

# Parallelization strategies for markerless human motion capture

Alberto Cano · Enrique Yeguas-Bolivar ·  
Rafael Muñoz-Salinas · Rafael Medina-Carnicer ·  
Sebastián Ventura

Received: 4 November 2013 / Accepted: 16 October 2014 / Published online: 12 November 2014  
© Springer-Verlag Berlin Heidelberg 2014

**Abstract** Markerless motion capture (MMOCAP) is the problem of determining the pose of a person from images captured by one or several cameras simultaneously without using markers on the subject. Evaluation of the solutions is frequently the most time-consuming task, making most of the proposed methods inapplicable in real-time scenarios. This paper presents an efficient approach to parallelize the evaluation of the solutions in CPUs and GPUs. Our proposal is experimentally compared on six sequences of the HumanEva-I dataset using the CMAES algorithm. Multiple algorithm's configurations were tested to analyze the best trade-off with regard to the accuracy and computing time. The proposed methods obtain speedups of  $8\times$  in multi-core CPUs,  $30\times$  in a single GPU and up to  $110\times$  using 4 GPUs.

**Keywords** Markerless motion capture (MMOCAP) · GPU · Tracking

---

A. Cano · E. Yeguas-Bolivar · R. Muñoz-Salinas (✉) ·  
R. Medina-Carnicer · S. Ventura  
Department of Computer Science and Numerical Analysis,  
University of Cordoba, Córdoba, Spain  
e-mail: rmsalinas@uco.es

A. Cano  
e-mail: acano@uco.es

E. Yeguas-Bolivar  
e-mail: eyeguas@uco.es

R. Medina-Carnicer  
e-mail: rmedina@uco.es

S. Ventura  
e-mail: sventura@uco.es

E. Yeguas-Bolivar · R. Muñoz-Salinas · R. Medina-Carnicer  
Maimonides Institute for Biomedical Research (IMIBIC),  
Córdoba, Spain

## 1 Introduction

MMOCAP is an emerging field with applications in areas like the animation industry [1], medical rehabilitation [2], and video surveillance [3], amongst others. The problem consists in determining the joints' angles of an articulated body model that best matches the pose of a subject recorded by one or several video cameras. It is a high-dimensional problem in which the evaluation of a single solution is a very time-consuming task. As a consequence, most of the proposed methods either require a high number of evaluations (leading to computing times unsuitable for real-time applications) or rely on simple human models (leading to suboptimal tracking results).

This paper presents an efficient approach to evaluate the solutions in the MMOCAP problem and three strategies to parallelize their computation. First, we propose a parallelization strategy based on Streaming SIMD Extensions (SSE), which increase the performance by processing multiple elements simultaneously. Second, a strategy based on a multi-threading approach which takes advantage of the parallel capabilities of multi-core CPUs is presented. Third, we propose a parallelization strategy that delegates computation on Graphic Processing Units (GPUs). In particular, our proposal can be parallelized in multiple GPUs making it very scalable. In addition, this work aims at evaluating multiple algorithm configurations to determine the one achieving the best trade-off between the model accuracy and the computing time. The higher the model accuracy, the better it fits to observations, but also, more computing time is required. Thereby, an experimental study is conducted to measure the performance and efficiency of the model with regard to the body model resolution and the number of evaluations of the algorithm.

The parallelization strategies have been evaluated on six sequences of the HumanEva-I dataset [4]. The experimental

results show the performance improvements of the different parallelization approaches namely,  $2\times$  for the SSE approach,  $4\times$  for the multi-threading approach, and  $8\times$  for the multi-threading + SSE approach. Specifically, GPUs have demonstrated to achieve high performance and significantly reduce the evaluation time, up to  $30\times$  when using 1 GPU,  $60\times$  when using 2 GPUs, and  $110\times$  when using 4 GPUs.

The remainder of this paper is structured as follows. Section 2 revises the related work. Section 3 formulates the problem of pose estimation, and describes the body models and the fitness function. Sections 4 and 5 present the parallelization strategies addressed. Section 6 shows the experimental results. Finally, Sect. 7 draws some conclusions.

## 2 Background

This section provides an overview of the related works. First, we review the main optimization approaches applied to the problem. Then, we focus on the most relevant parallelization strategies for the MMOCAP problem found in the literature.

### 2.1 Optimization approaches

The first solutions for the MMOCAP problem consist in the use of particle filters. In particular, the Condensation algorithm is the most prevalent of such algorithms and has been widely employed for the tracking task [5]. However, when applied to this problem, it has been repeatedly shown that it suffers from the *curse of dimensionality*. Therefore, Deutscher and Reid proposed the Annealed Particle Filter (APF) [6], which combines the ideas of the Condensation and the Annealed search so as to improve the tracking results. Corazza et al. propose also a custom version of adapted fast simulated annealing [7] for body tracking using as input data a visual hull reconstruction and an a priori model of the subject. Another popular approach for tracking articulated objects is the use of Partitioned Sampling (PS) [8]. The technique was initially employed for tracking several objects using particle filters, but then it was successfully applied to hand tracking. Unlike the APF, PS imposes a strong partition of the search space. Bandouch et al. proposed the Partitioned Sampling Annealed Particle Filter (PSAPF) [9] as an attempt to combine the strengths of PS and the Annealed Search. To do so, they incorporate the APF within a PS framework by applying an appropriate weighted resampling in each subspace. As they report, they are able to cope up with high-dimensional models, but at the cost of employing a very high number of evaluations per frame.

The MMOCAP problem is a continuous optimization problem for which Evolutionary Algorithms [10] have repeatedly proven to provide excellent results. John et al. [11] applied the Particle Swarm Optimization (PSO)

algorithm with great success, reporting relevant improvements over APF and PSAPF. The main advantages of the PSO algorithm become particularly evident when tracking fast movements, since it has demonstrated a good performance without requiring any motion prior. Zhao and Liu [12] proposed a Hierarchical Annealed Genetic Algorithm to infer the three-dimensional pose from a single monocular camera. Yeguas-Bolivar et al. [13] perform an experimental comparison of three relevant evolutionary algorithms namely Covariance Matrix Adaptation Evolutionary Strategy (CMAES) [14], Differential Evolution (DE) [15], and PSO [16], with two particle filters, namely APF [6] and PSAPF [9]. The results obtained show that the evolutionary algorithms evaluated performed significantly better than particle filters. In particular, the CMAES algorithm obtained the best performance.

In spite of the advances achieved over the last years, mobility limitations often are imposed to the body models employed so as to obtain reasonable performance in manageable computing times. For instance, there are works [17, 6, 18, 11] which employ models with no more than 32 degrees of freedom (DOF) and assume no mobility in dorsal spine, hands and feet. This simplification of the human anatomy allows a tractable computation of the model while achieving acceptable results for some applications. However, some other applications require a more precise modeling of the human body so as to measure biomechanical parameters [19, 20, 21, 22]. In such cases, the need of a high number of evaluations deters from using MMOCAP in real-time applications. Thus it would be desirable to reduce the time employed for evaluating solutions.

### 2.2 Parallelization approaches

Three main sources of parallelization can be exploited in current mass-produced hardware. First, most of the current processors include SIMD (Single Instruction Multiple Data) instructions, which provide a limited form of parallelism that can be exploited to obtain relevant improvements [23]. Second, multi-core CPUs are able to solve high-performance applications more efficiently using parallel computing [24]. Third, GPUs have gained an important role in the area of parallel computing [25, 26]. In particular, the Compute Unified Device Architecture (CUDA) [27] is a parallel computing architecture developed by NVIDIA. It has attracted increasing attention over the last few years, providing massive parallel computation for solving highly parallelizable high-dimensional optimization problems and data intensive tasks.

In recent years various approaches have been proposed for the MMOCAP problem using parallel techniques, and these based on GPU computing have gained much of the attention. Model-based object detection is tackled by GPU implementations of soft computing techniques [28], where

CUDA is used for accelerating a tracking algorithm based on adaptive appearance models and PSO. Based on an articulated 3D body model, in [29] the GPU is used to implement a real-time full-body tracking algorithm using a limited number of DOF. The method is directly based on the sequential approach presented in [11]. PSO is the most popular algorithm in parallel implementations because of its inherent parallel nature [30, 31, 32]. Nonetheless, some works like [33] apply other easily parallelizable meta-heuristics (e.g. DE) for a fast search and to reach good results in human body pose estimation.

Other approaches incorporate a multi-layer framework for model-based pose searching where a stochastic approach (e.g. PSO) can be inserted. In [34, 35], two layers of search, with an efficient GPU implementation, support robust and accurate pose recovery: a sampling algorithm with a weak dynamical model introducing a non-parameter niching technique into the particle filter and a hierarchical local optimization to refine the estimation of sampling. Body pose tracking is performed in 3D space using 3D data reconstructed at every frame. Another approach is proposed in [36] where a probabilistic filtering framework employs a highly accurate generative model with a discriminative model and the GPU is exploited to perform large numbers of likelihood evaluations efficiently. In this case, the human motion capture task is approached using time-of-flight sensors.

Finally, the work of Zhang et al. [35] proposes an evaluation strategy based on a volumetric reconstruction. The authors design a system that employs GPUs to speedup several steps of the evaluation process. However, the use of volumetric reconstruction (based on foreground silhouettes) has the problem of propagating the segmentation errors to the 3D space. So, almost perfect segmentations or robust methods to deal with inconsistent silhouettes [37, 38, 39] are required to obtain correct volumetric reconstructions. In contrast, other authors evaluate the foreground images directly [9, 17, 6] to deal with segmentation errors.

### 3 Problem formulation

Our problem can be formulated as estimating the pose  $x^t$  of a subject at each time step  $t$  from a set of synchronized and calibrated video cameras. For that purpose, a body model comprised of a *skin* model (triangular mesh) and a *skeleton* model (internal structure of articulations) is employed. The skeleton is modeled as a hierarchical structure where each node represents a joint which is subject to rotations in the three axes ( $R_x, R_y$  and  $R_z$ ). It is employed to apply the body movements to the skin model in such a way that the transformation of a node affects all its children. Figure 1a shows the skin and skeleton models employed in our work, whereas Fig. 1b depicts its hierarchical structure.

As can be seen, the hierarchical model has a root node (root joint) which defines the global rotations and translations. In total, our model is comprised of 16 joints, so that a complete transformation of such model is defined by 3 translation components ( $T_x, T_y$  and  $T_z$ ) plus  $16 \times 3$  rotations, i.e., a total of 51 parameters (DOF) constituting the dimension  $D$  of the problem.

$$x = \{T_x, T_y, T_z, R_x^1, R_y^1, R_z^1, \dots, R_x^{16}, R_y^{16}, R_z^{16}\} \in \mathbb{R}^D.$$

However, considering that some of these parameters correspond to invalid rotations (e.g. ankles have only 2 DOF), the final model employed in this work can be reduced to  $D = 39$  parameters.

A fitness function  $f(x)$  must be defined indicating the likelihood of a model configuration to be correct. For each new frame, the minimization procedure relies on the results obtained in the previous one so as to improve the results. For the first frame, an initial body configuration  $x^0$  is provided.

In this work, we propose  $f(x)$  as an optimized version of the silhouette matching function employed in most related works [9, 17, 6, 18, 11, 13]. In short, given a model configuration  $x$ , its projection (silhouette) in all the cameras is matched against the foreground information obtained by background subtraction. The degree of overlap between the real and synthetic silhouettes is measured aiming at maximizing it. The evaluation process can thus be divided into three main steps: foreground estimation, model projection and fitness evaluation, which are explained in detail below.

#### 3.1 Foreground estimation

In an initial phase, a background model capturing the color statistics of each pixel is created. This process is done prior to the recording of the scene. Then, using background subtraction techniques, the foreground images are obtained indicating which pixels belong to the moving objects in the scene. Let us denote by

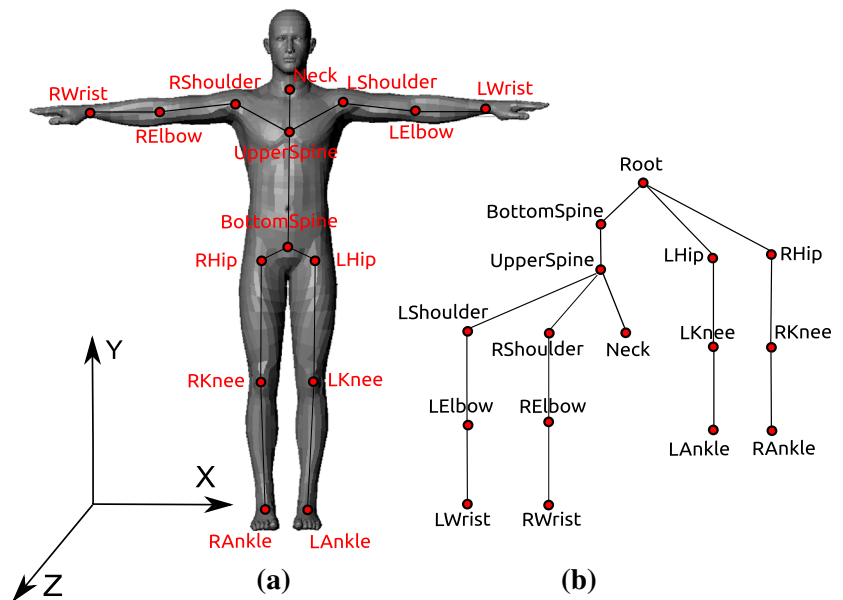
$$\mathcal{F}^t = \{\mathcal{F}_c^t | c = 1, \dots, N_c\}, \tag{1}$$

the set of foreground images obtained at time instant  $t$  with the  $N_c$  cameras available. A pixel  $\mathcal{F}_c^t(p)$  is 1 if it belongs to the foreground and 0 otherwise. The foreground images are employed for all the evaluations of the frame  $t$ , so that they are computed only once. In this work we have employed the background approach proposed by Horprasert et al. [40].

#### 3.2 Model projection

The projection of the model is the most repetitive task since it needs to be computed for each configuration  $x$ . It is comprised of two stages. First, it is necessary to calculate the three-dimensional position of the body meshes, according to the configuration  $x$ . Then, it is required to render

**Fig. 1** Body model employed: **a** Skin and skeleton models. The skin is a 3D model representing the surface of the body while the skeleton represents the internal structure of the articulations (joints and bones). **b** Hierarchical structure of the skeleton model. Each node represents a joint which is subject to rotations in the three axes. The transformation of a node affects all its children



the meshes in each image given that the camera parameters are known.

In this work, we propose a simplified projection approach that reduces the computing time. Instead of drawing the triangle meshes, we calculate the projection of its vertices and draw a rectangular patch around it.

Let us consider the  $N_v$  vertices of the triangle meshes that comprise the body model shown in Fig. 1(a). Each vertex  $v = [x, y, z]$  is assigned to a joint  $j$ , so that its movement affects all the vertices assigned to this joint (skinning). We denote by  $\mathcal{V}^j$  the set of vertices assigned to the joint  $j$ .

A three-dimensional transformation can be easily modeled in homogeneous coordinates as a  $4 \times 4$  matrix multiplication. This notation is specially appropriated since multiple transformations can be concatenated by multiplying the corresponding matrices. So, we can denote by  $T_j$  the matrix that transforms the vertices in  $\mathcal{V}^j$ .

The previous transformation produces the location of the model's vertices given by the configuration  $x$ . Afterwards, it is required to project the model onto the cameras. For that purpose, the extrinsic and intrinsic camera parameters are needed. These parameters are calculated prior to the sequence recording in a process called calibration.

The camera's extrinsics define the three-dimensional relationship between the camera reference system (CRFS) and a global reference system (GRFS) shared amongst all cameras. The camera extrinsics  $E_c$  is a  $4 \times 4$  matrix which translates a three-dimensional point (in homogeneous coordinates) from the GRFS to the CRFS. Once a point is expressed in the CRFS, the camera's intrinsics allow to determine its projection onto the camera image (pixel coordinates). Assuming a pin-hole model, the intrinsic matrix of camera  $c$  is defined as:

$$K_c = \begin{pmatrix} f_x & 0 & \rho_x \\ 0 & f_y & \rho_y \\ 0 & 0 & 1 \end{pmatrix} \quad (2)$$

where  $f_x, f_y$  are the focal lengths in both axes and  $\rho_x, \rho_y$  is the optical center.

Real cameras are always affected by distortion making the ideal pin-hole model invalid in realistic scenarios. Removing the distortion of a point is an iterative process that can be time consuming if applied to each vertex. However, it is possible to precompute the undistortion map for each camera, and apply it to the foreground image. Thus, the undistortion model is applied only once and we can assume in the following that the camera follows the ideal pin-hole model. As a consequence, the projection of a vertex  $v$  in the camera  $c$  can be completely expressed as:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = K_c \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} E_c T_j \begin{bmatrix} v^j \\ 1 \end{bmatrix} = \chi_c^j \begin{bmatrix} v^j \\ 1 \end{bmatrix} \quad (3)$$

where  $\chi_c^j$  is the matrix that projects all the points from  $\mathcal{V}^j$  to the camera  $c$ . The final camera coordinate is obtained as  $(x'/w', y'/w')$ . The main advantage of using this notation is that a single matrix  $\chi_c^j$  is employed for all the points in  $\mathcal{V}^j$ , making the model projection very fast.

Projecting exclusively the vertices instead of the triangles would produce a sparse set of points instead of a filled silhouette. Therefore, a small patch around each vertex projection is drawn so as to obtain a filled silhouette. The size of each patch is computed according to the distance of the model to the camera. The nearer the model from the camera, the larger the patch, and vice versa. The patch size

employed is the same for all vertices in a body part to avoid computation overhead. So, it is expected the area of the triangles not to diverge a lot from the mean to avoid leaving holes in the silhouette. Let  $l_t$  be the average length of the triangle sides of the body model employed. Then, a rectangle of similar area would have a side length  $l_r = \sqrt{l_t^2/2}$ . Considering that the average distance of the points to the camera is  $d$ , and that  $f = f_x \simeq f_y$ , the size in pixels of the rectangle  $p$  is given applying the pin-hole model as:

$$p \approx \frac{fd}{l_r} \tag{4}$$

In practice,  $d$  is not computed as the average distance but as the distance of a representative vertex of the body part. To avoid gaps in the projection it is important to have a mesh with vertices of similar area and vertices evenly distributed along the surface.

Finally, the vertex normals of the meshes can be used to determine whether they are seen from the “front” or the “back” side. These seen from the back side, can be ignored so that only those seen from the front are projected to generate the body silhouette. As a consequence, the number of vertices projected is reduced to half. To do that, let us consider the normal orientation to each vertex  $n$ . Given the configuration  $x$ , the new orientation  $n'$  can be obtained multiplying  $n$  by the upper  $3 \times 3$  matrix of  $E_c T_j$ . If  $n'_x > 0$ , the normal points towards the camera and the point must be projected. Otherwise, the point is ignored, saving time.

### 3.3 Fitness evaluation

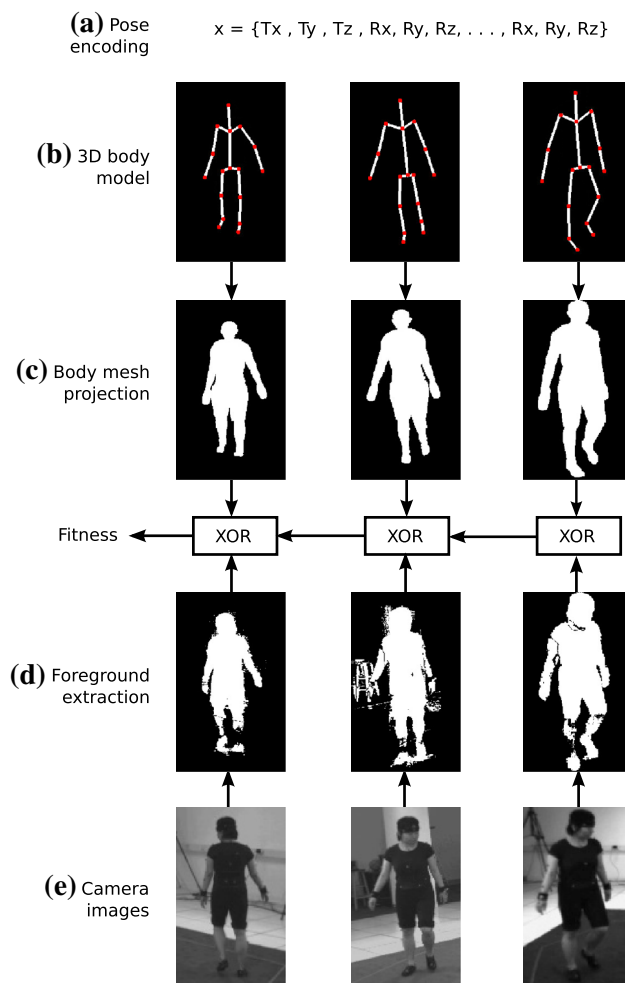
Using the above equations, the silhouettes of a body model configuration  $x$  are computed in all cameras. Let us denote by  $\mathcal{M}(x) = \{\mathcal{M}_c^x | c = 1, \dots, N_c\}$ .

these silhouette images, so that a pixel  $\mathcal{M}_c^x(p)$  is 1 if it belongs to the model’s silhouette and 0 otherwise.

Using the above-defined concepts, a model can be evaluated by matching the degree of overlap of its projected silhouette and the foreground image. Thus, let us define the evaluation function as:

$$f_c^t(x) = \frac{1}{2|\mathcal{D}(\mathcal{M}_c^x)|} \sum_{p \in \mathcal{D}(\mathcal{M}_c^x)} \mathcal{M}_c^x(p) - \mathcal{F}_c^t(p) + \frac{1}{2|\mathcal{D}(\mathcal{F}_c^t)|} \sum_{p \in \mathcal{D}(\mathcal{F}_c^t)} \mathcal{F}_c^t(p) - \mathcal{M}_c^x(p) \tag{6}$$

where the function  $\mathcal{D}(\cdot)$  indicates only these pixels with value 1. The first term of Eq. 6 accounts for these model points that project on foreground points, i.e., it decreases as the degree of overlap between the model and the mask increases. On the other hand, the second term of the equation accounts for the pixels of the foreground image that are not



**Fig. 2** Evaluation process: **a** encoding of the pose  $x$ . **b** 3D skeleton modeling of the pose  $x$ . **c** Model silhouette after mesh projection. **d** Foreground images obtained from **e** camera images, by background subtraction techniques and showing in white pixels belonging to the moving objects in the scene

covered by the model’s projection. Consequently, the function behaves as the logical XOR function of the two images, and it is evaluated  $f_c^t(x) = 0$  when the model projection fits exactly the foreground mask. On the contrary,  $f_c^t(x)$  tends to 1 as the degree of overlap decreases.

Due to illumination changes and color similarities between the subject and the background, it is unlikely to achieve a perfect match between the model and the foreground images. The use of multiple cameras helps not only to alleviate these problems but also to infer the three-dimensional configuration of the subject. The evaluations of the model in all the camera images are fused as:

$$f^t(x) = \frac{1}{N_c} \sum_c f_c^t(x). \tag{7}$$

Therefore, values of Eq. 7 near 0 indicate that  $x$  is a good solution and values near 1 indicate that  $x$  is a poor solution.

Figure 2 helps to clarify the above explanation. Given the pose encoding  $x$ , it summarizes the evaluation process that comprises the skeleton modeling, the body skin projection and the matching with the foreground images extracted from the multiple cameras.

The high complexity of the evaluation process is caused by the high number of operations carried out to generate the body model pose, the projection of the silhouette and the matching with the camera's foreground image. This process is repeated for every camera view and every tentative model in the population of the evolutionary algorithm. Moreover, the algorithm iterates to improve the fitness along a certain number of generations until the limit of the number of evaluations is reached. Eventually, the whole process is repeated for every frame in the video sequence. Consequently, this causes a high complexity and demands significant computation resources. Therefore, it is necessary to apply parallelization strategies to speedup this process.

#### 4 CPU parallelization strategies

This section presents the parallelization strategies proposed to speedup the evaluation of solutions using CPUs. First, the use of CPU's Streaming SIMD Extensions is presented. Second, the use of multi-threading on multi-core CPUs is described.

##### 4.1 Streaming SIMD extensions

The proper use of the SSE instructions has been shown to yield high-performance levels [23]. The SIMD nature of the SSE instruction set ideally suits for two major components of the evaluation process. The former comprises the vertex projection of the reference body model using the transformation matrix. The latter represents the XOR function of the projected body model image and the foreground image.

###### 4.1.1 Vertex projection

The vertex projection process involves the matrix multiplication between the vertices  $v^t$  and the transformation matrices  $\mathcal{X}_c^j$ . The computational complexity is due to many times this multiplication is performed. Code 1 shows the naive matrix multiplication, and its corresponding SSE instruction set. These SSE instructions are capable of calculating the multiplication and sum of four values concurrently. The `_mm_load_ps` function loads from memory four single-precision floating-point values, representing a row of the transformation matrix. The `_mm_mul_ps` function multiplies each of the four values of the transformation matrix with the vertex elements. The `_mm_hadd_ps` function performs a horizontal add, meaning that adjacent elements in the operand are added together.

Code 1: Naive and SSE matrix multiplication.

```
void matrixMul_naive(float* vT, float* v, float* M) {
    vT[0]=v[0]*M[0] + v[1]*M[1] + v[2]*M[2] + v[3]*M[3];
    vT[1]=v[0]*M[4] + v[1]*M[5] + v[2]*M[6] + v[3]*M[7];
    vT[2]=v[0]*M[8] + v[1]*M[9] + v[2]*M[10] + v[3]*M[11];
    vT[3]=v[0]*M[12] + v[1]*M[13] + v[2]*M[14] + v[3]*M[15];
}

void matrixMul_SSE(float* vT, float* v, float* M) {
    __m128 row1,row2,row3,row4,in_out,a,b,c,d;

    row1=_mm_load_ps(M);
    row2=_mm_load_ps(M+4);
    row3=_mm_load_ps(M+8);
    row4=_mm_load_ps(M+12);

    in_out=_mm_load_ps(v);
    a=_mm_mul_ps(row1,in_out);
    b=_mm_mul_ps(row2,in_out);
    c=_mm_mul_ps(row3,in_out);
    d=_mm_mul_ps(row4,in_out);

    a=_mm_hadd_ps(a,b);
    b=_mm_hadd_ps(c,d);
    in_out=_mm_hadd_ps(a,b);

    _mm_store_ps((float*) vT, in_out);
}
```

###### 4.1.2 Fitness evaluation

The fitness evaluation (Eqs. 6 and 7) can be performed by a pixelwise XOR operation that compares the body projections with the foreground images. Since image pixels are represented by 8-bit values, the SSE instruction set can provide an ideal speedup of 16. Code 2 shows the naive XOR function of the images, and its parallelization using the SSE instruction set.

The XOR function first loads 16 pixels (8-bit elements) from the two images into two 128-bit registers using the `_mm_load_si128` function. Then, the `_mm_sad_epu8` function computes the absolute difference of the 16 elements of the two registers. Since the feasible values are 8-bit integers (0 for black and 255 for white), the absolute difference function actually behaves as a logical XOR which indicates pixel error. Finally, we seek to count the number of errors of the whole image. Thereby, the `_mm_sad_epu8` function also sums the XOR values packing two partial semi-sums, which are eventually added to produce the error sum for the given 16 pixels. This process is embedded in a loop to process the complete image, and it is repeated for each of the camera views and each of the body projections.

Code 2: Naive and SSE XOR function.

```

int xor_naive(char* image_1, char* image_2, int imgSize) {
    int errorSum = 0;
    for (int p = 0; p < imgSize; p++)
        if(image_1[p] xor image_2[p]) errorSum++;
    return errorSum;
}

int xor_SSE(__m128i* image_1, __m128i* image_2,
           int imgSize) {
    int errorSum = 0;
    for (int p = 0; p < imgSize; image_1++, image_2++, p+=16) {
        __m128i r1 = _mm_load_si128 ( image_1 );
        __m128i r2 = _mm_load_si128 ( image_2 );
        __m128i res = _mm_sad_epu8 ( r1, r2 );
        errorSum += _mm_extract_epi16( res, 0 ) +
                   _mm_extract_epi16( res, 4 );
    }
    return errorSum/255;
}

```

Pixels of binary images might use a 1-bit representation to reduce the memory size. However, the time required to convert from a 8-bit representation to a 1-bit representation is high. Consider that mapping one bit into a 8-bit word requires a mask operation. Therefore, mapping 8 bits requires 8 mask operations on the same memory position, i.e., atomic operations that are performed in sequential order. Although the bitwise XOR would be faster, the memory load/store instructions using a mask would increase the total runtime.

#### 4.2 Multi-core CPU

Microprocessor industry have moved to multi-core architectures to continue to increase the computational power of their processors. Today, desktop CPUs are multi-core processors usually having four cores capable of processing multiple tasks concurrently. Taking advantage of the multiple cores of a CPU is a straightforward process using multi-threading directives. Open multi-processing (OpenMP) is an implementation of multi-threading, which forks a specified number of slave threads and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

An efficient and commonly used parallelization strategy using multi-core CPUs is the population parallel approach, which operates by multiple candidate solutions being evaluated in parallel by separate threads of execution. Thereby, the population is divided into multiple chunks that are evaluated in the multiple cores concurrently. Code 3 shows the loop for the evaluation of the individuals of the population. The `#pragma omp parallel for` directive enables automatically the concurrent execution of the

evaluation of each individual using multiple threads that take advantage of the multiple cores of the CPU.

Code 3: Population parallel approach using OpenMP.

```

void evaluate(float[][] population) {
    #pragma omp parallel for
    for(int i = 0; i < Nsolutions; i++) {
        evaluate(population[i]);
    }
}

```

## 5 GPU parallelization approaches

GPUs are intrinsically aimed for the parallel processing of computer images and there are many opportunities to use their power to speedup the evaluation of solutions. It must be noted that using GPUs requires a certain amount of communication and memory transfer overhead, which has to be minimized in order to avoid delays. This is a disadvantage of GPU computing which in some cases causes a CPU application to perform better than a GPU implementation even if the computation of the GPU is faster as the memory transfer and communication overhead of some applications cannot be hidden by the performance increase. In this work, we propose a GPU parallelization approach that proceeds as follows.

In an initial step, the reference body model (vertices and joints) is copied to the GPU. Then, for each frame, the foreground images are computed and transferred to the GPU memory. Then, the evolutionary algorithm is run on CPU and at some point it requires the solutions to be evaluated. The GPU is then employed to evaluate the solutions in parallel, producing the fitness values that are passed to the evolutionary algorithm to compute the next population set.

The work performed on the GPU is divided in the five stages that are outlined in Code 4, where  $N_{tb}$  is the number of threads per block of the grid (we employed 256 to maximize the GPU occupancy),  $H_{toD}$  represents host to device and  $D_{toH}$  device to host transfers. First, the set of solutions that comprise the population are transferred to the GPU. This small transaction is performed prior the evaluation and comprises  $N_s \cdot D \cdot \text{sizeof}(\text{float})$  bytes, where  $N_s$  stands for the number of solutions to be evaluated. Second, the projection matrices  $\chi_c^j$  required for the whole population are computed in parallel. Third, the projections of the body vertices are computed. Fourth, the fitness is evaluated using the XOR. Fifth, the fitness values

are transferred to CPU. This transaction comprises  $N_s \cdot \text{sizeof}(\text{float})$  bytes.

Due to space limitations, the kernel functions cannot be displayed in the article document, and the reader is referred to this website for further details.<sup>1</sup>

#### Code 4: GPU kernel calls.

```
// Copy population to GPU
cudaMemcpy(d_pop, h_pop, sizePopBytes, HtoD);

dim3 gridMatrix(Njoints, Ncameras);
matrix_computation <<< Nsolutions, gridMatrix >>>
    (d_tmatrix, d_pop, ...);

dim3 gridProjection(Nvertices/Ntb, Ncameras, Nsolutions);
vertex_projection <<< gridProjection, Ntb >>>
    (d_proj, d_vertices, d_tmatrix, ...);

dim3 gridFitness(Nsolutions, Ncameras);
fitness_evaluation <<< gridFitness, Ntb >>>
    (d_fitness, d_proj, d_img, ...);

// Retrieve fitness values from GPU
cudaMemcpy(h_fitness, d_fitness, sizeFitnessBytes, DtoH);
```

### 5.1 Computation of the $\chi_c^j$ matrices

As denoted in Sect. 3, a solution  $x$  encodes a translation and a set of rotations which produce a transformation matrix  $T_j$  for each of the body joints. These matrices are multiplied by the camera extrinsics  $E_c$  and intrinsics  $K_c$  to consider the cameras position and angles. The outcome is the  $\chi_c^j$  matrices that projects all the points from  $\mathcal{V}^j$  to the camera  $c$ . This process is implemented on the GPU in a 3D kernel to compute the transformation matrices for each body joint, solution, and camera view.

### 5.2 Vertex projection

Next, each vertex of the reference body model is multiplied with the transformation matrix  $\chi_c^j$  of its body joint to produce the motion effect and camera projection. This process is repeated for all the solutions encoded in the population of the algorithm. The computational cost of this function is given by the high number of vertices of the reference model that are multiplied with the transformation matrices for each camera view and solution. Fortunately, these

multiplications can be computed concurrently for every vertex, camera view and solution on the GPU. A GPU kernel function computes the vertex projections by means of a 3D grid of threads. The first dimension is devoted to represent every vertex of the skin, the second dimension handles the projection for each of the camera views, and the third dimension represents each of the solutions of the population. Thereby, the kernel handles  $N_v \cdot N_c \cdot N_s$  compute threads to project all the vertices for all the cameras and all the solutions. Eventually, each vertex projection results in a  $(x, y)$  point which is filled around with a small patch to obtain a closed silhouette projection, as described in Sect. 3.2. This kernel is time consuming since it comprises a very high number of threads, with multiple global memory loads and stores.

According to the CUDA programming guide [27], it is essential to guarantee the coalescing of global memory accesses to achieve maximum performance. Global memory loads and stores by threads are coalesced by the device into as few transactions as possible. Therefore, we guarantee that parallel threads running the same instruction access to consecutive locations in the global memory, which is the most favorable access pattern. This happens when loading the vertices from the reference pose  $\mathcal{V}$ , since consecutive threads compute consecutive vertices projections. Moreover, it is also more efficient to represent vertices in  $\mathcal{V}$  using a structure of arrays rather than using an array of structures, to improve the memory access pattern. Consequently, the vertices are stored as  $[x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n], [z_1, z_2, \dots, z_n]$  rather than  $[x_1, y_1, z_1], [x_2, y_2, z_2], \dots, [x_n, y_n, z_n]$ .

### 5.3 Fitness evaluation

The fitness function measures the degree of overlap between the projected model and the foreground images using the pixelwise XOR operation (see Eqs. 6 and 7). The GPU can be used to compute this process in parallel by means of a 2D grid of threads whose dimension depends on the image size ( $W \times H$ ) and the number of cameras  $N_c$ . Thereby, the kernel handles  $W \cdot H \cdot N_c \cdot N_s$  compute threads. This kernel computes a very simple XOR function among pixels but it comprises a massive number of threads. Furthermore, there are SIMD instructions available in CUDA that allow us to process multiple pixels at once. Specifically, the `__vabsdiff4()` function allows for evaluating the XOR on four pixels simultaneously, behaving similar to the SSE instructions.

Memory coalescing is achieved by consecutive threads computing consecutive pixels of the images, both when loading pixels from the foreground and projection images and when storing the XOR result. Finally, the results of the XOR function are summed in parallel, which is known as a

<sup>1</sup> Detailed information about the MMOCAP implementation, the GPU kernels source code and experimental results is available at: <http://www.uco.es/grupos/kdis/wiki/MMOCAP>.



reduction operation, to determine the error in the overlapping of the images.

#### 5.4 Multi-GPU

Similarly to the multi-core CPU approach presented in Sect. 4.2, we can take advantage of the presence of multiple GPU devices. The population of solutions to be evaluated can be divided into multiple chunks that are delegated to several GPUs. Therefore, each GPU is responsible for the evaluation of  $(N_s/N_{GPUs})$  solutions. Switching the compute context associated to the GPU device is as simple as using the instruction `cudaSetDevice(deviceID)`.

Moreover, the process of computing the evaluation using multiple GPUs is completely independent one from another. Therefore, there is no interdependency and communications between the GPUs during the evaluation process.

## 6 Results

The goal of our experimentation is two fold. On the one hand, we aim at analyzing the speedup obtained by each parallelization strategy. On the other hand, we examine different parameter configurations to determine the one with the most appropriate trade-off between accuracy and runtime.

The rest of this Section is structured as follows. First, the experimental setup and settings of the experiments are presented. Then, the experiments conducted to determine the speedup of the proposed parallelization strategies are presented. Finally, a trade-off evaluation of the method performance is analyzed and discussed.

### 6.1 Experimental setup

The experiments were run on a machine equipped with an Intel Core i7-3820 quad-core processor running at 3.6 GHz and 32 GB of DDR3-1600 host memory. The video cards used were two dual-GPU NVIDIA GTX 690 equipped each one with 4 GB of GDDR5 video RAM and 3,072 CUDA cores. The host operating system was GNU/Linux Ubuntu 12.10-64 bit along with CUDA 5.5, NVIDIA drivers 310.40, and GCC compiler 4.6.3.

The experimental study has been carried out using the HumanEva-I dataset [4], which has been actively used in the community in the last years [41, 42, 43]. It contains 7 calibrated video streams recorded at 60 FPS of 4 subjects performing different common actions (e.g. walking, jogging, gesturing, etc.). During the recordings, the subjects wore reflective markers placed at key positions of the

anatomy that were captured by a motion capture system. We have selected the *walking* and *gesturing* sequences of subjects *S1*, *S2* and *S3* for our evaluation.

A three-dimensional model of each subject has been created using the *makehuman* software [44]. For each video sequence, the model has been manually initialized to fit the subject in the first frame. Given the model in its initial position, we added points to the skin model corresponding to the locations of the reflective markers. Therefore, the error in subsequent frames can be obtained as the distance from these points to their ground-truth positions. As proposed by the creators of HumanEva, the error metric employed is the averaged absolute distance between the real positions of the  $n$  markers being tracked  $X$ , and their estimated positions  $\hat{X}$ :

$$d(X, \hat{X}) = \frac{1}{n} \sum_{i=1}^n \|x_i - \hat{x}_i\|. \quad (8)$$

Equation 8 provides an error measure in a single frame of the sequence. It is employed to calculate the tracking error of a complete sequence as the average of all its frames.

Finally, we have employed the CMAES algorithm, which has recently been reported to obtain the best results [13] for the MMOCAP problem.

### 6.2 Speedup analysis

The purpose of this experimentation is to analyze the performance, accuracy and scalability of the parallelization strategies. In addition, we analyze the performance of each strategy with regard to the body mesh resolution. The higher the number of model vertices, the more accurate its projection is, but also, more computing time is required. So, we are interested in determining the number of vertices to achieve an appropriate trade-off between accuracy and performance.

For that purpose, each parallelization strategy has been tested in each one of the selected HumanEva sequences 10 times, using 500, 1500, 3000 and 5000 fitness evaluations, and body meshes with 27393, 20544 and 13695 vertices. The CMAES algorithm has been run using the parameters indicated in [13].

Table 1 shows the average runtime (in milliseconds) of each strategy in evaluating a video frame, i.e., the time employed in evaluating all the solutions plus the time employed by the CMAES algorithm. Additionally, the table shows the average runtime employed exclusively in evaluating the solutions, and the speedups achieved as compared with the CPU naive approach. Each column represents a computation strategy whereas each row represents a configuration of the evolutionary algorithm regarding the number of evaluations to compute. Results are grouped by each of the three mesh resolution sizes.

**Table 1** Computing times (in ms) employed by each strategy in evaluating a frame, the fraction of that time employed in evaluating the solutions, and the speedup as compared with the CPU naive approach

Number of evaluations	Evaluation scope	Configuration setup						
		CPU naive	CPU SSE	CPU threads	CPU threads SSE	1 GPU	2 GPUs	4 GPUs
27,393 vertices								
500	Frame	2,061	1,080 (1.9 ×)	638 (3.2×)	332 (6.2×)	90 (22.9×)	56 (36.8×)	37 (55.7×)
	Solutions	2,046	1,065 (1.9×)	623 (3.3×)	318 (6.4×)	80 (25.6×)	45 (45.5×)	26 (78.7×)
1,500	Frame	6,195	3,228 (1.9×)	1,840 (3.4×)	941 (6.6×)	258 (24.0×)	148 (41.9×)	92 (67.3×)
	Solutions	6,141	3,173 (1.9×)	1,787 (3.4×)	887 (6.9×)	235 (26.1×)	125 (49.1×)	69 (89.0×)
3,000	Frame	12,188	6,293 (1.9×)	3,667 (3.3×)	1,869 (6.5×)	514 (23.7×)	293 (41.6×)	179 (68.1×)
	Solutions	12,076	6,184 (2.0×)	3,556 (3.4×)	1,767 (6.8×)	470 (25.7×)	249 (48.5×)	135 (89.5×)
5,000	Frame	20,569	10,491 (2.0×)	6,174 (3.3×)	3,119 (6.6×)	852 (24.1×)	484 (42.5×)	294 (70.0×)
	Solutions	20,387	10,292 (2.0×)	5,979 (3.4×)	2,941 (6.9×)	782 (26.1×)	414 (49.2×)	224 (91.0×)
20,544 vertices								
500	Frame	2,011	1,011 (2.0×)	562 (3.6×)	283 (7.1×)	81 (24.8×)	49 (41.0×)	33 (60.9×)
	Solutions	1,996	996 (2.0×)	547 (3.6×)	268 (7.4×)	70 (28.5×)	38 (52.5×)	22 (90.7×)
1,500	Frame	5,995	2,982 (2.0×)	1,702 (3.5×)	858 (7.0×)	235 (25.5×)	134 (44.7×)	84 (71.4×)
	Solutions	5,943	2,930 (2.0×)	1,649 (3.6×)	801 (7.4×)	212 (28.0×)	111 (53.5×)	61 (97.4×)
3,000	Frame	11,984	5,909 (2.0×)	3,391 (3.5×)	1,690 (7.1×)	452 (26.5×)	254 (47.2×)	158 (75.8×)
	Solutions	11,883	5,795 (2.1×)	3,272 (3.6×)	1,586 (7.5×)	408 (29.1×)	210 (56.6×)	114 (104.2×)
5,000	Frame	20,051	9,870 (2.0×)	5,647 (3.6×)	2,799 (7.2×)	747 (26.8×)	419 (47.9×)	259 (77.4×)
	Solutions	19,857	9,687 (2.0×)	5,450 (3.6×)	2,614 (7.6×)	678 (29.3×)	350 (56.7×)	190 (104.5×)
13,695 vertices								
500	Frame	1,706	825 (2.1×)	475 (3.6×)	226 (7.5×)	71 (24.0×)	42 (40.6×)	29 (58.8×)
	Solutions	1,692	810 (2.1×)	461 (3.7×)	210 (8.1×)	60 (28.2×)	31 (54.6×)	18 (94.0×)
1,500	Frame	5,128	2,441 (2.1×)	1,419 (3.6×)	672 (7.6×)	210 (24.4×)	118 (43.5×)	75 (68.4×)
	Solutions	5,072	2,386 (2.1×)	1,366 (3.7×)	621 (8.2×)	187 (27.1×)	95 (53.4×)	53 (95.7×)
3,000	Frame	10,150	4,826 (2.1×)	2,832 (3.6×)	1,341 (7.6×)	413 (24.6×)	230 (44.1×)	145 (70.0×)
	Solutions	10,032	4,711 (2.1×)	2,721 (3.7×)	1,224 (8.2×)	369 (27.2×)	186 (53.9×)	101 (99.3×)
5,000	Frame	17,002	8,029 (2.1×)	4,713 (3.6×)	2,203 (7.7×)	637 (26.7×)	351 (48.4×)	222 (76.6×)
	Solutions	16,810	7,834 (2.1×)	4,536 (3.7×)	2,006 (8.4×)	561 (30.0×)	280 (60.0×)	152 (110.6×)

The first column shows the times of the CPU naive approach, which demanded a minimum of 1.7 seconds for the simplest scenario: 500 evaluations and 13695 vertices. On the other hand, the most complex scenario (5000 evaluations and the highest number of vertices) demanded more than 20 seconds to process a single frame. It is shown that most of the frame evaluation time is devoted to the evaluation of the solutions. The second column shows the CPU times using the SSE instruction set to compute in parallel the vertex projection and the XOR function of the projected and captured images. The SSE instruction set enabled to approximately double the performance of the CPU naive code in all scenarios. The third column shows the CPU times using the multi-threading strategy and the population parallel approach. The multi-core CPU we used in the experimentation is comprised by four cores. Thereby, the performance obtained is nearly 4 times the CPU

naive sequential approach (small overhead is introduced due to thread creation/join procedure). The fourth column shows the performance of both CPU parallel approaches combined, resulting in significantly better performance. The frame evaluation time is reduced to 210 ms and 3.1 seconds, respectively, to the previous configuration scenarios. It is important to highlight that CMAES runtime is negligible when using CPU approaches, since the solutions' evaluation times are much higher in magnitude.

The remaining columns evaluate the performance of the GPU-based approach using one, two, and four GPU devices. The single GPU performance significantly reduces the computation time in all scenarios, and performs faster than the best CPU-based approach. The performance when using two and four GPU devices is increased and allows for reducing even further the evaluation time. At this point, it is essential to differentiate between the frame and

solutions’ evaluation times. After parallelization, the solutions’ evaluation times have been significantly reduced, but CMAES represents now a high percentage of the total runtime. Therefore, we focus specifically on the solutions’ evaluation times. It is shown that the 500 evaluations’ scenarios are reduced to 80, 45, and 26 ms when using 1, 2 and 4 GPUs, respectively. These results define multiple speedups as we relate CPU and GPU times. For instance, the single GPU performance as compared with the CPU parallel + SSE is as low as  $3.3\times$  faster and as high as  $4.0\times$ . On the other hand, the 4-GPUs performance as compared with the naive CPU is as low as  $78\times$  faster and as high as  $110\times$ .

As for the GPU kernels computing time, the application was profiled using the NVIDIA Visual Profiler software. It reported that 3 % of the duration was devoted to the computation of the  $\chi_c^j$  matrices, 77 % to the vertex projection, 12 % to the fitness evaluation, 6 % to memory initialization and 2 % to the memory transfers between host and devices memories.

As can be seen, the efficiency increases as the number of evaluations increase, which means the better performance of the GPU-based solutions on high demanding and complex configurations. However, we should also note the limitations of the GPUs configuration. The scalability to multiple GPUs is appropriate when splitting computation from one to two GPUs, but efficiency is reduced after using four GPUs. This decrease in the efficiency is due to the overhead and synchronization times among the multiple GPUs. Similar behavior is shown in computer games when using multiple GPUs.

Performance on video rendering and analysis is usually measured as the number of frames per second (FPS) that the system is capable of processing. Therefore, we should also report results in such terms. FPS values are obtained by means of the inverse of the total frame evaluation time.

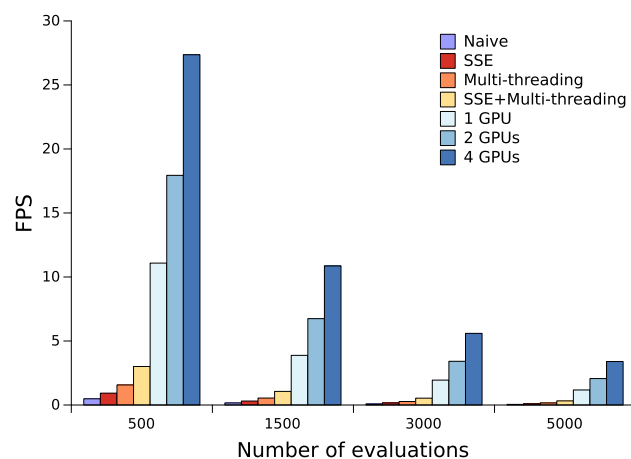


Fig. 3 FPS performance for different number of evaluations

Figure 3 shows the FPS for the different number of evaluations using a mesh size with 27393 vertices. Parallelization using GPUs is shown to perform as fast as 27.3 FPS, which is significantly faster than the naive CPU performance at 0.5 FPS, or the multi-threading CPU with SSE at 3.0 FPS for the same scenario when using 500 evaluations. Thereby, GPUs state a major step on increasing computation efficiency of the frame evaluation.

However, computation power of GPUs is still not enough to achieve real-time performance. Video sequences were originally recorded at 60 FPS, and therefore we should expect a compute system capable of processing at such high speeds. Nevertheless, advances on hardware manufacturing industry make us believe that the goal of real-time performance may be achieved within few years. Moreover, real-time performance may be also achieved using more GPU devices and by distributing computation. However, this would impact in the economic costs of the system as it would require to buy additional hardware devices. Therefore, we provide readers an idea on the best option to choose according to their computing time requirements and available budget.

### 6.2.1 Comparison with OpenGL

OpenGL is a widely used general purpose rendering engine that has been employed in the MMOCAP problem for rendering the models [45, 46, 47]. Thus, it is important to compare the performance of the proposed method with an OpenGL implementation for the same task.

In an initial step, the proposed OpenGL approach uploads the foreground images as textures to the GPU and creates vertex buffer objects for the vertices constituting the model. This reduces to minimum the CPU–GPU intercommunication. For each model to be evaluated, we first render the body model (as triangle meshes) in a texture buffer. Then, the texture buffer and the foreground texture are both applied to a quad covering the whole image. When applying the texture, a fragment shader computing the XOR function is employed. The fragment shader uses an

Table 2 Computing times employed for rendering a model and computing the XOR once

Method	Proj. (ms)	XOR (ms)	Total (ms)	Speedup
CPU	0.27533	0.27129	0.54662	–
CPU <sub>SSE</sub>	0.23171	0.02872	0.26043	2.10×
OpenGL	0.17992	0.05925	0.23917	2.29×
CPU <sub>thrds</sub>	0.07024	0.07010	0.14034	3.90×
CPU <sub>thrds+SSE</sub>	0.06487	0.00820	0.07307	7.48×
1 GPU	0.01796	0.00269	0.02065	26.47×
2 GPUs	0.00908	0.00136	0.01044	52.38×
4 GPUs	0.00458	0.00068	0.00526	103.91×

**Table 3** Error rate (in meters) for multiple configurations: number of evaluations and number of vertices

Video	27393 vertices				20544 vertices				13695 vertices			
	500	1,500	3,000	5,000	500	1,500	3,000	5,000	500	1,500	3,000	5,000
s1_gestures	0.0323	0.0317	0.0318	0.0310	0.0377	0.0390	0.0371	0.0381	0.0387	0.0386	0.0392	0.0394
s1_walking	0.0830	0.0486	0.0447	0.0428	0.0742	0.0541	0.0428	0.0467	0.0642	0.0550	0.0437	0.0442
s2_gestures	0.0784	0.0571	0.0498	0.0500	0.0833	0.0750	0.0669	0.0673	0.0828	0.0714	0.0730	0.0599
s2_walking	0.0696	0.0508	0.0376	0.0372	0.0716	0.0437	0.0383	0.0354	0.0640	0.0479	0.0504	0.0378
s3_gestures	0.0468	0.0379	0.0373	0.0372	0.0609	0.0393	0.0412	0.0393	0.0613	0.0400	0.0400	0.0397
s3_walking	0.1118	0.0730	0.0682	0.0665	0.1179	0.0799	0.0690	0.0675	0.1125	0.0981	0.0811	0.0787
Avg. Error	0.0703	0.0499	0.0449	0.0441	0.0743	0.0552	0.0492	0.0490	0.0706	0.0585	0.0546	0.0500
Ranks	9.50	4.83	2.83	1.50	10.67	7.33	4.83	4.50	10.50	8.00	7.50	6.00

atomic counter (option added in OpenGL 4.2) to count the number of pixels in both images that are different. Finally, the only value to be passed from GPU to CPU is the atomic counter (4 bytes). To take advantage of the parallelization capabilities of the GPU, our GPU implementation renders multiple models simultaneously. The code employed is publicly available from <http://www.uco.es/grupos/kdis/wiki/MMOCAP>.

Table 2 shows the computing times required to project and computes the XOR function once (in one view) for all the methods (including the CPU–GPU memory transfer times). In particular, for the OpenGL approach, the total time is divided into 75 % for rendering the texture and the rest for computing the XOR. As can be seen, the OpenGL approach is, in general, less competitive than the other approaches. Moreover, it is also interesting to highlight the performance of the SSE instructions when applied to the XOR.

An alternative OpenGL implementation employing points instead of triangles (using the same number of points than triangle vertices) has been also tested. The result is that the point-based implementation is significantly slower than the triangle-based one. A possible explanation is that the point primitive is less optimized than the triangle primitive in modern graphic cards. With regards to quality of the generated silhouettes with both methods, it is worth to mention that tests were run using the maximum resolution (27393 vertices), which implies small triangles since the vertices are very close to each other. So, the quality of the point-based silhouette is very similar to the triangle-based one. An example of the silhouettes obtained with our method can be seen in Fig. 2(c). The code available online lets the user to test both implementations.

Finally, it must be mentioned that it is possible to use multiple GPUs via SLI. However, this option was tested obtaining worse performance when enabled. Also, while some cards allow specific vendor extensions for OpenGL parallelization, this feature is limited to specific high end

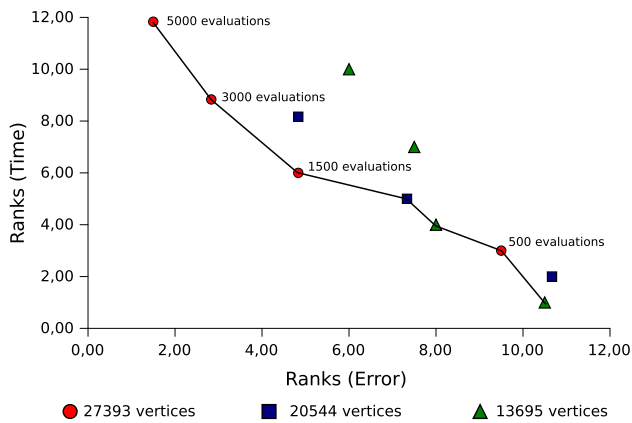
cards (e.g. the NVIDIA Quadro cards), whereas our GPUs are regular ones.

### 6.3 Error analysis

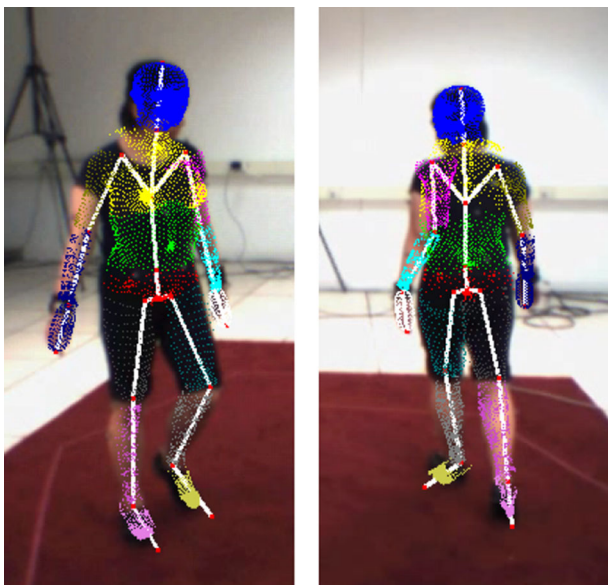
Accuracy and runtime of frame evaluation is a conflicting problem. To obtain more accurate models, the complexity and resolution of the 3D body model, and the number of evaluations of the algorithm are increased. However, this involves a higher number of calculations that conduct to longer runtime. Therefore, it is necessary to achieve a trade-off between the accuracy and the frame evaluation time.

Table 3 shows the error rate obtained on the six sequences of the HumanEva-I dataset. Errors are measured as indicated in Eq. 8 as the averaged absolute distance (in meters) between the real positions of the markers being tracked and their estimated positions. All experiments were repeated 10 times with different seeds and the mean error is provided. Each column belongs to a given configuration regarding the mesh size and the number of evaluations. These results correspond to the evaluation times shown in Table 1. Results clearly indicate that increasing the number of evaluations reduces the error, whereas decreasing the resolution of the mesh increases the error rate. The bottom rows show the average error for the six sequences and the ranks for each configuration. Rank values are obtained according to the Friedman’s statistical test [48, 49], which allow to perform a direct comparison of performance. The lower the rank value, the better the performance of the configuration. The best-ranked solution, which also obtains the lowest average error is the 5000 evaluations solution with the highest mesh size. However, this configuration is also the slowest according to Table 1 since it involves a very high number of evaluations. Therefore, we should find other configurations that provide a better trade-off.

The conflicting problem of obtaining the best accuracy at the lower computational cost can be addressed as a



**Fig. 4** Time vs. error plot of configurations. Solutions from/to the optimal Pareto front are linked together



**Fig. 5** Tracking results obtained in the walking sequence S1

multi-objective problem. Multi-objective optimization is concerned with the simultaneous optimization of more than one objective function [50]. Therefore, there is no single best solution to the problem, but a set of non-dominated solutions known as Pareto optimal front. Given a set of objective functions  $F = \{f_1, f_2, f_3, \dots, f_n\}$ , a solution  $s$  belongs to the front if there is no other solution  $s'$  that dominates it. A solution  $s'$  dominates  $s$  if and only if  $f_i(s')$  is equal or better than  $f_i(s) \forall f \in F$  and  $f_i(s')$  is strictly better than  $f_i(s)$  for at least one objective.

Figure 4 shows the multiple configurations evaluated, located according to their ranking with regard to the runtime and the error rate. Solutions belonging to the Pareto front are linked together. All solutions belonging to the Pareto front are said to be equally good. However, it is

known that extreme solutions are fast and inaccurate or slow and very accurate. Eventually, a single configuration with good trade-off should be provided by default. Thus, we would recommend the one having 1500 evaluations and 27393 vertices because it provides both accurate results and relatively fast runtime. This configuration agrees with the proposed in [13] as the best performance solution.

A visual example of the results obtained by the algorithm is presented in Fig. 5. It shows the tracking results obtained for two frames of the walking sequence of the subject S1 when employing the recommended configuration with 1500 evaluations. The body joints are linked with a white line and the vertices of the mesh are shown colored.

## 7 Conclusions

This paper presented an efficient and parallelizable approach to evaluate the solutions in the MMOCAP problem. Our approach consists in approximating the triangle body meshes by rectangular patches that are easily drawn and computed. In addition, strategies to parallelize the computation both in CPUs and GPUs were proposed. First, we proposed a strategy based on the CPU’s Streaming SIMD Extensions (SSE) instruction set, which demonstrated double performance. Second, we proposed an strategy using multi-threading on multi-core CPUs, which showed to speedup model evaluation up to 4 times. Third, we presented a GPU strategy scalable to multiple devices. A total of 4 GPUs were used to collaborate, providing a speedup of up to 110× faster than the naive CPU code. The parallelization approaches proposed also demonstrated better performance than an efficient OpenGL implementation.

Moreover, we experimented multiple algorithm’s configuration and body mesh resolution sizes, which allowed for analyzing the performance impact of varying the number of evaluations and the number of body model vertices. Accuracy and runtime were two conflicting objectives for the MMOCAP problem, and we established a Pareto front of solutions offering different levels of trade-off. Eventually, the user is allowed for selecting the configuration setup which best matches his needs in terms of obtaining accurate but slow solutions, or fast and less accurate solutions. A trade-off solution producing both accurate and fast results is proposed as recommended configuration.

As for the future work, it would be interesting to compare the performance of the proposal with FPGA-based implementations that allow for efficient single-bit operations.

**Acknowledgments** This research was supported by the Spanish Ministry of Science and Technology, projects TIN-2011-22408 and

TIN-2012-32952, and by FEDER funds. This research was also supported by the Spanish Ministry of Education under FPU grant AP2010-0042.

## References

- Multon, F., Kulpa, R., Hoyet, L., Komura, T.: Interactive animation of virtual humans based on motion capture data. *J. Vis. Comput. Animat.* **20**(5–6), 491–500 (2009)
- Zhou, H., Huosheng, H.: Human motion tracking for rehabilitation—a survey. *Biomed. Signal Process. Control.* **3**(1), 1–18 (2008)
- Moeslund, T.B., Hilton, A., Krüger, V.: A survey of advances in vision-based human motion capture and analysis. *Comput. Vis. Image Underst.* **104**, 90–126 (2006)
- Sigal, L., Balan, A.O., Black, M.J.: Humaneva: synchronized video and motion capture dataset and baseline algorithm for evaluation of articulated human motion. *Int. J. Comput. Vis.* **87**, 4–27 (2010)
- Isard, M., Blake, A.: Condensation—conditional density propagation for visual tracking. *Int. J. Comput. Vis.* **29**, 5–28 (1998)
- Deutscher, J., Reid, I.: Articulated body motion capture by stochastic search. *Int. J. Comput. Vis.* **61**(2), 185–205 (2005)
- Corazza, S., Mündermann, L., Chaudhari, A., Demattio, T., Cobelli, C., Andriacchi, T.: A markerless motion capture system to study musculoskeletal biomechanics: visual hull and simulated annealing approach. *Ann. Biomed. Eng.* **34**(6), 1019–1029 (2006)
- John, M., Michael I.: Partitioned sampling, articulated objects, and interface-quality hand tracking. In: Proceedings of the 6th European Conference on Computer Vision-Part II, ECCV '00, pp 3–19. Springer, London (2000)
- Jan, B., Florian, E., Michael B.: Evaluation of hierarchical sampling strategies in 3D human pose estimation. In: Proceedings of the 19th British Machine Vision Conference, pp. 1–10 (2008)
- Lozano, M., Molina, D., Herrera, F. (eds.): Special issue on scalability of evolutionary algorithms and other metaheuristics for large-scale continuous optimization problems. *Soft computing*, vol. 15. Springer, Berlin/Heidelberg (2011)
- John, V., Trucco, E., Ivekovic, S.: Markerless human articulated tracking using hierarchical particle swarm optimisation. *Image Vis. Comput.* **28**(11), 1530–1547 (2010)
- Zhao, X., Liu, Y.: Generative tracking of 3D human motion by hierarchical annealed genetic algorithm. *Pattern Recognit.* **41**(8), 2470–2483 (2008)
- Yeguas-Bolivar, E., Muñoz-Salinas, R., Medina-Carnicer, R., Carmona-Poyato, A.: Comparing evolutionary algorithms and particle filters for markerless human motion capture. *Appl. Soft Comput.* **17**, 153–166 (2014)
- Hansen, N.: The CMA evolution strategy: a comparing review. In: Lozano, J.A., Larranaga, P., Inza, I., Bengoetxea, E., (eds.) *Towards a new evolutionary computation. Advances on Estimation of Distribution Algorithms*, pp. 75–102. Springer, Berlin (2006)
- Kenneth, V., Price, R.M.S., Jouni A.L.: Differential evolution a practical approach to global optimization. In: *The Differential Evolution Algorithm*, pp. 37–134. Natural Computing Series. Springer, Berlin (2005)
- Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proceedings IEEE International Conference on Neural Networks, vol. 4, pp. 1942–1948 (1995)
- Chang, I.-C., Lin, S.-Y.: 3D human motion tracking based on a progressive particle filter. *Pattern Recognit.* **43**(10), 3621–3635 (2010)
- Gall, J., Rosenhahn, B., Brox, T., Seidel, H.-P.: Optimization and filtering for human motion capture. *Int. J. Comput. Vis.* **87**(1–2), 75–92 (2010)
- Cappozzo, A., Ugo D.C., Alberto L., Lorenzo C.: Human movement analysis using stereophotogrammetry: Part 1: theoretical background. *Gait Posture.* **21**(2), 186–196 (2005)
- Chiari, L., Ugo D.C., Alberto L., Aurelio C.: Human movement analysis using stereophotogrammetry: Part 2: Instrumental errors. *Gait Posture.* **21**(2), 197–211 (2005)
- Ugo, D.C, Alberto, L., Lorenzo, C., Aurelio, C.: Alberto Leardini, Lorenzo Chiari, and Aurelio Cappozzo. Human movement analysis using stereophotogrammetry: Part 4: assessment of anatomical landmark misplacement and its effects on joint kinematics. *Gait Posture.* **21**(2), 226–237 (2005)
- Leardini, A., Chiari, L., Ugo D.C., Aurelio C.: Human movement analysis using stereophotogrammetry: Part 3. soft tissue artifact assessment and compensation. *Gait Posture.* **21**(2), 212–225 (2005)
- Chitty, D.M.: Fast parallel genetic programming: Multi-core cpu versus many-core gpu. *Soft Comput.* **16**(10), 1795–1814 (2012)
- Creel, M., Goffe, W.L.: Multi-core CPUs, clusters, and grid computing: a tutorial. *Comput. Eco.* **32**(4), 353–382 (2008)
- Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Dist. Comput.* **68**(10), 1370–1380 (2008)
- Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. *Comput. Gr. Forum* **26**(1), 80–113 (2007)
- NVIDIA Corporation. NVIDIA CUDA Programming and Best Practices Guide. <http://www.nvidia.com/cuda> (2014)
- Rymut, B., Kwolek, B.: GPU-supported object tracking using adaptive appearance models and particle swarm optimization. In: Proceedings of the 2010 international conference on Computer vision and graphics: Part II, ICCVG'10, pp. 227–234 (2010)
- Krzyszowski, T., Kwolek, B., Wojciechowski, K.: GPU-accelerated tracking of the motion of 3D articulated figure. *Comput. Vis. Gr.* pp. 155–162 (2010)
- Luca, M., Spela, I., Stefano, C.: Markerless articulated human body tracking from multi-view video with GPU-PSO. In: Gianluca, T., Andy, M.T., Julian F.M., (eds.) *Evolvable systems: from biology to hardware*, vol. 6274, Lecture Notes in Computer Science, pp. 97–108 (2010)
- Rymut, B., Kwolek, B., Krzyszowski, T.: GPU-accelerated human motion tracking using particle filter combined with PSO. *Advanced concepts for intelligent vision systems. Lect. Notes Comput. Sci.* **8192**, pp. 426–437 (2013)
- Zhou, Y., Tan, Y.: GPU-based parallel particle swarm optimization. In: IEEE Congress on Evolutionary Computation, pp. 1493–1500 (2009)
- Ugolotti, R., Youssef S.G.N., Pablo M., Lvekovi, P., Luca M., Stefano C.: Particle swarm optimization and differential evolution for model-based object detection. *Appl. Soft Comput.* **13**(6), 3092–3105 (2013)
- Zheng, Z., Hock, S.S.: Cuda acceleration of 3D dynamic scene reconstruction and 3D motion estimation for motion capture. In: IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS), pp. 284–291 (2012)
- Zhang, Z., Hock, S.S., Chee K.Q., Jixiang, S.: GPU-accelerated real-time tracking of full-body motion with multi-layer search. *IEEE Trans. Multimed.* **15**, 106–119 (2013)
- Ganapathi, V., Plagemann, C., Koller, D., Thrun, S.: Real time motion capture using a single time-of-flight camera. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 755–762 (2010)
- Diaz-Mas, L., Madrid-Cuevas, F.J., Muñoz-Salinas, R., Carmona-Poyato, A., Medina-Carnicer, R.: An octree-based method for shape from inconsistent silhouettes. *Pattern Recognit.* **45**(9), 3245–3255 (2012)

38. Diaz-Mas, L., Muñoz-Salinas, R., Medina-Carnicer, R., Madrid-Cuevas, F.J.: Shape from silhouette using dempster-shafer theory. *Pattern Recognit.* **43**(6), 2119–2131 (2010)
39. Muñoz-Salinas, R., Yeguas-Bolivar, E., Diaz-Mas, L., Medina-Carnicer, R.: Shape from pairwise silhouettes for plan-view map generation. *Image Vis. Comput.* **30**(2), 122–133 (2012)
40. Horprasert, T., Harwood, D., Davis, L.S.: A statistical approach for real-time robust background subtraction and shadow detection. In: 7th IEEE International Conference on Computer Vision, Frame Rate Workshop (ICCV '99), pp. 1–19 (1999)
41. Grégory, R., Carlos O.-U., Martínez-del Rincón, J.: A spatio-temporal 2D-models framework for human pose recovery in monocular sequences. *Pattern Recognit.* **41**, 2926–2944 (2008)
42. Sundaresan, A., Chellappa, R.: Model driven segmentation of articulating humans in laplacian eigenspace. *IEEE Trans. Pattern Anal. Mach. Intell.* **30**, 1771–1785 (2008)
43. Zhao, X., Liu, Y.: Generative tracking of 3d human motion by hierarchical annealed genetic algorithm. *Pattern Recognit.* **41**, 2470–2483 (2008)
44. Manuel, B., Marc F., Jose C.: Makehuman Team. <http://www.makehuman.org/> (2014)
45. Maeda, T., Yamasaki, T., Aizawa, K.: Model-based analysis and synthesis of time-varying mesh. *Lect. Notes Comput. Sci.* **5098**, 112–121 (2008)
46. Schmaltz, C., Rosenhahn, B., Brox, T., Weickert, J., Wietzke, L., Sommer, G.: Dealing with self-occlusion in region based motion capture by means of internal regions. *Lect. Notes Comput. Sci.* **5098**, 102–111 (2008)
47. Shaheen, M., Gall, J., Strzodka, R., Van G.L., Seidel, H.P.: A comparison of 3D model-based tracking approaches for human motion capture in uncontrolled environments. *Appl. Comput. Vis.* pp. 1–8 (2009)
48. Demšar, J.: Statistical comparisons of classifiers over multiple data sets. *J. Mach. Learn Res.* **7**, 1–30 (2006)
49. García, S., Fernández, A., Luengo, J., Herrera, F.: Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: experimental analysis of power. *Inf. Sci.* **180**(10), 2044–2064 (2010)
50. Kalyanmoy D.: Multi-objective optimization. In: Edmund, K.B., Graham, K., (eds.) *Search methodologies*, pp. 273–316. Springer, Berlin (2005)