Real-time multiview human pose tracking using graphics processing unit-accelerated particle swarm optimization

Boguslaw Rymut[2] and Bogdan Kwolek[1,*,†]

[1] *AGH University of Science and Technology, Al. A. Mickiewicza 30, 30-059 Kraków, Poland*
[2] *Rzeszów University of Technology, Al. Powstańców Warszawy 12, 35-959 Rzeszów, Poland*

SUMMARY

This paper describes how to achieve real-time tracking of 3D human motion using multiview images and graphics processing unit (GPU)-accelerated particle swarm optimization. The tracking involves configuring the 3D human model in the pose described by each particle and then rasterizing it in each 2D plane. The Compute Unified Device Architecture threads rasterize the columns of the triangles and perform the summing of the fitness values of pixels belonging to the processed columns. Such a parallel particle swarm optimization (PSO) exhibits the level of parallelism that allows us to effectively utilize the GPU resources. Image acquisition and image processing are multi-threaded and run on CPU in parallel with PSO-based searching for the best pose. Owing to such task decomposition the tracking of the full human body can be performed at rates of 12 frames per second. For a PSO consisting of 1000 particles and executing 10 iterations the GPU achieves an average speedup of 12 over the CPU. Using marker-less motion capture system consisting of four calibrated and synchronized cameras, the efficiency comparisons were conducted on four CPU cores and four GTX GPUs on two cards.

KEY WORDS: GPGPU; CUDA; GPU-accelerated PSO; Model-based 3D tracking and pose recovery

1. INTRODUCTION

The first GPUs were built as graphics accelerators, supporting only specific fixed-function pipelines. Since the GPUs could only be programmed through a graphics rendering interface, they provided a very limited flexibility, even for those who knew software interfaces to graphics hardware such as OpenGL. Starting in the late 1990s, the GPUs became increasingly programmable and versatile architectures that can perform a wide range of data-parallel operations. Modern GPUs are throughput-oriented many-core processors that offer very high peak computational capabilities, and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data can be done in parallel. Full utilization of their potential requires fine-grained parallelism and exposing large amounts of computations with sufficient regularity of execution paths and memory access patterns. This means that it is crucial to extract data-parallelism and map it to the massive threading execution model advocated by GPUs.

Today's GPUs surpass the CPU's performance in many signal-processing-like applications [1]. Several recent studies report that general purpose GPUs (GPGPUs) are capable of obtaining significant speedups in comparison to homogeneous multi-core systems in the same price range. These papers initiated a passionate debate on the limits of GPU acceleration for various classes of applications [2]. A comparison of 14 various implementations showed speedups from $0.5\times$ to

*Correspondence to: Bogdan Kwolek, Department of Computer Science, Faculty of Computer Science, Electronics and Telecommunications, AGH University of Science and Technology, Al. A. Mickiewicza 30, 30-059 Krakow, Poland.
†E-mail: bkw@agh.edu.pl

$15\times$ (GPU over CPU). There is a common agreement that in order to achieve such speedups the algorithms to be executed on GPU should be carefully designed. Even though it is easier to program GPUs than ever, efficiently taking advantages of GPU resources still requires unique techniques.

CPUs are still the most frequently used hardware for image processing, given their versatility and tremendous speed. On the other hand, image processing algorithms are good candidates for GPU implementation, since many image processing operations have high inherent parallelism, which is frequently achieved through per-pixel operations. Many research reports confirmed this by showing GPU acceleration of many image processing algorithms [3]. A recent study [1] reports a 30 fold speed-up for low-level algorithms and up to 10 fold speed-up for high-level functions, which contain more overhead and many steps that are not easy to parallelize.

Non intrusive human pose tracking is a key issue in computer vision because of many possible applications in surveillance, human activity recognition, virtual reality, motion capture, etc. Due to variations in individual body shapes this is one of the most challenging problems in computer vision being at the same time one of the most computationally demanding tasks. Particle filtering is widely used in articulated motion tracking due to its ability to represent multi-modal probability distributions and to maintain multiple pose hypotheses. Several improvements of ordinary particle filter (PF) were proposed to achieve fast and reliable articulated motion tracking [4] as well as to obtain the initialization of the tracking [5]. 3D motion tracking can be perceived as dynamic optimization problem. Recently, particle swarm optimization (PSO) [6] has been successfully applied to achieve human motion tracking [7, 8]. In the approaches mentioned above, the motion tracking is achieved by a sequence of static PSO-based optimizations, followed by re-diversification of the particles to cover the possible poses in the next frame.

Although considerable work has been done on accelerating the PSO on GPU [9], very little of it is concerned with object tracking on GPU. In [10], a implementation of PSO in CUDA has been proposed. In [11], an approach that restricts the communication of a particle to its two closest neighbors and thus limits the communication between threads has been discussed. The authors of [12] compared three different variants of the PSO on GPU, but only parallelized the cost function. A PSO-based object tracking developed in [13] achieves about 40 fold speed-up of GPU over CPU. In [14], a multi-swarm PSO algorithm was selected to achieve a high degree of parallelism.

While extensive research has been done on human pose tracking, very little work has been carried out in real-time tracking. In [7] an approach to PSO-based full body human motion tracking on GPU and using single camera has been discussed. The 3D model with 26 DOF was constructed using cuboids, which were projected onto 2D plane and then rasterized in parallel. A single thread was responsible for matching the images containing the projected model with the extracted person. The tracking of the full human body was performed with 5 frames per second, whereas the speed-up of GTX280 over a CPU was about 15. A common approach to parallelize the PSO consists in executing a local swarm on every processor while optimizing the communication between the swarms. Mussi et al. [8] proposed an approach to articulated human body tracking from multi-view video using PSO running on GPU. Their implementation is far from real-time and roughly requires 7 seconds per frame. Recently, in [15] a framework for 3D model-based visual tracking using a GPU-accelerated particle filter has been presented. A hand was tracked using both synthetic and real videos. The authors reported a speedup of 9.5 and 14.1 against a CPU for image resolution of 96×72 and 128×96 using 900 and 1296 particles, respectively.

In this work we present an efficient algorithm that effectively utilizes the advantages of modern GPUs to achieve real-time full body tracking using a 3D human model. The 3D articulated human tracking is accomplished by a PSO algorithm running on GPUs. The presented approach to motion tracking follows the Black Box Optimization paradigm [16], according to which the search processes/particles investigate the hypothesis space of a model state in order to identify the hypothesis that optimally fits a set of observations. We show how the calculation of the fitness score, which is the most computationally demanding operation in the motion tracking, has been decomposed on the GPUs. The acquisition of the images and the image processing is executed in parallel on the CPUs. The human pose is estimated using four synchronized and calibrated cameras.

2. COMPUTATIONAL DEMANDS IN ARTICULATED MOTION TRACKING

The objective of articulated human tracking is to estimate the position and orientation of each limb of the human body undergoing tracking. The existing articulated tracking approaches can either be described as part-based bottom-up approaches or model-based generative top-down methods.

In bottom-up approaches different features describing the main body parts are first sought in the images and then assembled into a human body. Such approaches need to have robust part detectors. Since the publication of the Shotton et al. [17] a growing interest on model-free approaches can be observed. The method uses a random forest to classify features extracted from the depth image and to infer human poses. It is capable of capturing full-body motion at 30 fps in uncontrolled environments without imposing any requirements on the performer. It is also able to provide coherent pose estimations for occluded joints. The method uses Kinect, which is now a popular structured light device. However, it cannot recover 3D pose in strong sunlight. Moreover, structured-light based cameras like the Kinect have a limited field of view. The minimum range for the Kinect is about 0.6 m and the maximum range is somewhere between 4-5 m depending on how much error the application can tolerate.

The above-mentioned restrictions do not have multiview top-down approaches. Three dimensional model-based methods are generally more accurate in comparison to methods relying on learned mapping between pose exemplars and image primitives. All approaches to 3D model-based human pose tracking essentially rely on matching between the body model and image observations. This means that in such methods an evaluation function is calculated to measure similarities between projections of the 3-D model and the primitives extracted on the images. An articulated human body can be perceived as a kinematic chain consisting of at least eleven parts corresponding to key parts of the human body. Typically such a 3D human model consists of very simple geometric primitives like cylinders or truncated cones. On the basis of such geometrical primitives a lot of hypothetical body poses are generated to find the 3D pose that best matches the image descriptors. The computational overhead of 3D model-based methods arises from the necessity of searching in the high-dimensional space. Due to the high dimensionality of the search space, often a hierarchical search is conducted to reduce the amount of the computations. In order to cope with self-occlusions, multiple cameras are used frequently. In consequence, each of the 3D model configurations needs to be projected to each image plane, resulting in considerable computational demands.

The use of the particle filtering is by now a relatively standard approach in top-down approaches to 3D motion tracking [4]. The most important property of the particle filter is its ability to handle complex, non-Gaussian and multimodal posterior distributions. However, the number of particles required to adequately approximate the conditional density grows exponentially with the dimensionality of the state space. In particle filter-based human motion tracking each sample represents some hypothesized body pose. For a 3D model consisting of eleven geometric primitives we need around 26 parameters to describe the full body articulation. In such a high-dimensional space the particles usually do not cluster around the true pose and instead they migrate towards local maximas. In contrast, in PSO each particle follows simple position and velocity update equations. Thanks to interaction between particles a collective behavior of the swarm arises. It leads to the emergence of global and collective search capabilities, which allow the particles to gravitate towards the global extremum. In consequence, the number of the particles needed to obtain the assumed tracking accuracy is far smaller in comparison to ordinary particle filter. In [18] we demonstrated that a PSO built on 300 particles and executing 20 iterations allows us to obtain the tracking error smaller than 55 mm. The tracking was performed on image sequences with walking persons, which were acquired by four cameras with 25 fps.

While much effort has been devoted to developing functional likelihood models or objective functions, not much has been published on efficient implementations of PF-based or PSO-based motion tracking on GPU. Efficient implementations include work of Krzeszowski et al. [7], Mussi et al. [8], Brown & Capson [15], Rymut & Kwolek [19].

3. GPU COMPUTING

CUDA is a parallel computing platform and programming model invented by NVIDIA. Each function that is executed on the device is called a kernel. A CUDA kernel is executed by an array of threads. Blocks of threads are organized into one, two or three dimensional grid of thread blocks. Blocks are mapped to multiprocessors (MPs) and each thread is mapped to a single core. Within one thread block the threads are further divided in groups of 32 called warp. Such a group of threads within a block is launched together and usually executes together in a SIMD (Single Instruction, Multiple Data) fashion. When a warp is selected for execution, all active threads execute the same instruction but operate on different data. When one warp stalls, for example, because of a memory access, the warp scheduler can hide this latency by quickly switching to a ready warp. To achieve high computation efficiency, the number of threads residing on the MPs should be maximized to provide the warp scheduler with a large delivery of ready warps. A unique set of indices is assigned to each thread to determine to which block it belongs and its location inside it.

GPUs offer best performance gains when all processing cores are utilized and memory latency is hidden. In order to achieve this aim, it is common to launch a CUDA kernel with hundreds or thousands of threads to keep the GPU busy. The benefit of having multiple blocks per multiprocessor is that the scheduling hardware is capable to swap out a block that is waiting on a high-latency instruction and replace it with a block that has threads ready to execute. The context switch is very fast because the GPU does not have to store the state, as the CPU does when switching threads between being active and inactive. Thus, it is advantageous to have both high density of arithmetic instructions per memory access as well many more resident threads than GPU cores so that memory latency can be hidden. This permits the GPU to execute arithmetic instructions while certain threads are waiting for access to the global memory. Avoiding thread divergence is another technique to achieving high computation performance. The MPs can be programmed as scalar multiprocessors, but the members of a thread block that do not follow the same execution path are serialized.

Memory latency can be hidden by careful design of control flow as well as adequate design of kernels. The kernels can employ not only the global memory that resides off chip, but also they can use the shared memory residing on chip. This memory is shared between all the cores of the stream multiprocessor, see Fig. 1. Its latency is several times shorter than the latency of the global memory. Threads that are executing within the same block can cooperate using it, but threads from different block cannot cooperate via shared memory. Kernels only have read-only access to texture memory.

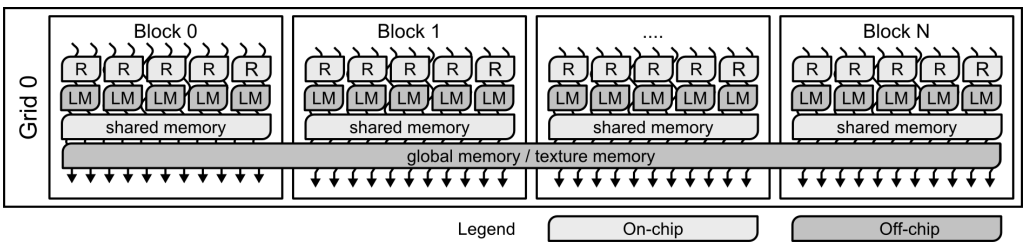

Figure 1. GPU thread and memory hierarchy. Threads are organized as a grid of thread blocks. Threads in a block are executed on the multiprocessor and have access to on-chip private registers (R) and shared memory. The global and local memories (LM) are off-chip.

Current GPU hardware can host tens of thousands concurrent threads, whereas the advanced PCs can run up to one hundred threads simultaneously. This is because in multi-CPU/multi-core systems, the number of threads physically residing in the hyperthreading hardware can not be larger than twice the number of physical cores. Switching between GPU threads does not introduce any overhead as the threads are residing in the hardware for their entire lifetime, whereas switching between CPU threads is costly since the scheduler loads the thread execution context from the RAM memory. Moreover, multi-CPU/multi-core systems require complex control logic to avoid data hazards. The widening performance gap between GPU and CPU can be attributed to the highly scalable nature of the GPU architecture.

4. PARALLEL PSO FOR OBJECT TRACKING

Particle Swarm Optimization (PSO) [6] is a bio-inspired meta-heuristic for solving complex optimization problems. The PSO is initialized with a group of random particles (hypothetical solutions) and then searches for optima by updating all particles locations. The particles move through the solution space and undergo evaluation according to some fitness function. Each particle iteratively evaluates the candidate solutions and remembers the personal best location with the best objective value found so far, making this information available to its neighbors. Particles communicate good positions to each other and adjust their own velocities and positions taking into account such good locations. Additionally each particle utilizes a best value, which can be:

- a global best that is immediately updated when a new best position is found by any particle in the swarm
- neighborhood best where only a specific number of particles is affected if a new best position is found by any particle in the sub-population

Typically, a swarm topology with the global best converges faster since all particles are attracted simultaneously to the best part of the search space. Neighborhood best permits parallel exploration of the search space and decreases the susceptibility of falling into local minima. However, such a topology slows down the convergence speed. Taking into account the faster convergence the topology with the global best has been selected for parallel implementation.

In the ordinary PSO algorithm the update of particle's velocity and position can be expressed by the following equations:

$$v_j^{(i)} \leftarrow w v_j^{(i)} + c_1 r_{1,j}^{(i)} (p_j^{(i)} - x_j^{(i)}) + c_2 r_{2,j}^{(i)} (p_{\mathrm{g},j} - x_j^{(i)}) \tag{1}$$

$$x_j^{(i)} \leftarrow x_j^{(i)} + v_j^{(i)} \tag{2}$$

where w is the positive inertia weight, $v_j^{(i)}$ is the velocity of particle i in dimension j, $r_{1,j}^{(i)}$ and $r_{2,j}^{(i)}$ are uniquely generated random numbers with the uniform distribution in the interval $[0.0,\ 1.0]$, $c_1,\ c_2$ are positive constants, $p^{(i)}$ is the best position that the particle i has found, p_{g} denotes best position that is found by any particle in the swarm.

The velocity update equation (1) has three main components. The first component, which is often referred to as inertia models the particle's tendency to continue the moving in the same direction. In effect it controls the exploration of the search space. The second component, called cognitive, attracts towards the best position $p^{(i)}$ previously found by the particle. The last component is referred to as social and attracts towards the best position p_{g} found by any particle. The fitness value that corresponds $p^{(i)}$ is called local best $p_{\mathrm{best}}^{(i)}$, whereas the fitness value corresponding to p_{g} is referred to as g_{best}. The ordinary PSO algorithm can be expressed by the following pseudo-code:

1. Assign each particle a random position in the problem hyperspace.
2. Evaluate the fitness function for each particle.
3. For each particle i compare the particle's fitness value with its $p_{\mathrm{best}}^{(i)}$.
   If the current value is better than the value $p_{\mathrm{best}}^{(i)}$, then set this value as the $p_{\mathrm{best}}^{(i)}$ and the current particle's position $x^{(i)}$ as $p^{(i)}$.
4. Find the particle that has the best fitness value g_{best}.
5. Update the velocities and positions of all particles according to (1) and (2).
6. Repeat steps $2-5$ until a stopping criterion is not satisfied (e.g. maximum number of iterations or a sufficiently good fitness value is not attained).

Our parallel PSO algorithm for object tracking consists of five main phases, namely initialization, evaluation, compute p_{best}, compute g_{best}, update and motion. At the beginning of each frame, in the initialization stage an initial position $x^{(i)} \leftarrow \mathcal{N}(p_{\mathrm{g}}, \Sigma)$ is assigned to each particle, given the location p_{g} that has been estimated in the previous frame. In the evaluation phase the fitness value of each particle is calculated using a cost function. The calculation of the matching score is the most time consuming operation of the tracking algorithm. The calculation of the matching score is discussed in Section 5.2, whereas the decomposition of this task into kernels is presented in Section 5.3. In the compute p_{best} stage the determining of $p_{\mathrm{best}}^{(i)}$ as well as $p^{(i)}$ takes place. This stage corresponds to operations from the point 3. of the presented above pseudo-code. The operations mentioned above are computed in parallel using available GPU resources, see Fig. 2. Afterwards, the g_{best} and its corresponding p_{g} are calculated in parallel. Finally, the update stage that corresponds to point 5. in the pseudo-code is done in parallel. That means that in our implementation we employ the parallel synchronous particle swarm optimization. The synchronous PSO algorithm updates all particle velocities and positions at the end of each optimization iteration. In contrast to synchronous PSO, the asynchronous algorithm updates particle positions and velocities continuously using currently accessible information.

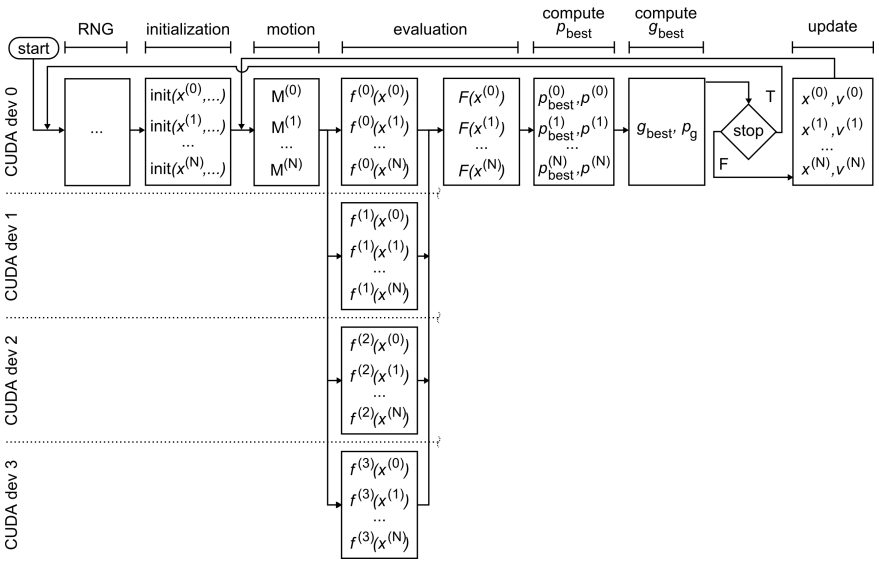

Figure 2. Decomposition of synchronous particle swarm optimization algorithm on GPU.

In order to decompose an algorithm into GPU we should identify data-parallel portions of the program and isolate them as CUDA kernels. In the initialization kernel we generate pseudo-random numbers using the curand library provided by the CUDA™ SDK. On the basis of the uniform random numbers we generate normally distributed pseudorandom numbers using Box Mueller transform based on trigonometric functions [20]. The normally distributed random numbers are generated at the beginning of each frame to re-distribute the particles around the pose in time $t-1$ and to calculate their velocities. Then the uniform random numbers r_1, r_2 for the optimal pose seeking are generated. This means that for every particle we generate $2 \times D \times K$ uniformly distributed random numbers, where D is dimension and K denotes the maximum number of iterations. They are stored in the memory and then used in the update kernel, see Fig. 2. At this stage the computations are done in $\lceil N/(2 \times W) \rceil$ blocks and W threads on each of them, where W denotes the number of cores per multiprocessor. In the compute p_{best} kernel and the update kernel the number of blocks is equal to $\lceil N/W \rceil$, whereas the number of threads in each block is equal to W. In the update kernel we constrain the velocities of the particles to the assumed maximal velocity values. In the motion stage the model's bone hierarchy is recursively traversed and the internal transformation matrices are updated according to the state vector of the particle.

5. GPU-ACCELERATED TRACKING OF HUMAN MOTION

At the beginning of this section we detail our approach to 3D model-based visual tracking of human motion. Afterwards, we present the cost function. Finally, we discuss the parallelization of the calculations of the cost function.

5.1. 3D Model-Based Human Motion Tracking

The articulated model of the human body has a form of kinematic chain consisting of 11 segments. The 3D model is constructed using truncated cones (frustums) that model the pelvis, torso, head, upper and lower arms and legs. The model has 26 DOF and its configuration is determined by position and orientation of the pelvis in the global coordinate system and the relative angles between the limbs. Each truncated cone is parameterized by the center of base circle A, center of top circle B, bottom radius $r1$, and top radius $r2$. Given the 3D camera location C and 3D coordinates A and B, the plane passing through the points A, B, C is determined. Since the vectors AB and AC lie in the plane, see Fig. 3, their cross product, which is perpendicular to the plane of AB and AC, is the normal. The normal is used to determine the angular orientation of the trapezoid to be projected onto 2D plane. Each trapezoid of the model is projected onto 2D image of each camera via modified Tsai's camera model. The projected image of the trapezoid is obtained by projecting the corners and then a rasterization of the triangles composing the trapezoid. Though projecting all truncated cones we obtain the image representing the 3D model in a given configuration.

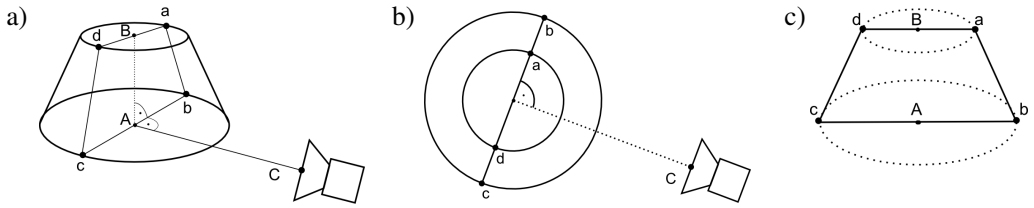

Figure 3. Extraction of trapezoid representing the truncated cone. a) side view, b) top view, c) camera view.

In each frame the 3D human pose is reconstructed through matching the projection of the human body model with the current image observations. In most of the approaches to articulated object tracking background subtraction algorithms are employed to extract the subject undergoing tracking. Additionally, image cues such as edges, ridges and color are often employed to improve the extraction of the person. In the presented approach the human silhouette is extracted via background subtraction. Afterwards, the edges are located within the extracted silhouette. Finally, the edge distance map is extracted [18]. The matching score reflects (i) matching ratio between the extracted silhouette and the projected 3D model and (ii) the normalized distance between the model's projected edges and the closest edges in the image. The objective function of all cameras is the sum of such matching scores. Sample images from the utilized test sequences as well as details of the camera setup can be found in [21]. Section 6 discusses in more detail the multi-thread processing of images acquired by four color cameras.

The motion tracking can by attained by dynamic optimization and incorporating the temporal continuity information into the ordinary PSO. Consequently, it can be achieved by a sequence of static PSO-based optimizations, followed by re-diversification of the particles to cover the potential poses that can arise in the next time step. The re-diversification of the particle i can be obtained on the basis of normal distribution concentrated around the best particle location p_g in time $t-1$, which can be expressed as: $x^{(i)} \leftarrow \mathcal{N}(p_g, \Sigma)$, where $x^{(i)}$ stands for particle's location in time t, Σ denotes the covariance matrix of the Gaussian distribution, whose diagonal elements are proportional to the expected velocity. The initial pose of the model is determined manually in the first frame. Given a set of prepared in advance models for typical human silhouettes, a model that best matches the images in the first frame is selected afterwards.

5.2. Cost Function

The most computationally demanding operation in 3D model-based human motion tracking is calculation of the objective function. In PSO-based approach each particle represents a hypothesis about possible person pose. In the evaluation of the particle's fitness score the projected model is matched with the current image observations. The fitness score depends on the amount of overlapping between the extracted silhouette in the current image and the projected and rasterized 3D model in the hypothesized pose. The amount of overlapping is calculated through checking the overlap degree from the silhouette to the rasterized model as well as from the rasterized model to the silhouette. The larger the overlap is, the larger is the fitness value. The objective function reflects also the normalized distance between the model's projected edges and the closest edges in the image. It is calculated on the basis of the edge distance map [21].

The fitness score for i-th camera's view is calculated on the basis of following expression: $f^{(i)}(x) = 1 - ((f_1^{(i)}(x))^{w_1} \cdot (f_2^{(i)}(x))^{w_2})$, where w denotes weighting coefficients that were determined experimentally. The same w_1 and w_2 were used in all experiments. The function $f_1^{(i)}(x)$ reflects the degree of overlap between the extracted body and the projected 3D model into 2D image from camera i. The function $f_2^{(i)}(x)$ reflects the edge distance map-based fitness in the image from the camera i. The objective function for all cameras is determined according to the following expression: $F(x) = \frac{1}{4}\sum_{i=1}^{4} f^{(i)}(x)$. In comparison to [8], where the fitness function is calculated through logical matching between the projected model and the images acquired, our algorithm permits far more accurate as well as reliable tracking. The edge information contributes towards more precise alignment. In the areas that are sparsely covered by the edges, the edge distance map gives useful information about edge location. The images acquired from the cameras are processed on CPU. The extracted foreground image and the distance map are then transferred onto the devices. Afterwards, they are mapped to the textures and then utilized by the PSO running on the GPU.

5.3. Parallelization of the Cost Function

In the evaluation phase of the PSO running on the GPU, see Fig. 2, we employ two kernels. In the first one the vertices of the 3D models are projected onto 2D image planes. Each model is projected to the image of each camera, and the total number of the projected models is equal to the number of cameras times the number of the particles. In the second one we rasterize the models and evaluate the objective functions. In our approach, in every thread block we rasterize eight or sixteen models, i.e. the models represented by eight or sixteen particles, as well as we calculate the fitness scores $f^{(i)}(x)$. Thus, the number of blocks is equal to the number of the particles divided by eight or sixteen, see Fig. 4 depicting a configuration with sixteen particles per block. The threads rasterize the columns of the triangles and perform the summing of the fitness values of pixels belonging to the processed columns. Taking into account the available number of registers, in each block we run 256 or 512 threads and each thread is in charge of processing out of several columns of the triangles.

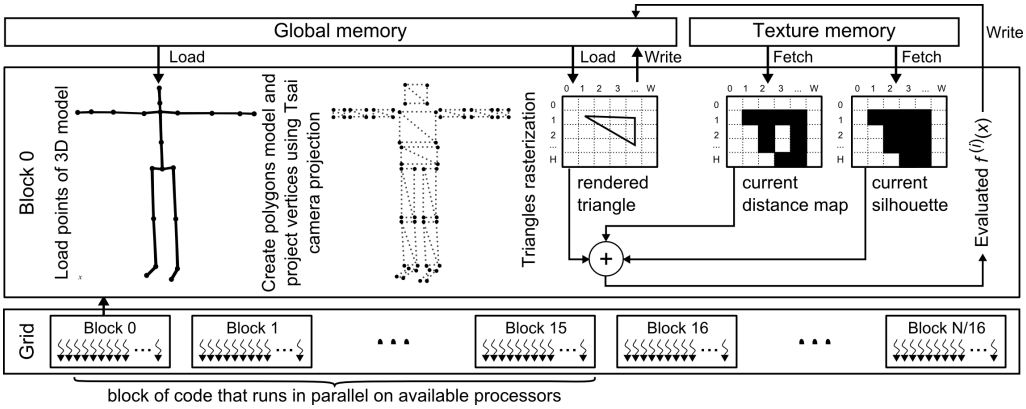

Figure 4. Calculation of the objective function on GPU.

The cost values of the objective function are summed using parallel reduction. The results from each column of the threaded block are stored in the shared memory. In the next stage, $W/2$ consecutive threads determine the sums of the two adjacent memory cells of the shared memory and then store the results in the shared memory. The next iteration employs $W/4$ threads to add the results of the previous iteration, and so on.

The rendering stage creates a two dimensional display of triangles given the transformed vertexes of the 3D model. The triangles are rasterized in order of their distance to the camera that is determined using parallel sorting. For each triangle a bounding box is determined. If the width of the bounding box is smaller than the blockDim - the number of threads executed in the block, the triangle is rasterized in a single step, see Fig. 5a, otherwise the triangle is rasterized in two or in more than two steps, see Fig. 5b.

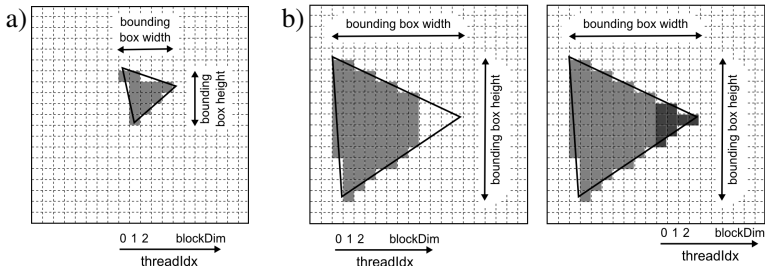

Figure 5. Triangle rasterization on GPU.

6. COMPUTATIONAL FRAMEWORK FOR REAL-TIME MOTION TRACKING

Figure 6 depicts the organization of the CPU threads in our real-time system for human motion tracking assuming that the tracking is done using four cameras. The images are acquired by four acquisition threads, which are supervised by an acquisition manager thread. The images are continuously acquired and written in a shared memory. A manager thread of image processing reads the images from the shared memory, which are then processed in processing threads. The results are stored in a shared memory, which is accessed by manager thread of the evaluation threads. Each evaluation thread executes the evaluation kernels on the GPU devices. When the GPUs return the best pose, the thread reads the foreground image as well as distance map from the shared memory, and then sends such images to the GPUs. The discussed thread organization allows us to process the images and to estimate the best human pose in parallel. It extends our previous work [19] by being able to track the 3D human motion in live video streams. The timing diagram of our soft real-time system for human pose tracking is shown in the next Section.

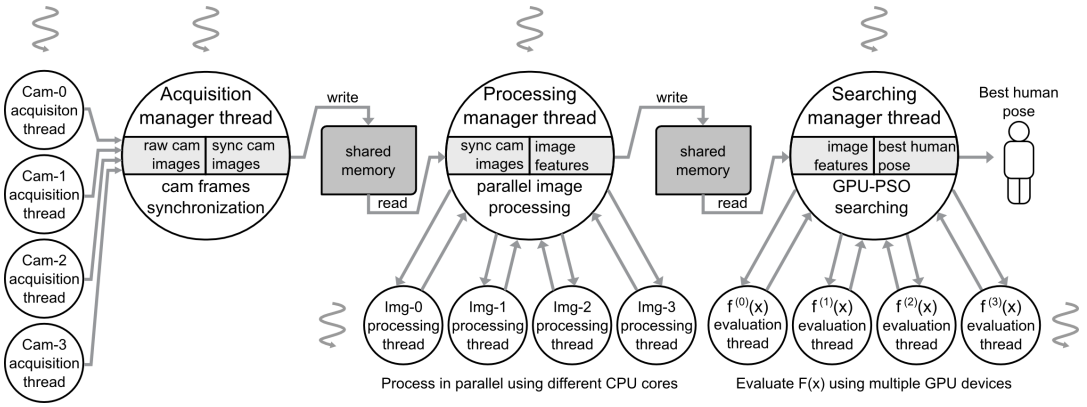

Figure 6. CPU-thread handling.

7. EXPERIMENTS

The experiments were conducted on a PC computer equipped with Intel Xeon X5690 3.46 GHz CPU (6 cores), with 8 GB RAM, and two NVidia GTX 590 graphics cards, each with 2×16 multiprocessors and 32 cores per multiprocessor. Each card has two GTX GPUs, each equipped with 1536 MB RAM and 48 KB shared memory per multiprocessor, see Fig. 7.

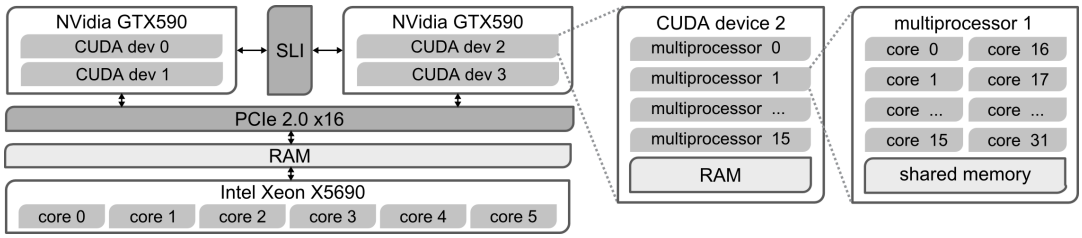

Figure 7. Hybrid multicore CPU-GPU computing platform utilized in experiments.

Figure 8 depicts a timing diagram for worst case execution time (WCET). The guaranteed human pose estimation time on image sequences from [19, 21] (of size 480×270) for the PSO with 256 particles and executing 10 iterations is equal to 96 ms. The image processing and transmission of the foreground image and the depth map from PC to GPUs takes maximum 10 ms.

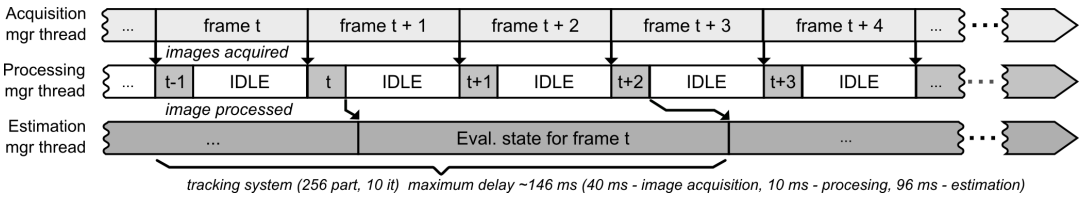

Figure 8. Timing diagram of the soft real-time system for human pose estimation.

Table I shows the computation time of the PSO, which has been obtained on CPU and GPU for 1,2, and 4 cameras, and executing 10 iterations. The results were obtained using an image sequence, which has been called P1 straight in [21], see also sample images in the discussed work, and Seq. 1 in [19]. The sequence consists of 125 frames (for each camera). The results were averaged over ten runs of the PSO with unlike initializations. For two cameras the computations were conducted on two CPU cores and two GPUs on single card, whereas for four cameras we employed 4 CPU cores and four GPUs, see also Fig. 6 and Fig. 7. As we can observe, for a system consisting of 2 cameras, the speedup that achieves the GPU over the CPU is between 4.5 and 12.7. For 4 cameras the

Table I. Average computation time [ms] of PSO for single frame of size 480×270 (Seq. P1 straight [21]).

	# part. (10 it.)	CPU [ms]	GPU [ms]	speedup
1 camera	100	130.0 ± 9.7	28.5 ± 11.2	4.6
	300	348.7 ± 16.8	51.8 ± 2.1	6.7
	1000	1130.0 ± 51.1	88.5 ± 3.6	12.8
2 cameras	100	131.1 ± 8.3	29.3 ± 13.2	4.5
	300	349.6 ± 15.9	52.5 ± 7.4	6.7
	1000	1147.4 ± 47.9	90.6 ± 3.5	12.7
4 cameras	100	169.5 ± 19.0	45.4 ± 11.5	3.7
	300	439.8 ± 53.0	61.7 ± 6.9	7.1
	1000	1378.1 ± 165.9	110.8 ± 10.2	12.4

processing time is larger in comparison to system with 2 cameras due to additional transmission and synchronization overhead between two GPU cards. For pose estimation, which is done on the basis of images from 4 cameras, using the PSO with 300 particles and 10 iterations, the system is able to execute 16 calls of the PSO-based searching for the best pose. Owing to parallel image processing it works at about 16 fps. The processing times on the CPU were obtained using an implementation presented in [21]. The mentioned above implementation uses fast scan line rendering, whereas the GPU implementation is based on point-in-polygon tests [22] that are executed in parallel. Thus, the speed-up of the GPU over CPU was calculated for the best CPU rendering algorithm. The speed-up for point-in-polygon based rasterization executed on CPU is far larger.

Figure 9 shows plots of number of frames per second vs. number of the particles for two image sequences from [21] (P1 and P2 straight). For the PSO consisting of 256 particles and executing 10 iterations the system estimates the pose at a speed of 13 fps. This means that the WCET time, see also Fig. 8, is larger about 19 ms than the average time for such a configuration of the PSO.

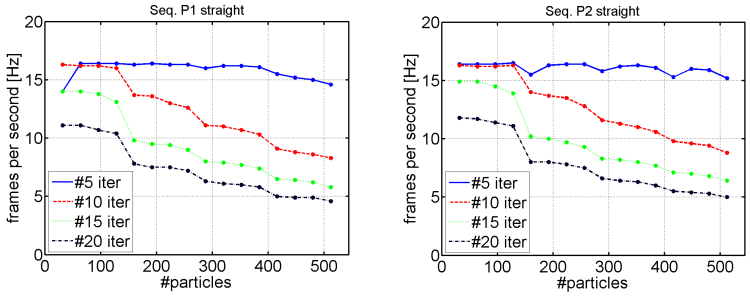

Figure 9. Number of frames vs. number of the PSO particles on Seq. P1 straight and P2 straight [21].

The plots shown in Fig. 10 illustrate the error of the pose estimation for both off-line and on-line tracking system. In the off-line tracking the system processed each frame from the Seq. P1 registered with 25 fps. In the on-line tracking the frames were read from the disk with the working frequency of the system. For soft real-time system the errors are larger since some input frames were skipped, i.e. due to larger inter-frame motions. The errors were calculated using ground-truth data [21] acquired

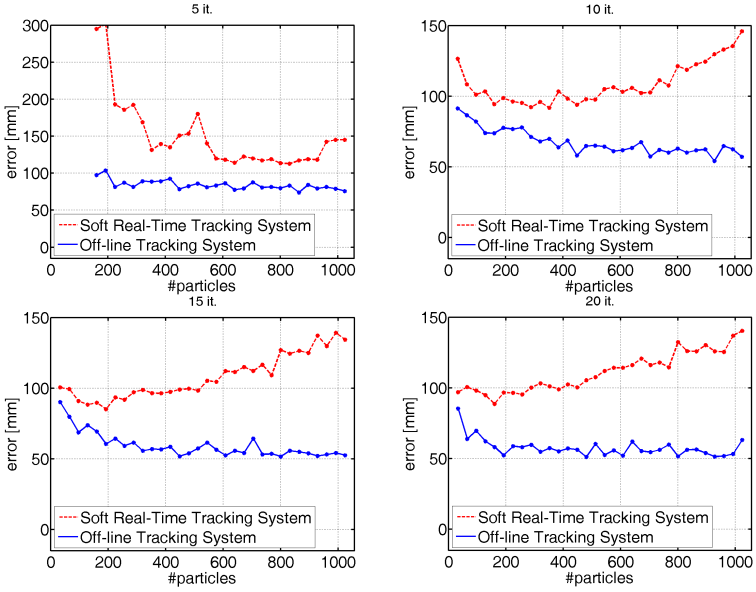

Figure 10. Tracking accuracy vs. number of particles on Seq. P1 straight [21].

by a commercial motion capture (moCap) system. For each frame they were computed as average Euclidean distance between individual (39) markers and the recovered 3D joint locations.

8. CONCLUSIONS

In this paper, we presented a GPU-accelerated algorithm for articulated human motion tracking in real-time. Image acquisition and image processing are multi-threaded and run on CPU in parallel with PSO-based searching for the best pose, which is executed on the GPUs. In a four camera setup the tracking of the full human body can be performed with 12 frames per second using a single CPU and two high-end graphics cards. The speedup of the human pose estimation running on GPU over CPU grows with the number of evaluations of the cost function, i.e. with number of the particles or with the number of the iterations. In consequence, on the GPU we can obtain more precise tracking.

Acknowledgment. This work has been supported by the National Science Center (NCN) within the research project N N516 483240.

REFERENCES

1. Pulli K, Baksheev A, Kornyakov K, Eruhimov V. Real-time computer vision with opencv. *Commun. ACM* 2012; **55**(6):61–69.
2. Lee VW, Kim C, Chhugani J, Deisher M, Kim D, Nguyen AD, Satish N, Smelyanskiy M, Chennupaty S, Hammarlund P, *et al.*. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *Proc. of the 37th Annual Int. Symp. on Comp. Arch.*, ISCA'10, ACM: New York, NY, USA, 2010; 451–460.
3. Castano-Diez D, Moser D, Schoenegger A, Pruggnaller S, Frangakis AS. Performance evaluation of image processing algorithms on the GPU. *Journal of Structural Biology* 2008; **164**(1):153 – 160.
4. Deutscher J, Blake A, Reid I. Articulated body motion capture by annealed particle filtering. *IEEE Int. Conf. on Pattern Recognition*, 2000; 126–133.
5. Wu C, Aghajan HK. Human pose estimation in vision networks via distributed local processing and nonparametric belief propagation. *Int. Conf. on Advanced Concepts for Intell. Vision Systems*, LNCS, Springer, 2008; 1006–1017.
6. Kennedy J, Eberhart R. Particle swarm optimization. *Proc. of IEEE Int. Conf. on Neural Networks*, IEEE Press, Piscataway, NJ, 1995; 1942–1948.
7. Krzeszowski T, Kwolek B, Wojciechowski K. GPU-accelerated tracking of the motion of 3D articulated figure. *Proc. of the 2010 Int. Conf. on Computer Vision and Graphics*, *LNCS*, vol. 6374, Springer, 2010; 155–162.
8. Mussi L, Ivekovic S, Cagnoni S. Markerless articulated human body tracking from multi-view video with GPU-PSO. *Proc. of the 9th Int. Conf. on Evolvable Systems: from biology to hardware*, Springer-Verlag, 2010; 97–108.
9. Kromer P, Platos J, Snasel V. A brief survey of advances in particle swarm optimization on graphic processing units. *2013 World Congress on Nature and Biologically Inspired Computing (NaBIC)*, 2013; 182–188.
10. De P Veronese L, Krohling RA. Swarm's flight: Accelerating the particles using C-CUDA. *Proceedings of the Eleventh Conference on Congress on Evolutionary Computation*, IEEE Press, 2009; 3264–3270.
11. Zhou Y, Tan Y. GPU-based parallel particle swarm optimization. *IEEE Congress on Evolutionary Computation, CEC'09.*, 2009; 1493–1500.
12. Laguna-Sanchez GA, Olguin-Carbajal M, Cruz-Cortes N, Barron-Fernandez R, Alvarez-Cedillo JA. Comparative study of parallel variants for a particle swarm optimization. *J. of Applied Res. and Technology* 2009; **7**(3):292–309.
13. Rymut B, Kwolek B. GPU-supported object tracking using adaptive appearance models and particle swarm optimization. *Int. Conf. on Computer Vision and Graphics*, *LNCS*, vol. 6375. Springer, 2010; II:227–234.
14. Solomon S, Thulasiraman P, Thulasiram R. Collaborative multi-swarm PSO for task matching using graphics processing units. *Proc. of the 13th Annual Conf. on Genetic and Evolutionary Computation*, 2011; 1563–1570.
15. Brown J, Capson D. A framework for 3d model-based visual tracking using a GPU-accelerated particle filter. *IEEE Trans. on Visualization and Computer Graphics* 2012; **18**(1):68–80.
16. Hansen N, Auger A, Ros R, Finck S, Pošík P. Comparing results of 31 algorithms from the black-box optimization benchmarking BBOB-2009. *Genetic and Evolutionary Computation Conf.*, GECCO'10, ACM, 2010; 1689–1696.
17. Shotton J, Sharp T, Kipman A, Fitzgibbon A, Finocchio M, Blake A, Cook M, Moore R. Real-time human pose recognition in parts from single depth images. *Communications of the ACM* 2013; **56**(1):116–124.
18. Krzeszowski T, Michalczuk A, Kwolek B, Switonski A, Josinski H. Gait recognition based on marker-less 3D motion capture. *10th IEEE Int. Conf. on Advanced Video and Signal Based Surveillance (AVSS)*, 2013; 232–237.
19. Rymut B, Kwolek B. Real-time multiview human body tracking using GPU-accelerated PSO. *Int. Conf. on Parallel Processing and Applied Mathematics (PPAM 2013)*, *Lecture Notes in Computer Science*, vol. 8384. Springer-Verlag: Berlin, Heidelberg, 2014; 458–468.
20. Box GEP, Muller ME. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics* 1958; **29**(2):610–611.
21. Kwolek B, Krzeszowski T, Gagalowicz A, Wojciechowski K, Josinski H. Real-time multi-view human motion tracking using particle swarm optimization with resampling. *Int. Conf. on Articulated Motion and Deformable Objects*, *Lecture Notes in Computer Science*, vol. 7378. Springer-Verlag: Berlin, Heidelberg, 2012; 92–101.
22. Heckbert PS ((ed.)). *Graphics Gems IV*. Academic Press Professional, Inc.: San Diego, CA, USA, 1994.