PARALLEL IMPLEMENTATION OF DEPTH-IMAGE-BASED RENDERING

Kun Xu, Xiangyang Ji, Ruiping Wang, Qionghai Dai

Department of Automation, Tsinghua University, Beijing, China
E-mails: xu-k05@mails.tsinghua.edu.cn, {xyji, rpwang, qionghaidai}@tsinghua.edu.cn

ABSTRACT

Depth-image-based rendering (DIBR) is a key step in 3D video generation. Parallel implementation of DIBR is able to improve rendering efficiency. General DIBR algorithms include two steps: pixel shifting (warping) and hole filling. There are memory correlations in these steps. To minimize memory conflict, we employ an auxiliary matrix to record maximum shifting distance. Implementation details on OpenMP and CUDA are presented and experimental results on GPU and multi-core CPU are compared.

Index Terms— DIBR, pixel shifting, hole-filling, parallel computation.

1. INTRODUCTION

In 3DTV technology framework, Depth image-based-rendering (DIBR) plays as a key component to generate virtual views of a scene from a video and its corresponding depth map [1]. It generates high-quality novel views and requires less computational cost than model-based rendering [2].

DIBR has to process a video shot pixel by pixel and frame by frame. High definite video clips, however, have millions of pixels in every frame, thus DIBR is a bottleneck in many 3D applications. A solution is to implement DIBR on parallel computing platforms.

There are several parallel computing systems. OpenMP and graphics processing unit (GPU) are two mainstream mode of parallel programming in PC and both of them could install and work in general computational platform, such as PC, laptop and portable devices. OpenMP is a compiler directive for the shared storage systems. It is convenient to transplant parallel code into C/C++ code. In recent years GPU has become more and more popular in high performance calculation (HPC) system. The performance and capability of GPU is remarkably increased [3]. Compute unified device architecture (CUDA) is a parallel programming environment on NVIDIA's GPU. It has been applied in many fields, such as image and video processing, bimolecular simulations and hydromechanics analysis.

In this paper, parallel implementation of DIBR on OpenMP and GPU is presented. Section 2 introduces the principle of DIBR and related work. Section 3 describes parallel algorithm designing and the optimization scheme of DIBR in parallel computing platforms. Section 4 demonstrates experiment result on OpenMP and CUDA and compares the results. Finally, Section 5 concludes the paper.

2. PRINCIPLE OF DIBR

DIBR efficiency and quality are extremely correlated with the quality of the depth map provided, therefore, some post-processing technologies for depth map are also included in DIBR. Zhang et al. employ symmetric and asymmetric Gaussian filters to reduce texture artifacts and distortion in vertical direction [4]. Experiment results from Shimono's group indicate that asymmetric Gaussian blur is able to remove cardboard effect [5]. Quang etc. employ bilateral filter and edge enhancement technique to optimize the quality of depth maps [6]. With the high quality of depth map, with little artifact and smooth object border, the DIBR algorithm could be very simple, typically include two steps: pixel shifting and hole filling.

Pixel shifting is a dual projection which maps original image plane to 3D scene space defined by corresponding depth map and then projects it to corresponding virtual view plane. Fehn described the projections in warping in detail [7]. In this paper, we consider binocular model, i.e. generating left and right virtual views. The geometry is shown in Fig. 1. Two viewpoints X_0 and X_1 are located on the x-axis. The distance between X_0 and X_1 is L. The goal of DIBR is to ensure each of the two viewpoints is correctly displayed in the screen plane P. Points A and B are the pixels in original image plane. Pixels with positive depth value are out of the screen, such as point A', or with negative depth value are deep in the screen, such as point B'. Pixel shifting step is used to calculate pixels' shift distance between virtual view and original image. For point A, the virtual pixel in X_0 and X_1 is marked as A_L and A_R, respectively. The shift distance for point A in X_0 and X_1 is calculated by following equation:

$$d_{\mathrm{AL}} = L_{\mathrm{AL}} \times \frac{D_{\mathrm{A}}}{(Z - D_{\mathrm{A}})} \quad (1)$$

$$d_{\mathrm{AR}} = L_{\mathrm{AR}} \times \frac{D_{\mathrm{A}}}{(Z - D_{\mathrm{A}})} \quad (2)$$

Generally, Z is much larger than D_A ($Z \gg D_A$). Then, $L_{AL} \approx L_{AR}$ and $d_{AL} \approx d_{AR}$. The equation (1) and (2) are simplified by one equation:

$$d_{\mathrm{AL}} = d_{\mathrm{AR}} = d_{\mathrm{A}} = \frac{L}{2} \times \frac{D_{\mathrm{A}}}{Z} \quad (3)$$

The shift distance for other pixels (such as B) is simplified as follows:

$$d = \frac{L}{2} \times \frac{D}{Z} \quad (4)$$

Generally, obtain a better visual comfort level, we often change the zero parallax plane (ZPP). The ZPP will impact the whole shift distance in image. The equation (4) is modified as:

$$d = \frac{L}{2} \times \frac{D-ZPP}{Z} \quad (5)$$

where ZPP is between 0 and 255 for consistence with 8 bits depth map. Therefore, the pixel value in virtual views and the shift distance have the following relationship:

$$v_{(x,y)} = v_{(x+d,y)} \quad (6)$$

In (6), v is the pixel value in color space. Therefore, the pixel value in virtual views can be completely settled.

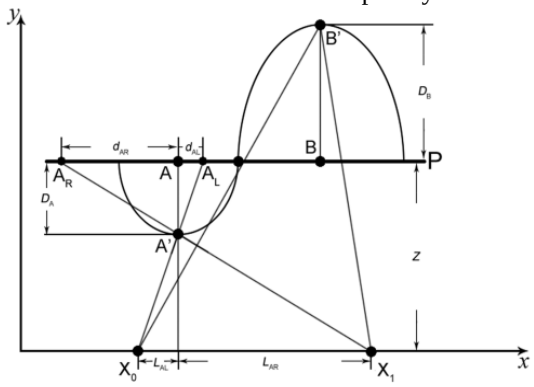

Fig. 1. Geometry of pixel shifting step.

The principle of disocclusion and hole-filling step is shown as Fig. 2. In Fig. 2, pixel A and C are adjacent pixels. C cannot be seen by view X_0 that means A occludes C in view X_0. In that case, the pixels between A and C in view X_1 are holes, because these pixels have no corresponding pixels in original image and they have no value in depth and color. There are several methods to fill holes, such as nearest neighbor, interpolation or In-painting algorithm. Zhang et al. use information of averaging textures from neighborhood pixels to fill holes [4].

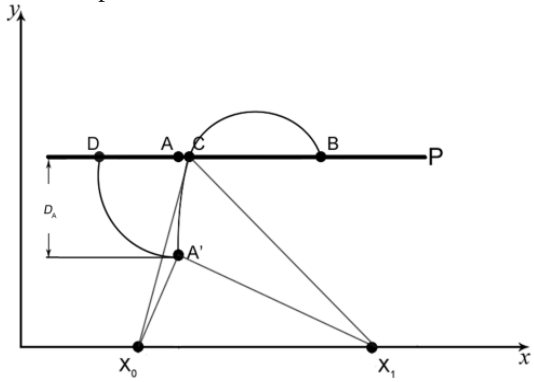

Fig. 2. Geometry of hole-filling step**.**

3. PARALLEL ALGORITHM DESIGN

In this section, we present the implementation details of DIBR on CUDA and OpenMP platforms.

3.1. Data Partition

DIBR must be processed pixel by pixel. In hole-filling step, with the nearest pixel filling method, the pixel value is much independent and there is little shared memory between each thread. Therefore, it is enough to use uniform partition method on image data in which each pixel is a minimum division unit. Because of no communication between each partition unit (thread), it is no need to design communication strategy between units.

Since DIBR is implemented on different parallel computing devices, we organize partitions into different granularity for further computation. The arrangement format of image data is a 2-D matrix. There are two viable data-partition methods for matrix: striped partition and checker board partition. In OpenMP, Processor is usually limited to have 2 or 4 cores, thus we bind each block to a processor. Consequently, we use striped way to organize data and make block number equal to processor number. Each separated block is bound to a processor. Nevertheless, GPU is multi-core processor with scalable parallel architecture in CUDA. Partitioning blocks into checker board is a better way. We bind a square block to a stream multi-processor and each pixel in block executes on a stream processor, so that every pixel value in virtual views is calculated on independent processor. As shown in Fig. 3, we partition image data into striped blocks whose number is the same as CPU number in OpenMP. DIBR is calculated by row in block. In contrast, we partition image data into blocks with 16×16 pixels in each block (192 or 256 elements in a block for the best) in CUDA.

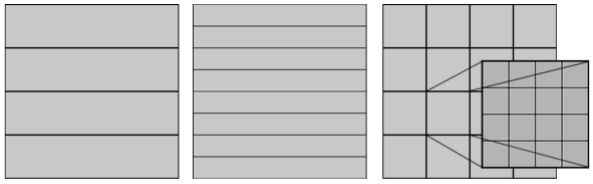

Fig. 3. Methods of data partition. (a) and (b) are situations of 4 and 8 cores in OpenMP, respectively. (c) is situation in CUDA**.**

3.2. Pixel Shifting

Input data include video frames and corresponding depth maps. A video frame is divided into 4 channels in order to increase addressing speed in storage, and the depth map is stored as float type in memory for higher computational accuracy. As mentioned in Section 2, warping step is a linear transforming process. The pixel shift distance (disparity) is calculated based on (5) using the depth value of the pixel with uniform coordinate. Then, the pixel value is filled with (6).

Occlusion and hole will emerge when the depth values between adjacent pixels are very different. For occlusions, one location shifts with more than one pixel; and for holes, no pixel shift to location in virtual view. Traditionally, we execute disocclusion and hole-filling steps to solve such

problems. The disocclusion search pixel and fill pixel with the maximum depth value. It means the filling operation will be carried out multi-time until reach the maximum depth location. That will seriously affect efficiency in parallel computation.

We propose a method to improve the parallel computation efficiency. We define a matrix M called "Shift Matrix" that pre-preserve shift distance of every pixel. We fill M with the following equation:

$$m_{(x,y)} = \max(D_{(x-d,y)}) \quad (6)$$

where m is the value of M in (x, y), and D is the depth value in depth map. According to (6), a pixel with the maximum depth in M will not fill with others. After this step, the maximum shift distance between original image and virtual view is confirmed. It can be predicted that the final location of shift pixel and the filling can be executed only once. This improvement could effectively reduce search time in shifting step. Meanwhile, pixel in M with no value indicates it is a hole.

Shifting operation for left eye view is illustrated in Fig. 4. The pseudo code is given in Algorithm 1.

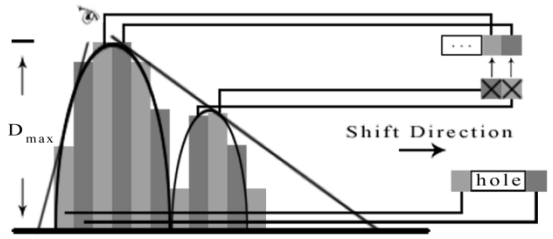

Fig. 4. The shift operation for left eye.

Algorithm 1 CUDA Parallel pixel-shifting (Left eye)

```
for each pixel (x, y) in block
    location←y*width+x
    d←abs(D[location]-D[location-1])
    if m[location+d]≤D
        m[location+d] ←D
        v[location+d] ←v[location]
    end if
end for
```

After this step, we get matrix M and half-finished virtual view image. Pixels no value in M are holes and their color value will be calculated in hole-filling step.

3.3. Hole-filling

Hole-filling step is an ill-posed problem. This means we compute pixel value in the hole regions only based on estimation. There are several methods for hole-filling. In parallel camera system, using pixels in same row to estimation value in hole is enough, we just consider nearest neighbor and interpolation algorithm. Using nearest neighbor is more efficient than interpolation and may achieve good effect when holes are small or discontinuous. Furthermore, it only needs to search one direction to fill pixel but interpolation with two directions. Since the search distance is unknown, two directions' search is probable out of block boundary and leads to memory conflict. In conclusion, nearest neighbor method is in favor of parallel computing.

In this step, the partition mode is the same as before. According to M, pixel's coordinates belong to holes are known, which makes it is sample to calculate their color value all alone. In left virtual view, the color value of pixel on holes equals to the nearest pixel on the left whose value exist in M, and the nearest pixel on the right for right view in the same way. That means, for every hole-pixel, we search M from the closest pixel to further pixel until find a pixel with value in M. Then, we fill hole-pixel with the value of it. In practice, OpenMP partitions image data into blocks in strip that could do searching operation directly. However, it is necessary to limit the search scope called "warp distance" in CUDA. This scope is used to avoid addressing length crossing block boundary and its value could estimate by parameters in (5). The over-long addressing leads to memory conflict and decreases computing efficiency. The executing process of hole-filling step in CUDA is described in Algorithm 2.

Algorithm 2 CUDA Parallel hole-filling (Left eye)

```
for each pixel (x, y) in block
    location←y*width+x
    if m[location] is NULL
        for i from x-1 to x-warpDistance
            if m[location+i] is not NULL
                v[location] ←v[location+i]
                m[location] ←m[location+i]
            end if
        end for
    end if
end for
```

Finally, the boundary cutting process for image is implemented to eliminate boundary discontinuity and visual discomfort. The pixels in cut boundary are set to black. The final virtual views are obtained then.

The matrix M has two functions in the algorithm. The first is to avoid memory writing for multi-threads simultaneously. The second is to provid reference mark in hole-filling. We can directly fill holes with M.

In order to avoid pointer rollback and memory conflict, the search direction in rows is unidirectional in virtual view rendering. It is left-to-right in left views and right-to-left in right views. Furthermore, uchar4 data type and shared memory are used in CUDA to store temporary variable and row data to fast computation speed.

The main contribution of this paper is: we propose a DIBR algorithm for parallel computation, which is on-line and of high quality. The algorithm is a common model, and any field needs to generate virtual view with depth map can execute this DIBR module. Meanwhile, Using GPU as parallel processing unit, it can work on PC for family multimedia applications.

4. EXPERIMENTAL RESULTS

We use the PC with Intel i7 920 CPU of 8-core 2.67GHz for OpenMP programming and GPU of NVIDIA GTX 285 with 30 multiprocessors for CUDA to experiment for 5 terms, including 1-core, 2-core, 4-core, 8-core CPU and GPU. In addition, we test image data in 4 resolutions (640×480, 720×576, 1280×720, and 1920×1080). The higher resolution represents larger amount of data volume. In pixel shifting step, we choose L = 0.06m and Z = 0.5m for general situation and the maximum depth value as ZPP which makes all objects deep in the screen plane in equation (5). The experimental results are shown in Fig. 5.

As shown in Fig. 5, after comparison with other OpenMP results, the CUDA speed is 12-15 times faster by 1-core processor, and nearly 3 times by 8-core processor. Because of limits of hardware structure, the 8-core running time is not 2 times by 4-core. The running time of CUDA is faster-than-real-time in almost 200 fps for 1080HD video, which is totally enough for on-line video display. Furthermore, the calculation time is linear-increased by resolution (data volume), which indicates the computation speed of DIBR is directly related with processor number.

Obviously, the proposed DIBR algorithm is executed quickly in parallel computation system. The results in OpenMP and GPU both have prominent enhancement. The calculation in CUDA is much effective and has greater performance than CPU.

The rendered virtual view is high-quality and has little coarse artificial effect. Fig. 6 shows the original image, depth map and virtual views using CUDA in 1920×1080 resolution.

5. CONCLUSION

In this paper, we propose a parallel DIBR scheme in PRAM model and compare executing speed in multi-core CPU and GPU. Experiment results show that parallel computation for the DIBR algorithm is effective. Whereas, because of using the nearest neighbor method, the border of objects in image is blocky artificial when depth map pixels are discontinuous which causes holes in virtual view huge and hard to fill. Next, we consider to apply filter which is parallelizable in a fixed window, such as 3×3 or 5×5, to improve this distortion.

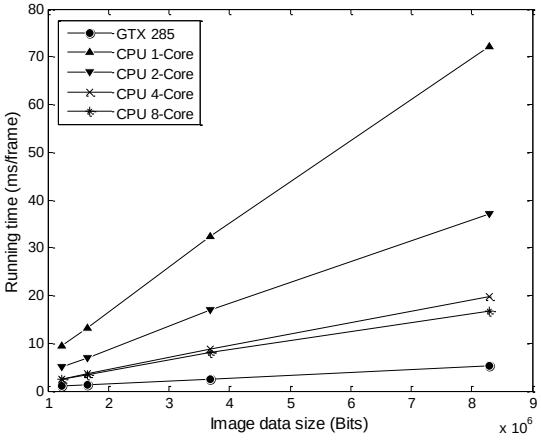

Fig. 5. Comparison running time of DIBR in OpenMP and CUDA.

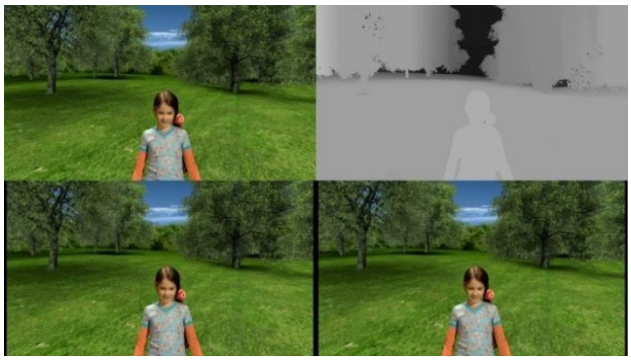

Fig. 6. Original image, depth map and rendered virtual left/right view in 1920×1080 Resolution. In DIBR, the parameters are L = 0.06m, Z = 0.5m and ZPP = 173, respectively.

6. ACKNOWLEDGEMENT

This work was supported by the National Basic Research Project of China (973 Program, No.2009CB320905), the Project of NSFC (No.60933006 & No. 60972013).

7. REFERENCES

[1] C. Fehn, R.D.L. BARRE, and S. Pastoor. Interactive 3-DTV—Concepts and Key Technologies. Proceedings of the IEEE, Vol. 94, No. 3, pages 524–538. March 2006.

[2] M.H. Lee, and I.K. Park. Accelerating Depth Image-Based Rendering Using GPU. MRCS 2006, LNCS 4105, pages 562–569. 2005.

[3] D.O. John, H. Mike, L. David, G. Simon, E.S. John, and C.P. James. GPU Computing. Proceedings of the IEEE, vol. 96, No. 5, pages 879–899. May 2008.

[4] L. Zhang, J.W. Tam. Stereoscopic Image Generation Based On Depth Images for 3DTV. IEEE Transactions on Broadcasting, Vol. 51, pages 191–199. June 2005.

[5] K. Shimono, W.J. Tam, C. Vazquez, F. Speranza, R. Renaud. Removing the Cardboard Effect in Stereoscopic Images Using Smoothed Depth Maps. Proc. SPIE, vol. 7524, pages 75241C1-75241C 8. 2010.

[6] H.N. Quang, N.D. Minh, J.P. Sanjay. Depth Image-based Rendering with Low Resolution Depth. IEEE International Conference on Image Processing, pages 553-556. 2009.

[7] C. Fehn. A 3D-TV Approach Using Depth-Image-Based Rendering (DIBR). Proc. of Visualization, Imaging, and Image Processing, pages 482-487. 2003.