3D Objects Tracking by MapReduce GPGPU-Enhanced Particle Filter

Jieyun ZHOU†‡, Student Member, Xiaofeng LI†, Haitao CHEN†, Rutong CHEN†, and Masayuki NUMAO††, Nonmembers

SUMMARY Objects tracking methods have been wildly used in the field of video surveillance, motion monitoring, robotics and so on. Particle filter is one of the promising methods, but it is difficult to apply to real-time objects tracking because of its high computation cost. In order to reduce the processing cost without sacrificing the tracking quality, this paper proposes a new method for real-time 3D objects tracking, using parallelized particle filter algorithms by MapReduce architecture which is running on GPGPU. Our methods are as follows. First, we use a Kinect to get the 3D information of objects. Unlike the conventional 2D-based objects tracking, 3D objects tracking adds depth information. It can track not only from the x and y axis but also from the z axis, and the depth information can correct some errors in 2D objects tracking. Second, to solve the high computation cost problem, we use the MapReduce architecture on GPGPU to parallelize the particle filter algorithm. We implement the particle filter algorithms on GPU and evaluate the performance by actually running a program on CUDA5.5.

key words: objects tracking, particle filter, 3D, MapReduce, GPGPU

1. Introduction

In recent years, visual tracking research is getting popular among researchers in the fields of video surveillance, robot vision, etc. A large amount of research papers focus on the improvement of efficiency and accuracy of tracking algorithms. Among them, particle filter approach is most attractive due to its high robustness and effectiveness, because of the ability of solving non-linear and non-Gaussian dynamic problems.

Objects tracking using GPU is also a very hot field. Paper [1] describes a method of 3D tracking using particle filter in DOF (Degrees Of Freedom) and gestures. It focuses on the changes of gesture of one object having many DOFs. But it has to train the whole video sequence to learn the background before tracking. Paper [2] provides another particle filter method on GPU and proposed a cluster histogram named co-occurrence histogram which contains both color information and location information to improve robustness. However it only deals with a video sequences from color cameras. Paper [3] discusses a MapReduce method on GPUs for particle filters. The research is on face recognition and tracking with 2D face information and is not suitable for other objects.

In our study, we make use of 3D information and do not need a training process in advance. Our system provides a real-time tracking based on color and depth information.

In this paper, rigid objects are considered with some tolerance on small changes in shape. The targets can be bottles, books, bags, and human faces. An object is naturally in three dimensions. But traditionally, trackings are carried out in 2D with projection of the objects, in this way we lose a lot of effective information. By using of the 3D information, object tracking is more robust and effective [4].

We use GPU for the tracking in the following three ways. Firstly, the parallel computing ability of GPU is utilized to support a large number of transition particles. Secondly, we are able to compute histograms for all particles on GPU. Finally, we calculate the similarity function and sort the particles by their weights on GPU. Besides an XYZ-related weighted histogram is proposed in this paper to segment color and depth image into subimages of different weights on GPU. Optimizing parallel reduction and Bitonic sort in CUDA methods are used to reduce the computing complexity. From the experiment result our tracking method is very real-time and robust.

This paper is organized as follows. Section 2 describes the background, including the particle filter and the platform CUDA. Section 3 presents three proposed techniques. The experiment result is showed in Sect. 4. And the Sect. 5 draws the conclusion.

2. Background

2.1 Particle Filter

Particle filter is a tracking method based on Monte Carlo sampling. Two kinds of variables are defined in this method, state variables and measurement variables. In hidden Markov model we can’t observe the states of the model directly, and only the measurement variables are available. From the measurement model we can conjecture the state of the model. The definition of the state model has been given in [5], the state sequence \( \{x_k, k \in N\} \) of a target given by

\[
x_k = f_k(x_{k-1}, v_{k-1})
\]  

Copyright © 2015 The Institute of Electronics, Information and Communication Engineers
where $f_k$ is the nonlinear system function, $x_k$ is the state vector, $v_k$ is the noise of this system. $k$ is the index of time step. In tracking problem, state model means actual movement of the target object.

The measurement model is a measurement from the state model

$$z_k = h_k(x_k, n_k)$$

(2)

where $z_k$ is the measurement vector, $n_k$ is the measurement noise caused by measure method. $h_k$ is the measurement function. In tracking problem, measurement model means the estimated movement of target candidate. The possibilities of target candidates are calculated by the likelihood function.

Algorithm 1 Basic of Particle filter

| 1: for $i = 1$ to $N$; | 2: Initial Particles $x^i_1$; |
| end for | 3: |
| 4: for very frame do | 5: |
| 6: $x^i_{t+1 \text{prev}} = f(x^i_{t-1 \text{prev}}, u^i_t)$; equation(1) | 7: |
| 8: Weight $w^i_k = p(z_k | x^i_{t+1 \text{prev}})$; equation(6) | 9: |
| 10: Normalize weight; | 11: |
| 12: Resampling; | |
| end for |
| end for |

A particle filter typically consists of two stages, prediction and update.

- The prediction stage, based on the known system model $p(x_k | x_{k-1})$ predicts the state probability density function (pdf), forwarding from one measurement time to the next. The $p(x_k | x_{k-1})$ is a dynamic model which defines how the particle moves.

$$p(x_k | x_{k-1}) = \int p(x_k | x_{k-1}) p(x_{k-1} | x_{k-1}) dx_{k-1}$$

(3)

- The update operation uses the latest measurement to modify the prediction pdf. This is achieved by using Bayes’ theorem and is a mechanism for updating knowledge about the target state in the light of extra information from new data.

$$p(x_k | z_{1:k}) = \frac{p(z_k | x_k) p(x_k | x_{1:k-1})}{p(z_k | x_{1:k-1})}$$

(4)

where the denominator is a normalizing constant,

$$p(z_k | x_{1:k-1}) = \int p(z_k | x_k) p(x_k | x_{1:k-1}) dx_k$$

(5)

where $p(z_k | x_k)$ is a likelihood function which defines the similarity of measurement model and state model. The similarity between target model and target candidate is also defined by $p(z_k | x_k)$. And we define the likelihood function as the weight of a particle, that is

$$w_k = p(z_k | x_k)$$

(6)

From the Eq. (3) (4) (6) and the known system model $p(x_k | x_{k-1})$ we can easily get the state sequence or the target model. Fig. 1 shows how the transition works. The tracking method based on particle filter is listed in Algorithm 1.

2.2 GPGPU Calculation by Compute Unified Device Architecture (CUDA)

General-purpose computing on graphics processing units (GPGPU) [6] uses the graphics processing units (GPU) to calculate the data for CPU. GPU is usually used in image processing, but how to migrate an algorithm from CPU to GPU is still a challenge.

CUDA is a parallel computing platform and program model invented by NVIDIA. It enables a large increase of computing performance by migrating data from CPU to GPU and using parallel computing ability of GPU. A new execution model has been used is the single instruction multiple thread (SIMT). SIMT uses 32 parallel threads to create, manage and execute, which are called a warp. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path.

Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vectors are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. In fact, many non-image-processing algorithms can be accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology.

2.3 Kinect for 3D Information

Kinect is a motion sensing input device by Microsoft for the Xbox 360 video game console and Windows PCs [7]. Based around a webcam-style add-on peripheral for the Xbox 360 console, it enables users to control and interact with the Xbox 360 without the need to touch a game controller, through a natural user interface using gestures and spoken...
commands.

The device features a RGB camera, a depth sensor (IR), a multi-array microphone, and a motor to adjust camera angles. It provides three kinds of information, a color image, a depth image and a skeleton.

With Kinect our object tracking method works on 3D information, with RGB image for normal tracking and depth information to help locating of the object.

3. MapReduce Architecture Based 3D Tracking on GPU for Improvement

In our method, tracking by particle-filter algorithm is realized by three approaches, filtering theory, similarity function, and partial differential equation.

The first approach is based on the filtering theory. Objects tracking problem can be converted to a probability density function estimation problem, which is often solved by Kalman filter [8] or particle filter [9]. The merit of these filters is that they are able to deal with the occluded objects problem. But the drawback is their very high computing complexity and the shortcoming is their difficulty in dealing with the occluded objects problem. So nowadays a lot of researchers concentrate on how to decrease the computing complexity. The second approach is often based on the mean-shift objects tracking [10]. The probability similarity function between the target model and the target candidate is used to calculate the mean-shift iteration by gradient descent algorithm. The advantage is its relatively low computing complexity and the shortcoming is its difficulty in dealing with the occluded objects problem. The last approach is based on partial differential equation. Changing into a functional optimization problem, the target tracking is to solve the functional extremum from a partial differential equation.

Because of the time consuming, it is a critical issue for tracking object in a real-time. In this section, three techniques are proposed, a) 3D self-adaptive tracking window, b) parallelization on the transition part, histogram part and the likelihood part, c) XYZ-related weighted histogram.

The following definitions are used in the subsequent discussion.

(1) Tracking window. A rectangular in a image containing the target object. It is chosen according to some rules, often manually, in the first frame. Some of image pixels in the window are sampled as particles.

(2) Particle. The basic unit of the particle filter algorithm. It moves in frames to indicate the location of the tracking window.

(3) Block. The basic unit in GPU for parallel computation.

(4) Thread. The basic unit in GPU for serial and parallel computation. Units communicate with each other.

3.1 3D Self-Adaptive Tracking Window

Conventional particle filter tracking method uses the random number generator to get random sizes of the tracking windows [5]. It takes N random numbers as N window sizes and computes similarity function values on these windows. The window which has the maximum value should be the target window. This method may have some shortcomings. Even if the particles location is the center of the target window, the similarity function value can become very small because the window size is different. Correct window size is not always obtained and this often leads to failure of tracking. To solve this problem, we propose a method of 3D self-adaptive tracking window.

With depth information, we can estimate the size of the tracking window adaptively instead of using N random numbers. Suppose a target of size L is firstly at distance of \( d_1 \) and then at \( d_2 \), as shown by T1 and T2 in Fig. 2 (a).

From the geometric relationship, we have

\[
\frac{H_1}{L} = \frac{f}{d_1}, \quad \frac{H_2}{L} = \frac{f}{d_2}
\]

where \( H_1 \) and \( H_2 \) are the target window sizes in images respectively, and \( f \) is the camera focus. So,

\[
H_2 = \frac{d_1}{d_2}H_1
\]

If the target is off the horizontal line by \( a \), as shown by T3, we have,

\[
\frac{b + H_3}{a + L} = \frac{f}{d_1}, \quad \frac{b}{a} = \frac{f}{d_1}
\]

where \( b \) is the offset in image and \( H_3 \) is the target window size. Then we have,

\[
\frac{H_3}{L} = \frac{f}{d_1}
\]

which means the image size is only related to the distance. Kinect has a motor to control the angle between camera and the horizontal plane. Assume the angle is controlled to a value of \( \theta \), as show in Fig. 2 (b), we denote the new distance and effective height of the target by \( d'_1 \) and \( L' \) respectively. As \( \theta \) is not very large, we have \( d'_1 = d_1 \cos \theta \) and \( L' = L \cos \theta \) roughly. Similarly to the above, the new window size is
Fig. 3 Difference of getting particles between basic and the proposed method.

\[
H_1' = \frac{L_f'}{d_1'} = \frac{L_f}{d_1} = H_1
\]

So the angle will not affect the window size. Generally we have

\[
H_2' = \frac{d_1'}{d_2'} H_1' = \frac{d_1 \cos \Delta \theta}{d_2'} H_1
\]

where \(\Delta \theta\) is the angel change of the Kinect between two positions of the target. Based on Eq. (12), we can get the size of each particle iteratively. The geometric relationship between window sizes and depths is direct and robust, so the sizes calculated by Eq. (12) are more robust and accurate than the conventional approach of random window sizes.

The main difference between the basic version and our proposed one is how to get particles. It is compared in the algorithms shown in Fig. 3.

For each window, a threshold \(D_{th}\) is setup based on the depth of the center, and weights \(w_i\) for other points in the window are given by

\[
\omega_i = \begin{cases} 
1 & \text{when } d_i \leq D_{th} \\
0 & \text{when } d_i > D_{th} 
\end{cases}
\]

where \(d_i\) is the depth of \(i^{th}\) pixel in the window.

3.2 Parallelization in Particle Filter

GPGPU recently becomes more and more popular due to its high parallel computing ability and lower cost [11]. CUDA utilizes the blocks and the threads of GPU to realize the parallel computing. There exist some designs of particle filter on GPU. But which part of the algorithm to be migrated and how to parallelize the algorithm to reduce the time of execution is very challenging.

As a particle algorithm repeats the same kind of calculation many times, it is naturally suitable to be done on GPGPU which is very good at parallel computation [4], [12]. In this paper, we adopt two methods of implementing particle filter on a GPU. Method 1 works in parallel on blocks. GPUs may have very different properties [13]. When the number of blocks is less than the number of SMs (Stream Multiprocessor), parallelization is straight forward. Otherwise, if the number of blocks is larger, the blocks need to be divided into teams which are processed serially, while the blocks in a team are processed parallely. Method 2 works in parallel on threads in a block. All threads in a block are processed parallely and the number of threads varies on GPUs and CUDA visions. In the following experiment we compare the two methods.

Figure 4 shows the two different methods on GPUs. They work in different parallel units, method 1 in blocks and method 2 in threads.

• Initialization of particles (CPU). This is done on CPU. First we select a tracking window and produce \(N\) random points (A, B). The random points form a set and is sent to GPU and stored in each block (method 1) or thread (method 2). The present location \((x_0, y_0)\) is also sent to each block (method 1) or thread (method 2). To calculate the size and histogram for each particle’s window, we also send the depth and color information which are gotten by Kinect. And the normalized histogram of target in the first frame is also send to GPU as a template.

• Transition of particles (GPU). In this part, the random points in each block (method 1) or thread (method 2) are used to sample new particles, according to the second-order autoregressive dynamics which give new positions of the particles \((x_p, y_p)\). The data on GPU is stored in \(N\) blocks (method 1) or thread (method 2) and they are independent and work parallely in the same time.

• Histogram calculation (GPU). For each particle, with the depth information from Kinect, the size of the area is calculated on GPU. As a target candidate, we find the histogram of the area. The algorithm is shown in Algorithm 2. In method 1 \(N\) histograms are calculated in \(L\) bins, one histogram for a block, exploiting \(L\) threads. In method 2 histograms are also calculated in shared threads in blocks. Algorithm 2 shows a simple code snippet for histogram calculation on a sequential processor. To compare the histograms we have to normalize them, for this reason we sum the value of a histogram in its bins. To do the sum efficiently, we create a new parallel computation in each block, making use of the parallel computing capability of threads. In this way, we create small parallel computation in the big parallel one. Now it is good to use the Parallel Reduction [14] in each thread of each block. So we divide each block into \(L\) threads such that one thread for a bin, and we sum the \(L\) bins in a parallel way and normalize the histogram in each block.

**Algorithm 2 Histogram calculation**

1: // bins is the amount of bin
2: for \(i = 0\) to data_length
3: // data[] should be normalized between 0 to 1
4: \(bin = data[i] \times (bins - 1)\)
5: \(H[bin]++\)
6: end for
XYZ-related weighted histogram.
In objects tracking field, histogram calculation is an important part. Paper [15] introduces some methods in
histogram calculation on GPU. Based on them we propose a XYZ-related weighted histogram which makes the best use of GPU and the 3D position information which is lost in traditional histogram calculation. In our method the weight is not only considered on the x and y axis from color information but also on z axis from depth information. The weight varies on different positions in XYZ direction. The main idea is that the weight near center is larger than the surroundings in x,y axis. And the weight in z axis is given in Eq. (13), it decreases the impact of the background on the far end.

As shown in Fig. 5, a target image is divided into \( m \times n \) subimages. Let \( S_i \) be the \( i^{th} \) subimage. \( z_i = (x_i, y_i) \) be the center point of a tracking window, and \( z_0 = (x_0, y_0) \) the beginning point of the \( i^{th} \) subimage.

\[
d = \|z_i - z_0\| \tag{14}
\]

where \( \| \| \) is the norm operation. Here we use a 2-norm which is the Euclidean distance. The weight of each subimage in xy axis \( w_{xy} \) is defined as

\[
\omega_{xy} \propto \frac{1}{d} \tag{15}
\]

Every subimage’s histogram is calculated in a parallel way on GPU. We can get \( m \times n \) histograms, and the final histogram is gotten based on Eq. (13) (15).

\[
H_{final} = \frac{1}{C} \sum_{i=1}^{mn} \omega_{xy} \times \sum_{j=1}^{\text{pixel}} H_{ij} \omega_{z_j} \tag{16}
\]

where \( H_{ij} \) is the histogram of the \( i^{th} \) subimage. \( H_{final} \) is the final histogram of this target, and C is the normalization coefficient.

Our proposed method uses not only the GPU’s parallel computation ability but also the position information.

- **Likelihood calculation (GPU).** In each block (method 1) or thread (method 2), the likelihood between the histogram of a particle and the template target histogram is calculated on GPU. Then the likelihoods are converted to weights by normalization conducted on GPU. To do the normalization, likelihoods of all blocks (method 1)
or threads (method 2) are summed up. An optimizing parallel reduction is applied here again in CUDA to improve the processing speed and make tracking very fast. Before resampling part of the particle filter, we use a parallel sort method, Bitonic sort [16] that sorts by the particle’s weights. When the sorted weights obtained, they are sent from GPU to CPU.

- **Resampling.** It is processed on GPU to produce more particles on larger weight area, meanwhile remove the particles which have small values of weights. In conventional method all the particles have to be sorted on CPU by their weights to find the particles which have larger weights. Now in our method the sort step has already done in the calculation of likelihood on GPU. The pseudocode for method 1 is showed in Algorithm 3.

**Algorithm 3** Particle filter designed on GPU

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>for ( i = 1 ) to ( N );</td>
</tr>
<tr>
<td>2:</td>
<td>Initial Particles ( x_i ) ;</td>
</tr>
<tr>
<td>3:</td>
<td>end for</td>
</tr>
<tr>
<td>4:</td>
<td>for very frame do</td>
</tr>
<tr>
<td>5:</td>
<td>Move data from CPU to GPU; // Host to device</td>
</tr>
<tr>
<td>6:</td>
<td>Calculate histogram; // GPU code, calculate in each block or thread</td>
</tr>
<tr>
<td>7:</td>
<td>Calculate weight;</td>
</tr>
<tr>
<td>8:</td>
<td>Normalize weight;</td>
</tr>
<tr>
<td>9:</td>
<td>Calculate SUM, optimizing parallel reduction;</td>
</tr>
<tr>
<td>10:</td>
<td>Normalize weight and Bitonic sort;</td>
</tr>
<tr>
<td>11:</td>
<td>Move particles from GPU to CPU; //Device to host</td>
</tr>
<tr>
<td>12:</td>
<td>Resampling;</td>
</tr>
<tr>
<td>13:</td>
<td>end for</td>
</tr>
</tbody>
</table>

3.3 MapReduce Architecture on GPU for Particle Filter

MapReduce method is a very hot method in data mining. It was originally proposed by Google for the ease of development of web search applications on a large number of CPUs [17]. This method is based on two steps, Map and Reduce.

Phoenix [18] and Mars [19] are two well-known MapReduce frameworks on parallel systems. The former is designed for shared memory systems in C++ while the latter works on GPU and Mars-CUDA is said to be 22 times faster than the CPU-based MapReduce, Phoenix. To adapt to most applications, Mars-CUDA is designed to support dynamic memory situations by counting the output sizes of MAP, GROUP, REDUCE and so on in advance. In our tracking application, the output sizes of all particles are known and fixed. So it is not necessary to deal with dynamic situations and we apply a hand-tuned CUDA implementation, which could be more efficient. In our proposed method each parallel unit is responsible for a Map or a Reduce task. Each parallel unit is like one worker. First we split our data to each parallel unit as the step of map. Each parallel unit has a pair of key/value \((k_1, v_1)\). \(k_1\) stands for the index of particles. The pseudo codes are showed in Algorithm 4.

**Algorithm 4** MapReduce on particle filter

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>MAP(key, value)</td>
</tr>
<tr>
<td>2:</td>
<td>{</td>
</tr>
<tr>
<td>3:</td>
<td>// key, index of chunk</td>
</tr>
<tr>
<td>4:</td>
<td>// value, contents in chunk (particle’s property)</td>
</tr>
<tr>
<td>5:</td>
<td>for each particle in chunk;</td>
</tr>
<tr>
<td>6:</td>
<td>EmitIntermedia ((k_1, v_1)) ;</td>
</tr>
<tr>
<td>7:</td>
<td>end for</td>
</tr>
<tr>
<td>8:</td>
<td>}</td>
</tr>
<tr>
<td>9:</td>
<td>EmitIntermedia ((k_1, v_1))</td>
</tr>
<tr>
<td>10:</td>
<td>{</td>
</tr>
<tr>
<td>11:</td>
<td>((k_1, v_2))=Transition+Likelihood((k_1, v_1)) ;</td>
</tr>
<tr>
<td>12:</td>
<td>Emit ((k_1, v_2));</td>
</tr>
<tr>
<td>13:</td>
<td>}</td>
</tr>
<tr>
<td>14:</td>
<td>REDUCE((k_1, v_2))</td>
</tr>
<tr>
<td>15:</td>
<td>{</td>
</tr>
<tr>
<td>16:</td>
<td>//aggregation with the different particle</td>
</tr>
<tr>
<td>17:</td>
<td>(k_2=)BitonicSork (k_1) by (v_2.weight);</td>
</tr>
<tr>
<td>18:</td>
<td>Emit ((k_2, v_2));</td>
</tr>
<tr>
<td>19:</td>
<td>}</td>
</tr>
</tbody>
</table>

\[ k_1 = N_{\text{index}} \] \hspace{1cm} (17)

And \(v_1\) is the properties of each particles. Which is a set of current positions, previous positions and the size of original tracking window. There are different workers in one task.

\[ v_1 = \{ \]
\[ x_0, y_0; //Current x, y coordinate size; \]
\[ x_{\text{pre}}, y_{\text{pre}}; //Previous x, y coordinate \} \hspace{1cm} (18) \]

After Mapping step the algorithm’s transition part works as in Fig. 4. In the transition process we calculate the intermediate result \((k_1, v_2)\). \(v_2\) is a set of new positions, new scales and sizes of tracking windows and likelihoods of histogram of each particles.

\[ v_2 = \{ \]
\[ x_p, y_p; //New x, y coordinate size_0; //New size weight = \text{likelihood}; \} \hspace{1cm} (19) \]

Then \(v_2\) is assigned to workers again for the Reduce, which is the main part of the likelihood calculation described above. In the process we merge the intermediate result \((k_1, v_2)\) to \((k_2, v_2)\). \(k_2\) is the new key which is the new index after Bitonic sort. It is easy to complete the following resampling.

\[ k_2 = N_{\text{new_index}} \] \hspace{1cm} (20)

The pseudo codes are showed in Algorithm 4.

4. Experiment Result

In the following part the experiment environment is Windows 7 and Visual Studio 2012 combined with OpenCV 2.4.6, gsl 1.8, and CUDA 5.5. The camera is Kinect 360. The GPU of this experiment platform is NVIDIA Quadro.
2000D. Our purpose is to evaluate the performance of our proposed method and compare with the conventional one. The SM of this GPU is 32 and thread number is 1024. The particle number is selected to 128.

Figure 6 (a) shows the conventional algorithm of particle filter with the target of a toy. (b) shows the method 1 of 3D tracking by Kinect with the target of a toy. (c) shows the method 2 of 3D tracking by Kinect with the target of a toy. (d) shows another case of the conventional algorithm of particle filter with the target of a box. (e) shows the method 1 of 3D tracking by Kinect with the target of a box. (f) shows the method 2 of 3D tracking by Kinect with the target of a box. The yellow windows show each particles’ area, and the red window shows the final location of the target.

Figure 7 shows the experiment results with large number of particles, 1024 particles. The given frame is randomly chosen on number of 5, 58, 128, and 172. Figure 8 displays the tracking paths of the conventional one and our method 2, and the accurate path of the moving target. Then Table 1 lists the SD-err (standard derivation in error) of different tracking in x, y positions and sizes, in which CPU means the conventional algorithm. Four kinds of particle number are used in the experiments. During the experiments, the conventional algorithm sometimes loses the target and the tracking window often does not fit the target. But the proposed methods show that the target window matches the target well due to self-adaption by distance, which is especially obvious in method 2. Along with the 2D-images, the depth data provided by Kinect establishes 3D-information, though simple, it is very helpful to the robustness of the tracking.

We compare the whole time costs in Fig. 9. ‘GPU+CPU’ means our proposed methods while ‘CPU’ means the conventional one. The experiment results show that the proposed algorithms have better real-time abilities. Also method 2 is much better. Then we evaluate each part of our methods.
Figure 10 shows the time cost for transition part of conventional method on CPU and two proposed methods on GPU. The evaluation are done for N=32, 64, 128, 256, 512, 1024 particles respectively.

Figure 11 shows the histogram calculating part. The difference in performances are obvious.

From above results we see that method 2 has less time cost. The main reason is that the numbers of the blocks and threads affect the efficiency. As mentioned above, when the number of blocks is less than the number of SMs (Stream Multiprocessor), parallelization is straightforward. Otherwise, when the number of blocks is larger, the blocks will be divided into groups in serial, and the blocks in a group be processed parallelly. The SM number in our GPU is 32. Therefore method 1 working in large number of blocks is limited in parallelization. And method 2 working in large number of threads has no such problem and its threads run in parallel to give a better performance. In the following experiment we will show the performance of method 2 on different GPUs.

In the following, two GPUs of NVIDIA GeForce 9300M and NVIDIA Quadro 2000D are considered. We compare the time costs of each part of particle filter algorithm of the proposed method 2 on the two GPUs. Table 2 shows the different properties of the GPUs. Because of the limit of threads on GPU1 the particles are selected to 32, 64, 128, 256, 512 for the evaluation. The experiment results in Fig. 12 shows that GPU2 has a better performance in tracking.

Before the calculation on GPU, a transfer of data between host and GPU is necessary. In our experiment, the size of image is 640*480. To reduce the overhead of transferring, we do not transfer the whole RGB and depth images into GPU simply. The data that has to be transferred is well selected. It includes the current locations \((x_c, y_c)\) and previous locations \((x_p, y_p)\) of all particles, the location of target on the previous frame \((x_0, y_0)\), the height and width \((w, h)\) of the first and most recent tracking windows, the RGB and depth images in the windows, the histograms. Due to the moving-velocity of the target, a range of image around the current target position is also transferred to the GPU, The range size is application-dependent and we use one fourth in \(x\) and in \(y\) in our experiment. The time cost of data transfer is shown in Table 3. The experiment is carried out under the same settings of that for Fig. 7. It shows that the overhead takes a small portion of the time cost for each frame in most cases of particle numbers, which is not a problem for real-time applications. If very large number of particles is used, the time cost may be much more than the frame interval of 33 ms and makes realtime tracking impossible.
5. Conclusion

In this paper we propose two methods to implement particle filters on GPUs. In the methods three techniques are used. They are, a) 3D self-adaptive tracking window, b) parallelization on the transition, histogram computing and the likelihood calculation, c) XYZ-related weighted histogram. From the experiments we have done on CUDA5.5, the proposed techniques reduce global operation largely. The methods provide significant speedup and show a very good robustness.

References

Rutong Chen was born in 1990. He received his B.S. degree from the University of Electronic Science and Technology of China. Now is a master at School of Information and Communication in the University of Electronic Science and Technology of China. His current research focuses on image processing and error analysis of Multiple camera system.

Masayuki Numao received the B.S., M.S., in electric engineering, and Ph.D. Degrees in information science and technology from the University of Tokyo in 1981, 1983, and 2000, respectively. His research interests include intelligent informatics, data mining, and security and privacy. He is currently Professor of Computer Science at the University of Electro-Communications, specializing in Big Data analysis from sensor networks. Dr. Numao is a member of IEEE Computer Society, Information Processing Society of Japan, and Japan Society of AI.