Real-time Multi-view Depth Generation Using CUDA Multi-GPU

Eunsang Ko, Yunseok Song, and Yo-Sung Ho
Gwangju Institute of Science and Technology (GIST)
123, Cheomdangwagi-ro, Buk-gu, Gwangju 500-712, Rep. of Korea
E-mail: {esko, ysong, hoyo}@gist.ac.kr

***Abstract*—In this paper, we propose a real-time multi-view depth generation method using compute unified device architecture (CUDA) multi-graphics processing units (GPU). The objective is to generate multi-view depth maps in real-time. We employ eight color cameras and three depth cameras. After capturing multi-view color and depth data, we warp the depth information to color camera positions. Then joint bilateral filtering (JBF) is performed to fill empty regions. Such a procedure is accelerated by CUDA which is one of general-purpose computing on GPU (GPGPU). As a result, depth maps of eight views are generated at a rate of 23 frames per second (fps) on a single GPU computer. When using a multi-GPU computer, depth generation at 34 fps was achieved.**

Keywords—CUDA Multi-GPU; Multi-view Depth Generation;

I. INTRODUCTION

As the broadcasting system has grown, 3D contents have been developed at the same time. The 3D contents allow the viewer to experience natural depth perception. In general, multi-view depth maps are required to generate the 3D contents. Multi-view depth maps are acquired from multi-view camera system. Depth maps contain distance value between camera and object. Depth maps can be captured by depth cameras via time-of-flight (ToF) [1, 2].

However, due to low-resolution of depth camera, depth maps must be up-sampled more high-resolutions for creating 3D contents [3]. Besides, the number of depth cameras can be used up to three and each depth map is captured using one computer because of ToF frequency problem. Therefore, for generating multi-view depth maps, original depth maps are required image processing such as 3D warping and up-sampling [2].

Although up-sampler is slow, we can obtain up-sampled depth maps in real-time using compute unified device architecture (CUDA). CUDA has developed by NVIDIA Corporation. CUDA is one of general-purpose computing on graphics processing units (GPGPU). Actually, a lot of applications have achieved their objective using CUDA because GPU has many stream multiprocessor (SM), each SM can execute threads at the same time. Also, CUDA applied application can parallelize and optimize the operation using CUDA multi-GPU because of extendibility of CUDA [4, 5].

In this paper, we propose a real-time multi-view depth generation method using CUDA multi-GPU. The objective is to generate multi-view depth maps in real-time using color and depth images from a multi-view camera system.

Fig. 1 Multi-view camera system

II. MULTI-VIEW DEPTH GENERATION SYSTEM DESIGN

We set up the multi-view camera system as shown in Fig. 1. Eight color cameras (Basler Pilot piA1900-32gc GigE) are positioned above three depth cameras (Mesa Imaging SR4000). After configuring the multi-view camera system, we progress camera calibration, capture a single object at a blue-screen studio, and multi-view image rectification. Camera calibration is the process of estimating camera parameters. Multi-view image rectification is the process of correcting alignment discrepancies in color cameras onto a common image plane [2].

Next, for up-sampling the depth images, we warp the depth information to corresponding color camera positions. So we can acquire depth maps with the color image resolutions as long as the number of color cameras. Then we execute joint bilateral filtering (JBF) to fill empty regions using object information that is obtained by distinguishing blue-screen color.

Fig. 2 represents the overall procedure of multi-view depth generation.

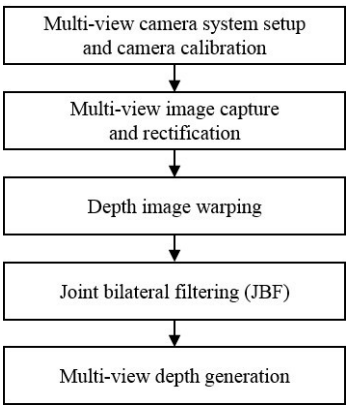

Fig. 2 Overall procedure of multi-view depth generation

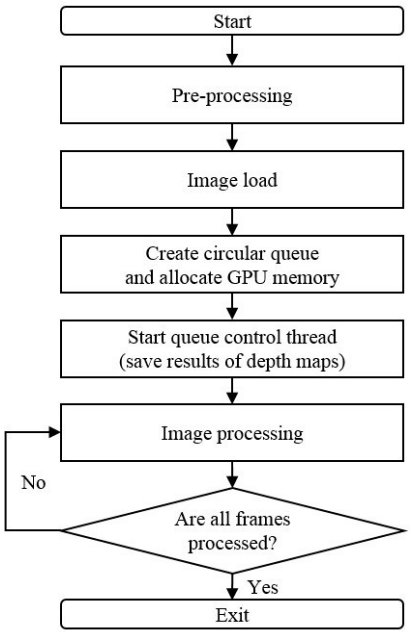

Fig. 3 Overall procedure of proposed multi-view depth generation method

III. Proposed Multi-view Depth Generation Method

The proposed multi-view depth generation method is processed by CUDA multi-GPU. Fig. 3 represents the overall procedure of proposed method. This method operation can be divided into the following five steps.

A. Pre-processing

Pre-processing comprises three steps: checking the number of color cameras, depth cameras, and graphics card, and pre-calculating warping matrices and Gaussian filter table for 3D warping and JBF, respectively. Pre-processing steps are essential for fast image processing.

1) Checking the number of cameras and graphics card: First, we check the number of color cameras, depth cameras and graphics card. Multi-GPU applied application is fast, but there are lots of restrictions. A typical restriction is each device cannot share their memory because each GPU has their own memory. Therefore, one depth camera cannot devide by multi-GPU because we need to access neighbor pixels of image in JBF. So the number of GPUs is decided as long as we use the depth cameras. Next, we find corresponding the depth cameras and color cameras. For example, when we use eight color cameras and three depth cameras, we can use up to three GPUs, one and another depth camera is warped to three color camera positions, and the other depth camera is warped to two color cameras, respectively.

2) Pre-calculating warping matrices: Equations (1) and (2) represent the 3D warping process. Subscripts *l* and *r* denote left and right, these mean source and destination, respectively. *A*, *R*, *T*, M_w, and *m* are intrinsic matrix, rotation matrix, translation matix, 3D image point, and 2D image point, respectively [2]. In this equation, before the 3D warping process, $R_l^{-1} \cdot A_l^{-1}$ and $R_l^{-1} \cdot t_l$ is calculated each depth camera and $A_r \cdot R_r$, and $A_r \cdot t_r$ is calculated each color camera, in advance respectively. Thus, we transfer these matrices to each GPU memory in advance for fast 3D warping process.

$$M_w = R_l^{-1} \cdot A_l^{-1} \cdot m_l - R_l^{-1} \cdot t_l \quad (1)$$

$$m_r = A_r \cdot R_r \cdot M_W + A_r \cdot t_r \quad (2)$$

3) Pre-calculating Gaussian filter table: (3), (4), and (5) represent the formulas necessary for Gaussian filter table generation. This table is used to generate the depth values in JBF process. (*u*, *v*) and (*x*, *y*) are image coordinates. We can create this table by inputting radius and σ value. Thus, we transfer this table to constant memory of each GPU in advance for fast JBF process.

$$g(u,v) = \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \quad (3)$$

$$u = \{x-r, \cdots, x+r\} \quad (4)$$

$$v = \{y-r, \cdots, y+r\} \quad (5)$$

B. Image load

First, we load all color and depth images that will be processed before starting image processing because image load from files is slow. Next, we convert all loaded images to 4-byte data types because GPU memory access and operation are fast at 4-byte units. Color images are represented by pixel values in 4-byte float types per channel. Depth images contain camera-to-object distance data, stored in 4-byte float types. We create arrays that collect converted color and depth data. Thus, we simplify the image data transfer to each GPU memory using the 4-byte float type arrays.

C. Circular queue for GPU memory

We use a circular queue that is implemented using the C++ standard template library (STL) for fast image processing [6]. All spaces of the circular queue are allocated as GPU memory. The results of generated depth maps are saved to single space of circular queue. Each index of the circular queue is decided by modulo operation of current frame number by circular queue size. All spaces of the circular queue can be reused if results of generated depth maps are completely processed. Fig. 4 shows an example of the circular queue memory status. Indices 0 and 1 of the circular queue are completely processed. Indices of 2, 3 and 4 were generated depth maps but these are being waited for saving the results of generated depth maps, and the remainder is never used. Thus, the circular queue in the figure shows that the sixth frame is being processed.

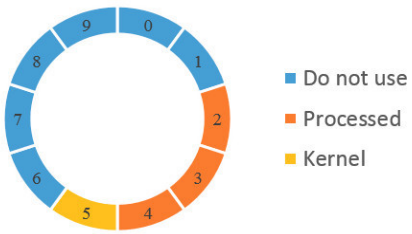

Fig. 4 Circular queue memory status

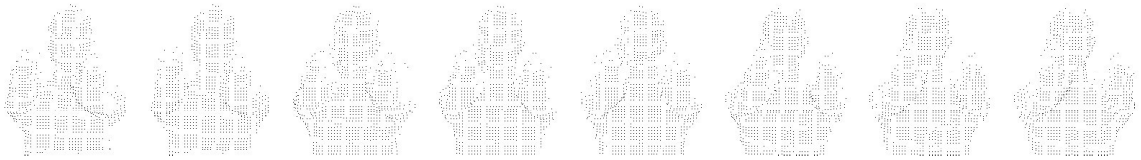

Fig. 5 Results of generated depth maps after 3D warping

Fig. 6 Results of generated depth maps after JBF

D. *Image processing*

Image processing comprises three steps: transferring the 4-byte float type arrays of color and depth data to each GPU memory, executing the 3D warping and JBF by calling a GPU kernel function, and receiving results of generated depth maps. Each GPU kernel function is called at the same time using parallel patterns library (PPL). The PPL partitions work in an optimal way given the available number of computing resources [7]. Thus, we reduce a GPU starvation using the PPL.

E. *Save results and free memory*

After one frame is completed image processing, results of generated multi-view depth maps are saved as image files, respectively. For promptly image processing of next frame, image saving is handled by CPU thread that was called before starting the image processing. This thread continually watches the circular queue whether multi-view depth maps are generated. This thread is exited with freeing used memories after saving all frames of generated multi-view depth maps.

IV. Experimental Results

We proceeded with five sets of experiments for the evaluation of multi-view depth generation performance using CUDA multi-GPU. The sets are defined in Table 1. We used the graphics card for all cases, GeForce GTX TITAN. The captured resolutions are 1280×720 and 176×144 for color and depth images, respectively. The filter radius for JBF is 15, and the number of processed frames is 90. Fig. 5 and Fig. 6 show the results of generated depth maps after 3D warping and JBF, respectively.

The depth generation speed only include image processing step in the proposed multi-view depth generation method operation. Set 2 and 5 show optimal conditions of multi-GPU is best for generating multi-view depth maps when one depth camera is processed by one GPU, respectively. Set 1 and 3 represent multi-view depth generation speed using single-GPU, respectively. On the other hand, set 4 shows double-GPUs performance is better than single-GPU, but the image processing cannot be optimized due to the GPU starvation that two color cameras and one depth camera are processed later.

TABLE I. Multi-GPU Experiment Methods

Set	*Color Cameras*	*Depth Cameras*	*The number of GPUs*	*Fps*
1	6	2	1	23.05
2	6	2	2	33.49
3	8	3	1	17.14
4	8	3	2	21.19
5	8	3	3	34.08

V. Conclusion

In this paper, we proposed a CUDA multi-GPU implementation for real-time generation of high-resolution multi-view depth maps. We warp low-resolution depth information to color camera positions and then execute JBF. We employ multi-GPU to accelerate the computation. As a result, eight views of depth maps were generated at 34 fps.

When watching multi-view depth generation speed, we can use real-time generated multi-view depth maps for generating 3D contents by implementing a module that directly receive the color and depth data from cameras.

Acknowledgment

This research was supported by the 'Cross-Ministry Giga KOREA Project' of the Ministry of Science, ICT and Future Planning, Republic of Korea (ROK). [GK13C0100, Development of Interactive and Realistic Massive Giga-Content Technology]

References

[1] J. Jung and Y.S. Ho, "Multi-view Image Generation Using Single-view Color Image and Low-resolution Depth Map," International Conference on Embedded Systems and Intelligent Technology (ICESIT), pp. 114-117, Jan. 2013.

[2] Y. Song, D.W. Shin, E. Ko, and Y.S. Ho, "Real-time Depth Map Generation Using Hybrid Multi-view Cameras," submitted to Asia-Pacific Signal and Information Processing Association (APSIPA), Dec. 2014.

[3] S.Y. Kim and Y.S. Ho, "Fast Edge-Preserving Depth Image Upsampler," IEEE Transactions on Consumer Electronics, vol. 58, no. 3, pp. 971-977, Aug. 2012.

[4] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, July 2010.

[5] CUDA Toolkit Documentation v6.0, http://docs.nvidia.com/cuda

[6] Vector Class, http://msdn.microsoft.com/en-us/library/9xd04bzs.aspx

[7] PPL, http://msdn.microsoft.com/en-us/library/dd470426.aspx