REAL-TIME PARALLEL REMOTE RENDERING FOR MOBILE DEVICES USING GRAPHICS PROCESSING UNITS

Wucherl Yoo, Shu Shi, Won J. Jeon, Klara Nahrstedt, Roy H. Campbell

University of Illinois at Urbana-Champaign
Department of Computer Science
201 N Goodwin Ave, Urbana, IL, 61801, USA
Email: {wyoo5,shushi2,wonjeon,klara,rhc}@illinois.edu

ABSTRACT

Demand for 3D visualization is increasing in mobile devices as users have come to expect more realistic immersive experiences. However, limited networking and computing resources on mobile devices remain challenges. A solution is to have a proxy-based framework that offloads the burden of rendering computation from mobile devices to more powerful servers. We present the implementation of a framework for parallel remote rendering using commodity Graphics Processing Units (GPUs) in the proxy servers. Experiments show that this framework substantially improves the performance of rendering computation of 3D video.

Keywords— GPGPU, Parallelization, 3D Video, Remote Rendering, Mobile Devices

1. INTRODUCTION

As an emerging technology, 3D video has attracted many research interest. Existing projects like TEEVE [1] indicate a bright future for 3D video in the next generation of tele-communication and tele-immersion applications. Currently, 3D video-based applications are implemented on powerful workstations due to the requirements of massive computation capability and network bandwidth.

In order to meet the increasing demands of mobile computing, we have proposed a proxy-based remote rendering framework to stream 3D video to mobile devices over wireless networks in our previous work [2]. A proxy server, which is assumed to have sufficient computation and network resources, receives and pre-processes the original 3D video streams for mobile devices in the framework. Each 3D video frame is rendered and converted into multiple 2D image-based reference frames on the proxy server. The reference frames can be compressed for wireless transmission and rendered by efficient 3D warping algorithm on mobile devices. The selection of reference frames is critical to the visualization quality of 3D video rendering on mobile devices. Our previous work [2] discussed several algorithms on how to select the most appropriate reference frames. However, it did not address how to efficiently implement those computation intensive search algorithms.

In this paper, we present a parallel design of the proxy server in the remote rendering framework. We show how 3D warping and reference frame selection algorithms can be efficiently implemented on CUDA [3], a parallel computation environment for GPUs provided by NVIDIA. In addition, we present how components running on the GPU and the CPU can be effectively integrated to prevent unnecessary data transfer in-between. We compare the performance of the parallel implementation of the proxy server on CUDA with the CPU implemented version. Experimental results show that the performance improvement can be up to 13X. We believe our whole framework can provide not only 3D video streaming but also other generic dynamic 3D content rendering for mobile devices.

For the rest of the paper, we first briefly introduce background information: the TEEVE architecture, our proxy based remote rendering framework, reference frame selection algorithms, and the CUDA platform in Section 2. The design and implementation details of our parallel proxy server framework on CUDA are presented in Section 3 and Section 4. We evaluate our design in Section 5. Section 6 summarizes the related work, and Section 7 concludes the paper.

2. BACKGROUND AND MOTIVATION

2.1. 3D Video and TEEVE

Unlike traditional 3D graphics, 3D video is captured and reconstructed from the real world. 3D attributes allow the video objects to be rendered at arbitrary viewpoint or merged with a virtual 3D background. In general, "3D video" has been used in the literature for both stereo video and free viewpoint video [4]. Stereo video focuses on providing different views from each eye to give the impression of 3D depth. It is closely related to stereo vision and 3D display technologies. Free viewpoint of 3D video allows viewers to interactively select different viewpoints. Instead, this paper focuses on the 3D video with the attribute of free viewpoint navigation. Several approaches to 3D video have been proposed in the past [5][6][7]. They make use of multiple views of the same scene. Multiple cameras placed at different

978-1-4244-7493-6/10/$26.00

positions capture all facets of the object so that the user can freely navigate the scene.

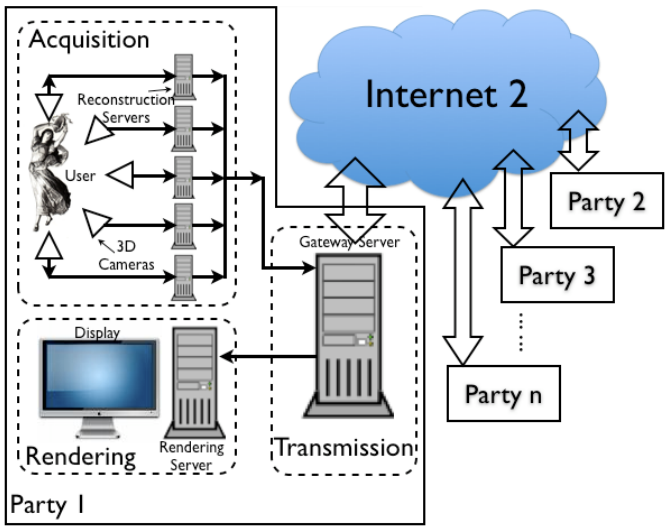

Fig. 1. TEEVE Architecture.

TEEVE project [1] aims to provide a seamless 3D video tele-immersive experience for a broader audience. Figure 1 illustrates the system architecture of TEEVE. The system comprises of three parts that are acquisition, transmission, and rendering. For the acquisition part, 3D cameras placed around objects are used to generate 3D video streams. Both point clouds and triangular meshes have been implemented to represent 3D video streams. For the transmission part, a gateway server aggregates and exchanges all 3D video streams with other gateway servers over high-speed Internet 2. For the rendering part, which is the focus of this paper, a rendering server receives streams from the gateway server and renders the scene according to the user requests. In order to support mobile devices as viewing and view-controlling devices, the remote rendering framework is implemented.

2.2. Remote Rendering Framework

Remote rendering is widely used in the scenario that the local device does not have sufficient rendering capability. In our previous proxy-based remote rendering framework [2], the proxy server can record the rendered 3D scene as a 2D image and the mobile devices directly display the received 2D image frames. However, this simple design can suffer from large interaction delay. When the mobile user wants to change the rendering viewpoint, the user is required to send the request to the proxy server first and wait until the proxy server sends back the 2D image rendered at the updated rendering viewpoint. Many factors such as the network round trip time, traffic congestion, and buffering can contribute hundreds of milliseconds or even seconds to the interaction delay. Such large interaction delay becomes a barrier for many applications that require frequent user interaction, such as games and real-time immersive communication.

In order to reduce the interaction delay, an image-based rendering mechanism called 3D warping has been used. The 3D warping was first proposed by McMillan in [8]. It allows mobile devices to warp the received 2D images to the new rendering viewpoint immediately and efficiently. Instead of waiting for the response from the proxy server, the 3D warping on mobile devices can reduce the interaction delay significantly. The only modification is that the proxy server needs to send the depth information of each pixel in addition to the 2D color image. We name the frame with both 2D color image and depth image as a *reference frame*.

A critical deficiency of 3D warping is that warping errors are generated when the occluded objects in the scene become visible in the new rendering viewpoint. There is no pixel in the input image to refer to when drawing the new image. It is also called the *exposure* problem.

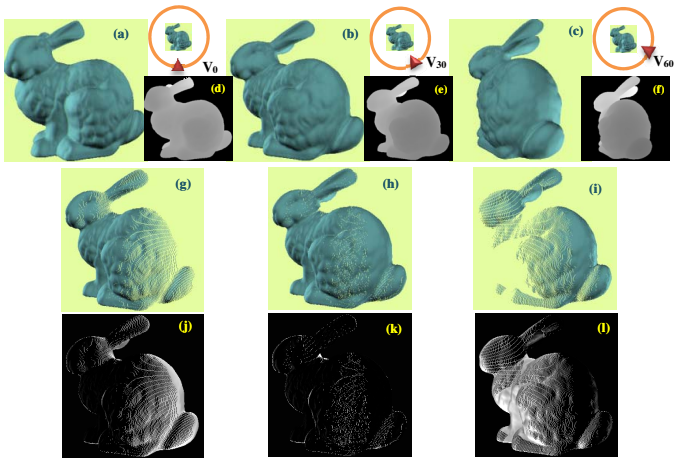

Fig. 2. Double Warping. (a),(b) and (c) Color maps rendered at viewpoints v_0, v_{30} and v_{60}; (d),(e) and (f) Depth map of the image rendered at viewpoint v_0, v_{30} and v_{60}; (g) Image warped from v_0 to v_{30}; (h) Image double warped from v_0 and v_{60} to v_{30}; (i) Image warped from v_{60} to v_{30}; (j),(k) and (l) The difference between (g),(h) and (i) from (b)

In our previous work [2], we found that the warping error can be reduced to negligible levels by warping twice from two carefully selected reference frames. Figure 2 illustrates the performance of the double warping approach. Therefore, our goal is to run a reference selection algorithm on the proxy server to select the best reference frames that generate the least warping error when used for warping on mobile devices. We summarize the reference frame selection problem formally in the following subsection.

2.3. Reference Frame Selection

We denote F as the original 3D video frame, C_i as the 2D color image rendered from F for the viewpoint V_i, D_i as the depth image rendered from F for the viewpoint V_i, R_i as the reference frame for the viewpoint V_i where $R_i = \{C_i, D_i\}$, and $W_k^{i,j}$ as the result color image of warping both R_i and R_j to the

viewpoint V_k. We define two basic operations: *rendering* and *warping*.

$$R_i = rendering(F, V_i)$$

$$W_k^{i,j} = warping(R_i, R_j, V_k)$$

When R_i and R_j are selected as reference frames and the actual rendering viewpoint V_k is given, the warping error $E_k^{i,j}$ is calculated as

$$E_k^{i,j} = W_k^{i,j} - C_k$$

The reference frame selection problem can be formally described as: given a range $[a, b]$ where the actual rendering viewpoint can be, we need to select two reference frames F_i and $F_j (a \leq i < j \leq b)$. The maximum warping error caused by these two reference frames is bounded by:

$$\max_k \left\{ E_k^{i,j} < T \mid k \in (i, j) \right\}.$$

We have proposed three search algorithms [2] that simplify the problem above by fixing one reference frame such that $F_i = F_a$. Using this simplification, our algorithms only need to find the position of F_j. The *Full Search Algorithm* lists all $j \in (a, b]$ as a possible reference frame position, and for every j, the algorithm examines every $E_k^{i,j}$ (where $k \in (i, j)$) to guarantee error margins are smaller than the threshold. The *Median Search Algorithm* lists all j position but only checks the warping error at the median viewpoint $E_{\frac{i+j}{2}}^{i,j}$. The *Fast Median Search Algorithm* is based on the *Median Search Algorithm*. However, it does not check for every possible j positions, but increases the value exponentially. Therefore, the *Fast Median Search Algorithm* successfully reduces the complexity of both rendering and warping from $O(n^2)$ to $O(\log n)$.

Since rendering 3D video is a real-time task, each video frame has to be processed within bounded time to achieve the required video frame rate. Given the limited time and computing resources to select reference frames, the proposed algorithms may not complete and find the most appropriate reference frames. To evaluate the performance of the algorithms, the *search range* is defined as the maximum distance between selected reference frames that the algorithms can compute, without compromising real-time requirements. Intuitively, the $O(N^2)$ *Full Search Algorithm* will generate shorter *search range* than the $O(log(n))$ *Fast Median Search Algorithm*.

2.4. CUDA and Parallelism

The Compute Unified Device Architecture (CUDA) [3] is a parallel programming model for GPUs developed by NVIDIA. The hardware model of CUDA consists of streaming multiprocessors (SMs), multiple memories with Non-Uniform Memory Architecture (NUMA). The extended Single Instruction Multiple Data (SIMD) model of SMs allows multiple threads to be executed independently when the results of *branch* instruction are different among the threads.

The programming model of the CUDA can control a hierarchical memory space on the non-uniform memories. For instance, programmers can specify the type of memories for *read* and *write* operations considering the characteristics of the data. It is crucial to partition the data and to fit them to the non-uniform memories to reduce the overhead of the data transfer between the main and the device memory.

Programmers can manage creation, termination, and scheduling of threads. Threads can be launched in the form of three-level dimensional hierarchies called *kernels*. The programmers manage thread blocks scheduled on SMs and threads in the blocks on multiple SIMD processor cores in a SM. Explicit synchronization operations are required when threads on different processors access a shared region of the device memory. For performance optimization, parallel operations need to be carefully allocated to the hierarchy of the SIMD processors of the GPU putting into consideration that synchronization operation may incur idle cycles. Hence, CUDA programming in the current state requires detailed knowledge about the GPU hardware.

3. DESIGN

Figure 3 illustrates our system design and interactions among the servers and the mobile devices. The proxy server can select reference frames by considering viewpoint changes from mobile devices. The reference frame selection is done by using three modules that are designed to exploit parallelism. The rendering module renders 3D model of video streams received from the gateway server. The depth extraction module generates the depth image from the rendered image. The 3D warping module computes a warped image to another viewpoint from an original viewpoint. The depth extraction module and the 3D warping module are designed using the CUDA architecture.

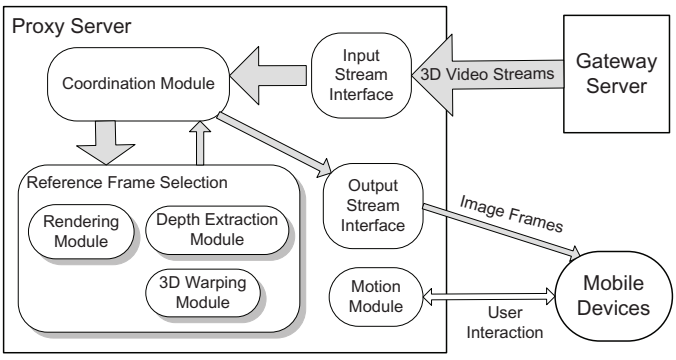

Fig. 3. Proxy Server Design.

Reference Frame Selection

The proxy server searches and selects the reference frames that have bounded possible warping errors to reduce the *exposure* problem. The reference frames are selected by the searching algorithms in Section 2.3. The warping error, $E_k^{i,j}$ is computed by summing up the differences between each pixel value of the warped image $W_k^{i,j}$ and the matching pixel of the original

image C_k. Two modules are used for searching the reference frame: depth extraction and 3D warping, and most of the required computation is in these modules. Fortunately, these modules can be implemented efficiently on GPUs by exploiting data-parallelism thanks to the independence of computations on pixels in the images.

3D Warping

For visualizing 3D models, the remote rendering framework renders the color image C_i from a viewpoint V_i of a frame F. After rendering, it extracts the depth information, D_i from each pixel of C_i. Since every pixel of both C_i and D_i is independent with respect to each other, the depth image D_i can be extracted in parallel from partitioned groups of pixels using separate threads.

```
PROCEDURE Parallel3DWarping (C_i, D_i, V_i, V_k)
S_block = NPI / (NTB × NT)
PARALLEL FOR l = 0 TO NTB - 1
    PARALLEL FOR m = 0 TO NT - 1
        P_start = S_block × (l × m + m)
        P_end = P_start + S_block - 1
        FOR n = P_start TO P_end
            n' = Cal_Warped_Pos ( C_i[n], D_i[n], V_i, V_k )
            V_k[n'] = C_i[n]
        END
    END
END
```

Table 1. Pseudo-code of 3D Warping

Table 1 shows the pseudo-code for the 3D warping module optimized for GPUs. The warped color image (W_k^i) to another viewpoint (V_k) is computed from the reference frame (F_i). For parallel computation, C_i is partitioned into blocks of pixels of the same size. Each thread calculates the warped position corresponding to n'-th pixel of W_k^i from n-th pixel of C_i. It repeats the calculation on the partitioned block of C_i from P_{start}-th pixel to P_{end}-th pixel.

To assign the threads to those parallel operations, we consider the number of pixels of the image (NPI), the number of SMs (NS), and the number of processor cores (NC) in a SM. Threads execute the same parallel operations in a thread block. The number of thread blocks (NTB) and the number of threads (NT) in a thread block are the parameters to this algorithm. We partition a rendered color image into uniform sizes of $NPI/(NTB \times NT)$ and allocate the partitioned data into each thread. We set NTB to NS and NT to twice the number of the processor cores (NC). These numbers for NTB and NT show the best performance in our experimental environment.

The calculation of the warped position is composed of 20 arithmetic operations. The absence of *branch* operations means that the parallel warping operations on the SMs at each independent pixel can happen in synchrony. Therefore, 3D warping operations can be implemented efficiently on GPUs by executing parallel operations using hundreds of processing elements without unnecessary blocking due to divergent control flow depending on the input data.

4. IMPLEMENTATION

Figure 4 illustrates our implementation of the parallel proxy server framework. The proxy server renders 3D models using OpenGL after copying 3D video streams from the main memory. Then, it selects the reference frames by executing its CUDA implemented modules. The selected frames are copied from the device memory to the main memory to transfer them to mobile devices. We implement the modules in the GPU by extracting additional parallelism from the previous CPU implementation of reference selection algorithms. In this section, we explain how we efficiently allocate the resources of the modern GPU based on the parallelism of the program and how we organize the data flow to minimize the overhead of memory accesses and data copies.

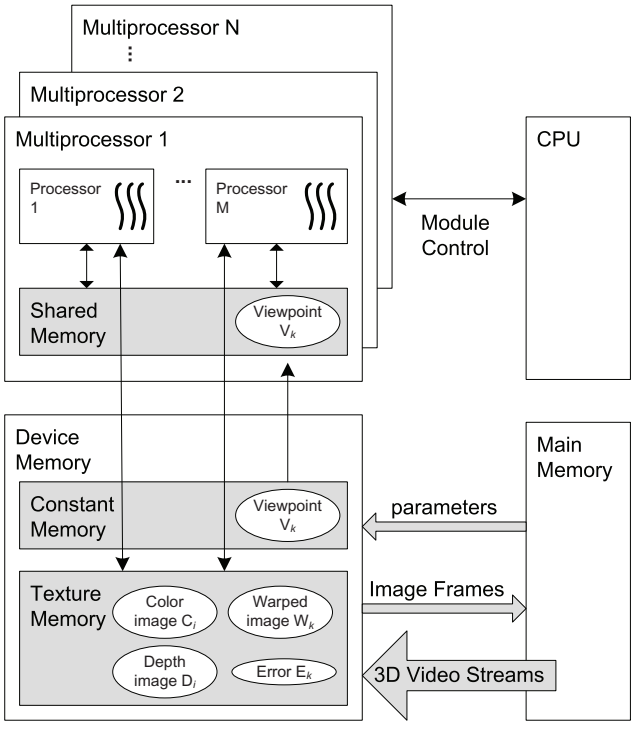

Fig. 4. Proxy Server Framework.

4.1. Resource Allocation on GPU Modules

GPU resource allocation is crucial to achieve most performance out of the GPU. While creating too small number of threads in SMs may under-utilize the GPU, launching too many number of threads in SMs may increase scheduling overhead thus also under-utilize the GPU. Reasonable dimensions of CUDA

kernels (*e.g.*, the number of threads, thread, blocks, and grids) needs to be allocated while considering the hardware specification of SMs in order to maximize the use of shared and cached data. If possible, input and output data need to be partitioned into independent blocks to reduce dependencies among parallel operations. This partitioning is crucial to reduce the overhead of the synchronization operations. In addition, we need to allocate most frequently accessed data into lower latency memory with smaller sizes such as the `constant memory` or the `shared memory` in the cores of SMs to reduce the overhead of memory accesses.

Rendering and Extracting Depth

Each 3D video frame from the gateway server is rendered by the proxy server. The proxy server renders an image from OpenGL 3D model after copying the video frames from the main memory to the device memory. The rendered image is copied from the graphics pipeline of the OpenGL to the `texture memory`. The depth extraction module computes the depth image, D_i from the rendered color image, C_i. Each thread in a core computes depth information from the partitioned blocks of the color image. The extracted depth image is stored in the `texture memory` so that any threads in SMs can access any portion of it. Parameters such as viewpoint V_k are computed on CPU and sent to the `constant memory`. The `constant memory` is a read-only region that can be cached to the local `constant cache` of the processors in a SM so that all processors can share the same data with much less latency. The size of the `constant memory` in our experimental environment is 65KB, which is sufficient to store the viewpoint parameters which are hundreds of bytes (B).

3D Warping

The 3D warping module reads input images (C_i, D_i) and warps the color image into the output image (W_k^i). Since the input and output images reside in the `texture memory`, any thread can access any portion of the images. The size of the local `constant memory` is sufficient to store multiple viewpoint parameters, thus threads in the same SM can load multiple viewpoints parameters with reduced latency.

Each thread in a SM computes warped pixel positions of the partitioned block of the image. It copies the original pixel that contains color information to the warped pixel position of the target warped image. If the input images can be stored in the `shared memory`, they can be exclusively accessed by separate SMs at lower reading latency. The size of both images are hundreds of kilo-bytes (KB) thus we would need to partition these input images to fit into the 16KB `shared memory`. However, since the computed warped position of the output image can come from any pixel of the input image, there is no consistent way to partition the input image. Therefore, we keep the input images in the `texture memory` instead of `shared memory`.

4.2. Data Transfer

Careful memory allocation of the GPU helps to reduce the cost of data transfer. In addition, we need to reduce the size and number of times of unnecessary data copies between the main memory and the device memory for performance.

We use rendered image on the GPU when extracting depth image. The framework extracts depth image directly on the GPU without copying the contents to the main memory. Unless the device memory of the GPU is fully occupied, we can eliminate two data copies between the main memory and the deice memory by caching the images. Unfortunately, a rendered image, C_i from the graphics pipeline of the OpenGL needs to be copied from the OpenGL pixel buffer to the `texture memory` of CUDA. This is because OpenGL and CUDA do not share the memory space on GPU in current CUDA environment.

We can reduce the number of data copies between two memories by caching computed depth images and copied rendered images on the `texture memory` in the GPU. Without copying the rendered color image and depth image to main memory, 3D warping operation is computed by directly accessing cached images on the `texture memory`. Thirdly, using faster memories reduces the data transfer cost. Viewpoint parameters in the `constant memory` can be accessed with little latency.

5. EXPERIMENTAL RESULT

Our parallel proxy server framework is implemented with CUDA SDK version 2.3. We run our proxy server on a machine that has an Intel Xeon Quad-Core processor running at 2.33GHz with 3GB main memory and an NVIDIA GeForce 9800 GX2 GPU with 256 cores and 1GB device memory.

Modules	Resolution	CPU (ms)	GPU (ms)
Depth Extraction	320 x 240	5.22	0.20
without Data Copy	640 x 480	5.65	0.70
Depth Extraction	320 x 240	5.33	0.40
with Data Copy	640 x 480	5.78	1.38
3D Warping	320 x 240	3.47	0.02
without Data Copy	640 x 480	13.89	0.05
3D Warping with	320 x 240	3.59	0.52
Data Copy	640 x 480	14.40	1.70

Table 2. Performance of Modules on CPU/GPU

Table 2 shows the execution time of the GPU modules compared with the CPU modules. The execution time "without data copy" represents only the computation time of the modules. For this measurement, we store the data (parameters, input, and output) in the main memory and the device memory. The execution time "with data copy" includes transfer time of the data in addition to the computation time. The computation time of the two modules on GPUs shows improvement of performance up to 13 times even with the overhead of data transfer.

Module	Display Resolution	Frame Rate (fps)	Search Range		
			Full	Median	Fast
CPU	320 x 240	10	5	6	14
	640 x 480	5	4	6	14
GPU	320 x 240	10	8	10	38
	640 x 480	5	9	12	39

Table 3. Performance of Proxy with CPU/GPU Modules

The average *search range* with the three reference frame selection algorithms is shown in Table 3. The table shows overall performance of the proxy server when using either the CPU or the GPU modules. The frame rate represents possible visualizing frame rate achieved from our experimental mobile device, Nokia N 800. This frame rate acts as real-time requirements when searching the reference frame. Using the GPU modules, the *search range* is increased up to 2.8 times compared with using the CPU modules. These experimental results show that our GPU modules can substantially improve the performance of the proxy server. The GPU modules can offload CPU burden on the framework since the GPU works separately while the CPU can handle other tasks such as communicating with the mobile devices or calculating warping errors. If we integrated and ran both the CPU modules and the GPU modules simultaneously, the performance would be improved further. We leave this as our future work.

6. RELATED WORK

It is common practice to use hardware acceleration when designing a remote rendering system. The WarpEngine [9] proposed a hardware architecture to accelerate 3D warping operations. Our design is partially inspired from their idea of using SIMD array to compute 3D warping and image interpolation. However, we use the CUDA platform rather than building from scratch. Moreover, our framework selects reference frames to reduce the exposure problem, which is not addressed by the WarpEngine. Lamberti and Sanna [10] presented a remote rendering framework based on clusters. The server can run rendering and video encoding concurrently in distributed nodes to improve performance. Exploiting the parallelism of the program and splitting the work into different servers are similar design philosophies as our work. However, the authors focused more on different timing factors of video streaming in remote rendering. The RealityServer [11] is a recent remote rendering project. The platform is built on NVIDIA GPUs to provide 3D rendering as web services. While the RealityServer aims to visualize 3D scenes with remarkable realism, our work targets real-time 3D applications.

7. CONCLUSION

We have designed and implemented a parallel proxy server framework for remote-rendering dynamic 3D video contents. CUDA-implemented modules on our proxy server compute the reference frame selection and the 3D warping operation in parallel. Our proxy server shows substantial performance improvement by using the optimized GPU modules even with data transfer overhead. Based on our experiences, proxy-based remote rendering framework shows promising performances for generic 3D rendering applications for mobile devices in addition to 3D video applications.

8. REFERENCES

[1] Zhenyu Yang and et al., "TEEVE: The next generation architecture for tele-immersive environment," in *ISM '05*, December 2005, pp. 112–119.

[2] Shu Shi, Won J. Jeon, Klara Nahrstedt, and Roy H. Campbell, "Real-time remote rendering of 3d video for mobile devices," in *MM '09*. 2009, pp. 391–400, ACM.

[3] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[4] Aljoscha Smolic and et al., "3D video and free viewpoint video - technologies, applications and mpeg standards," in *ICME*. 2006, pp. 2161–2164, IEEE.

[5] Wojciech Matusik and Hanspeter Pfister, "3D TV: a scalable system for real-time acquisition, transmission, and autostereoscopic display of dynamic scenes," in *SIGGRAPH '04*, 2004, pp. 814–824.

[6] André Redert and et al., "ATTEST: Advanced three-dimensional television system technologies," in *3DPVT '02*, June 2002, pp. 313–319.

[7] Stephan Würmlin and et al., "3D video recorder: a system for recording and playing free-viewpoint video," *Comput. Graph. Forum*, vol. 22, no. 2, pp. 181–194, 2003.

[8] Leonard McMillan and Gary Bishop, "Plenoptic modeling: an image-based rendering system," in *SIGGRAPH '95*, 1995, pp. 39–46.

[9] Voicu Popescu and et al., "The warpengine: an architecture for the post-polygonal age," in *SIGGRAPH '00*, 2000, pp. 433–442.

[10] Fabrizio Lamberti and Andrea Sanna, "A streaming-based solution for remote visualization of 3D graphics on mobile devices," *IEEE Trans. Vis. Comput. Graph.*, vol. 13, no. 2, pp. 247–260, 2007.

[11] "Nvidia realityserver," http://www.nvidia.com/object/ realityserver.html.