The 2010 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 18-22, 2010, Taipei, Taiwan

Using GPUs to improve system performance in Visual Servo systems

Chuantao Zang, Koichi Hashimoto
Graduate School of Information Sciences,
Tohoku University, Sendai, Japan
{chuantao,koichi}@ic.is.tohoku.ac.jp

Abstract— **This paper describes our novel work of using GPUs to improve the performance of a homography-based visual servo system. We present our novel implementations of a GPU based Efficient Second-order Minimization (GPU-ESM) algorithm. By utilizing the tremendous parallel processing capability of a GPU, we have obtained significant acceleration over its CPU counterpart. Currently our GPU-ESM algorithm can process a 360×360 pixels tracking area at 145 fps on a NVIDIA GTX295 board and Intel Core i7 920, approximately 30 times faster than a CPU implementation. This speedup substantially improves the realtime performance of our system. System reliability and stability are also greatly enhanced by a GPU based Scale Invariant Feature Transform (SIFT) algorithm, which is used to deal with such cases where ESM tracking failure happens, such as due to large image difference, occlusion and so on. In this paper, translation details of the ESM algorithm from CPU to GPU implementation and novel optimizations are presented. The co-processing model of multiple GPUs and multiple CPU threads is described in this paper. The performance of our GPU accelerated system is evaluated with experimental data.**

I. INTRODUCTION

Visual servo consists in controlling a robot using feedback image information [1]. Among various image based visual servo (IBVS) systems, one category has been designed to exploit the Cartesian information from the homography between two images of a planar object. Several control strategies decompose the homography to explicitly reconstruct the motion of a camera (translation and rotation), such as [2] [3] [4]; some schemes directly use the homography to control the robot without such decomposition in [5] [6] [7]. Among all these methods, the system performance greatly depends on the estimation accuracy and robustness of the homography.

In most cases homography solution is a typical minimization problem of sum of squared differences (SSD) between a region in a reference image and a warped region in a current image [6]. Many nonlinear optimization approaches have been proposed to deal with this least square optimization problem with different kinds of approximations, such as Standard Newton method, Gauss-Newton methods, Levenberg-Marquardt method and so on [8]. Among these solutions, the efficient second-order minimization (ESM) algorithm is an elegant idea which adopts a more adequate approximation of the computational costly Hessian matrix than other methods' approximations [9]. By performing a second order approximation of the SSD problem with only first order derivative, this method can get a high convergence rate and avoid local minima close to the global one. Because of these merits, it has been used in different applications.

However, when considering a realtime visual servo system, the main requirements of the tracking algorithms are efficiency, accuracy and stability. In Malis's paper [6], only the stability and convergence rate of the ESM algorithm are evaluated. As far as we know, no processing speed information is mentioned. From our experience, with the increase size of a tracking area, for example, an area of 300×300 pixels, the ESM computation still takes too much time and induces a relative low processing speed. In a typical visual servo system, this will cause a larger image displacement in the two continuous images as the camera is mounted on a moving end-effector. As we will mention later, ESM can not process this kind of large image difference well.

And there are also other limitations about the ESM algorithm. In such cases where a large difference exists between a current image and a reference image, i.e. only a small overlapping area exists in this pair of images, the ESM tracking algorithm will fail to find the right homography solution because it can not obtain enough information from this relative small overlapping area. In practical applications of visual servo, usually there is a large difference between the current image and the reference image because the end-effector can be far from the desired pose.

To solve these problems, we adopt the GPUs as coprocessors to increase the tracking speed and stability of our system. Our contributions are mainly as follows.

Firstly, we present a GPU based ESM algorithm (GPU-ESM) to address the need for faster visual tracking algorithms. After several novel optimizations on the GPU code, we succeed in achieving substantial acceleration over its CPU implementation. For a tracking area of 360×360 pixels, our GPU-ESM can work at 145 fps, approximately 30 times faster than the CPU application. This allows for a realtime visual tracking system with a higher speed camera. With such camera there will be a smaller difference between the continuous frames and this smaller difference will make the ESM tracking result more reliable.

Secondly, to improve the system reliability and robustness, we adopt Lowe's Scale Invariant Feature Transform (SIFT) algorithm [10] on GPU (GPU-SIFT) to find the homography in such cases where GPU-ESM failure happens due to large translation or rotation or even occlusion. We get an approximately 20 times speed up than on CPU.

Thirdly, we proposed a co-processing strategy to combine the GPU-ESM and GPU-SIFT algorithms for system reliability. If GPU-ESM tracking failure happens, the system will automatically use the homography result from the GPU-

SIFT. As the ESM algorithm is adaptive, i.e. a homography from the previous loop will be the initial value of the current loop, ESM tracking can continue working with the homography from the SIFT algorithm. Therefore, the whole system can work smoothly with high reliability and robustness at a high processing speed.

The rest of this paper is organized as follows. Section II reviews the relative works about the ESM and SIFT algorithms. Section III introduces the translation details of the GPU-ESM so as to fully utilize the parallel architecture of a GPU. The co-processing model between multiple CPUs and GPUs is also described in this part. Section IV lists the optimization methods in our GPU implementations. Section V shows the experiments to evaluate our proposed system. Section VI is the conclusion part.

II. RELATED WORKS

A. ESM algorithm

ESM algorithm was first proposed by Malis in 2004 [11]. By performing a second order approximation of the least square problem with only first order derivative, it has shown great potential in a variety of applications, such as visual tracking of planar object [6] and deformable object [12], visual servo in [7] [11] etc. In our visual servo applications, we adopt the homography-based visual control scheme [6] for a planar object. For simplicity, we review the homography-based visual control scheme with ESM algorithm [6].

First we suppose a planar object is projected in a reference image I^* with some "Template" region of q pixels. Tracking the reference template in a current image I consists in finding the homography transformation $\overline{G}$ that transforms each pixel P_i^* of the template into its corresponding pixel in the current image I, i.e. finding the homography $\overline{G}$ such that $\forall i \in \{1,2,...q\}$:

$$I(w(\overline{G})(P_i^*)) = I^*(P_i^*) \tag{1}$$

Suppose we have had an approximation $\widehat{G}$ of $\overline{G}$, the problem consists in finding an incremental transformation $G(\mathbf{x})$ (where the 8×1 vector $\mathbf{x}$ contains a local parameterization), such that the difference between the region in the image I (transformed with the composition $w(\widehat{G})\circ w(G(x))$ and the corresponding region in the image I^* is null. Therefore, tracking consists in finding the vector $\mathbf{x}$ such that $\forall i \in \{1,2,...q\}$, the image difference

$$y_i(x) = I(w(\widehat{G})\circ w(G(\mathbf{x}))(P_i^*)) - I^*(P_i^*) = 0 \tag{2}$$

Let $\mathbf{y}(\mathbf{x})$ be the $q\times 1$ vector containing the differences:

$$\mathbf{y}(\mathbf{x}) = [y_1(\mathbf{x}), y_2(\mathbf{x}), ...y_q(\mathbf{x})]^\top \tag{3}$$

The problem consists in finding $\mathbf{x} = \mathbf{x_0}$ verifying:

$$\mathbf{y}(\mathbf{x_0}) = \mathbf{0} \tag{4}$$

We linearize the vector $\mathbf{y}(\mathbf{x})$ around $\mathbf{x} = \mathbf{0}$ using a second-order Taylor series approximation:

$$\mathbf{y}(\mathbf{x}) = \mathbf{y}(\mathbf{0}) + \mathbf{J}(\mathbf{0})\mathbf{x} + \frac{1}{2}\mathbf{x}^\top\mathbf{H}(\mathbf{0})\mathbf{x} + \mathbf{O}(\|\mathbf{x}\|^3) \tag{5}$$

where $\mathbf{J}(\mathbf{0})$ and $\mathbf{H}(\mathbf{0})$ are the Jacobian matrix and Hessian matrix at $\mathbf{x} = \mathbf{0}$, separately. In the ESM algorithm, the Hessian matrices of vector $\mathbf{y}(\mathbf{x})$ are replaced by a first-order Taylor Series approximation of vector $\mathbf{J}(\mathbf{x})$ about $\mathbf{x} = \mathbf{0}$:

$$\mathbf{J}(\mathbf{x}) = \mathbf{J}(\mathbf{0}) + \mathbf{x}^\top\mathbf{H}(\mathbf{0}) + \mathbf{O}(\|\mathbf{x}\|^2) \tag{6}$$

Then Eq. 5 becomes

$$\mathbf{y}(\mathbf{x}) \approx \mathbf{y}(\mathbf{0}) + \frac{1}{2}(\mathbf{J}(\mathbf{0}) + \mathbf{J}(\mathbf{x}))\mathbf{x} \tag{7}$$

For $\mathbf{x} = \mathbf{x_0}$, we have

$$\mathbf{y}(\mathbf{x_0}) = \mathbf{y}(\mathbf{0}) + \frac{1}{2}(\mathbf{J}(\mathbf{0}) + \mathbf{J}(\mathbf{x_0}))\mathbf{x_0} = \mathbf{0} \tag{8}$$

With some mathematical proof in [6], the sum of Jacobian matrix $\frac{1}{2}(\mathbf{J}(\mathbf{0}) + \mathbf{J}(\mathbf{x_0}))$ can be written as one matrix $\mathbf{J}_{esm}$. Therefore, for $\mathbf{x} = \mathbf{x_0}$, we have

$$\mathbf{y}(\mathbf{x_0}) = \mathbf{y}(\mathbf{0}) + \mathbf{J}_{esm}\mathbf{x_0} = \mathbf{0} \tag{9}$$

The solution $\mathbf{x_0}$ can be obtained by:

$$\mathbf{x_0} = \mathbf{J}_{esm}^+\mathbf{y}(\mathbf{0}) \tag{10}$$

$\mathbf{J}_{esm}^+$ is the pseudoinverse matrix of $\mathbf{J}_{esm}$. The homography $\overline{G}$ can be calculated with this $\mathbf{x_0}$ with Lie Algebra operation. With this homography $\overline{G}$, an elegant visual servo scheme was proposed with a special task function and control law, whose computation only depend on the $\overline{G}$ [6]. In our visual servo system, we adopt this homography-base control strategy.

B. SIFT algorithm

Among the feature based matching approaches, the SIFT [10] algorithm has been demonstrated to have a good performance with respect to variations in scale, rotation, and translation. However, it involves a computationally intensive high-dimensional descriptor extraction and is difficult to apply for realtime applications. To improve its performance and accelerate processing speed, various algorithms have been proposed, including PCA-SIFT [13] and SURF (Speed up Robust Features) [14]. And there are also GPU SIFT implementations [15] [16]. In our system the GPU based implementation SiftGPU provided by Changchang Wu is selected [17] and extended to find a homography solution. It exploits the processing power of a GPU to achieve a significant speedup (about 20 times) over the CPU implementations.

Besides, our combination of the GPU-SIFT and GPU-ESM algorithms is following the "Incremental Focus of Attention" (IFA) architecture [18]. When conditions are good, GPU-ESM tracking is accurate and precise; as conditions deteriorate, more robust but less accurate GPU-SIFT algorithm takes over. By this means we can realize robust, adaptive, real-time visual tracking applications.

III. System implementation

Our GPU implementations of the ESM and SIFT algorithms are carried out on a desktop with Intel Core i7-920 2.67GHz, 3GB RAM and a NVIDIA GTX295 board. The GTX295 board integrates two GTX280 GPUs inside and has 896MB GPU RAM for each GPU. Development environment is Windows XP (sp2) with NVIDIA's CUDA (Compute Capability 1.3).

A. GPU programming

For hardware details, each GTX280 has 30 Streaming Multiprocessors (SMs) and each SM has 8 Scalar Processor (SP) cores. In CUDA, functions are expressed as kernels and the smallest execution unit on a GPU is a thread. In CUDA the thread hierarchy is defined as follows: a CUDA kernel is a grid of threads. A grid is composed of several blocks of threads and a block is composed of several threads. As one CUDA kernel just completes a certain task like an ordinary C function, multiple CUDA kernels are needed to realize different functions in one algorithm.

B. GPU-ESM

Our GPU-ESM algorithm is categorized into 6 CUDA kernels.

1) Warping. This kernel warps the template image to the current image with a known homography.

2) Gradient. This kernel calculates the image intensity gradient in the X and Y directions.

3) JESM. It obtains the $\mathbf{J}_{esm}$ matrix in the Eq. 9.

4) Solving. This kernel finds a solution $\mathbf{x_0}$ of the equations

$$\mathbf{J}_{esm}\mathbf{x_0} = \mathbf{y}(\mathbf{0}) \tag{11}$$

5) Updating. This kernel updates the homography with solution $\mathbf{x_0}$ from the "Solving" kernel.

6) Correlation. This kernel calculates the correlation of the warped area and the template area to determine when to stop the ESM loops. As an iterative minimization algorithm, such threshold is necessary to stop the loops.

C. GPU-SIFT

We transfer Changchang Wu's code to our GPU-SIFT algorithm and extend it to a homography solution with RANSAC. The processing loop for an image is as follows.

1) SIFT feature extraction. The SIFT features in the current image and the reference image are extracted in this step. The extraction from a reference image can be computed in advance as it stays the same during the tracking process.

2) SIFT feature matching. The corresponding pairs of feature points in the two images are matched in this step.

3) Solving. After applying Changchang Wu's code in the previous steps, the RANSAC method is exploited to find the homography between the corresponding pairs of feature points. RANSAC shows a better performance than the ordinary least squares methods as it can effectively remove the outliers (the mismatched pairs of feature points).

D. Co-processing model of CPUs and GPUs

We assigned GPU-SIFT and GPU-ESM on separate GTX280 GPU. As a GPU can only run as a coprocessor of a CPU, corresponding CPU threads to control each GPU are needed. This is implemented by utilizing a multi-thread programming model on the CPU.

The final important item is how to effectively combine the GPU-ESM and GPU-SIFT to improve the system performance. Both algorithms run simultaneously on two GPUs. But the homography from GPU-SIFT is only used when GPU-ESM fails to track the object. Therefore it is important to set some criteria to detect whether the current GPU-ESM tracking fails or not. If GPU-ESM tracking failure is detected, the GPU-ESM thread will automatically read the homography result from the GPU-SIFT thread. From our practical experience, we adopt such criteria:

1) Image coordinate changes of the tracking area. For simplicity, the template area is selected with a rectangular shape. If the coordinates of any one of the four corner points change larger than a preset threshold (10 pixels in our application), it will be treated as an ESM tracking failure.

2) Correlation from the kernel "Correlation". The Zero mean Normalized Cross Correlation (ZNCC) is used as:

$$\frac{\sum_{k=1}^{q}(I(k)-\overline{I})(I^*(k)-\overline{I^*})}{\sqrt{\sum_{k=1}^{q}(I(k)-\overline{I})^2\sum_{k=1}^{q}(I^*(k)-\overline{I^*})^2}} \tag{12}$$

where $\overline{I}$ and $\overline{I^*}$ are the mean values of the warped area I and template area I^*, respectively. If the correlation value is smaller than a preset threshold(0.6 in our application), it will be treated as tracking failure.

In GPU-ESM algorithm, after each processing loop for one image, such homography quality evaluation is carried out with the above criteria. If either of the criteria is reached, GPU-ESM will alternatively load the homography from GPU-SIFT. In our visual servo application, after detecting a tracking failure, the robot will not move until ESM tracking successfully continues working. Therefore the negative effect of the control information discontinuities can be removed.

IV. Optimization

In this section, we describe our optimization techniques in our GPU-ESM implementation. Though CUDA uses C language with several extensions which make it easier than other GPU languages, to make GPU code highly proficient, carefully optimization must be exploited and several important factors must be considered.

A. Code parallelization

Algorithms must be carefully parallelized so that they can fully use the parallelism of a GPU, otherwise no obvious speedup can be obtained. Amdahl's law [19] specifies the maximum speedup (S) that can be expected by parallelizing portions of a sequential program as

$$S = \frac{1}{(1-P)+\frac{P}{N}} \tag{13}$$

where P is the fraction of the total serial execution time taken by the portion of code that can be parallelized, N is the number of processors on which the parallel portion code runs. When P is small, i.e. the code is not effectively parallelized, no matter how many cores you use (N), there will be little improvement. So maximizing the amount of code that can be parallelized is the most important factor. For simplicity, we only take the matrix multiplication to show the important role of the code parallelization.

With CUDA Profiler we find that the kernel "Solving" takes most of the running time. Therefore optimization of this kernel is important. In the CUDA kernel "Solving", we need to solve the overdetermined equation Eq. 11:

$$\mathbf{J}_{esm}\mathbf{x_0} = \mathbf{y}(\mathbf{0})$$

$\mathbf{J}_{esm}$ is $M \times 8$, $\mathbf{y}(\mathbf{0})$ is $M \times 1$, solution $\mathbf{x_0}$ is 8×1. M is the number of pixels in the template area. With least square methods, it consists in solving the following equation:

$$\mathbf{J}_{esm}^{\top}\mathbf{J}_{esm}\mathbf{x_0} = \mathbf{J}_{esm}^{\top}\mathbf{y}(\mathbf{0}) \tag{14}$$

$\mathbf{J}_{esm}^{\top}$ is the transpose matrix of $\mathbf{J}_{esm}$. Therefore, GPU implementation of matrix multiplication ($\mathbf{J}_{esm}^{\top}\mathbf{J}_{esm}$) and ($\mathbf{J}_{esm}^{\top}\mathbf{y}(\mathbf{0})$) are necessary. At first we tried the CUBLAS library. It is simple to use but the result is not satisfactory (see Fig. 3). Therefore specific CUDA kernels are needed to obtain a better performance. We use block matrix multiplication and compare two different block division methods in Fig. 3.

In method 1, each CUDA block is composed of 512 threads and each block calculates a block matrix multiplication (8×64 by 64×8) to get an 8×8 intermediate result (Fig. 1). After all the blocks obtaining their results, a second summary kernel will calculate the final result by summarizing all the intermediate results.

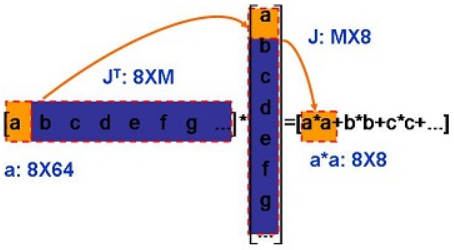

Fig. 1. Method 1

In method 2, each block contains 256 threads and calculates 1 element of the result (Fig. 2). The number of blocks is 64.

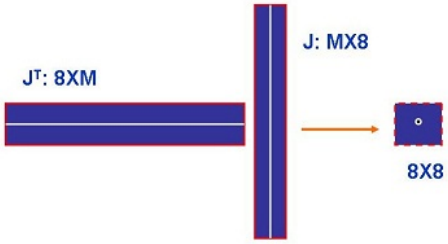

Fig. 2. Method 2

With CUDA Profiler these two methods' running time are shown in Fig. 3. The template pixel number M is selected from 64×64 to 384×384 pixels. The logarithmic scales are used on both the horizontal and vertical axes.

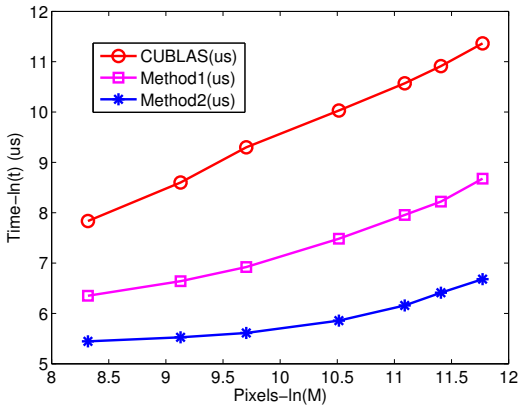

Fig. 3. Comparisons on processing time with M

It shows that method 2 is even faster than method 1 though both run on the same GPU. To investigate the difference in detail, we take $M = 300 \times 300 = 90000$ for an example. The block number in method 1 is $90000/64 \approx 1407$ and two steps are needed to get the final result. Meanwhile method 2 only needs 64 blocks and can get the result in one step. Limited by CUDA's compute capability, a GTX280 GPU can run at most 120 blocks simultaneously when 1 block contains 256 threads. So in method 2, all these 64 blocks can run parallel on a GPU. While in the method 1, it needs block scheduling because of so large a block number (1407) and it has to save all the 1407 intermediate results in global memory and load them again for next summary operation. As we will mention later, this kind of global memory fetching takes too much time. Method 1 essentially does not fully parallelize the code as some blocks have to wait for the accomplishment of other blocks during the block scheduling. Therefore, with fully code parallelization in method 2, significant speedup has been obtained.

B. Memory optimization

Memory optimization mainly consists in using different kinds of GPU memory to accelerate the application. CUDA provides a hierarchy of memory resources. Among them, commonly used are register, shared memory, global memory and texture memory.

In our GPU-ESM, we intensively utilize the fast shared memory instead of the long-latency global memory. In kernel "JESM", the computing of $\mathbf{J}_{esm}$ matrix needs several intermediate results based on the image gradient. So we first load the image gradient data into the shared memory of each block and then continue other computation on them. By using this "cache" like strategy, we reduce this kernel's processing time from 300us to 50us (300×300 size).

We also use texture memory in kernel "warping" because CUDA provides such bilinear filter function. We only need to set the filter mode to bilinear. When fetching the texture memory, the returned value is computed automatically based

on the input coordinates. Though the running time is similar to our own kernel but with it we can skip its programming.

C. Memory coalescing

By memory coalescing a half warp of 16 GPU threads can complete 16 global data fetching in as few as 1 or 2 transactions. In our application, we also intensively use this technique. For example, in the method 2 of matrix multiplication ($M = 90000$), each block (256 threads) needs to calculate the sum of 90000 products. As $90000 = 255 \times 352 + 240$, each tread will process 352 data (except the last thread with only 240 data). One common idea is using a "for-loop" in each thread:

$$for(k = threadID * 352; k < (threadID + 1) * 352; k++)$$

$$sum+ = data[k];$$

Each thread will process a continuous addressing memory, e.g. thread 0 accumulates data[0]∼data[351]; thread 1 processes data[352]∼data[703], and so on. To use memory coalescing, the code is changed to follows:

$$for(k = threadID; k < 90000; k+ = 352)$$

$$sum+ = data[k];$$

In this case, thread 0 accumulates data[0], data[352]...; thread 1 processes data[1], data[353]... and so on. Though both "for-loops" have the same performance for a CPU thread, speedup really happens on the GPU.

GPU memory is accessed in a specific block mode, i.e. each GPU memory access will load data from a block of continuous address memory space. For example, 16 threads can load $data[0] \sim data[15]$ simultaneously by 16 GPU threads. In the latter method, the loaded 16 data can be parallel processed by 16 GPU threads. Meanwhile, in the former method, only one of the 16 loaded data is used by one thread while all the other data is deserted. For each of the other threads, they must invoke their own GPU memory access to fetch the data they need. Therefore using memory coalescing strategy, we can substantially reduce the total number of GPU global memory access. Reducing such global memory access (usually takes several hundred GPU cycles) provides us with great performance gain.

V. Experiments

A. Running Time Proportions

We compare our GPU-ESM with CPU-ESM on the same desktop. The running-time proportion of each kernel is listed in Table I. The template area is set to $M = 300 \times 300$ pixels.

TABLE I
Running Time Proportions

Kernel	GPU Time(us)(%)	CPU Time(us)(%)
Warping	67.1(7.9%)	6942.4(23.6%)
Gradient	55.5(6.5%)	1552.1(5.3%)
JESM	74.9(8.8%)	6405.6(21.7%)
Solving	408.7(48.0%)	11692.5(39.7%)
Updating	19.6(2.3%)	4.4(0.01%)
Correlation	226.3(26.6%)	2853.9(9.7%)

B. Comparision of Processing speed

This part we compare the processing speeds of GPU-ESM and CPU-ESM with the same image sequence which has already been loaded into the CPU memory. Their processing frame rates are shown in Fig. 4 and Fig. 5. The template is selected from 32×32 to 360×360 pixels. The logarithmic scale is only used on the horizontal axe in both figures.

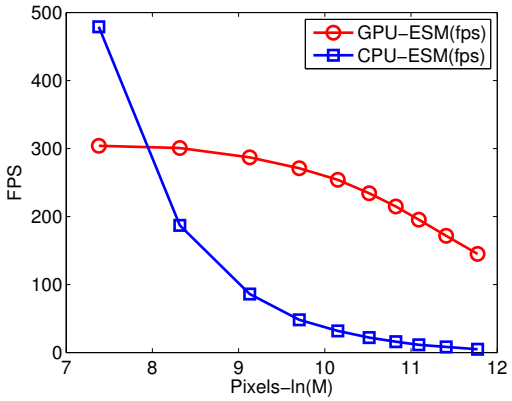

Fig. 4. FPS Comparision respecting to ln(M)

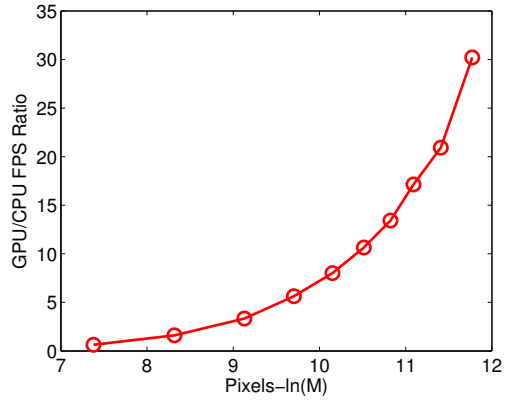

Fig. 5. GPU/GPU FPS Ratio respecting to M

It shows that using a GPU can greatly accelerate the ESM algorithm. Meanwhile, as the "GPU/CPU FPS Ratio" increases with M, it also shows that a GPU is more preferable for highly parallel processing.

C. Combination Performance

Experiments are carried out to evaluate the whole system. Images sequences extracted from the GPU-ESM and CPU-ESM tracking are shown separately in Fig. 6 and Fig. 7. Tracking area is a 200×200 window shown in $t = 0s$. The windows in the first row of Fig. 6 and Fig. 7 are warped back and shown in the second row. Despite illumination changes and image noises, the warped windows should be close to the template when the tracking is accurately performed. When the object moves slowly, both work well in tracking the object. With the increase of the moving speed, CPU-ESM can not track it well (Images at the boundary time are shown at $t = 9.8s$. From $t = 10.56s$ the warped area is obviously different from the template) while GPU still performs tracking well. At $t = 21.54s$ occlusion happens, with GPU-SIFT's homography the GPU-ESM can continue

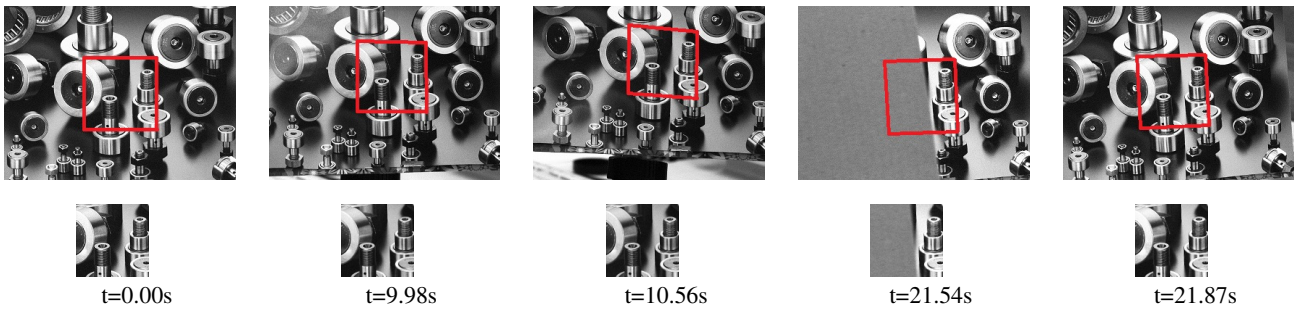

Fig. 6. GPU-ESM tracking. The second row shows the warped images from the boxed region in current images (the first row). The red quadrilateral shows that GPU-ESM can track the fast moving object even when occlusion happens.

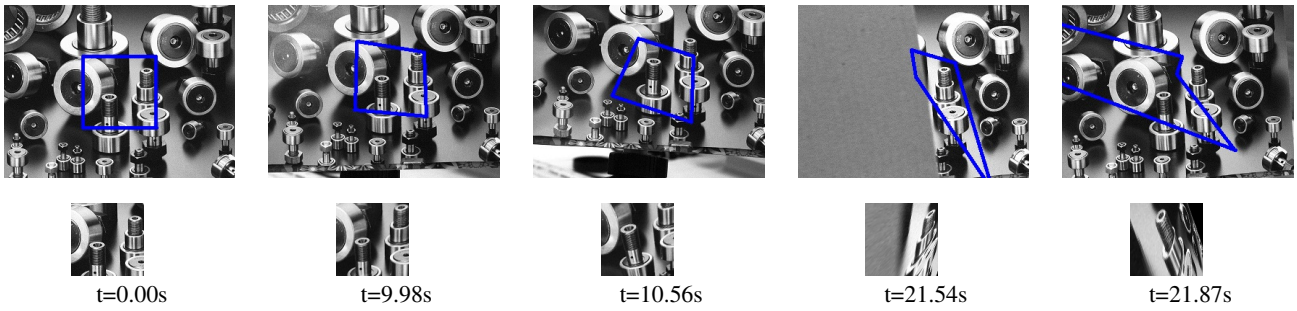

Fig. 7. CPU-ESM tracking. The change of warped images shows that CPU-ESM tracking can not track the same moving object as in Fig. 6 when the object moves fast or occlusion happens.

tracking the object during the occlusion(The red quadrilateral of GPU-ESM at $t = 21.54s$ means the tracking result is accurate). After the occlusion is moved($t = 21.87s$), GPU-ESM can continue tracking while CPU-ESM fails. Therefore the effectiveness of our combination strategy is confirmed.

VI. CONCLUSIONS

In this paper, CUDA implementations of GPU-ESM and GPU-SIFT are presented in a homography-based visual servo system. By utilizing optimization techniques including code parallelization, memory optimization and memory coalescing, GPU provides us a better system performance with a higher processing speed and reliability. Experimental results validate the effectiveness of our CUDA applications. By investigating the optimization techniques adopted in our implementations in detail, this study also makes contribution to the general purpose GPU computation community.

REFERENCES

[1] S. Hutchinson, G. D. Hager, P. I. Corke, "A tutorial on visual servo control", *IEEE Trans. on Rob. and Autom*, vol. 12, no. 5, pp. 651-670, 1996.

[2] O. Faugeras, F. Lustman, "Motion and structure from motion in a piecewise planar environment", *International Journal of Pattern Recognition and Artificial Intelligence*, 2(3): 485-508, 1988.

[3] M. Vargas, E. Malis, "Visual servoing based on an analytical homography decomposition", Joint 44th IEEE Conference on Decision and Control and European Control Conference, Seville, Spain, 2005.

[4] E. Malis, M. Vargas, "Deeper understanding of the homography decomposition for vision-based control", Research Report 6303, INRIA, 2007.

[5] Y. Fang, W. Dixon, D. Dawson, P. Chawda, "Homography-based visual servoing of wheeled mobile robots", *IEEE Trans. on Systems, Man, and Cybernetics - Part B*, 35(5): 1041-1050, 2005.

[6] S. Benhimane, E. Malis, "Homography-based 2d visual tracking and servoing", *International Journal of Robotic Research*, 26(7): 661-676, 2007.

[7] G. Silveira, E. Malis, "Direct Visual Servoing with respect to Rigid Objects", IEEE/RSJ International Conference on Intelligent Robots and Systems, San Diego, USA, October 2007.

[8] Shum, H. Y, Szeliski, R, "Construction of panoramic image mosaics with global and local alignment", *International Journal of Computer Vision*, 16(1): 63-84.

[9] S. Benhimane, E. Malis, "Real-time image-based tracking of planes using efficient second-order minimization", IEEE/RSJ International Conference on Intelligent Robots Systems, Sendai, Japan, 2004.

[10] D. G. Lowe, "Distinctive image features from scale-invariant keypoints", *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91-110, Nov. 2004.

[11] E. Malis, "Improving vision-based control using efficient second-order minimization techniques IEEE International Conference on Robotics and Automation", New Orleans, USA, April 2004.

[12] E. Malis, "An efficient unified approach to direct visual tracking of rigid and deformable surfaces", IEEE/RSJ International Conference on Intelligent Robots and Systems, San Diego, USA, October 2007.

[13] Y. Ke, R. Sukthankar, "PCA-SIFT: A more distinctive representation for local image descriptors", *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 511-517, 2004.

[14] H. Bay, Y. Tuytelaars, G. L. Van, "SURF: Speeded up robust features", *Computer Vision and Image Understanding*,vol. 110, pp. 346-359, 2008.

[15] S. Sinha, J.M. Frahm, M. Pollefeys, Y. Genc, "Feature tracking and matching in video using programmable graphics hardware", *Machine Vision and Applications*, March 2007.

[16] S. Heymann, K. Muller, A. Smolic, B. Froehlich, T. Wiegand, "SIFT implementation and optimization for general-purpose GPU", in WSCG'07, 2007.

[17] http://www.cs.unc.edu/ ccwu/siftgpu/

[18] K. Toyama, G. Hager, "Incremental Focus of Attention for Robust Vision-Based Tracking", *Int'l J. Computer Vision*, 35(1): 45-63, Nov, 1999.

[19] Amdahl, Gene, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", *AFIPS Conference Proceedings*, (30): 483-485, 1967.