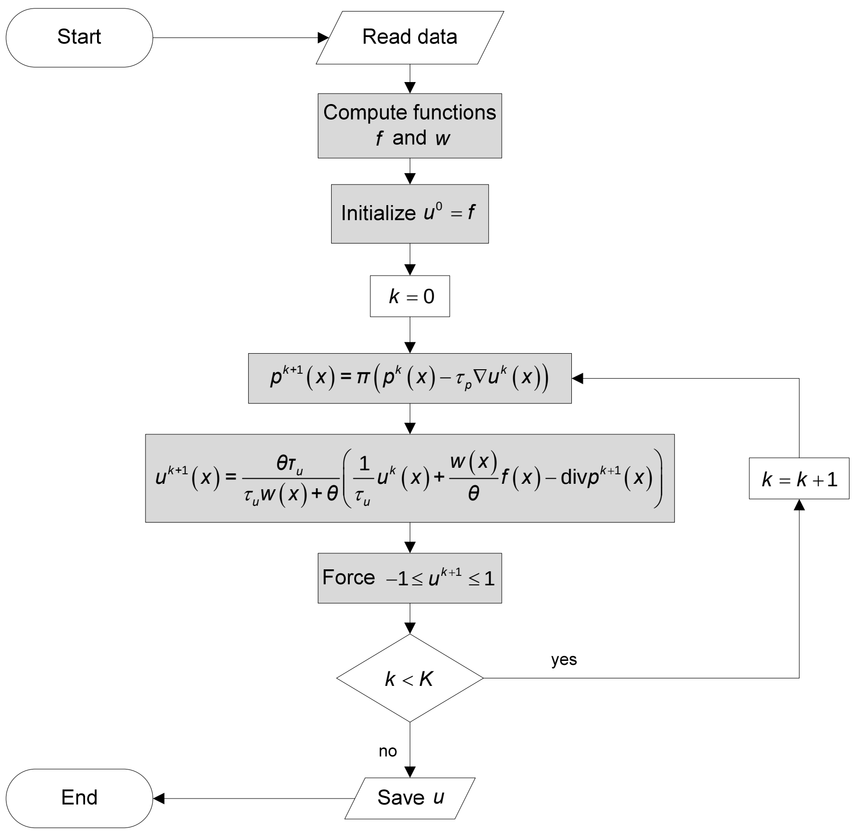
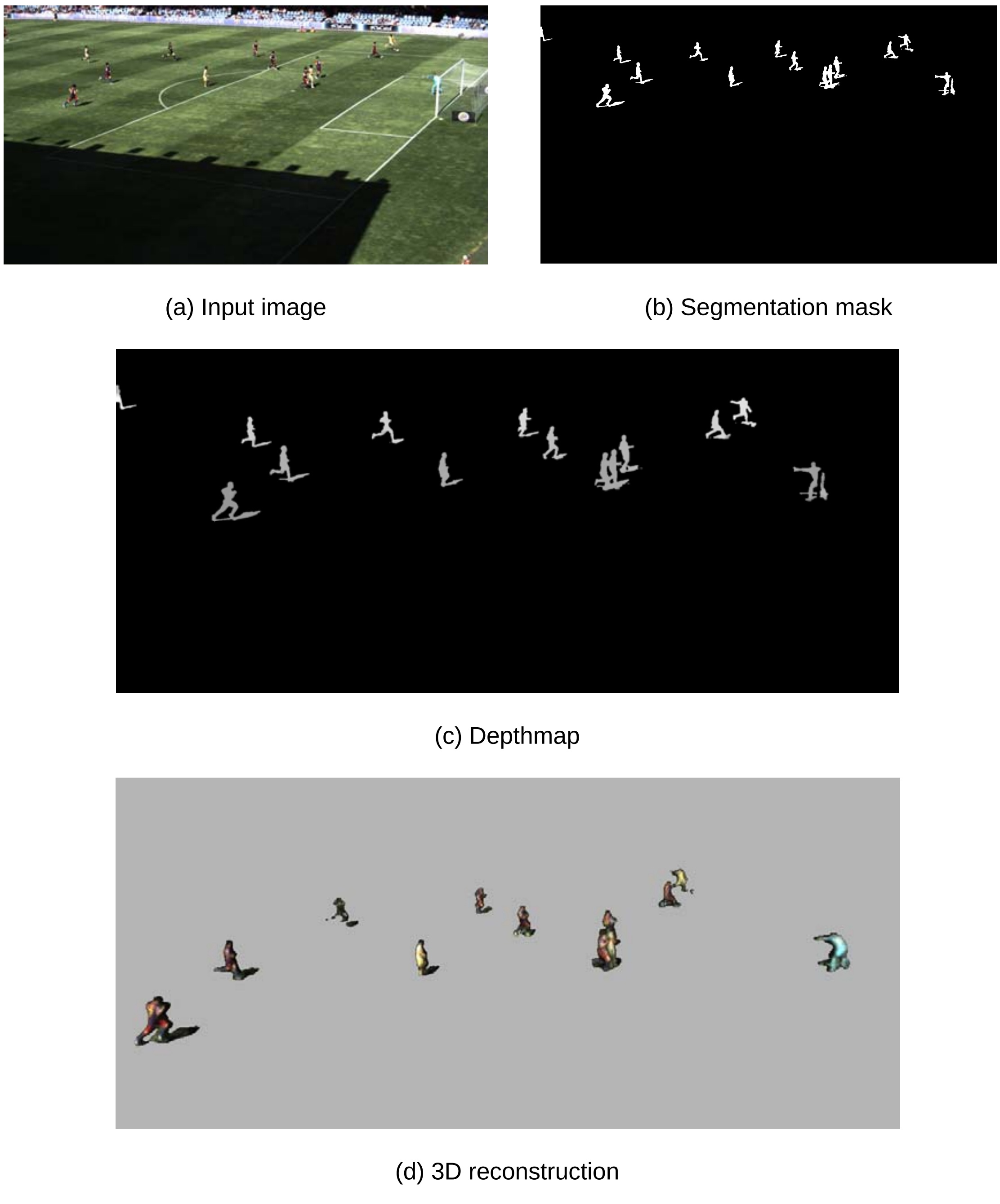

Start
Read data
Compute functions $f$ and $w$
Initialize $u^0 = f$
$k = 0$
$p^{k+1}(x) = \pi\left(p^k(x) - \tau_p \nabla u^k(x)\right)$
$u^{k+1}(x) = \frac{\theta\tau_u}{\tau_u w(x) + \theta}\left(\frac{1}{\tau_u}u^k(x) + \frac{w(x)}{\theta}f(x) - \operatorname{div}p^{k+1}(x)\right)$
Force $-1 \le u^{k+1} \le 1$
$k < K$
yes
$k = k + 1$
no
Save $u$
End

**Figure 14: flowchart for the 3D reconstruction algorithm.**

Using as input the camera calibrations and depthmaps computed as described in Sections 3.2 and 4.1.2 for the seven images, we compute a 3D reconstruction with the algorithm described in Section 3.3, using 20 iterations of the primal-dual solver. The steps depicted in gray in Figure 14 correspond to CUDA kernels. We have used three different discretizations for the 3D space, using grids of $100 \times 100 \times 20$, $200 \times 200 \times 40$ and $400 \times 400 \times 80$ voxels. These discretizations are defined so that the discretization steps are similar in all three directions. A sample result corresponding to a discretization of $400 \times 400 \times 80$ voxels is shown in Figure 15. The input images, segmentation masks and depthmaps shown are the same as in Figure 10. we reproduce them here for commodity.

The performance of the CUDA implementation has been analyzed by measuring the execution times of each operation in the algorithm, and compared them with the execution times of an equivalent sequential algorithm. The system used is the same as in Section 4.1.2. Table 10 summarizes the obtained results for several sizes of the 3D discretization. It can be clearly observed that the memory transfers between CPU and GPU are severely penalizing the algorithm performance. If the memory transfers are ignored, then the performance raises to reasonable values. Note that ignoring memory transfers makes sense in this case, because the depthmaps, that were computed with the CUDA code described in Section 3.2 are already hosted in GPU memory at the end of the execution, and do not need to be uploaded from RAM memory to GPU memory. In the worst case, data transfers can be overlapped with GPU computation of the 3D reconstruction of the previous frame. In the example shown in this Section, for voxel sizes bigger than $200 \times 200 \times 40$, the data transfer cost is smaller than the cost of the 3D reconstruction, and therefore the data transfer for a frame can be completed before the computation of the 3d reconstruction for that frame starts. In any case, the measures of the computational cost of the memory transfers have been computed for completeness of the efficiency studies.


(a) Input image

(b) Segmentation mask

(c) Depthmap

(d) 3D reconstruction

**Figure 15: Sample 3D reconstruction computed from depths**

| 3D Grid dimension (voxels) | GPU time (s) | GPU time (without memory transfers) (s) | CPU time (s) | Speedup (with memory transfers) | Speedup (without memory transfers) |
|---|---|---|---|---|---|
| $100 \times 100 \times 20$ | 0.11807 | 0.02177 | 0.715 | 6.05 | 32.84 |
| $200 \times 200 \times 40$ | 0.21767 | 0.11517 | 6.145 | 28.23 | 53.34 |
| $400 \times 400 \times 80$ | 0.87337 | 0.71067 | 45.440 | 52.03 | 63.94 |

**Table 10: Execution times and performance comparison of GPU and CPU codes for 3D reconstruction**

A more detailed analysis can be done from the data in Table 11, Table 12 and Table 13. The total execution times shown in row 8 include the memory transfers. The timings in Table 10 are obtained by taking the data corresponding to rows 1, 2, 4, 6 and 7 in these tables, discarding rows 1 and 7 for the case where data transfers are not considered.

The cost for data transfer from CPU to GPU corresponds to uploading the depthmaps, segmentation masks and camera data to the GPU, and does not depend on the voxel resolution. The total data transferred corresponds to 14 FullHD float single-channel images, which amounts to nearly 111 Mb of data. The computed time of 95ms is in accordance with the 1.33 GB/s bandwidth estimated for CPU to GPU transfers in the GTX 280 card. The GPU to CPU data transfer (row 7) moves the computed volumetric 3D reconstruction to RAM. The amount of data moved equals to the product of the grid sizes in each direction (float elements). The timings obtained in the three test cases are also in accordance with the 1.40 GB/s bandwidth estimated for GPU to CPU transfers in this GPU.

Regarding processing kernels, the primal-dual iteration reaches an speedup of around 100 for big grid sizes, which is a fairly good result. In contrast, the computation of the initial data $w$ and $f$ obtains at most an speedup of 25. This is justified by the nature of the computation, which involves several conditional statements. This implies that not all the threads in a block execute the same operations, and some threads can be idle waiting for the slowest one to complete.

| | Process | GPU time (ms) | GPU time % | CPU time (ms) | CPU time % | Speedup |
|---|---|---|---|---|---|---|
| 1 | Allocate GPU memory and CPU to GPU data transfer | 95 | 80.46% | | | 0.00 |
| 2 | Computation of $w$ and $f$ | 10.8 | 9.15% | 225 | 31.47% | 20.83 |
| 3 | Primal iteration (average over 20 iterations) | 0.284 | | 12.5 | | 44.01 |
| 4 | Primal iteration (total cost for 20 iterations) | 5.68 | 4.81% | 250 | 34.96% | 44.01 |
| 5 | Dual iteration (average over 20 iterations) | 0.2645 | | 12.0 | | 45.37 |
| 6 | Dual iteration (total cost for 20 iterations) | 5.29 | 4.48% | 240 | 33.57% | 45.37 |
| 7 | GPU to CPU data transfer and free GPU memory | 1.3 | 1.10% | | | 0.00 |
| 8 | Total time | 118.07 | 100% | 715 | 100% | 6.05 |

**Table 11: Execution times and performance comparison for individual parts of the GPU and CPU algorithms for 3D reconstruction. 3D grid size $100 \times 100 \times 20$**

| | Process | GPU time (ms) | GPU time % | CPU time (ms) | CPU time % | Speedup |
|---|---|---|---|---|---|---|
| 1 | Allocate GPU memory and CPU to GPU data transfer | 95 | 43.64% | | | 0.00 |
| 2 | Computation of $w$ and $f$ | 61.1 | 28.07% | 1235 | 20.10% | 20.21 |
| 3 | Primal iteration (average over 20 iterations) | 1.3595 | | 129 | | 94.89 |
| 4 | Primal iteration (total cost for 20 iterations) | 27.19 | 12.49% | 2580 | 41.98% | 94.89 |
| 5 | Dual iteration (average over 20 iterations) | 1.344 | | 116.5 | | 86.68 |
| 6 | Dual iteration (total cost for 20 iterations) | 26.88 | 12.35% | 2330 | 37.92% | 86.68 |
| 7 | GPU to CPU data transfer and free GPU memory | 7.5 | 3.45% | | | 0.00 |
| 8 | Total time | 217.67 | 100% | 6145 | 100% | 28.23 |

**Table 12: Execution times and performance comparison for individual parts of the GPU and CPU algorithms for 3D reconstruction. 3D grid size $200 \times 200 \times 40$**

| | Process | GPU time (ms) | GPU time % | CPU time (ms) | CPU time % | Speedup |
|---|---|---|---|---|---|---|
| 1 | Allocate GPU memory and CPU to GPU data transfer | 95 | 10.88% | | | 0.00 |
| 2 | Computation of $w$ and $f$ | 342.3 | 39.19% | 8860 | 19.50 | 25.88 |
| 3 | Primal iteration (average over 20 iterations) | 9.223 | | 916.5 | | 99.37 |
| 4 | Primal iteration (total cost for 20 iterations) | 184.46 | 21.12% | 18330 | 40.34 | 99.37 |
| 5 | Dual iteration (average over 20 iterations) | 9.1955 | | 912.5 | | 99.23 |
| 6 | Dual iteration (total cost for 20 iterations) | 183.91 | 21.06% | 18250 | 40.16 | 99.23 |
| 7 | GPU to CPU data transfer and free GPU memory | 67.7 | 7.75% | | | 0.00 |
| 8 | Total time | 873.37 | 100% | 45440 | 100% | 52.03 |

**Table 13: Execution times and performance comparison for individual parts of the GPU and CPU algorithms for 3D reconstruction. 3D grid size $400 \times 400 \times 80$**

The theoretical computational cost of the initialization (computation of $w$ and $f$) is linear with respect to the number of voxels, and the one of the primal-dual iteration loop is linear with respect to both the number of voxels and the number of iterations. Thus, the number of voxels $V$ that can be initialized in a given time $t$ is given by $V = t \cdot S_I$, where $S_I$ is the number of voxels that can be initialized in a second. In turn, the number of voxels for which $K$ iterations can be done in a given time is given by $V = \frac{t \cdot S_L}{K}$, where $S_L$ is the number of voxels for which a single primal-dual iteration can be done in a second. From these expressions, the following expression can be given for the number of voxels for which a 3D reconstruction can be computed in a given time:

$$V = \frac{S_I \cdot S_L}{S_L + KS_I} t \qquad (12)$$

Estimations of $S_I$ and $S_L$ for the CPU and GPU codes are given in Table 14 and Table 15. As an example, assume that, for the GPU code, the values of $S_I$ and $S_L$ are equal to the speeds computed on a 3D grid of $400 \times 400 \times 80$ voxels (which is a conservative choice), i.e., $S_I = 37.42$ Mvoxels/second and $S_L = 694.93$ Mvoxels/second. If a real-time 3D reconstruction is desired we can perform it on a grid of approximately $\frac{37.42 \cdot 10^6}{1+0.053847K} \cdot 0.040$ voxels. Values of $V$ for various values of $K$ are provided in Table 16. The values for the equivalent grid are computed assuming that the 3D space to be reconstructed has the same dimension in the $X$ and $Y$ directions, and that its dimension in the $Z$ direction is five times smaller.

| 3D grid dimension | Initialization time (sec.) | Initialization speed (Mvoxels/second) | Primal – dual iteration time (sec.) | Primal-dual speed (MVoxels/second) |
|---|---|---|---|---|
| $100 \times 100 \times 20$ | 0.225 | 0.8889 | 0.0245 | 8.1633 |
| $200 \times 200 \times 40$ | 1.235 | 1.2955 | 0.2455 | 6.5173 |
| $400 \times 400 \times 80$ | 8.860 | 1.4447 | 1.8290 | 6.9974 |

**Table 14: Performance analysis for the 3D reconstruction algorithm (CPU version)**

| 3D grid dimension | Initialization time (msec.) | Initialization speed (Mvoxels/second) | Primal – dual iteration time (sec.) | Primal-dual speed (MVoxels/second) |
|---|---|---|---|---|
| $100 \times 100 \times 20$ | 0.01080 | 18.52 | 0.000548 | 364.96 |
| $200 \times 200 \times 40$ | 0.06110 | 26.19 | 0.002703 | 591.93 |
| $400 \times 400 \times 80$ | 0.3420 | 37.42 | 0.018419 | 694.93 |

**Table 15: Performance analysis for the 3D reconstruction algorithm (GPU version)**

| Number of iterations $K$ | Number of voxels $V$ | Approximate equivalent grid |
|---|---|---|
| 0 | 1496800 | $195 \times 195 \times 39$ |
| 10 | 972910 | $165 \times 165 \times 33$ |
| 20 | 720670 | $150 \times 150 \times 30$ |
| 40 | 474590 | $130 \times 130 \times 26$ |

**Table 16: Real-time processing capabilities of the 3D reconstruction algorithm with respect to the number of primal dual iterations**

## *4.2 Multi-GPU view synthesis algorithms*

Machines hosting more than one CUDA-capable GPU are nowadays common due to the low cost per gigaflop provided by GPUs. The CUDA language allows for the use of several GPUs in a program, by defining as much CPU execution threads as GPUs the system has. Each CPU thread handles a different GPU and threads can be executed concurrently. For real concurrency it is required that the number of CPU cores is equal or higher than the number of GPUs, which is the case in most systems.

For the analysis presented in this Section the system used is composed by

- 2x Quad Core Intel Xeon E5520 @ 2.27 GHz processor
- 24 GB RAM memory
- 2 nVidia GTX285 GPU with 1 GB of global memory each

Each GPU is connected to the motherboard through a PCI Express bus, so that data transfers between CPU and GPU can be done simultaneously for both GPUs.

The most favourable scenario for multi-GPU parallelization is when each CPU thread/GPU pair does not need to share data with any other pair. This is due to the fact that data communication between GPUs cannot be achieved directly, but through an intermediate transfer to the CPU RAM via the PCI express bus. The goal is therefore to design implementations that allow for different GPUs to operate on independent data. For performance, a load imbalance between GPUs is necessary.

In the context of view synthesis algorithms, the most natural situation in which independent threads can be launched over different data corresponds to the case where a video sequence has to be processed. If the work to be done for each frame does not depend on other frames, then the computation corresponding to different frames can be assigned to different GPUs on a per-frame splitting fashion. On the other hand, some algorithms can be split into tasks to be done for each camera (for example, background subtraction and depth computation), and thus a multi-GPU per-camera splitting is allowed. In both cases each GPU handles a similar and independent image dataset, and thus the computational cost is likely to be similar for consecutive frames of adjacent cameras, resulting in a good load balance.

For the estimation of the potential performance of a multi-GPU setting corresponding to the situations described above, a multi-GPU version of the gray scale conversion GPU program considered in Section 4 has been implemented (the source code of this program is listed in Appendix 3). The program takes as input two colour images and launches two CPU threads, one per image, which invoke the GPU gray scale conversion code on the two available GPUs. The workload is nearly the same for both GPUs. Table 17 shows the execution times of the algorithm when run on the multi-GPU system, using one or two GPUs. For a better comparison, the single-GPU program is a modification of the code used in Section 4 that processes two images sequentially. The execution times presented in Table 17 show that the execution time in the multi-GPU case is nearly one half of the single-GPU case, for the data transfers and the kernel execution parts, indicating that in the multi-GPU case each thread performs nearly as an independent single-GPU execution.

There is still an overhead due to the generation and handling of the CPU threads. This overhead is shown in the last row of Table 17, and has been estimated to be approximately equal to 60 ms per thread, by measuring the wall clock execution time difference between the multi-GPU and the single-GPU versions of the program, and also includes potential delays due to the different execution times of the different threads. It is worth mentioning that since no communication between different threads is required, if the program is processing a video sequence, each GPU will process its assigned frames without the need of re-launching the thread for every frame, and therefore the extra cost is only paid once, at the beginning of the program execution.

| **Process** | **Execution time in ms (1GPU)** | **Execution time in ms (2GPU)** |
|---|---:|---:|
| Upload data from CPU to GPU | 13.2 (6.6 ms per image) | 6.7 |
| Kernel execution | 0.66 | 0.33 |
| Download data from GPU to CPU | 10.8 (5.4 ms per image) | 5.5 |
| Parallelization Overhead | 0 | 120 |

**Table 17: Execution times for the single-GPU and multi-GPU grayscale conversion programs. Both codes operate on two images.**

We also note very small differences for the (per-image) data transfer timings of the single-GPU and multi-GPU parts. This may be due to many reasons, like the delay induced by the job scheduler of the operating system, or to hardware latency when both PCI Express buses are activated at the same time. Anyway, these differences are of the order of tenths of microseconds, and do not have a significant influence on the performance.

From these results, we can conclude that when operating on independent data, the use on a multi-GPU platform allows for essentially multiplying the performance by the number of GPUs in the system. This is the case for the depth estimation algorithm, which can be ported to a multi-GPU system because the depth estimation for each camera and frame is independent of each other. There are two main ways for the multi-GPU implementation of the depth estimation problem in a video sequence:

- Assigning to each GPU the depth computation for all cameras corresponding to a frame, different from GPU to GPU. Each GPU is responsible for the computation on a frame, and because the input data is likely to be similar for frames that are close in time, all GPU are expected to have a similar workload.
- Assigning to each GPU the depth computation of one or more cameras corresponding to a frame, the same for all GPUs. For efficiency, the number of cameras should be a multiple of the number of GPUs used in the computation. This approach is adequate for the case where the depth of a given camera and frame is constrained by the depth of the same camera at previous frames. In this approach the load balance can be slightly degraded because different cameras can have different workloads. The workload of a camera depends on the number of pixels in its segmentation mask, which can vary from one camera to another. A load balancing mechanism can be implemented. For example, the camera assigned to each GPU can vary from one frame to another, so that after a certain number of frames all GPU have processed all cameras.

In the case of the 3D reconstruction problem, the most natural implementation in a multi-GPU system is to make each GPU to compute the reconstruction for a different frame, using the input of all cameras for that frame. If time constraints are introduced, so as the 3D reconstruction at a frame depends on previous frames, then the 3D domain can be split in parts, so that each GPU processes a part of the 3D volume. This approach requires data exchanges between GPUs, which have to be done via intermediate transfers to the RAM memory, and represents an important bottleneck for performance. An interesting approach for this situation could be to fix a GPU that computes the 3D reconstruction while the rest is computing the depth corresponding to the next frame or doing something else.

Multi-GPU execution can be also applied to other contexts. As an example, consider the splitting technique described in Section 4.1.2, where the initial depth computation problem, defined on the whole image, is divided into smaller problems corresponding to single players or groups or player that are close in the 3D space. In this case, each subproblem can be handled by a separate GPU, up to the final phase where all depths are merged in a single depthmap. This latter merging process is done by a CPU code that can be run in parallel with the depth computation of the next frame. However, the workloads could be unbalanced in this situation, and it is a better approach to assign all subproblems corresponding to a camera and frame to a single GPU.

## 5 Conclusions

In this report a preliminary analysis on the feasibility of parallelizing view synthesis algorithms has been performed by analyzing the performance of some processes involved in the novel view synthesis computation, in particular background subtraction, depth computation and 3D reconstruction from depths. A careful analysis of single-GPU implementations of the aforementioned algorithms has been used to study its advantages and drawbacks and to point out approaches to improve the performance at the weakest points of the parallel implementation. A more qualitative analysis has been performed for the case of multi-GPU implementations of view synthesis algorithms, using the conclusions of the single-GPU case and the analysis of a simple multi-GPU program.

For the case of background subtraction, the implementation in CUDA proposed by (Pham et al., 2010) obtains good results, much faster than the CPU implementation. The asynchronous execution proposed reduces delays produced by memory transfers and optimize occupation of CPU and GPUs. These delays are noticeable in the non-optimized CUDA implementations of the depth estimation and the 3D reconstruction algorithms, which also highlight the importance of a proper handling of optimization aspects like multiprocessor occupancy, divergence, memory access patterns, data binning, etc.

The depth estimation algorithm has been used to illustrate a situation in which the use of priors like the players' position in the pitch can help in reducing the data dimensionality of the problem. A similar situation appears when segmentation masks for the input images are available and the computation can be restricted to segmented zones.

The importance of a high ratio between the number of operations to be done and the size of the input data to be transferred to the GPU has been shown using the 3D reconstruction algorithm. Performance improvements are clearly observed as the dimension of the 3D voxel grid is growing, given that the amount of data transferred is constant.

From the results of the presented study, based on particular approaches of some of the main algorithms for novel view synthesis in both single-GPU and multi-GPU platforms, we can extrapolate the following aspects that are of special importance when dealing with the generation of novel view synthesis in video sequences:

- To take advantage of the possibility of overlapping GPU computations with internal data transfers, CPU computations and disk reads/network transfers.
- To reduce the data dimensionality to reduce the computational effort, and to increase the number of operations to be done with respect to the input data.
- To design multi-GPU algorithms that take advantage of the data independency among cameras and frames, wherever possible, so as to reduce the GPU to GPU data transfers and to maximize load balance between GPUs.
- To optimize the programs in terms of GPU memory access, divergence, occupancy, etc.

These conclusions will guide further project developments related to view synthesis algorithms and are valuable inputs in the design of the FINE 3D video server and its API.

## 6 References

(Baeza et al., 2011) A. Baeza, P. Gargallo, N. Papadakis, and V. Caselles, A narrow band method for the convex formulation of discrete multi-label problems, *Multiscale Modeling and Simulation*, 8(5), pp. 2048-2078, 2010.

(Baeza et al., 2011) A. Baeza, V. Caselles, A. Bosch, and S. Sagàs, *First report on fast and accurate methods for depth computation, 3D reconstruction and inpainting*, FINE deliverable D4.2, 2011.

(Benezeth et al., 2008) Y. Benezeth, P. Jodoin, B. Emile, H. Laurent, and C. Rosenberger, Review and evaluation of commonly-implemented background subtraction algorithms, in IEEE International Conference on Pattern Recognition, pp. 1-4, 2008.

(Boykov et al, 2001) Y. Boykov, O. Veksler, and R. Zabih, Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern. Anal. Mach. Intell., 23* (11), pp. 1222-1239, 2001.

(Calderara et al., 2006) S. Calderara, R. Melli, A. Prati, and R. Cucchiara, Reliable background suppression for complex scenes, Proceedings of the 4th ACM International Workshop on Video Surveillance and Sensor Networks - VSSN '06, p. 211, 2006.

(Collins, 1996) R. Collins, A space-sweep approach to true multi-image matching. *Proc. Comp. Vis. and Pat. Recogn. (CVPR'96)*, pp. 358-363, 1996

(Curless & Levoy, 1996) B. Curless, and M. Levoy, A volumetric method for building complex models from range images, *Proc. SIGGRAPH*, pp. 303-313, 1996.

(Elgammal et al., 2000) A. Elgammal, D. Harwood, and L. Davis, Non-parametric Model for Background Subtraction, Proc. ECCV, pp. 751–767, 2000.

(Kolmogorov & Zabih, 2001) V. Kolmogorov, and R. Zabih, Computing visual correspondence with occlusions via graph cuts. *IEEE Int. Conf. Comp. Vis. (ICCV'01)*, 2, pp. 508-515, 2001.

(Latour et al., 2011a) P. Latour, P. Pacôme, and V. Rachelle, 3D video server, first version. FINE Deliverable, D6.2, 2011.

(Latour et al., 2011b) P. Latour, P. Pacôme, and V. Rachelle, Digitizing, storage and delivery platform. FINE Deliverable, D3.4, 2011.

(Liu & Sun, 2010) J. Liu, and J. Sun, Parallel Graph-cuts by Adaptive Bottom-up Merging, *Proceedings of CVPR*, 2010.

(McFarlane & Schofield, 1995) N. J. B. McFarlane, and C. P. Schofield, Segmentation and tracking of piglets in images, Machine Vision and Applications, 8 (3), pp. 187-193, 1995.

(NVIDIA, 2010) NVIDIA Corporation, CUDA Best Practices Guide. NVIDIA, 2010.

(Oliver et al., 2000) N. M. Oliver, B. Rosario, and A. B. Pentland, A Bayesian computer vision system for modeling human interactions, IEEE Transactions on Pattern Analysis and Machine Intelligence, 22 (8), pp. 831-843, 2000.

(Papadakis & Caselles, 2010) N. Papadakis, and V. Caselles, Multi-label depth estimation for graph cuts stereo problems, Journal of Mathematical Imaging and Vision, 38 (1), pp. 70-82, 2010.

(Parks & Fels, 2008) D. H. Parks and S. S. Fels, Evaluation of Background Subtraction Algorithms with Post-Processing, in IEEE Fifth International Conference on Advanced Video and Signal Based Surveillance, pp. 192-199, 2008.

(Pham et al., 2010) V. Pham, P. Vo, H. Vu Thanh, B. Le Hoai, GPU Implementation of Extended Gaussian Mixture Model for Background Subtraction, IEEE-RIVF 2010 International Conference on Computing and Telecommunication Technologies, Vietnam National University, 2010.

(Piccardi, 2004) M. Piccardi, Background subtraction techniques: A Review, in IEEE International Conference on Systems, Man and Cybernetics, pp. 3099-3104, 2004.

(Pock et al., 2008) T. Pock, T. Schoenemann, D. Cremers, and H. Bischof, A convex formulation of continuous multi-label problems, Proc. ECCV'08, Marseille, France, 2008.

(Radke et al., 2005) R. J. Radke, S. Andra, O. Al-Kofahi, and B. Roysam, Image change detection algorithms: a systematic survey, IEEE Transactions on Image Processing, 14 (3), pp. 294-307, 2005.

(Rockafellar, 1976) T. Rockafellar, Monotone operators and the proximal point algorithm. SIAM J. Control and Optimization, 14, pp. 877-898, 1976.

(Scharstein & Szeliski, 2002) D. Scharstein, and R. Szeliski, A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. Int. J. Comput. Vis., 47 (1), pp. 7-42, 2002

(Stauffer & Grimson, 1999) C. Stauffer and W. E. L. Grimson, Adaptive Background Mixture Models for Real-Time Tracking", Proc. CVPR, 1999.

(Sun et al., 2003) J. Sun, N.-N. Zheng, and H.-Y. Shum. Stereo matching using belief propagation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 25(7), pp. 787-800, 2003.

(Vineet & Narayanan, 2008) V. Vineet and P. J. Narayanan, Cuda cuts: Fast graph cuts on the GPU. In Proceedings of CVPR Workshops, 2008.

(Wren et al, 1997) C. Wren, A. Azarbayejani, T. Darrell, and A. Pentland, Pfinder: realtime tracking of the human body, IEEE Transactions on Pattern Analysis and Machine Intelligence, 19 (7), pp. 780-785, 1997.

(Yang et al., 2009) Q. Yang, L. Wang, R. Yang, H. Stewénius, and D. Nistér, Stereo matching with color-weighted correlation, hierarchical belief propagation and occlusion handling. IEEE Trans. Pattern Recogn.Mach. Intell., 31 (3), pp. 492-504, 2009.

(Zach et al, 2007) C. Zach, T. Pock, and H. Bischof, A globally optimal algorithm for robust TV-L1 range image integration, Proc. ICCV'07, 2007.

(Zivkovic, 2004) Z.Zivkovic, Improved adaptive Gausian mixture model for background subtraction, International Conference Pattern Recognition, Vol.2, pp. 28-31, 2004.

(Zivkovic & van der Heijden, 2006) Z.Zivkovic, F. van der Heijden, Efficient adaptive density estimation per image pixel for the task of background subtraction, Pattern Recognition Letters, 27(7), pp. 773-780, 2006.

## Appendix 1. A sample GPU code for colour to grayscale image conversion

```
#include"CImg.h"

#define BLOCK_DIM 8

using namespace cimg_library;

void convert_to_grayscale(CImg<float> &im, CImg<float> &im_g);
__global__ void grayscale(float *color_image,
                          float *grayscale_image,
                          int imagesize_x,
                          int imagesize_y);

int main(int argc, char *argv[]) {

  if(argc<2) {
    printf("\nUSAGE: %s input image\n", argv[0]);
  }
  else {
    CImg<float> im(argv[1]);
    CImg<float> im_g(im.width, im.height, 1);
    convert_to_grayscale(im, im_g);
    im_g.save_png("grayscale.png");
  }
  return 0;

}

void convert_to_grayscale(CImg<float> &im, CImg<float> &im_g) {

  float *color_image;
  float *grayscale_image;
  cudaMalloc((void **)&color_image,
    im.width * im.height * 3 * sizeof(float));
  cudaMalloc((void **)&grayscale_image,
    im.width * im.height * sizeof(float));
  cudaMemcpy(color_image, &im[0],
    im.width * im.height * 3 * sizeof(float),
    cudaMemcpyHostToDevice);

  cudaMemset(grayscale_image, 0 ,
    im.width * im.height * sizeof(float));
  dim3 gd(im.width / (BLOCK_DIM -1), im.height / (BLOCK_DIM -1));
  dim3 bd(BLOCK_DIM, BLOCK_DIM);
  grayscale<<<gd, bd>>>(color_image, grayscale_image,
                        im.width, im.height);
  cudaThreadSynchronize();
  cudaMemcpy(&im_g[0], grayscale_image,
    im.width * im.height * sizeof(float),
    cudaMemcpyDeviceToHost);
  cudaFree(color_image);
  cudaFree(grayscale_image);

}

__global__ void grayscale(float *color_image,
                          float *grayscale_image,
                          int imagesize_x,
                          int imagesize_y) {

  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int pos_channel0 = x + y * imagesize_x;
  int pos_channel1 = pos_channel0 + imagesize_x * imagesize_y;
  int pos_channel2 = pos_channel1 + imagesize_x * imagesize_y;
  if(x < imagesize_x && y < imagesize_y) {
    grayscale_image[pos_channel0] = 0.3 * color_image[pos_channel0]
      + 0.59 * color_image[pos_channel1]
      + 0.11 * color_image[pos_channel2];
  }

}
```

## Appendix 2. A sample CPU code for color to grayscale image conversion

```
#include"CImg.h"

using namespace cimg_library;

void convert_to_grayscale(CImg<float> &in, CImg<float> &out);

int main(int argc, char *argv[]) {

  if(argc<2) {
    printf("\nUSAGE: %s input image\n", argv[0]);
  }
  else {
    CImg<float> im(argv[1]);
    CImg<float> im_g(im.width, im.height, 1);
    convert_to_grayscale(im, im_g);
    im_g.save_tiff("grayscale.tif");
  }
  return 0;
}

void convert_to_grayscale(CImg<float> &in, CImg<float> &out) {

  for(int j = 0; j < in.height; j++)
    for(int i = 0; i < in.width; i++)
      out(i,j,0) = 0.3 * in(i,j,0)
        + 0.59 * in(i,j,1)
        + 0.11 * in(i,j,2);

}
```

## Appendix 3. A sample Multi-GPU code for color to grayscale image conversion

```
#include <ctime>
#include <time.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include"CImg.h"

#define BLOCK_DIM 8

using namespace cimg_library;

__global__ void grayscale(float *color_image,
                          float *grayscale_image,
                          int imagesize_x,
                          int imagesize_y);

typedef struct thread_descriptor {
  CImg<float> *im;
  CImg<float> *im_g;
  pthread_t tid;
  unsigned int gid;
}thread_descriptor_t;

void *convert_to_grayscale(void *ptr) {

  thread_descriptor_t *td = (thread_descriptor_t *) ptr;

  CImg<float> *im = td->im;
  CImg<float> *im_g = td->im_g;

  unsigned int gid = td->gid;

  int width = im->width;
  int height = im->height;

  dim3 gd(width / (BLOCK_DIM -1), height / (BLOCK_DIM -1));
  dim3 bd(BLOCK_DIM, BLOCK_DIM);

  float *color_image;
  float *grayscale_image;

  cudaSetDevice(gid);

  cudaMalloc((void **)&color_image,
    width * height * 3 * sizeof(float));
  cudaMalloc((void **)&grayscale_image,
    width * height * sizeof(float));
  cudaMemcpy(color_image, im->data,
    width * height * 3 * sizeof(float),
    cudaMemcpyHostToDevice);
  cudaMemset(grayscale_image, 0 ,
    width * height * sizeof(float));

  grayscale<<<gd, bd>>>(color_image, grayscale_image,
                        width, height);
  cudaThreadSynchronize();

  cudaMemcpy(im_g->data, grayscale_image,
    width * height * sizeof(float),
    cudaMemcpyDeviceToHost);
  cudaFree(color_image);
  cudaFree(grayscale_image);

  return NULL;
}

__global__ void grayscale(float *color_image,
                          float *grayscale_image,
                          int imagesize_x,
                          int imagesize_y) {

  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  int pos_channel0 = x + y * imagesize_x;
  int pos_channel1 = pos_channel0 + imagesize_x * imagesize_y;
  int pos_channel2 = pos_channel1 + imagesize_x * imagesize_y;

  if(x < imagesize_x && y < imagesize_y) {
    grayscale_image[pos_channel0] =
        0.3 * color_image[pos_channel0]
      + 0.59 * color_image[pos_channel1]
      + 0.11 * color_image[pos_channel2];
  }

}

int main(int argc, char *argv[]) {

  int ndevices;
  cudaGetDeviceCount(&ndevices);

  if(argc < ndevices + 1) {
    printf("\nERROR: number of input images should be %d\n",
ndevices);
    exit(1);
  }

  else {
    thread_descriptor_t tds[ndevices];

    CImgList<float> im(ndevices);
    CImgList<float> im_g(ndevices);

    for(int n = 0; n < ndevices; n++) {
      im[n].load(argv[n+1]);
      im_g[n].assign(im[n].width, im[n].height, 1);
    }
    for(int n = 0; n < ndevices; n++) {
      tds[n].im = &im[n];
      tds[n].im_g = &im_g[n];
      tds[n].gid = n;

      int err = pthread_create(&tds[n].tid, NULL,
                               convert_to_grayscale, &tds[n]);
      if(err!=0) {
        printf("\nError creating thread %d\n", n);
        exit(1);
      }
    }

    for(int n = 0; n < ndevices; n++) {
      pthread_join(tds[n].tid, NULL);
    }


    for(int n = 0; n < ndevices; n++) {
      char name[200];
      sprintf(name, "grayscale%d.png", n);
      im_g[n].save_png(name);
    }

  }
  return 0;
}
```