2011 8th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)
Nov. 23-26, 2011 in Songdo ConventiA, Incheon, Korea

Optical Flow Computation on a Heterogeneous Platform

Jinsoo Oh[1], Eun-Jin Im[2*], and Kyoungro Yoon[3]
[1, 2]School of Computer Science, Kookmin University, Seoul, 136-702, Korea
(Tel : +82-2-912-4760; E-mail: astive84@gmail.com, ejim@kookmin.ac.kr)
[3]School of Computer Science and Engineering, Konkuk University, Seoul, 143-701, Korea
(Tel : +82-2-450-4129; E-mail: yoonk@konkuk.ac.kr)

bstract - **Unmanned Aerial Vehicles (UAV) are finding their way to a wide range of safety-critical missions. Our collaborative research team of computer scientists and aerospace engineers has worked on developing hardware and software of UAV. One of vital components in software used in UAV is image stabilization. The constantly-shaking images taken from the UAV, resulted from the vehicle's motion, need to be stabilized to perform its mission. In this paper, we present our implementation of image stabilization software. Our research is focused on using state-of-the-art Graphic Processing Unit (GPU) to improve the performance of the image stabilization software. The stabilizer estimates motion of the vehicle by calculating optical flow between successive two frames. In this study, we parallelized the calculation of the optical flow, which is identified as a computational bottleneck of the entire image stabilization process. Using the massive parallelism of NVIDIA C2060 GPU with 448 cores, we could improve the overall performance of image stabilizer.**

Keywords – GPU, CUDA, Optical Flow, Video Stabilization.

1. Introduction

Lately a variety of Unmanned Aerial Vehicles (UAV) are manufactured and serve a range of safety-critical missions, including military and disaster-control missions. Our collaborative research team of computer scientists and aerospace engineers has worked on developing hardware and software of UAV. "Fig. 1" shows one of UAVs, having 8 propellers, developed by our research team. One of vital components in software used in UAV is image

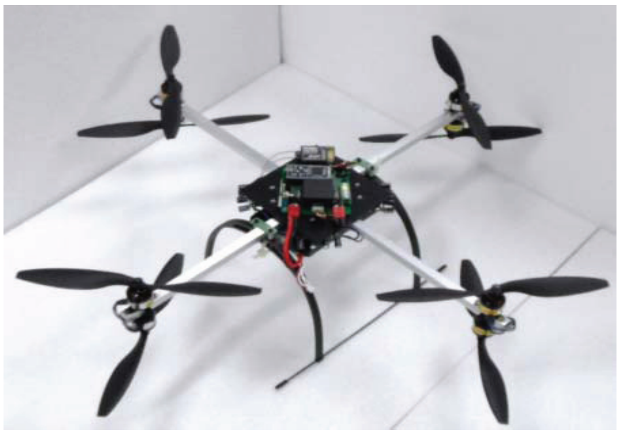
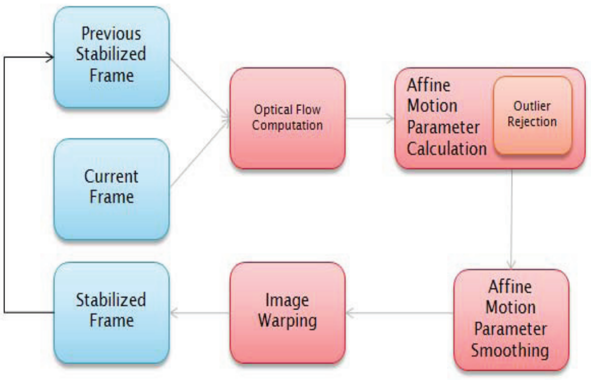


Fig. 1. Unmanned Aerial Vehicle, developed in our research team


Previous Stabilized Frame
Current Frame
Stabilized Frame
Optical Flow Computation
Affine Motion Parameter Calculation
Outlier Rejection
Affine Motion Parameter Smoothing
Image Warping
Fig. 2. Video Stabilization Structure

stabilization[1]. Most UAVs rely on images taken from cameras on UAVs in performing their mission. The lightweight UAV results in constantly-shaking images. This shaking motion degrades the quality of the video. The video stabilization is needed to overcome this degradation of video. There have been many studies for video stabilization. Generally video stabilization is categorized into two types; mechanical video stabilization and digital video stabilization. In the mechanical video stabilization, the video is stabilized by adjusting the motion of camera lens in the opposite direction to the angular velocity of the motion of the object. In digital video stabilization[2], the motion of two adjacent frames are estimated using the incoming video signals and the image data from charge-coupled device (CCD) are compensated. Global motion estimation of the camera is at the core in digital video stabilization, but there is no perfect solution to this. Commonly used camera motion estimation uses optical flow between two frames. There are two types of optical flows; a dense optical flow is calculated for all pixels of images, while sparse optical flow is computed for some of the pixels, designated as characteristic points. Because sparse optical flow calculates the motion of some of the pixels, processing time is shorter, but the accuracy is lower[3]. Our research is focused on using state-of-the-art Graphic Processing Unit (GPU) to improve the performance of the image stabilization software. The stabilizer estimates motion of the vehicle by calculating dense optical flows between successive two frames. This paper describes dense optical flow computation using NVIDIA compute unified device architecture (CUDA). In this study, we parallelized the calculation of the optical

[*] Eun-Jin Im is a corresponding author.

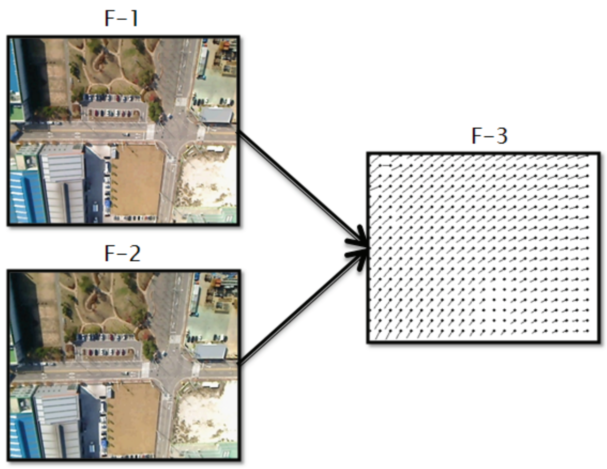


Fig. 3. Optical Flow Computation

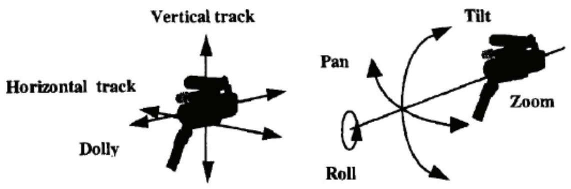


Fig. 4. Camera Motion

flow, which is identified as a computational bottleneck of the entire image stabilization process. We adopted Farneback algorithm[4] in computing dense optical flows. In the related works, the video stabilization and NVIDIA CUDA architecture are introduced. The following sections, we describe the parallelization of the video stabilization and its evaluation, and we conclude.

2. Related work

2.1 Video Stabilization

"Fig. 2" shows the flow of video stabilization software. First, the optical flow between each pair of frames. Then affine parameters are calculated using the computed optical flow. The next step is to remove outliers as a result of comparison between those affine parameters and optical flow of all pixels. The affine parameters recomputed then, and smoothed. The stabilized image frame is created after image warping in the final step. Each step is described further in the following.

A. Optical Flow Computation

Optical flow is the pattern of apparent motion of objects in a visual scene caused by the relative motion between a camera and the scene. "Fig. 3" shows the example of optical flow computing. F-1 was move toward northeast. So F-2 is a picture that after F-1 moved. In this case calculating optical flow, each pixel of the arrow indicates the north-east

represented like F-3. In video velocity of pixel is related with the amount of pixels to move between two adjacent frames. Calculating amount of all pixels to move is dense optical flow. Optical flow algorithms have been developed in the last few years. Horn and Schunck algorithm[5] is an iterative algorithm where the accuracy depends largely on

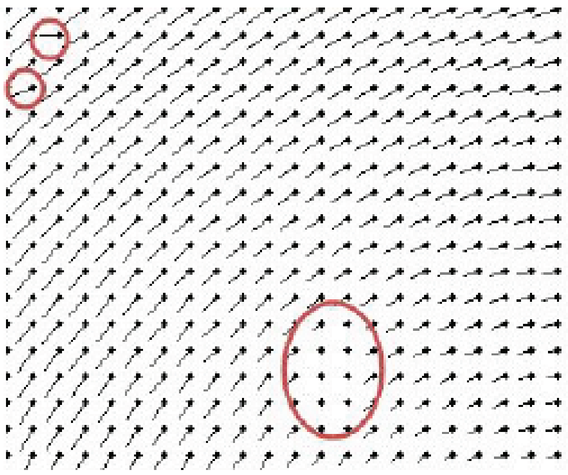


Fig. 5. Example of outliers

the number of iterations. The Lucas-Kanade algorithm[6] is good tradeoff between accuracy and processing efficiency. Farneback algorithm is accurate and fast tensor-based optical flow algorithm.

In recent years, a number of different schemes have been proposed to implement optical flow algorithms in real-time. An example Farneback optical flow algorithm is developed and implemented using FPGA technology[7].

B. Affine Motion Parameter Calculation

Affecting the video screen of the camera motion is seven kinds, horizontal translation, vertical translation, dollying, tilting, panning, rolling, zooming[8]. "Fig. 4" shows the camera motion of seven kinds. But if use the camera which moving in front of display device as a point device, horizontal translation and panning, tilting and vertical translation, darling and zooming has a same effect. So camera motion can define basic four motion (horizontal translation, vertical translation, dollying, rolling). And these camera motions are estimated by combination of affine parameter. Affine parameter calculate using least square method[9]. Least square method is calculates the value that the sum of square of value between actual value and measured value was minimum.

C. Outlier Rejection

"Fig. 5" is shows the example of outlier. Circled pixel has a different optical flow, this is outlier. Outlier is sensitive factor for calculating affine parameter. To getting the correct affine parameters, the outliers must be removed. If error is greater than the limit by comparing the optical flow and affine parameter, it is judged as outliers, as shown in "Fig. 5". The procedure is as follows removal of outliers.

- Calculate affine parameter from calculated optical flow set.
- Calculate error between two points by comparing the optical flow and affine parameter.
- If error is greater than the limit, recalculate affine parameter with optical flow set that error removed.
- Until outliers can't found, repeat procedure two, three.

D. Affine Motion Parameter Smoothing

Affine parameters are calculated from camera motion. These values has intended and unintended movement of the camera motion. So affine motion parameter smoothing[10] removes unintended camera movements.

E. Image Warping

Image Warping is the process that make stabilized frame using affine parameter. Using six the affine parameters makes the calculation of each pixel.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad (1)$$

As in (1), a and e are influenced to zoom in and zoom out, b and d are influenced to rotation of horizontal. And c affects the horizontal movement, f affects the vertical movement. x and y mean that the original location, x' and y' mean that destination location.

F. OpenCV

Open Source Computer Vision (OpenCV)[11] provides a library of programming functions for real time computer vision. This library is widely used in application related to computer vision. In our CPU implementation, we used OpenCV library to compute the optical flow

2.2 CUDA (Compute Unified Device Architecture) as a GPU architecture

Graphics Processing Unit (GPU) is a specialized circuit designed to rapidly manipulate and alter memory in such a way so as to accelerate the building of images in a frame buffer intended for output to a display GPU are used in embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPU for algorithms where processing of large blocks of data is done in parallel. Compute Unified Device Architecture (CUDA) is a parallel computing

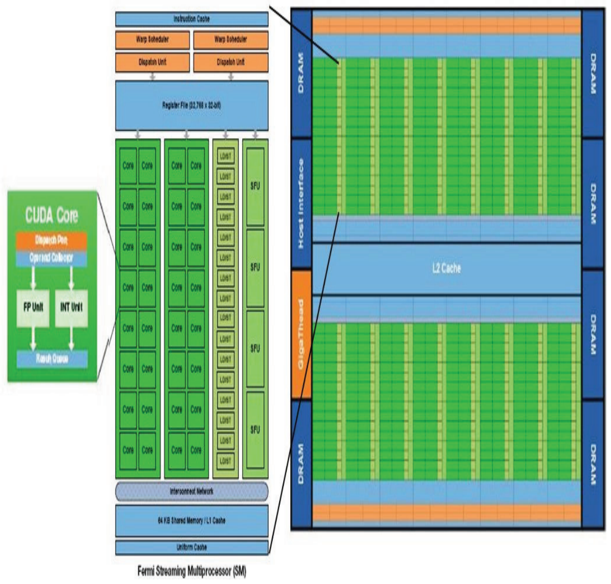


Fig. 6. Fermi GPU Architecture

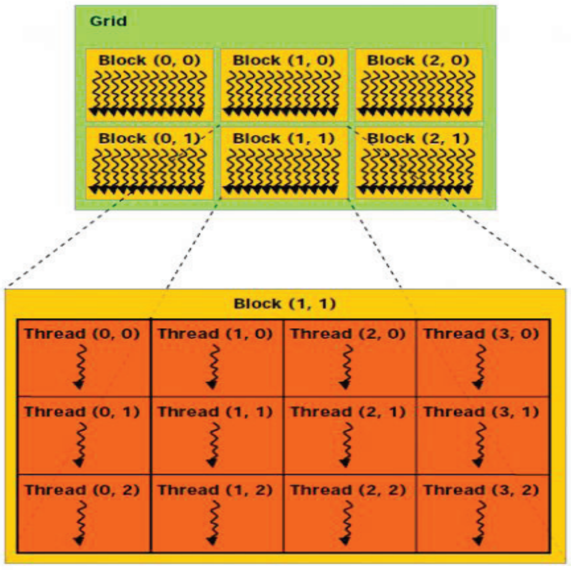


Fig. 7. CUDA thread hierachy

architecture developed by NVIDIA. "Fig. 6" illustrates state-of-the-art Fermi GPU architecture, designed by NVIDIA. There are many lightweight computational cores, organized in a number of groups, called 'streaming multiprocessor (SM)". The cores in the same SM share high-speed on-chip memory (registers and shared memory/L1 cache) and instruction scheduler/issuer. The memory in CUDA is organized in hierarchy, from high-speed on-chip registers and shared memory/ L1 cache, to global L2 cache, and then to low-speed off-chip DRAM called global (device) memory. A part of on-chip memory is used as read-only constant memory and texture memory. The L1 and L2 caches are the latest addition to the newest Fermi architecture. "Fig. 6" shows the memory architecture of CUDA. CUDA comes with its programming framework, a suite of a compiler, runtime libraries, driver APIs, and development tools such as profiler and debuggers, and CUDA refers to this programming framework sometimes. In the following section, we describe the CUDA programming framework, in relation to its CUDA architecture.

2.3 CUDA as a programming framework

CUDA is the computing engine in NVIDIA graphics processing unit that is accessible to software developers through variants of industry standard programming languages[12], The programmability of GPUs in general-purpose programming language is essential in using GPUs in general-purpose computing. In developing parallel program on NVIDIA GPU, programmers have choice of using a low-level C API directly and of using a C interface for CUDA, which is a runtime API built on top of CUDA driver API, compiled through a PathScale Open64 C compiler. In CUDA program, a C function which runs on CUDA is called "kernel", and the kernel is executed in a number of parallel "threads" on GPU cores. As with CUDA hardware hierarchy, the CUDA threads are organized in a hierarchy. "Fig. 7" illustrates the hierarchy

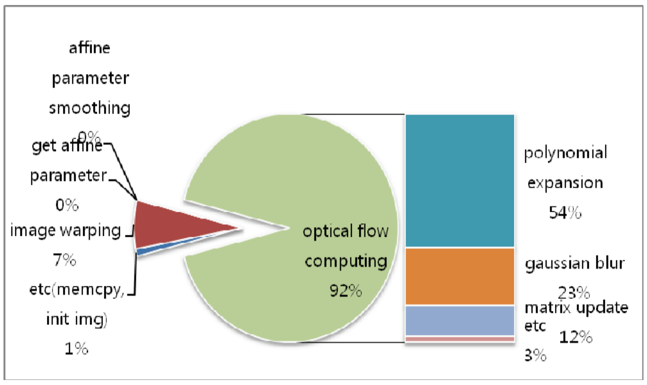


Fig. 8. Timing Breakdown of Video Stabilization

of CUDA threads. A number of threads are organized in a thread block, and a number of thread blocks compose one grid. When the kernel is executed on a GPU, the individual thread runs on a core, while a block of thread is assigned to a SM. A SM can run multiple number of thread blocks, and can schedule those thread blocks to hide its memory latencies. Threads in the same thread block shares high-speed shared memory and L1 cache, and those threads in the same thread block can synchronized. Since the threads in the same thread blocks are scheduled and issued in a unit of 32 threads, called 'warps', the control flows of the threads within a warp are advised to be converged for the sake of the performance.

3. Video Stabilization using CUDA

Video stabilization software should satisfy two important factors, quality of resulting video and the processing speed, at the same time. Previously described video stabilization method used the optical flow algorithm whose emphasis is put on the quality of the stabilized video at the cost of lower performance. In this section, we try to implement optimal video stabilization, which improves both the quality of the stabilized video and processing speed. We have expertise in parallelization and analysis of computing-intensive applications [13][14]. To improve processing speed, we parallelized the implementation using CUDA. In doing so, we located the performance bottleneck of the CPU implementation of optical flow algorithm, by measuring the consumed time in each component. We describe the process in detail

3.1 Identifying a Performance Bottleneck in Video Stabilization

located by measuring the elapsed time of each component in the code. Video stabilization process is composed of optical flow computation, affine parameter calculation, outlier rejection, affine parameter smoothing, and image warping as illustrated in "Fig. 2". When the timing of each function is measured, as shown in "Fig. 8", we could confirm that the optical flow computation occupies 92% of total processing time. So this location is the bottleneck of processing time. We further refined our timing to measure the consumed-time of components in optical flow computation. Optical flow computation consists of the following three functions, Polynomial Expansion, Matrix Update, and Gaussian Blur. The

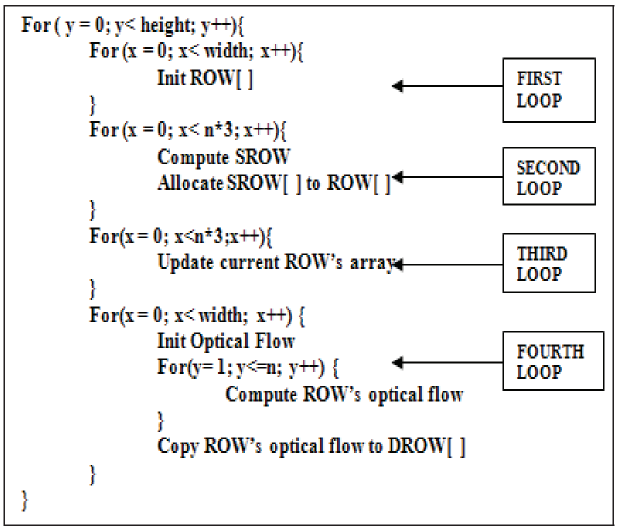

```
For ( y = 0; y< height; y++){
        For (x = 0; x< width; x++){
                Init ROW[ ]
        }
        For (x = 0; x< n*3; x++){
                Compute SROW
                Allocate SROW[ ] to ROW[ ]
        }
        For(x = 0; x<n*3;x++){
                Update current ROW's array
        }
        For(x = 0; x< width; x++) {
                Init Optical Flow
                For(y= 1; y<=n; y++) {
                        Compute ROW's optical flow
                }
                Copy ROW's optical flow to DROW[ ]
        }
}
```

Fig. 9. Serial Implementation of Polynomial Expansion

Polynomial Expansion function calculates optical flow between previous frame and current frame. The Update Matrix function computes single optical flow, by combining optical flow of previous frame and current frame. Gaussian Blur function eliminates the noise from the resulting image. In the refined timing, we noted that the Polynomial Expansion function occupied 54% of the time spent in optical flow computation, as depicted in "Fig. 8". As a result, improving Polynomial Expansion function on Optical Flow Computation would have a significant effect on overall processing speed.

3.2 Decomposing the Polynomial Expansion function

As a result of the previous analysis, we found out that the polynomial expansion function should be parallelized, in order to maximize the performance benefit. The Polynomial Expansion function is composed of four loops, and we parallelized those four loops as separate GPU kernels. The code in "Fig. 9" illustrates those four loops in polynomial expansion. In this code, ROW is an array of one row in the original image frame, and SROW is a temporary array used in calculating the values of ROW. DROW is an array of one row in the optical flow-computed frame using the values of ROW. In the first loop, array ROW is initialized. The second loop is used to copy SROW to ROW. The values of ROW are updated in the third loop. In the fourth loop, optical flows are computed and they are copied to DROW. All four loops iterate as many as 352 times, given the particular video resolution.

3.3 Parallelizing the Polynomial Expansion function

In parallelizing the polynomial expansion on GPU, those four loops in serial code are implemented as four separate GPU kernels. In this process, additional operations occur, which is a memory copy between the CPU and GPU. These operations take a lot of time and affects performance. In our experiment, we exclude this overhead of memory copy when measuring performance because memory copy and GPU computation can be overlapped. "Fig. 10" shows the CUDA kernel implementation of the polynomial expansion.

4. Evaluation

```
Void __global__ initROW()
        {
                Init ROW[ ]
        }

Void __global__ allocateSROW()
        {
                Compute SROW
                Allocate SROW[ ] to ROW[ ]
        }

Void __global__ updateROW ()
        {
                Update current ROW array
        }

Void __global__ computeOpticalFlow()
        {
                Init Optical Flow
                For(y= 1; y<=n; y++) {
                        Compute ROW's optical flow
                }
                Copy ROW's optical flow to DROW[ ]
        }
```

Fig. 10. CUDA kernels implementation of the Polynomial Expansion

In this section, we evaluate the impact of parallelization on the performance of the video stabilization. Those parameters that have impact on the performance include the number of blocks, the number threads per block, and use of the shared memory. The effects of those parameters are analyzed in this section.

4.1 The effects of the number of blocks and the number of threads on performance

The effects of the number of blocks and the number of threads per block are also dependent on the specific GPU implementation. In our experiment, we used NVIDIA Tesla-C2050 GPU, an implementation of the latest Fermi architecture. Tesla-C2050 have 14 streaming multiprocessors (SMs) and each SM consists of 32 cores. A CUDA thread block is executed on a SM, and each SM can schedule multiple number of thread blocks. To hide memory latency, it is advisable to have a multiple of the number of SMs as a number of thread blocks. The maximum repetition count of loop in Polynomial Expansion function is 352 for the configuration we use for our application. Therefore we measured performance, changing the number of threads per block and the number of blocks, taking those application-dependent configuration into account. As a result, we found four combinations of number of blocks and number of threads per block, 1 X 352, 11 X 32, 14 X 26, 42 X 9. Table 1 summarizes the combinations of the number of blocks and the number of threads per block used in our experiment.

4.2 The effects of using shared memory on performance

The effect of using shared memory on performance is strongly related to memory usage pattern. Access speed of the shared memory is significantly faster than that of global device memory, which is an off-chip memory. Thus, the shared memory on GPU is often used as a user-controlled cache. As such, we optimized our GPU

Table 1 Combination of blocks and threads per block and shared memory.

	Block	Thread	Shared Memory	Frame/Sec
GPU	1	352	O	20
	1	352	X	20
	11	32	O	22
	11	32	X	22
	14	26	O	22
	14	26	X	21
	42	9	O	24
	42	9	X	22
CPU				21

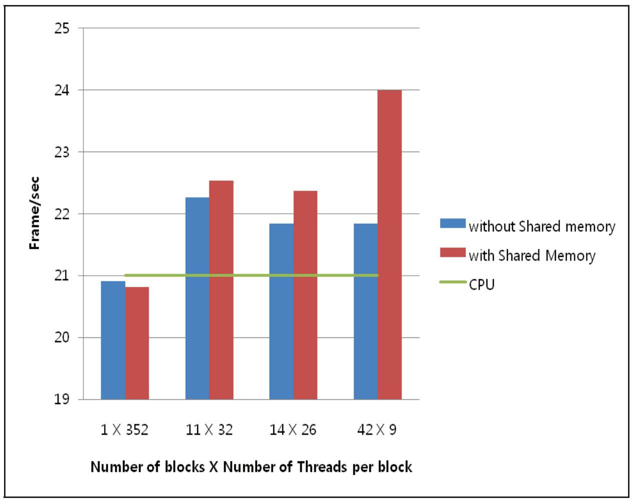


Fig. 11. Performance of Video Stabilization

implementation using shared memory as often as possible. Table 1 summarizes the measured performance of video stabilization on GPU with/without using the shared memory as a user-controlled cache.

4.3 Performance comparison between CPU and GPU

For our GPU implementation, we have searched the best combination of the number of blocks, the number of threads per block, and use of shared memory. In "Fig. 11", the performance of video stabilization on GPU is compared to the performance of CPU implementation, varying the execution configuration of GPU. The horizontal green bar is a reference performance of CPU implementation, which is 21 frames/s. In the leftmost bar, if we use only one thread block on GPU, it is slower than CPU. In such case, only 1/14 of the GPU resource is utilized while the remaining 13/14 resource is wasted since a block runs on a SM no matter how many threads the block has. As we increase the number of blocks to 11, 14, and 42, the performance improves. It is also confirmed that the performance is improved when the shared memory is used. In our experiment, the best execution configuration was when the number of blocks is 42 and the number of threads per block is 9, with shared memory. In fact, the total number of thread in this particular experiment is 352, and it means that the inherent parallelism in this application is much lower than the potential of the deployed GPU, which has 448 computational cores. If we had a larger video resolution, we could have expected a larger speedup, due to the abundant parallelism, inherent in the application.

5. Conclusion

In this paper, we presented an efficient real-time video stabilization. First we implemented the video stabilization algorithm on CPU, and then analyzed the consumed time of each component in this algorithm to locate the computational bottleneck of the algorithm. Then we parallelized the bottleneck in NVIDIA Fermi GPU (C2050). As a result, we showed that the performance is improved. The number of processed frames per second is improved to 24 (frames/s) from 21 (frames/s). And it can be further improved because our implementation parallelized only a part of video stabilization algorithm. The memory copy overhead between CPU and GPU can be also improved using an asynchronous data transfer between CPU and GPU. By implementing double buffering in GPU side, the computation and memory transfer and be overlapped to enhance the performance further. We expect that the performance of the overall software will improve through an implementation of asynchronous data transfer between CPU and GPU using double buffering.

Acknowledgement

This research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency)" (NIPA-2011-C1090-1131-0003) and by Basic Science Research program through NRF funded by the Ministry of Education, Science and Technology under grant number 2011-0005992.

References

[1] H. T. Cho, “Real-Time Stabilization Method for Video Acquired by Unmanned Aerial Vehicle”, Master’s thesis, Konkuk university, Feb 2010.

[2] H. C. Chang, S. H. Lai, and K. R. Lu, “A robust real-time video stabilization algorithm,” J. Vis. Commun. Image Represent., vol. 17, no. 3, pp. 659–673, Jun. 2006.

[3] H. Shen, Q. Pan, Y. Cheng, and Y. Yu, “Fast Video Stabilization Algorithm for UAV,” in proceeding of the IEEE International Conference on IntelligentComputing and IntelligentSystems, Vol.4, pp.542-546, 2009.

[4] G. Farneb¨ack, “Two-frame motion estimation based on polynomial expansion,” in Proc. of the 13th Scandinavian Conf. on Image Analysis, June-July 2003, vol. 2749 of LNCS, pp. 363–370

[5] Horn, B. K. P. , Schunck, B. G., “Determining opticalflow”, Artificial Intelligence 17,1981, 185–204.

[6] B. D. Lucas, and T. Kanade, “An iterative image registration technique with an application to stereo vision,” IJCAI, 1981.

[7] Z. Wei, D-J. Lee, B. Nelson, and M. Martineau, “A fast and accurate tensor-based optical flow algorithm implemented in FPGA”, Proceddings of the IEEE Workshop on Applications of Computer Vision(WACV ’07), p.18, Feb 2007

[8] M. V. Srinivasan, S . Venkatesh, and R. Hosie, “Qualitative estimation of camera motion parameters from video sequences,” Pattem Recognit., vol. 30, no. 4, pp. 593-606, Apr. 1997.

[9] Donald W. Marquard, “Analgorithm for leant-squares estimation of nonlinear parameters,” Journal of the Society for Industrial and Applied Mathematics, 11(2), June 1963.

[10] Savitsky A, Golay MJE, “Smoothing and dIfferentiation of data by simplified least squares procedures,” Analytical Chemistry, 1964.

[11] G. Breards, A. Kaehler, "Learning OpenCV," O'REILLY, 2008.

[12] NVIDIA, NVIDIA CUDA Compute Unified Device Architecture: Programming Guide (Version 3.2), 2010

[13] K, Madduri, E. J. Im, K. Ibrahim, S. Williams, S. Ethier, L. Oliker, “Gyrokinetic Toroidal Simulations on Leading Multi- and Manycore HPC System”, Proceedings of Supercomputing 2011, to appear.

[14] K, Madduri, E. J. Im, K. Ibrahim, S. Williams, S. Ethier, L. Oliker, “Gyrokinetic Particle-in-Cell Optimization onEmerging Multi- and Manycore Platforms”, Journal of Parallel Computing, to appear in 2011.