

# A CUDA-Supported Approach to Remote Rendering

Stefan Lietsch\* and Oliver Marquardt\*\*

University of Paderborn  
Paderborn Center for Parallel Computing

**Abstract.** In this paper we present the utilization of advanced programming techniques on current graphics hardware to improve the performance of remote rendering for interactive applications. We give an overview of existing systems in remote rendering and focus on some general bottlenecks of remote visualization. Afterwards we describe current developments in graphics hardware and software and outline how they can be used to increase the performance of remote graphics systems. Finally we present some results and benchmarks to confirm the validity of our work.

## 1 Introduction

With today's ever increasing computational power and the possibilities to simulate and visualize more and more complex systems, it is obvious that efficient technologies are needed to make the results of such simulations and visualizations available for a broad audience. Remote rendering and remote visualization are techniques to fulfill this need for visual data. Since the early 90s researchers work on the topic of transporting the rendered images or visual data to the clients of remote users to let them analyze and work with this data. There are many systems that focus on several different aspects of remote rendering / visualization. Some allow comfortable administration of remote servers and others focus on interactively showing remote users images that were rendered on powerful workstations. But all of those systems need to deal with vast amounts of data that needs to be transferred. This is because visual data is mostly pixel or voxel based. Therefore even single images which are displayed for only a fraction of a second can consist of one or more Megabytes of data. To provide a smooth animation, 30 or more frames are needed every second. All have to get from the GPU of the server, over the network, to the GPU and finally to the display of the user. There are already many approaches to minimize the amount of data that needs to be transferred for example through prefetching certain data to the client or by introducing level of detail mechanisms and compression of all kind. But there is still no solution on how to enable users to interact with a remote system as if it was local. However recent developments in graphics hard- and

---

\* e-mail: [slietsch@upb.de](mailto:slietsch@upb.de)

\*\* e-mail: [marquardt@upb.de](mailto:marquardt@upb.de)

software technology open up new possibilities to access, compress and transfer visual data on and to the graphics cards. The introduction of PCI Express for Graphics (PEG) for example cleared the way to efficiently download rendered images from the graphics card and send it over the network. This was a serious improvement over the asynchronous AGP-Bus used until then, which only granted very slow access to the relevant memory areas on the card. With the so called *Frame Buffer Extension* and the *Render Buffer Objects* of the OpenGL 2.0 specification [1] efficient methods to enable offscreen rendering and the read-back of rendered images were introduced. Besides, the only recently published CUDA framework by NVIDIA [2] opens up new ways on also using graphics hardware for compression or filtering techniques to reduce the size of the data that needs to be transferred already on the graphics card. An example of how all these techniques could be used to improve the abilities of remote rendering is given in this paper.

## 2 Remote Rendering/Visualization - An Overview

There are different classes of remote visualization for different tasks. In the following we will introduce three classes that group the most common existing systems. This should help to understand what the problems of the different classes are, and to which class the approach proposed in this paper belongs.

### 2.1 Client-Side Rendering

Systems where the application runs on a server but the graphics data (polygon meshes, textures, volumes) is sent to the client and rendered there belong to this class. The best known system in this class is a remote X-Server which allows to start X-based applications on a remote server as if they were local. The main disadvantage, however, is that the rendering power of the server system is not used at all while all rendering work is done by the client. This is fine for small desktop applications but insufficient for complex visualization applications that require certain rendering power. Another popular representative of this class is the Chromium [3] framework which is able to transfer the whole OpenGL stream from a server to a client. This is only one feature of Chromium but has the same problem as the remote X-Server, namely it does not use the servers rendering power. Chromium is a very flexible framework and is not only designed to do remote rendering. It also offers a lot of features to support all kind of distributed and parallel rendering tasks. But for this work we only consider the remote rendering capabilities and classify them as Client-Side Rendering.

### 2.2 Server-Side Rendering for 2D and Administration

This class contains all systems that are mainly used for server administration and remote control. They render the images on the server side, compress them and, send them to the client where they can be viewed with simple viewers. They

also work with low bandwidth connections and mostly show the whole desktop of the remote server. They perform well with simple 2D and little interactive applications (e.g. administration and settings dialogs) but they are insufficient for highly interactive 3D applications such as a driving simulator or an interactive 3D viewer. The most popular representatives of this class are the Virtual Network Computing Framework (VNC) [4] and Microsoft's Terminal Service architecture [5]. Both offer possibilities to adapt the remote visualization to the available bandwidth and client device. However they are not optimized to achieve high frame rates and low latency for interaction.

### 2.3 Server-Side Rendering for 3D application (with and without transparent integration)

The third class of systems is specialized on displaying interactive 3D applications on remote clients. The server (or servers) with a lot of graphics processing power renders a 3D application (e.g. OpenGL-based). Every frame is read back, compressed (lossy or lossless) and send to a client. Mostly, only the window of the 3D application is processed, to save bandwidth and processing power. Popular systems of this class are the SGI Viz-Server [6] and VirtualGL [7]. All of these systems are optimized to provide high frame rates with the available bandwidth. This is supported by the possibility to choose different resolutions and compression methods as well as certain level of detail mechanisms. Most systems allow a transparent usage of existing applications (mostly OpenGL-based). This guarantees a comfortable and universal utilization. On the opposite, there are systems that are tightly coupled to a certain application to further increase performance by adjusting the rendering process dynamically to fit the needs of remote visualization. The Invire prototype presented in this paper belongs to this class and will support both transparent and non-transparent integration. Furthermore it utilizes advanced hard- and software mechanisms to achieve high performance remote visualization.

## 3 Advances in Computer Graphics Hard- and Software

As mentioned before graphics hardware significantly changed in the last decade. The GPUs evolved from highly specialized and slowly clocked graphic processors to highly parallel, multi-purpose, fast co-processors which in some tasks outperform the CPU by far. This fact inspired a lot of programmers to try to port their application on the graphics hardware to gain significant performance increases. However the main problem was that the general purpose applications needed to be transferred into a graphics domain since the graphics cards could only be programmed through graphics APIs such as OpenGL or DirectX. There where efforts to develop more general APIs, mainly driven by the GPGPU<sup>1</sup> consortium.

---

<sup>1</sup> [www.gpgpu.org](http://www.gpgpu.org)

However one of the most important step towards general usage of graphics hardware was the introduction of the Compute Unified Device Architecture (CUDA) by NVIDIA in 2006 (see [2]).

### 3.1 CUDA

CUDA is a combination of software and hardware architecture (available for NVIDIA G80 GPUs and above) which enables data-parallel general purpose computing on the graphics hardware. It therefore offers a C-like programming API with some language extensions. The architecture offers support for massively multi threaded applications and provides support for inter-thread communication and memory access. The API distinguishes between *host* and *device* domains and offers access to fast caches on the device side. The implemented method of thread partitioning allows the execution of multiple CUDA applications (*kernels*) on one GPU. Each *kernel* on the host device has access to a *grid* of thread *blocks*. A block consists of a batch of threads that can be synchronized and is organized by one-, two- or three-dimensional IDs. This allows to uniquely identify each thread and assign tasks to each thread. All threads inside one block have access to a fast shared memory space to exchange data.

Another feature of the CUDA architecture is the interoperability with graphic APIs (OpenGL and Direct3D) which allows to use, for example, rendered images as input to CUDA kernels. Since this data already resides on the graphics device it only needs to be copied on the device to be processed by CUDA. This offers great possibilities for e.g. online image compression which is one topic of this paper (see section 4.3).

### 3.2 Render Buffer Objects

Another important technique in this field is the introduction of Frame Buffer and Render Buffer Objects in the OpenGL 2.0 specification [1]. They replace the slow pBuffer mechanisms for off-screen rendering. This technique offers, among others, a fast way to render to specified memory regions other than the framebuffer. This can be used e.g. to implement rendering servers for a remote rendering framework as proposed in this paper.

## 4 Invire - A Concept for an Interactive Remote Visualization System

In this section we introduce the concept for a new system called Invire (INteractive REmote VISualization). It belongs to the third class of remote rendering systems and focuses on high interactivity, maximized performance and best visualization results. We therefore utilized the new advances in graphics technology described in chapter 3 to improve compression and readback performance. The goal was to achieve adequate frame rates (above 20 FPS) and unrestricted interactivity for the remote visualization of high-resolution (1000x1000 pixels and

above) 3D applications. We choose to use lossless compression techniques first to maximize graphics quality and because of their high potential for parallelization. We also work on integrating lossy compression methods which allow to guarantee a fixed bandwidth utilization. The main limitation, however, still is the network connection between server and client. Uncompressed image data is too large to be transferred over current internet or even LAN connections. An example shows that to achieve 25 FPS for a remote visualization with a resolution of 1000x1000 pixels we would need a network connection that has a bandwidth of 100 MB/s:  $bandwidth = size\ of\ one\ frame * frames\ per\ second = (1000 * 1000 * 4 \frac{byte}{frame}) * 25 \frac{frame}{second} = 100 \frac{MByte}{second}$ . Even Gigabit Ethernet is limited to about 80 MB/s in practical usage. That leads to the conclusion that a compression rate of more than 0.1 is needed to realize our goal on a system with a Fast Ethernet connection (100 MBit/s).

#### 4.1 The Architecture

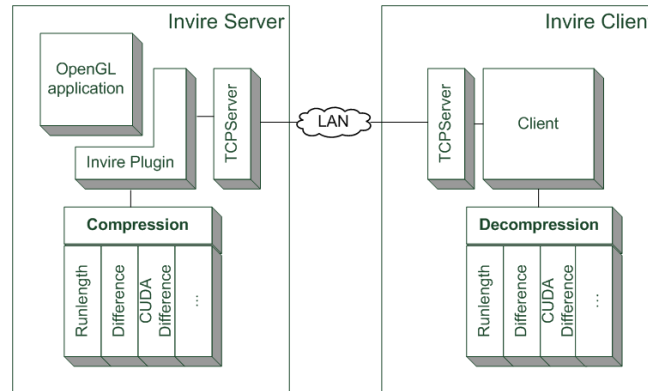
The overall architecture of Invire (as shown in Fig. 1) is kept simple to maximize the pure remote visualization performance. The server side offers an *Invire Plugin* which can easily be integrated in existing OpenGL applications. This allows the passing of parameters from the host application to Invire. We also work on a transparent integration possibly on the basis of VirtualGL. The *Invire Plugin* reads the current OpenGL context into a Renderbuffer Object (see section 3) which is passed to the *Compression* facility. This part of the software is implemented as modular collection of compression algorithms that can be exchanged arbitrarily. This allows us to compare the different methods and eventually combine them to achieve higher compression rates. After the compression of a rendered frame, it is passed to the *TCPServer* where a header is generated. The header contains information about the image size, compression. Afterwards header and compressed data is sent to the *Invire Client*. It receives the compressed frame and passes it to the *Decompression* facility together with the information from the header. After decompression the frame is displayed by the client.

#### 4.2 Image Transfer

To transfer the images a TCP socket connection is established between server and client. This socket remains active until either client or server cancels the transfer. The advantage of TCP sockets is that the correct order and the integrity of the image data is guaranteed. After a socket connection is established the protocol overhead is minimal and allows a good utilization of the available bandwidth.

#### 4.3 Image Readback and Compression

The most important parts of Invire are the readback and compression methods. We implemented a simple run-length (RLE) and difference encoding using



**Fig. 1.** Cuda memory and thread concept

standard programming techniques to have a basis for comparison (for more information on data and image compression see [8]). We shortly present the two basic compression methods and afterwards describe our CUDA-based difference compression algorithm in detail. The use of relatively simple and lossless compression methods has several reasons. First of all, they are very fast since both RLE and difference compression only need to go over the input once. Additionally the difference compression has a very good potential for parallelization as shown in the following. However, the main problem of lossless compression is that the achievable compression rate strongly depends on the input frame. If it is highly arbitrary in terms of consecutive colors or frame-to-frame difference then compression ratios can even be above 1.0. Nevertheless, typical visualization data (CAD, scientific visualization, VR) is often very regular in one or both terms. Therefore one can expect good compression rates for this application. There are also more sophisticated compression methods (e.g. statistical such as Huffman Coding or dictionary based such as LZ) but they cannot meet the requirements of interactive applications because of their comparably long run times or high memory consumption.

**Run-length encoding** After reading back the image data of the current context to a given location in (host) memory, the run-length algorithm goes over the array, counts consecutive pixels with same color values and writes the sum followed by the actual color information into a new array. The decompression can be done by writing the amount of pixels with the same color in a new array consecutively. Afterwards it can be displayed by the OpenGL function `glDrawPixels()`. The RLE algorithm can encode and decode  $n$  pixels in  $O(n)$  time.

**Difference compression with index** Another basic technique for lossless image compression is the difference compression. The current and the last frames

are compared pixel wise and only the pixels that are different are saved. Additionally the position of the pixels that changed is needed to decompress the current frame. This can be most efficiently done by an index which maps one bit to every single pixel of a frame. If the bit is 1 the pixel has been changed and the saved pixel value at the position *#of preceding 1s in index* is needed to update the pixel of the last frame. This requires  $O(n)$  time to compress and decompress  $n$  pixels.

**Difference compression with index using CUDA** The difference compression with index method described before is very suitable for parallel execution (on  $k$  threads with  $n > k$  pixels). Especially creating the index, as well as the copying of the pixel data can be done in  $O(n/k)$  time. Therefore we decided to implement this method in parallel and choose to use the CUDA architecture. It is able to process the data directly on the graphics hardware and allows highly parallel execution. Since CUDA offers several SIMD multi processors we decided to split the frames into  $m = \frac{n}{k}$  blocks which can be processed independently. This helps to optimize the workload on the graphics hardware. Each block consists of  $k$  threads which are working in parallel on one multiprocessor and which have access to a fast, shared memory. Each thread processes one pixel which is assigned to it by its thread and block index (thid and bid). Fig. 2 shows the three main steps of the algorithm. In the first step the index is generated by simply comparing corresponding pixels of the last and the current frame. When all threads have finished, a parallel compaction method (the second step) based on the stream compaction algorithm by [9] is invoked on the shared memory. This algorithm requires  $O(\log k)$  time in parallel to compute the amount of empty spaces to its left for every item of the s\_result array. It needs to run for all  $m$  blocks. With this information the pixels can now be stored in an array without empty spaces which is copied to a position in the global result array which is determined by the block index (bid). Additionally the number of stored pixels is written to a global array (g\_changedPixels). After all blocks finished the second step, the g\_changedPixels array is used to compute the absolute position in memory for each blocks partial result (step 3). This is also done in parallel based on the scan algorithm presented by [10] which sums up all items to the left of the current value in  $O(\frac{m}{k} * \log \frac{m}{k})$  time in parallel. Finally the partial results of the blocks are copied to the calculated memory locations and then the final compressed frame and the index are readback from the graphics hardware to be send to the client. The decompression at the client side is done sequentially as described above or can be parallelized equal to the compression using CUDA if available. Finally the whole algorithm can be run in

$$O(2 * (m) + m * \log k + \frac{m}{k} * \log \frac{m}{k})$$

time in parallel with  $n$  = number of pixels,  $k$  = number of threads,  $m$  = number of blocks and  $n = m * k$ .

```

divide frame in blocks of 256 pixels and
copy them to shared memory (s_current);
for all blocks do parallel
  for all threads do parallel
    if(compare(s_current[thid],s_last[thid]))
      s_index[thid]=0;
    else
      s_index[thid]=1;
      s_result=s_current[thid];
  g_index[bid*bw+thid]=s_index[thid];
  switch s_current and s_last;
  syncthread;

  for all threads do parallel
    compaction(s_result);
  for all threads do parallel
    g_result[bid*bw + thid] = s_result[thid];
    g_changedPixels[bid] = #changed pixels;
syncblocks;

  for all threads do parallel
    computeAbsoluteMemoryPosition(g_changedPixels);
  result: g_AbsoluteMemoryPosition[];
syncblocks;

  for all threads do parallel
    g_compResult[g_AbsoluteMemoryPosition[bid]+thid] = g_result[thid];
syncblocks;

send(g_index,g_compResult);

```

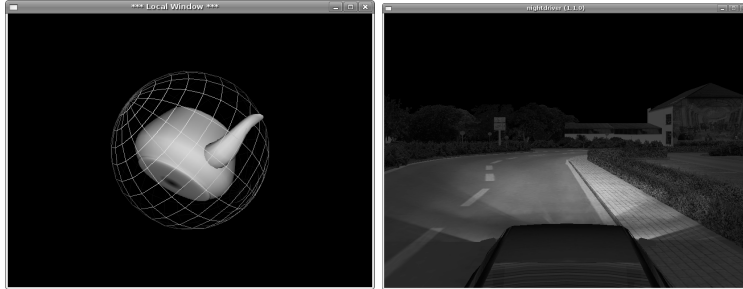
**Fig. 2.** Pseudocode of the parallel "difference with index" compression. Abbreviations: bid = Block index, bw = Block width, thid = Thread Index, s\_ = shared, g\_ = global

## 5 Benchmarks and Results

To proof the theoretical concept and to compare the described compression algorithms we prototypically implemented the Invire system and tested it in a sandbox environment. The server part runs on a computer equipped with a CUDA ready Geforce 8800 GTS graphics card by NVIDIA. It has 12 multiprocessors, a wrap size<sup>2</sup> of 32. That leads to  $k = 12 * 32 = 384$  threads that can run concurrently. As testing scenario we chose two OpenGL based applications (see Fig. 3) . The first is a simple rotating teapot on black ground and the second is a driving simulator called Virtual Night Drive (VND) [11]. The VND is

<sup>2</sup> Number of threads that are executed in parallel on one multiprocessor

specialized on simulating automotive headlights at night and uses the shaders of the graphics card to calculate the luminance intensity pixel wise. The first teapot application is highly regular and has a uniform black background, whereas the VND is relatively arbitrary through its textured and lighted scene and its unpredictable movement. The charts in Fig. 4 show some interesting results. For



**Fig. 3.** Test cases: simple teapot, Virtual Night Drive with headlight simulation

both test application we measured the average frame rate for various resolutions and compression methods. Fig. 4A) shows the teapot application with resolutions from  $16 \times 16$  pixels up to  $1024 \times 1024$  pixels. For resolutions below  $256 \times 256$  the overhead for compressing and decompressing the frames is obviously too big so that no compression method performs better than just sending uncompressed images. But from  $256 \times 256$  on the size of the transferred images is the limiting factor. The RLE and the difference without CUDA method perform nearly similar and are a bit faster than the version with CUDA up to the resolution of  $512 \times 512$ . There the difference coding with CUDA takes the lead and outperforms the other compression methods by a factor of about two (RLE 14 FPS, difference 10 FPS and difference with CUDA 21 FPS). This can be explained by the computational overhead the CUDA-based algorithm introduces (mainly the compaction methods). But for high resolutions this overhead amortizes and we can nearly double the frame rate. The VND application shows similar results. The interesting thing here is, that all algorithms produce slower frame rates because of the poor compressibility of the input data. But still the CUDA-based approach provides the best results for the highest resolution. In comparison with another remote graphics system of the same class -VirtualGL- Invire performed slightly worse (about 20% fewer FPS in the VND application) but that can be explained of the use of lossy compression versus lossless compression techniques.

## 6 Conclusion and Outlook

In this paper we presented a remote rendering system that takes advantage of recent advances in computer graphics hard and software. We could show that

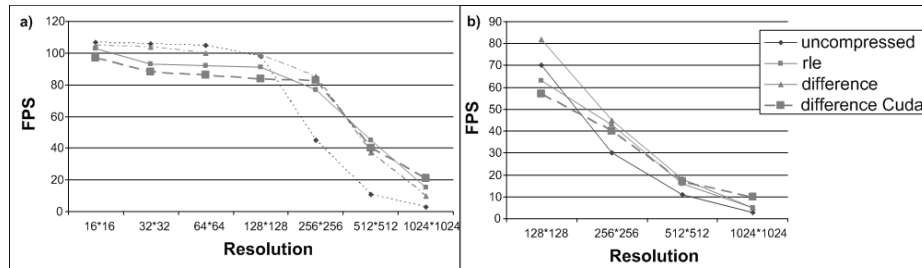


Fig. 4. Benchmarking results with A) Teapot and B) Virtual Night Drive

the speedup of a parallel compression method surpasses the resulting computational overhead for high resolutions (1024\*1024 pixels and above). However, the implementation that was used for the performance benchmarks still is in a prototypical stage. There are some programming optimizations to make and especially the integration of high quality lossy compression techniques promises good results.

## References

1. Segal, M., Akeley, K.: The OpenGL Graphics System: A Specification Version 2.0. Silicon Graphics (2004)
2. NVIDIA: NVIDIA CUDA - Compute Unified Device Architecture. Website: <http://developer.nvidia.com/object/cuda.html> (2007)
3. Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P., Klosowski, J.: Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics* **21** (2002) 693–702
4. Richardson, T., Stafford-Fraser, Q., Wood, K., Hopper, A.: Virtual network computing. *IEEE Internet Computing* **2** (1998) 33–38
5. Microsoft: Microsoft Windows Server 2003 Terminal Services. Website: <http://www.microsoft.com/windowsserver2003/technologies/terminalservices/default.mspx> (2007)
6. SGI: SGI OpenGL Vizserver - Visual Area Networking. Website: <http://www.sgi.com/products/software/vizserver/> (2007)
7. VirtualGL: The VirtualGL Project. Website: <http://www.virtualgl.org/> (2007)
8. Salomon, D.: Data Compression: The Complete Reference, 3rd Edition. Springer (2004)
9. Horn, D.: Stream Reduction Operations for GPGPU Applications. In: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional (2005) 573–583
10. Harris, M.: Parallel Prefix Sum (Scan) with CUDA. Website: <http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/scan/doc/scan.pdf> (2007)
11. Lietsch, S., Berssenbruegge, J., Zabel, H., Wittenberg, V., Eikermann, M.: Light simulation in a distributed driving simulator. In: *Proceedings of the 2nd International Symposium on Visual Computing, 2006*. Volume 4291 of *Lecture Notes in Computer Science.*, Berlin Heidelberg New York, Springer-Verlag (2006) 343–353