Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 231–240

DOI: 10.15439/2015F410
ACSIS, Vol. 5

Fast GPU and CPU computing for Head Position Estimation

Michał Szkudlarek[1] and Maria Pietruszka[2]
Institute of Information Technology
Lodz University of Technology
ul. Wólczańska 215, 90-924 Łódź, Poland
[1] michal.szkudlarek@dokt.p.lodz.pl, [2] maria.pietruszka@p.lodz.pl

***Abstract*—The head movement based control methods in the 3D graphic applications requires the real-time face position estimation. Therefore, the tracking method at the high speed and with the minimal latency is needed. This is especially hard to achieve when the face is tracked with the use of the high resolution video image on mobile devices. In the article, we present several methods for an acceleration of the face position estimation method based on the fuzzy skin color classifier and other color-based face tracking methods. The acceleration is achieved through a highly parallel GPU computation, the precalculation of the classifier weights and through the combined computations on the GPU and the CPU. The achieved computation time is independent of the used skin color classification method, allowing for use of very complex classifiers. The presented methods provides the robust head position tracking on the high resolution video image of 1920x1080 pixels, at 300 frames per second, on the mobile device with a low computing power.**

I. Introduction

THE head position tracking can be used as a multiplatform control method on desktop computers, laptops, game consoles and hand-held mobile devices. In the latter case it is especially important as the available touch-based control methods are not suitable for many interactive applications, due to the low precision and the fingers obscuring the screen. Moreover, when we are using the hand-held mobile device, the change of a relative head position does not require the actual movement of the head and can be changed by the rotation of the device.

The face tracking can be used, inter alia, for three-dimensional imaging technique called the *Head-Coupled Perspective* (*HCP* abbreviated) [1][2][3][9] which gives the impression of the depth of the presented 3D scene by dynamically linking the perspective of the rendered scene with the current head position. The rough head tracking in three degrees of freedom is sufficient for this imaging technique. For the depth sensation more important than the face tracking accuracy are its smoothness (i.e. the lack of the jittering and the unnoticeable difference between two consecutive estimated head positions), a high frequency of tracking and its low latency (i.e. short time between the actual head movement and the application response). These requirements are difficult to achieve in the case of the face tracking on the high resolution video image. Also it must be taken into account, that the head tracking is just one of the costly calculations that the interactive application needs to perform in a real time. On the mobile devices the additional limitation is their low computing power.

On the mobile devices it is possible to estimate the relative position of the user's face on the basis of other internal device sensors like the accelerometer or the gyroscope [6][7]. The rotation and the orientation of the device can be determined based on the readings from these sensors and therefore the head position relative to the screen can be estimated. The information from these sensors is delivered quickly (with frequency of 200 Hz on Android devices) and do not require time consuming computations [19]. Moreover, the tracking range of the sensors is not limited by the camera field of view. The drawback of the sensor based tracking is that their readings are noisy, what is especially noticeable in the *HCP* technique where a slight head movement results in the big displacement of the far background of the scene. Filtering of the sensor signals takes into account earlier sensor readings and generates the additional delay in the application response to the head movement. Yet, it may be unnoticeable with use of the Extended Kalman Filter [19]. Still, sensors like accelerometer or gyroscope are not able to notice the actual movement of the head itself, omission of which can destroy the 3D impression. Therefore, the additional use of the video based head tracking can highly improve the quality of the sensor based tracking [6][7]. Unfortunately, the face tracking methods chosen in these articles, are too slow. The tracking is several times slower than the sensors input, so its usage is limited to the occasional sensor reading correction. The use of a very rapid head tracking method may provide the actual head movement tracking, as well as the signal noise reduction without the delay generation.

The article presents the face tracking method that is able to find face coordinates in a few milliseconds and is based on the fuzzy skin color classifier first proposed in [1]. The low computational time is achieved through the parallel calculations on the GPU and the CPU and through the precalculation of all possible color weights assigned by the fuzzy classifier.

In the next section the existing rapid head tracking method are discussed. The third section presents the face position

978-83-60810-66-8/$25.00

estimation method with use of the fuzzy skin color segmentation. In the fourth section we describe precisely the methods for accelerating the color-based head position tracking. The fifth section presents the test results of the proposed solutions and the sixth section contains the conclusions and considered future work.

II. Related Works

Fast head position finding in the camera image is possible with various head tracking and detection methods. They can be based, among others, on the background subtraction [1][2][4], template matching [2], Haar-like feature based learning [16][17][18] or the Local Binary Pattern based learning [9][15].

In [2], for the head tracking the template matching method is used, which compares the low resolution face image with the input frame fragments. To reduce the search space, the background subtraction is performed to reject still, unchanged pixels. This method of the face tracking proved to be rapid, but the used subtraction is sensitive to the illumination changes, which is frequent in the natural light. It is not applicable on the mobile devices in which camera is not still and moves with the device. Additionally, the template matching method is sensitive to the head tilting, a natural movement when controlling the application with head movement.

For the fast head detection in the camera image, the most popular method is based on the Haar-like features, proposed by Viola and Jones [18]. Method is based on the detection of simple rectangular features, used then for the AdaBoost learning algorithm. With the use of the Integral Image, wherein each pixel represents the sum of values of all pixels from the input image above and to the left of it, it is possible to quickly evaluate the features on the given position and scale. Unfortunately the calculation of the Integral Image is time consuming and requires *2*N* operations for the image of *N* pixels. The complexity of the feature calculations is of *O(M*N)* for features of *M* scales. In [16] the acceleration of this method is proposed, with use of the GPU, but still it reaches only 19 fps for an image of 1280x960 pixels. In [17] a head tracking method based on Haar-like feature detection is proposed, working at theoretical 500 fps. The search of the entire frame of 512x512 pixels can be executed at 200 fps, on a rather powerful GPU (934 GFLOPS). The search frequency of the high-resolution image of 1920x1080 pixels, on the hand-held device would not exceed 20fps.

A faster tracking method, also based on the AdaBoost learning, but using the Local Binary Pattern is proposed in [9]. The LBP method considers the surroundings of the pixels. Thanks to the calculation on GPU and CPU [15], for a picture of 1024x1024 pixels, method can process at 10fps, but on the mobile device of very low computational power. On current hand-held devices even high-resolution image could be processed at about 25 fps.

Due to the low computational complexity and the possibility of use on the mobile devices, the face tracking methods based on the skin color classification are important[1][4][5][14]. Such a classifiers extract from the RGB space (or other color space) a subspace containing the RGB values corresponding to the possible shades of the skin color. Fuzzy classifiers that determines the pixel degree of membership may provide a smoother tracking with the same resolution image [1]. Multiplatform version of the method described in [1] is presented in Section III.

Despite the linear complexity of the skin color based tracking method, analysis of high resolution images still can be very time consuming, especially on hand-held mobile devices. In this article, we proposed accelerating methods, enabling face tracking on frame of 1920x1080 pixels at about 300 fps, even on the devices with a low processing power.

III. The Fuzzy Skin-Color Classifier

To maintain the multiplatform usability of the fuzzy head tracking method, first proposed in [1], and to allow its usage on hand-held mobile devices, the omission of the background subtraction is necessary. It is impossible to use due to the continuous motion of the hand-held device and hence its camera.

In the proposed multiplatform head tracking method, the user's head position is determined based on the degree of membership $\mu_S(p)$ of all the pixels to the fuzzy skin-color pixels multiset *S:*

$$S = \left\{ \left\langle p, \mu_S(p) \right\rangle : p \in X_{RGB} \right\} \quad (1)$$

$$\mu_S(p) = \max\left(0, \min\left(1, p_R \cdot f_R + p_G \cdot f_G + p_B \cdot f_B + f_D\right)\right) \quad (2)$$

Where:

$X_{RGB} = \{<r,g,b>: r,g,b \in [0,255]\}$ – the multiset of all the image pixels.

$p=<p_R, p_G, p_B>$ – the components *R, G, B* of the pixel *p*.

$f=<f_R, f_G, f_B, f_D>$ – the vector of the color filter.

The vector of the color filter, specified in the classifier calibration, defines a plane in the RGB space. This plane separates most of the skin color pixels from the background pixels that we are trying to discard. The filter vector specifies also the "positive" side of that plane and tells us which of the two subspaces contains skin-color pixels. The fuzziness of the multiset *S* provides a gradient boundary between the subspaces. The further the pixel color resides from the plane on its "positive" side, the higher is its degree of membership.

The usage of the RGB space instead of other color model, in which the skin colors are easier to distinguish and isolate, is motivated by the additional computation time needed to perform the transformation from the original RGB image provided by the camera to the more preferable color model.

The usage of just two components (e. g. H and S in HSV model) does not compensate this time in the processing. As the skin tone is separable in the RGB space, the computational complexity remains the major factor in choosing the color model.

To determine the position of the face on the image and its distance from the camera, it is necessary to calculate the cardinality of the S multiset, defined as the sum of the degrees of membership of all N image pixels:

$$|S| = \sum_{i=1}^{N} \mu_S(p_i) \tag{3}$$

We are using all the N pixels of the image as the skin-color face pixels, because the background pixels with the degrees of membership equal 0 does not affect the results. This degrees (Eq. 2) are also the weights of the pixels used to determine the coordinates C_X and C_Y of the centroid C of all the skin-color pixels, calculated as the ratio of the weighted sum of the image pixel coordinates to the cardinality of multiset S:

$$C_X = \frac{1}{|S|} \sum_{i=1}^{N} \mu_S(p_i) \cdot x_i \tag{4a}$$

$$C_Y = \frac{1}{|S|} \sum_{i=1}^{N} \mu_S(p_i) \cdot y_i \tag{4b}$$

Where:

x_i, y_i – coordinates of pixel p_i

The found centroid coordinates are considered the coordinates of the head center in the image and are used to determine face position in the plane parallel to the screen and the camera. The ratio of the cardinality of the S multiset to the cardinality of multiset of all the image pixels can be used as the A measure of the face area in the image and used to calculate the head distance from the screen:

$$A = \frac{|S|}{|X_{RGB}|} \tag{5}$$

What is, according to (Eq. 3):

$$A = \frac{1}{N} \sum_{i=1}^{N} \mu_S(p_i) \tag{6}$$

The algorithm of finding the face position on the camera image is shown in the Fig. 1. The non-modified algorithm is later referred to as "*Version CPU 1*" in comparison with the accelerated variants of the algorithm.

The filter f (Eq. 2), in contrast to most of the skin-color classifier methods, is not designed to match all colors that may belong to the skin of any person, of any race, skin tone, in all possible lightning condition. The primary objective of the filter f is to extract the user face from the current environment visible in the camera. It requires determining the optimal values of the filter parameters for each application usage.

```
sumOfWeights = 0; CX = 0; CY = 0;
For each p in XRGB
{
  weight = max(0, min(1, pR ·fR + pG ·fG + pB ·fB + fD));
  CX += weight * p.xCoordinate;
  CY += weight * p.yCoordinate;
  sumOfWeights += weight;
}
CX = CX / sumOfWeights;
CY = CY / sumOfWeights;
A = sumOfWeights / N;
```

Fig. 1 The algorithm of finding the face position on the camera image (pseudocode).

Manual finding of the optimal values may be hard task for the user, and certainly it is uncomfortable and time-consuming. Therefore, in [8] we proposed the automatic method of finding the optimal values of skin-color filter. The method of automatic parameters calculation is based on the analysis of the image with arbitrarily marked area of the face. The user moves the head to place it in the oval-shaped mask visible in the preview of the camera. Once approved, several consecutive frames of the video image are capture for the analysis. Pixels of the obtained images are then use as factors in the objective functions $G(f)$ which maximum is searched. To accelerate the computation of the optimum parameters the clustering of the input data transformed to the RGB space is performed with fast grid-based clustering method proposed in [8] where the entire process is described in detail.

The automatic parameters calculation allows for extracting most of the face pixels even from a difficult background with colors close to the skin tone or in the poor lightning. The automatically calculated parameters provide the method stability, ensuring that the $|S|$ value in the Equations (4a) and (4b) does not tend to zero. Still, in some particularly difficult conditions the method does not filter out all of the background pixels, assigning a small part of these pixels with high weights. In practice, the tracking is then still effective. Although the unfiltered pixels "attract" the found centroid to their center of gravity, reducing the amplitude of the estimated head movement determined by the tracking, still the direction of the motion is preserved and its speed is proportional to the actual speed of the head movement. As a result, the perspective of the virtual scene can still be coupled with the head movements, resulting in an immersive sense of depth in the Head-Coupled Perspective technique or effective control in other applications. Therefore, although the method is error-prone in cases when additional skin-colored body parts (e.g. neck or chest) or other faces are visible to the camera, the method still provides sufficient results for proposed applications. Especially on the hand-held mobile devices, the user head is always the biggest skin-colored object visible to the camera and cannot be dominated by other objects. When the head leaves the field of view, due to the device rotation, it may be also recorded by other internal sensors which can then substitute the head tracking.

To minimalize the resulting errors of classification in the particularly difficult conditions, we can utilize more computationally complex skin-color models, like described in [14] Gaussian model or used in [10] elliptical Gaussian chrominance probability density function. We can also transform pixel colors to another color space, e.g. *HSV,* normalized *RGB* or *CIE-XYZ* (with all components divided by the sum of all components), or proposed in [10] the *STV* space. The possible skin colors in these spaces are easier to separate [10] and the impact of the lightning on the classification is reduced. Unfortunately, both methods of improvement increase the computational cost of calculating the pixels weights, almost proportionally increasing the computation time, which actually need to be reduced. Therefore, besides the direct computation time reduction, it is desirable to decouple the head tracking cost from the complexity of the used skin color classifier.

IV. Accelerating the Head Position Tracking

The computational cost of the presented above head tracking method is of the order *O(N)*, where *N* is the number of image pixels. The method requires a few operations per pixel, and each is processed once and individually. Although the linear computation cost seems to be low, for high resolution video image of 1920 x 1080 pixels the processing of the full frame in the real time is hard to achieve, even without the additional CPU load.

The accelerating methods that decrease the problem size, reduce at the same the tracking quality. The downscaling of the input image [15][16] requires additional computation, decreases the number of input data and reduces the angular resolution of the tracking. Searching for the face only in the neighborhood of the previously found head position [9][16][17] makes the tracking sensitive to the fast motion of the head, providing just a slight acceleration, due to the large area occupied by the face on the image of the narrow-angle camera.

The described below accelerating methods reduce the computational cost of the head tracking without decreasing the number of the analyzed pixels, to maintain high quality of the tracking. When the lower quality of the tracking is allowed, these methods can be successfully combined with the problem size reduction for the further tracking acceleration.

A. The Head Tracking Acceleration on the GPU

The acceleration of the head tracking can be achieved by transferring the calculations to the graphics processor unit, which allow for the parallel analysis of many pixels. Besides the possible computation time reduction, this approach may additionally decrease the energy usage of the mobile devices during the frame processing [9]. The use of the Nvidia® CUDA® framework allow for using the full capabilities of graphics processors compatible with this architecture. Such GPUs are widely installed on the laptop computers and they are recently available on the hand-held mobile devices. The tests of the methods described below are performed on the laptop computer equipped with the Intel® Core™ i5-2450M mobile processor and the Nvidia® GeForce® GT 630M mobile GPU.

As the classification of the pixel colors in our method is performed for each pixel individually, the acceleration is seemingly easy to achieve by performing the computations on the graphics card. The parallelization of the calculation of the pixels degrees of membership to the skin-color multiset *S* is trivial on GPU. Each thread must calculate the weight of one pixel, according to Equation 2. A part of the kernel (i.e. a function in CUDA executed in parallel by multiple threads on GPU) responsible for the pixels weights calculation is as follows:

```
uint index=threadIdx.x + blockIdx.x*blockDim.x;
Pixel p=frame[index];
float weight=f[0]*p.r + f[1]*p.g + f[2]*p.b+ f[3];
if(weight>0.0f){ if(weight>1.0f)weight=1.0f; }
else weight=0.0f;
[…]
```

A constant delay is generated by the transfer of the image to the GPU memory. For an image of 1920x1080 pixels, the transfer takes about 2 milliseconds.

For the further analysis, the calculation of the head position is split into two parts:

1) Calculating the degrees of membership from Eq. 2.
2) Finding the centroid coordinates (Eq. 4) and the sum of all the pixel weights need in Eq. 6.

On the CPU, the second part of the calculation represents only a small fraction of all the computations. Conversely, on the GPU the computation of the second part can be several times longer than the first part. The time of 2) on the graphics processor depends largely on the used algorithm. Below, different methods of accelerating computation on GPU are compared.

1) Version GPU 1 – Atomic adding to one global value

In the naive approach to calculate the coordinates of the centroid, all the threads can add its calculated values directly to the output sums, using the *atomic* add function:

```
[…]
atomicAdd(x,weight*(index%width));
atomicAdd(y,weight*(index/width));
atomicAdd(w,weight);
```

Atomic functions in CUDA allows multiple threads to modify common data, with the guarantee of receiving the correct result, which means that every thread perform the operation exactly once, without data loss from the simultaneous access to the output value.

Unfortunately, in this approach the summing is performed sequentially, as only one thread at a time increases the output sum. Moreover this variant requires a very frequent access to the very slow global memory. It is very inefficient approach, which do not use the full power of the GPU and prolongs the calculation compared to the CPU computing.

2) Version GPU 2 – Atomic adding to the shared memory

The acceleration can be obtained when first the thread values are atomically added to local sums of the blocks in the faster shared memory, and then only this local sums are atomically added to the global output values, by one thread per block:

```
__shared__ float sX,sY,sW;
sX=0.0f;
sY=0.0f;
sW=0.0f;
[…]
atomicAdd(&sX,weight*(index%width));
atomicAdd(&sY,weight*(index/width));
atomicAdd(&sW,weight);

__syncthreads();

if( 0 == threadIdx.x )
{
  atomicAdd(x,sX);
  atomicAdd(y,sY);
  atomicAdd(w,sW);
}
```

In this approach a part of the internal aggregations is executed in parallel between several thread blocks. Therefore this variant is more than three times faster than the *Version GPU 1*.

3) Version GPU 3 – Parallel aggregation with divide and conquer

Although in *Version GPU 2* part of the summation is performed in parallel between blocks, still only one thread per block can add its value to the local sum at the same time. To execute more parallel addition inside a thread block it is possible to use the divide and conquer method, decreasing the complexity order of the summation for each *N*-thread block from *O(N)* to *O(logN)*. In this case, half of the block threads must sum up pairs of values calculated by the consecutive threads. In the next steps, the sums from previous step are summed in pairs until there is only one final sum of all the block values (Fig. 2).

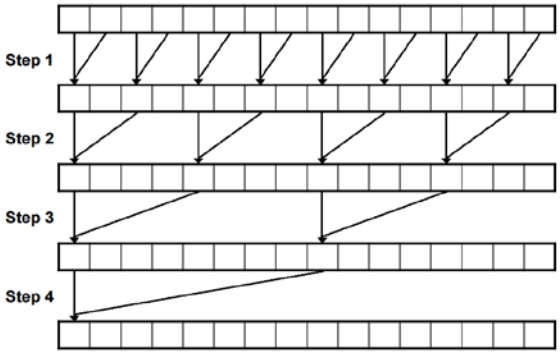

Fig. 2 The parallel aggregation in block

The kernel realizing directly this approach is as follows:

```
__shared__ float tempX[THREADS_PER_BLOCK];
__shared__ float tempY[THREADS_PER_BLOCK];
__shared__ float tempW[THREADS_PER_BLOCK];
[…]
uint s,temp;
for(s=1;s<THREADS_PER_BLOCK;s<<=1)
{
  if(0==threadIdx.x%(s<<1))
  {
    __syncthreads();
    temp=threadIdx.x+s;
    tempX[threadIdx.x]+=tempX[temp];
    tempY[threadIdx.x]+=tempY[temp];
    tempW[threadIdx.x]+=tempW[temp];
  }
}
if( 0 == threadIdx.x )
{
  atomicAdd(x,tempX[0]);
  atomicAdd(y,tempY[0]);
  atomicAdd(w,tempW[0]);
}
```

This approach decrease the computational time by 40% compared with the *Version GPU 2*. The acceleration is less than expected due to the highly divergent branching of the code in the condition `(0==threadIdx.x%(s<<1))`, what is discussed by Harris [11] on an analogous example.

4) Version GPU 4 – Without divergent branching

Further acceleration can then be obtain by alternatively engaged threads, as shown in Fig. 3.

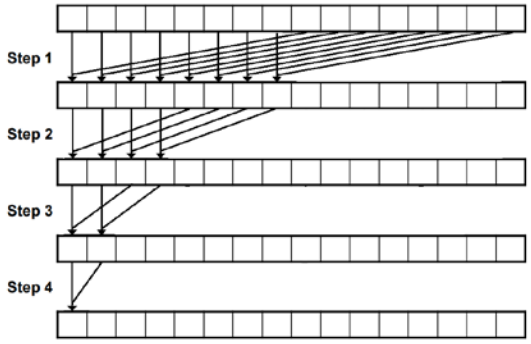

Fig. 3 The parallel aggregation without the divergent branching and bank conflicts

This approach omits the problem of divergent branching, without additional shared memory bank conflicts. Analogous solution for similar problem was used in [13].

The internal aggregation in this version looks as follows:

```
[…]
uint s,temp;
for(s = THREADS_PER_BLOCK>>1; s>0; s>>=1)
{
     if(threadIdx.x<s)
     {
       __syncthreads();
       temp=threadIdx.x+s;
       tempX[threadIdx.x]+=tempX[temp];
       tempY[threadIdx.x]+=tempY[temp];
       tempW[threadIdx.x]+=tempW[temp];
     }
}
if( 0 == threadIdx.x )
{
  atomicAdd(x,tempX[0]);
  atomicAdd(y,tempY[0]);
  atomicAdd(w,tempW[0]);
}
```

This variant is 2.5 times faster than *Version GPU 3*.

5) Version GPU 5 – Multiple aggregation per thread

In the *Version GPU 4* during the iterative aggregation an average of ¾ of the block threads is idle, from the half in the first iteration, to all but one in the final aggregation. At the same time, due to the limit of thread number per block, we receive a large number of blocks. The possible solution, proposed by Harris for similar problem [11], is to use threads to iteratively add up a greater number of values. In our case it requires also the calculation of more pixel weights (i. e. degrees of membership to skin color multiset) per thread. Such a solution can significantly better harness the GPU computational power. Before we get to the inter-thread aggregation, all the threads are engaged in a long non-synchronized work.

After this modification, kernel is as follows:

```
__shared__ float tempX[THREADS_PER_BLOCK];
__shared__ float tempY[THREADS_PER_BLOCK];
__shared__ float tempW[THREADS_PER_BLOCK];
uint ti = threadIdx.x;
uint index = ti+blockIdx.x * THREADS_PER_BLOCK;
uint gridSize = THREADS_PER_BLOCK * gridDim.x;
Pixel p;
float weight;
tempX[ti]=0;  tempY[ti]=0;  tempW[ti]=0;
while(index<N)
{
  p=frame[index];
  weight=f[0]*(p.r)+ f[1]*(p.g)+ f[2]*(p.b)+ f[3];
  if (weight>0){if(weight>1.0f) weight=1.0f;}
   else weight=0.0f;
  tempX[ti]+=weight*(index%width);
  tempY[ti]+=weight*(index/width);
  tempW[ti]+=weight;
  index+=gridSize;
}
uint s,temp;
for(s= THREADS_PER_BLOCK>>1;s>0;s>>=1)
{
  if(ti<s)
  {
    __syncthreads();
    temp=ti+s;
    tempX[ti]+=tempX[temp];
    tempY[ti]+=tempY[temp];
    tempW[ti]+=tempW[temp];
  }
}
if( 0 == ti )
{
  atomicAdd(x,tempX[0]);
  atomicAdd(y,tempY[0]);
  atomicAdd(w,tempW[0]);
}
```

Due to the reduced number of blocks, this variant is 2.5 times faster than *Version GPU 4*.

6) Version GPU 6 – Double memory access

Further acceleration by about 10% can be obtained by adding two values per iteration and grouping operations. It is the result of the GPU's memory access, where reading two consecutive 4-bytes words has a similar cost to reading just one word [12]. The number of blocks is halved:

```
uint index = ti+blockIdx.x*THREADS_PER_BLOCK * 2;
uint gridSize = gridDim.x*THREADS_PER_BLOCK * 2;
[…]
while(index<N)
{
 index2=index + THREADS_PER_BLOCK;
 p=frame[index];
 p2=frame[index2];
 weight=f[0]*(p.r)+ f[1]*(p.g)+ f[2]*(p.b)+ f[3];
 weight2=f[0]*(p2.r)+f[1]*(p2.g)+f[2]*(p2.b)+f[3];
 if (weight>0){ if(weight>1.0f) weight=1.0f; }
  else   weight=0.0f;
 if(weight2>0){if(weight2>1.0f) weight2=1.0f;}
  else weight2=0.0f;
 tempX[ti]+=weight*(index%width)
           +weight2*(index2%width);
 tempY[ti]+=weight*(index/width)
           +weight2*(index2/width);
 tempW[ti]+=weight + weight2;
 index+=gridSize;
}
```

7) Version GPU 7 – Aggregation in local memory

In versions *GPU 5* and *GPU 6* designed on the basis of [11], highly ineffective is the iterative aggregation of all the values calculated by the thread to the shared memory. During this aggregation other threads do not need any access to this temporary sum. Our proposed solution is to aggregate the thread values in its local memory, as it is faster than the shared memory. Only the final thread sum should be copied to the shared memory for access of other threads.

```
[…]
float tX,tY,tW;
tX=0;  tY=0;  tW=0;
while(index<N)
{
 […]
 tX+=weight*(index%width)+weight2*(index2%width);
 tY+=weight*(index/width)+weight2*(index2/width);
 tW+=weight + weight2;
index+=gridSize;
}
tempX[ti]=tX;
tempY[ti]=tY;
tempW[ti]=tW;
[…]
```

This approach decrease the computational time by 10% compared with the *Version GPU 6*.

8) Version GPU 8 – Multiple pixels per thread with atomic adding to shared memory

The divide and conquer parallelization in the *Version GPU 4* leads to a four times faster computing, compared to *Version GPU 2,* where the tread values are atomically added to the block sum. But since each thread aggregates weights of hundreds of pixels, the synchronization of threads before each iteration of inter-thread summing is very time consuming. It appears that the atomic summation to one value is more effective in that case. Even though the acceleration of such approach is only about 3%, the additional profit is the reduction of the shared memory occupancy and a slight kernel code simplification:

```
__shared__ float tempX,tempY,tempW;
if(0==threadIdx.x)
{
  tempX=0; tempY=0; tempW=0;
}
uint ti=threadIdx.x;
uint index=ti+blockIdx.x* THREADS_PER_BLOCK *2;
uint gridSize = THREADS_PER_BLOCK *2*gridDim.x;
Pixel p;
float weight,weight2;
unsigned int index2;
float tX,tY,tW;
tX=0;  tY=0;  tW=0;
while(index<N)
{
 index2=index + THREADS_PER_BLOCK;
 p=frame[index];
 p2=frame[index2];
 weight=f[0]*(p.r)+ f[1]*(p.g)+ f[2]*(p.b)+ f[3];
 weight2=f[0]*(p2.r)+f[1]*(p2.g)+f[2]*(p2.b)+f[3];
 if(weight>0) { if(weight>1.0f)  weight=1.0f;}
  else weight=0.0f;
 if(weight2>0){if(weight2>1.0f) weight2=1.0f;}
 else  weight2=0.0f;
 tX+=weight*(index % width)
    +weight2*(index2 % width);
 tY+=weight*(index / width)
    + weight2*(index2 / width);
 tW+=weight + weight2;
 index+=gridSize;
}
atomicAdd(&tempX,tX);
atomicAdd(&tempY,tY);
atomicAdd(&tempW,tW);
__syncthreads();
if( 0 == ti )
{
  atomicAdd(x,tempX[0]);
  atomicAdd(y,tempY[0]);
  atomicAdd(w,tempW[0]);
}
```

The final version of the kernel is almost 45 times faster than the *Version GPU 1*.

B. The Head Tracking Acceleration on the CPU

Although the computation of head position on the GPU can be very fast (with frame computation below 5 ms), it can be insufficient in the 3D graphic applications like video games, where the 3D rendering loads the GPU to its limits. To maintain the performance of the application, we may need to limit the GPU computations.

Unfortunately, the head position estimation on the CPU is time consuming, as its computational cost is of the order *O(N)*. Even with an effective implementation of the algorithm (Fig. 1), on the testing platform the 30 fps is not achieved (see Table 1). Therefore it is desirable to accelerate also the CPU computing.

1) Version CPU 2 – The weights precalculation

The acceleration on the CPU can be obtain with the precalculation of all the possible values of $\mu_S(p)$ (Eq. 2) and referring to them instead of calculating this value for every input pixel individually. The number of all the RGB colors with 8-bits channels is limited and amounts to 256^3. It is 8

times more values than in a frame of 1920x1080 pixels, but the precalculation can be performed only once before the start of the application, as the colors weights change only when the filter *f* values change.

The precalculated weights assigned by the classifier to all the RGB values, can be stored in an array *W*[], in which the pixel *p* weight is located at the given position:

$$W\left[p_R \cdot 256^2 + p_G \cdot 256 + p_B\right] = \mu_S(p) \tag{7}$$

With the indexing relevant to the input pixel format (i. e. its byte order), the integer value written on the four bytes of the pixel is also the position in the array *W,* at which the pixel weight is stored.

With the use of the precalculated weights, the CPU computations are faster only by 35%. Even though this solution is interesting for other reason. As was mentioned in the previous section, it is desirable to decouple the computing time from the color classification method, for the possible classification improvement without the reduce of the tracking speed. With this solution, the usage of more complex color model or the transformation to the other color space, increases only the once performed precalculation time and the tracking time remain unchanged, so that aim is fully achieved.

Applying this concept for the GPU is not recommended, as it results in prolonged, twice as long calculation time. This is caused by the "random", irregular access to the array *W,* as the adjacent pixels can have different colors, distant in the RGB space. As a result the global memory access time is increased, slowing the entire computing. Moreover, on the GPU computation cost is already almost independent of the used color model. A slight gain from the precalculation may be achieved only with use of a very complex color classifiers.

2) Version CPU 3 – The precalculation and multicore processing

The parallelization of the calculations is not limited to the GPU computing. Most modern CPUs have at least two independent processing units (called *cores*). Therefore, the equal distribution of the calculations to more threads may results in almost direct proportional time reduction.

By dividing the problem, i.e. the input frame, between multiple CPU cores, we can compute pixel weights from each part of the image, and the centroid of these pixels. The centroid of all the partial centroids is also the centroid of the whole image.

3) Version GPU 8 + CPU 3– The joint GPU and CPU calculations.

As in the above example, the problem can be divided into two parts, one of which is implemented on the CPU, and the second on the GPU. Such division may be dictated by the GPU limitation, or by the need for utilizing all the available computing power for the further tracking acceleration.

In the second case, in order to achieve the greatest acceleration, the division of the problem between the CPU and the GPU should not be symmetrical as in multicore processing. For the highest resulting speed, the two processors should complete their computations at the rather same time. If average times of the full frame computation on CPU and GPU are respectively T_C and T_G, and they analyze sub-images I_C and I_G of the image I ($I = I_C + I_G$), than the sizes of the sub-images should be:

$$I_C = \frac{T_G}{T_C + T_G} \cdot I \tag{8a}$$

$$I_G = \frac{T_C}{T_C + T_G} \cdot I \tag{8b}$$

Hence, in our case, at the testing laptop, the highest speed is achieved when 70% of the image pixels is processed on the GPU (with *Version GPU 8*) and 30% on the CPU (with *Version CPU 3*). The comparison of the computation times of all the methods is presented in the next section.

4) Other possible acceleration methods

The possible further acceleration of the computation may be achieved by the replacement of the floating point operations with the fixed point calculations and especially integer operations. In order to do this, we must upscale filter *f* values by *u* to the integer values. The maximum weight assigned by the classifier (1 in the Equation 2) must also be upscaled by the same *u* value. This way the *S* multiset would no longer meets the definition of the fuzzy set, but the results of head position estimation would not change if we only divide the *A* (Eq. 5 and 6) by the *u*. The problem of this solution is that we may must restrict the maximum resolution of the image or the resolution of the classifier (i.e. number of different weights it may assign) to ensure that we do not exceed the maximum of the 32 bits variables. Yet, it may be acceptable in some applications and the resulting acceleration may be significant.

Another way to accelerate CPU computation may be the usage of Streaming SIMD Extensions which allow for parallel processing of up to four values. However the usage of SSE may be restricted and may depends on the target platform. Altough this solution was not applied it is worth consideration.

V. RESULTS

The fast head position tracking methods proposed in the Section IV were compared with the original algorithm of *Version CPU 1* shown in Fig. 1. The tests was performed on the laptop computer equipped with the mobile processor Intel® Core™ i5-2450M (2x2,5GHz) with processing performance of 34,5 GFLOPS and the mobile GPU Nvidia® GeForce® GT 630M with 307,2 GFLOPS processing performance.

The comparison of the average computation times for frames of 1920x1080 pixels is shown in the Fig. 4 and in the

Table 1. Although the times of CPU and GPU should not be compared due to the different architectures, the scale of the possible acceleration of the head tracking can be seen in the Fig. 4. The computations on the GPU (*Version GPU 8*) are over seven times faster, including the time of transfer to the GPU memory, and it allows for tracking at 200 frames per second. In an architecture where the camera image is saved directly in the GPU memory, without need for additional copying, over 300 fps can be achieved.

The acceleration methods basing on the CPU also give good results and over three times faster computing compared to the original algorithm. Using CPU with more processing cores, the further acceleration is possible.

When the processing powers of GPU and CPU are not restricted, the method combining the calculations on both units may result in the greatest tracking speed. On the testing platform the analysis of the full frame was performed in about 3,6 milliseconds (Fig. 4 and Table 1).

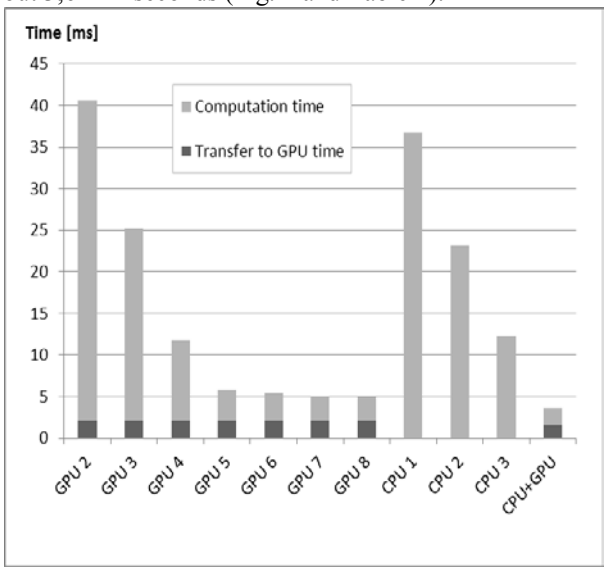

Fig. 4 The processing times of one frame of 1920x1080 pixels (average of 100 trials).

VI. Conclusions and Future work

Although the device used in the tests was not a hand-held device, its CPU and GPU has respectively 50% and 15% lesser processing performance than the Nvidia® Shield™ Tablet with Tegra K1 mobile processor equipped with CPU ARM Cortex-A15 R3 (4x2,3GHz), with the processing performance of 70,4 GFLOPS and CUDA-enabled GPU Kepler with 364,8 GFLOPS processing performance. Also, the popular smartphone processors Qualcomm® Snapdragon™ 810 has a similar processing performance. Hence, the testing laptop computer represents well the computing power of the today hand-held devices.

The achieved head tracking times leave a large margin of error for the real-time tracking with over 60 fps, even in the case of a highly loaded GPU and CPU or when performed on the devices with a lot less computing power. With the achieved tracking times it becomes possible to use the found head position for effective Head-Coupled Perspective implementation in combination with the interactive 3D applications on hand-held mobile devices.

The future work includes testing the proposed methods on the actual hand-held devices and designing the model for determining the head position relative to the device with use of the internal sensors (accelerometer and gyroscope) combined with the proposed fast head tracking method.

References

[1] M. Szkudlarek, M. Pietruszka, "Head-Coupled Perspective in Computer Game", In: Journal of Applied Science, vol. 21, no. 2, pp. 165-179, 2013.

[2] J. Rekimoto, "A vision-based head tracker for fish tank virtual reality-VR without head gear," Virtual Reality Annual International Symposium, 1995. Proceedings, IEEE, pp.94-100, 1995.

[3] J. Francone, "Using the User's Point of View for Interaction on Mobile Devices", 23rd French Speaking Conference on Human-Computer Interaction, pp. 41-48, New York, 2011.

[4] W. Gaver, G. Smets, K. Overbeeke, "A Virtual Window on media space". In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '95), New York, pp.257-264, 1995.

[5] A. Bulbul, "A Face Tracking Algorithm for User Interaction in Mobile Device", CyberWorlds, 2009. CW '09. International Conference on, IEEE, pp. 385 – 390, 2009.

[6] J. Hwang, J. Jung, and G. J. Kim. "Hand-held virtual reality: A feasibility study", in ACM Virtual Reality Software and Technology, pp. 356–363, 2006.

[7] N. Joshi, A. Kar, and M. Cohen. "Looking at you: fused gyro and face tracking for viewing large imagery on mobile devices". In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12), pp. 2211-2220, 2012.

[8] M. Szkudlarek, M. Pietruszka, "Fast Grid-Based Clustering Method for Automatic Calculation of Optimal Parameters of Skin Color Classifier for Head Tracking". In Proceedings of 2015 IEEE 2nd International Conference on Cybernetics (CYBCONF), 2015.

[9] M. B. Lopez, J. Hannuksela, O. Silven, F. Lixin, "Head-tracking virtual 3-D display for mobile devices," Computer Vision and Pattern Recognition Workshops (CVPRW), 2012 IEEE Computer Society Conference on , pp.27-34, 2012.

[10] J.-C. Terrillon, M. David, "Automatic detection of human faces in natural scene images by use of a skin color model and of invariant moments," Automatic Face and Gesture Recognition, Proceedings.

Table I.
The processing times of one frame of 1920x1080 pixels (average of 100 trials).

	GPU 1	GPU 2	GPU 3	GPU 4	GPU 5	GPU 6	GPU 7	GPU 8	CPU 1	CPU 2	CPU 3	CPU + GPU
Transfer time [ms]	2,05	2,05	2,05	2,05	2,05	2,05	2,05	2,05	-	-	-	1,54
Calculation [ms]	131,34	38,59	23,19	9,68	3,81	3,36	3,06	2,94	36,74	23,12	12,34	2,12
Total time [ms]	133,39	40,64	25,24	11,73	5,86	5,41	5,11	4,99	36,74	23,12	12,34	3,66
CPU 1 / Total time	0,28	0,90	1,46	3,13	6,27	6,79	7,19	7,36	1,00	1,59	2,98	10,04
Frames per Second	7,50	24,61	39,62	85,25	170,65	184,84	195,69	200,4	27,22	43,25	81,04	273,22

Third IEEE International Conference on,pp.112-117, 1998.
[11] M. Harris, “Optimizing Parallel Reduction in CUDA”, NVIDIA Developer Technology, 2007.
[12] J. Luitjens, S. Rennich, “CUDA Warps and Occupancy”, GPU Computing Webinar, 2011.
[13] D. Xie, L.Dang, R. Tong, “Video Based Head Detection and Tracking Surveillance System”, 9th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2012), IEEE, pp. 2832-2836, 2013.
[14] Y.-W. Wu, X.-Y. Ai, "Face Detection in Color Images Using AdaBoost Algorithm Based on Skin Color Information," Knowledge Discovery and Data Mining, 2008. WKDD 2008. First International Workshop on , pp.339-342, 2008.
[15] M. B. López, H. Nykänen, J. Hannuksela, O. Silvén, M. Vehviläinen, “Accelerating image recognition on mobile devices using GPGPU”, IS&T/SPIE Electronic Imaging. International Society for Optics and Photonics, pp. 78720R-78720R, 2011.
[16] B. Sharma, R. Thota, N. Vydyanathan, A. Kale, "Towards a robust, real-time face processing system using CUDA-enabled GPUs," High Performance Computing (HiPC), 2009 International Conference on, pp.368-377, 2009.
[17] I. Ishii, H. Ichida, T. Takaki, "GPU-based face tracking at 500 fps", Image Processing (ICIP), 2011 18th IEEE International Conference on, pp.557-560, 2011.
[18] P. Viola, M. Jones, "Rapid object detection using a boosted cascade of simple features," Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on, vol.1, pp.I-511,I-518, 2001.
[19] J. Gośliński, M. Nowicki, P. Skrzypczyński, "Performance Comparison of EKF-Based Algorithms for Orientation Estimation on Android Platform," Sensors Journal, IEEE, vol. 15, no. 7, pp. 3781 - 3792, 2015.