Chapter 9

Appendix B - Unpublished papers

9.1 gTiledVNC - a 22 Mega-pixel Display Wall Desktop Using GPUs and VNC

This paper is unpublished.

gTiledVNC - a 22 Mega-pixel Display Wall Desktop Using GPUs and VNC

Yong LIU [1], John Markus BJØRNDALEN and Otto J. ANSHUS
Department of Computer Science, University of Tromsø

Abstract. A display wall, a high resolution, wall-sized display, uses Virtual Networked Computing (VNC) to implement a high resolution desktop because virtual frame buffer resides in the main memory. However, the performance of common VNC servers decreases heavily with increasing desktop resolutions since they use CPU to handle compute-intensive tasks such as compression of pixels for network transmission. Running parts of the VNC server on a graphics processing unit (GPU) can potentially increase the performance by taking advantage of the highly parallel architecture of the GPU. gTiledVNC using GPU has been implemented, where zero-copy buffer is used as VFB on integrated GTX 295. Pixel encoding and the operations related to virtual frame buffer are implemented on the GPU using CUDA. The initial implementation shows a slow-down compared to the CPU-only implementation. We measured and compared the performance under different cases of using the GPU in order to locate the bottleneck of the performance. The results show that pixel comparison has a big performance penalty when each thread does pixel encoding for one 16x16 pixel tile on the GPU.

Keywords. display wall desktop, high resolution, CUDA, GPU, CPU-intensive

Introduction

A display wall is a wall-size display in University of Tromsø, whose display area used is up to $18m^2$. The area is much more than that of regular desktops. The display comprises many tiles, each of which is driven by one projector. All tiles are put together into the seamless display. The display wall is specially well-suited for communication in front of it for a large area. As a result, it can easily support the collaboration of multiple users. A high resolution desktop is required to be implemented in order to utilize the large display area efficiently.

The display wall is a high resolution and distributed display. For this, the existing desktop systems cannot support the display wall directly. There are many available solutions to utilizing a large display wall at different levels, including an application level, a graphical level [6] and a pixel level [1]. At an application level, each tile runs a copy of the application. It's necessary to keep synchronization between each tile. But applications must be rewritten for the display wall. It's more flexible to implement it at a graphi-

[1]Corresponding Author: Department of Computer Science, University of Tromsø, 9037, Tromsø, Norway; E-mail: yongliu@cs.uit.no.

cal library level. However, the dependence on a specific graphical library limits the range of applications.

We believe that it's the best way to implement a high-resolution desktop at a pixel level, which can provide a universal desktop interface and support running applications without modifications. The advantage is to reuse the existing applications at the display wall. The disadvantage is that one single desktop server has to handle all operations and transfer data over network. As a result, it's necessary to compress data because of a large number of pixels at the limited network bandwidth. An X server based on virtual network computing (VNC) can satisfy those requirements. A high resolution X VNC server is used as our display wall desktop. the X VNC server uses virtual frame buffer (VFB) in the memory, which can support any resolution in comparison with the limited resolutions on GPUs. However, VNC is typically successful at a low resolution desktop over network. VFB is difficult to benefit from graphical cards in the cluster. Using CPU instead of GPU is efficient at low resolution [4]. When the number of pixels increases, VNC will lead to the bottleneck of CPU and network at a display wall. High resolution at display wall has an impact on the performance when CPU handles pixel operations and compression. The performance has to be improved when VNC is used to create the display wall desktop.

A display wall desktop is a compute-intensive system. We have done performance benchmark for display wall desktop in [8], where we located the bottleneck of performance and used Intel SSE2 to speedup memory copies. But the speedup factor depends on Intel SSE2 and is limited. A multi-thread architecture introduces in [10] in order to improve the performance further, which can benefit from Multi-core CPUs. However, the cost of multi-core CPUs is expensive. That paper also demonstrates the problem of display consistency.

CUDA (Compute Unified Device Architecture) [11], a general-purpose parallel computing platform, allows us to utilize manycore GPUs at a low cost, compared with the cost of multicore CPUs. Compute-intensive high resolution display wall desktop can benefit from that capability. gTiledVNC has been implemented as high resolution desktop with CUDA, where VFB resides in GPU memory in order to avoid multiple data copies from host to device. Zero-copy buffer[11] used as VFB makes CPU and GPU work together seamlessly. The performance of the initial version is much lower than that of using CPU when the code is ported directly into CUDA. The results show that pixel comparison has a big performance penalty when each thread does pixel encoding for one 16x16 pixel tile on the GPU.

This paper is organized as follows. Section 2 is related work. Section 3 presents the problem using CPU at a display wall desktop. Section 4 is the architecture using GPU for a display wall desktop. Section 5 is the experimental results. Section 6 is discussion. Section 7 is our conclusion.

1. Related Work

Distributed Multihead X, XDMX [3], makes it possible to create a high-resolution display wall desktop. However, the lack of data compression limits the resolution of desktop. MultiStream [9] is a cross-platform display sharing system using multiple video streams to create lossy videos on-the-fly of a desktop, and then stream the videos to a

display wall or PCs. The system is transparent to the applications by using standard media players and video stream formats. However, when used to create a video of the large desktop of a display wall, capturing and encoding the pixels are CPU intensive, and the performance suffers.

WireGL [2] uses a cluster of off-the-shelf PCs with a high speed network, which provides a parallel interface to the visualized graphical system in order to address the slow interface bottleneck between the host and graphical system. WireGL is the first implementation of OpenGL in a cluster, which extends OpenGL by adding barriers and semaphores to the OpenGL APIs. WireGL is a sort-first render, sort-last architectures. It is made of client library and pipeserver. Client library replaces the original OpenGL APIs. The client applications must be aware of the parallel architecture, which means that it's necessary to rewrite applications.

Chromium [6] makes a set of individual tiles of a display wall to appear as a single display to the applications. Display output from applications is transparently redirected to the display wall. The model is flexible in that it can be used both for single process and multiple process parallel applications. The applications don't have to be changed. A limiting factor is that the applications must depend on OpenGL.

SAGE [7] is a flexible and scalable software approach, where display output is sent from applications to a display wall. However, applications are required to be rewritten with the SAGE application interface library. Xinerama [12] is an extension to the X window system using multiple graphical devices to create one large virtual display. However, it requires the graphical devices to coexist on a single computer. These approaches are designed to let an application display to a display wall. They are not suitable as a way to do a high resolution display wall desktop.

We have profiled high resolution display wall desktop and use Intel SSE2 instructions to speed up the performance [8]. We also used multi-threading and server update pushing to improve the performance at display wall desktop further, where the desktop can benefit from multi-core CPUs [10].

2. Problems Using CPU for VNC

Display wall desktop uses VNC model to implement high resolution X server, where VFB resides in the host memory. X VNC server handles X server operations as well as VNC operations. It uses CPU instead of GPU to do with them. VNC packs and transfers pixels in addition to keyboard and mouse events over network. More description detail of display wall desktop is given in [8] and [10]. VFB in host makes it independent of graphic cards. However, VNC model is typically efficient at low resolution. With increasing the number of desktop resolution, the requirement of CPU cycles becomes more and more and the performance decreases heavily at display wall. There are two main bottleneck, pixel bitblt related to X operations and pixel encoding related to VNC, when X VNC server is used to build high resolution display wall desktop.

The critical operations of X server are pixel bitblt at display wall desktop, which usually involves in a source drawable and a destination drawable. The typical X operation is PutImage, which updates an image into VFB. The source and destination are combined bitwise according to the specified raster operation (ROP) and the result is then written to the destination. The ROP is essentially a boolean formula. The source pixels are written into the destination one pixel by one pixel when ROP is copy.

The time T_{bitblt} of pixel bitblt is as follows: $T_{bitblt} = N_{pixels} * T_{ROP\ on\ one\ pixel}$, where N_{pixels} is the number of the involved pixels and $T_{ROP\ on\ one\ pixel}$ is the time of handling one pixel. So the bitblt time highly depends on the number of pixels. From that formula, it's easy to understand the improvement in [8], where $T_{ROP\ on\ one\ pixel}$ is shortened by the Intel SSE2 instructions.

Pixel encoding is another bottleneck. Hextile is the most efficient encoding at display wall desktop. Hextile encoding uses pixel comparison to look for the rectangle regions of the same colors in tiles whose size is 16x16. Time Complexity of Hextile encoding is $T(256n)$, where n is the number of pixels.

Display wall desktop used in University of Tromsø is 22 megapixels. From above discussion, display wall desktop is a compute-intensive application, which is hard to overlap the computation on CPU. When Intel SSE is used, only $T_{ROP\ on\ one\ pixel}$ is shortened for memory copy. The speedup factor is limited. Multi-thread architecture is more efficient. The more CPUs are required in order to improve the performance further. However, the cost of more CPUs is expensive.

3. Display Wall Desktop Using CUDA

3.1. Reasons Using CUDA and Challenges

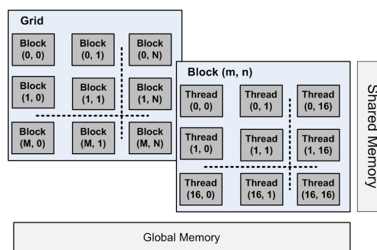

Figure 1. CUDA Computing Model

CUDA (Compute Unified Device Architecture) is a general-purpose parallel computing architecture, which uses NVIDIA GPUs. CUDA provides a software environment to use C as development language, which is easy to work together with display wall desktop. The idea behind CUDA is that GPU is specialized for compute-intensive, highly parallel computation. CUDA is well-suited to address problems that can be expressed as data-parallel computations. Display wall desktop is highly compute-intensive and data-parallel, which is expected to benefit from GPU.

GPU is viewed as a computing device in CUDA, which is able to execute a very high number of threads in parallel. Kernels, which are data-parallel compute-intensive portions of applications running on the host, are off-loaded onto the device. Those kernels are executed many times and independently on different data. They can be isolated into a function that is executed on the device as many different threads. Pixel bitblt is to handle each pixel independently so that it is typically data-parallel. Pixel encoding is also to compress independent pixels regions.

There are two separate memory spaces in host and GPU, host memory and device memory. CUDA applications have an execution pattern: Data is copied from host to device before CUDA computing, and data is copied back to host after CUDA computing. Large data copying at display wall desktop will have a heavy impact on the performance. The challenge is to decrease the times of memory copying between the host and GPU. Pixel comparison is another challenge for the performance.

Figure 1 demonstrates the data model with CUDA, where a thread block is a batch of threads that work on one pixels region at display wall desktop. Each thread has a private local memory. All threads have access to the same global memory. Global memory and local memory are not cached in CUDA. Accesses to them are expensive. CUDA provides shared memory as fast as registers to make threads cooperate and share data efficiently in one block. It also supports synchronizing their execution to coordinate memory accesses.

3.2. Pixel Bitblt Using CUDA

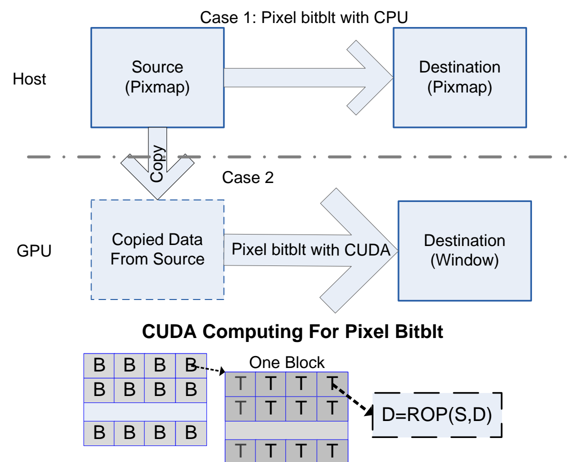

Figure 2. Pixel Bitblt with CUDA

In order to decrease the times of memory copies, VFB of X VNC server is put into the GPU device's memory at display wall desktop. CUDA 2.2 supports zero-copy buffer on integrated GPUs, where the CPU and GPU memory are physically the same and the buffers in host can be mapped into the CUDA address space. VFB uses zero-copy buffer in order to avoid the overhead of memory copy between host and GPU. Zero-copy buffer makes CPU and GPU work seamlessly.

X server uses a drawable as a descriptor of a surface that graphics are drawn into. X server supports two types of drawables, either a window on the screen or a pixmap in memory. Pixmap is an off-screen graphic area in host memory. Window is an area on the screen that can be drawn into by graphic routines. VFB is used as the screen in X VNC server. As a result, window is referred directly to VFB in display wall desktop. All

graphic operations work on drawables, and operations are available to copy patches from one drawable to another. All operations related to window will be handled with GPU.

Graphic routines are divided into four cases. When both the source and destination drawables are pixmap, CPU will do with pixel bitblt. The other cases will be done with CUDA. When one of drawable is pixmap, the data of pixmap is copied to GPU before CUDA computing. There is no memory copy when both drawables are windows.

CUDA supports that the maximum number of threads per block is 512. Because ROP on pixels is independent each other in pixel bitblt, it's easy to do with one operation on one pixel in one thread. When both the source and destination of bitblt are windows, it will speed up 512 times theoretically, because of 512 pixels each time in Figure 2. The typical case is putimage, where the source is a pixmap and the destination is a window. At that case, there is one memory copying between the host memory and the device memory in that case.

3.3. Hextile Encoding Using CUDA

Hextile is relatively fast compared with other pixel encoding methods at display wall. There are the following steps in hextile encoding. (1) In order to improve the compression ratio, the background and foreground pixels are selected firstly in each 16x16 tile. (2) The sub-rectangles are looked for in one tile, which are the same colors and not the background pixel. (3) When a sub-rectangle is found, pixels are encoded and the sub-rectangle is filled with the background pixel. If the length of final encoded data is more than the length of raw data, raw encoding will be used. Although hextile is an efficient pixel compression, the time complexity is very high when CPU is used to implement it at high resolution case.

When hextile encoding is implemented with CUDA, the method used is that one gird implements hextile encoding and each thread is to finish encoding pixels in one 16x16 tile. This method is very direct when the codes are ported from CPU to CUDA, because the pixels used by threads are independent on each other in VFB. Figure 3 demonstrates hextile encoding with CUDA. After each thread has done hextile encoding, synchronization has to be used. The reason is that the length of compressed data in one tile is not fixed. The position of each tile in data buffer cannot be calculated until all information of the lengths is collected. After each thread gets its position in data buffer, the encoded data can be written into data buffer.

The challenge is that the whole process is involved in a lot of pixel comparison and memory copies. Those operations have an impact on the performance of CUDA. The background pixel can efficiently increase compression ratio. However, the only first two different colors are used as the background and foreground pixels, not the two most numbers of same pixels, because of time complexity.

4. Experimental Results

gTiledVNC has been implemented based on TightVNC, where the codes include a CUDA interface library and some modifications for window bitblt and hextile encoding. The display wall computers run 32-bit Rocks 4.1 Linux on a cluster of Dell Precision WS 370s, each with 2GB RAM and a 3.2 GHz Intel P4 Prescott CPU. The server node

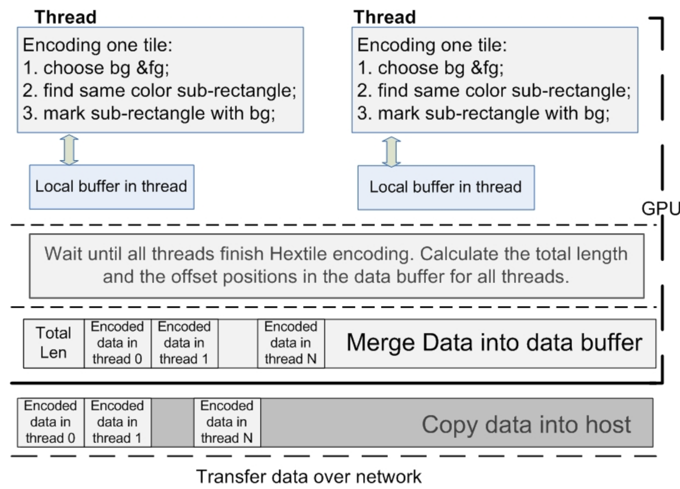

Figure 3. Hextile Pixel Encoding Implementation Using GPU

has 2.5GB RAM, 3.2 GHz CPU and one integrated GeForce GTX 295. That node runs 64-bit Ubuntu 9.04 and installs CUDA 2.2. We use one video, which is 3-megapixel each frame, 30 FPS, 3000-frame and MPEG4 encoding to emulate updates to a high-resolution desktop. The video plays with ffplay [5] on the server. It plays back on display wall with vncviewer.

The result is showed in the figure 4. FPS in Y-axis is the number of frames per second seen in front of display wall. X-axis represents different cases using GPU. Case A is the result using CPU-only with VFB in host. VFB uses zero-copy buffer in the other cases. Case B is when VFB uses zero-copy buffer and CPU does all computation. Case C only uses GPU to do bitblt. Case D only uses GPU to do hextile encoding. Case E uses GPU to do bitblt and hextile, where each block has 15 threads and one tile is copied into shared memory firstly. VFB using zero-copy buffer has no impact on performance because FPS of A and B is the same. Hextile encoding using GPU is mainly bottleneck because FPS of D is almost the same as FPS of E.

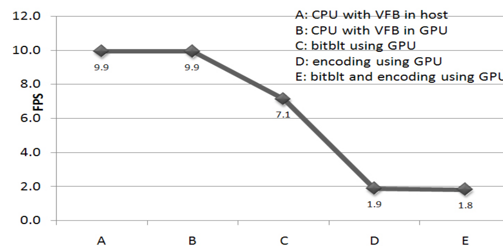

Figure 4. Performance of Display Wall Desktop Using GPU

The table 1 is the result when local memory is used as pixel buffer for one tile each thread. FPS still keeps stable with increasing the number of threads per block. That shows hextile encoding using GPU cannot benefit from multi-core GPUs now. FPS of using local memory is slower than FPS of using shared memory, because of shared memory is fast than local memory on GPU.

Threads per block	15	64	128	192	256
FPS	1.6	1.6	1.6	1.6	1.6

Table 1. Performance of Display Wall Desktop.

5. Discussion

CUDA has two distinct memory spaces. It interchanges data by coping data between CPU and GPU. However, high resolution desktop means a great number of pixel copies in discrete GPUs, which leads to the big overhead. CUDA supports zero-copy mode in integrated GPUs. When zero-copy buffer is used as VFB, the overhead is little compared with the buffer in host according to the experimental result. The advantage is that zero-copy buffer implements data interchange without buffer copies. The disadvantage is that it depends on integrated GPUs and consumes the memory in host.

Hextile encoding using CUDA has a large negative impact on the performance. The reason is pixel comparison and memory operation. An amount of pixel comparison leads to heavy performance penalty on GPU. When each thread does hextile encoding for one tile, encoding is in need of memory buffer. Each thread consumes 1KB buffer to copy pixels in one tile. The performance of using shared memory is better than that of using local memory, because shared memory is as fast as registers. However, it supports 16KB for one block. As a result, the number of threads per block is only up to 15 when shared memory is used for hextile encoding.

The performance of using GPUs is much lower than 15 FPS using multi-CPUs [10]. There are 480 processors in GeForce GTX 295, but FPS does not vary with increasing the number of threads per block. It cannot utilize the capacity of GPU efficiently when the code is ported directly from CPU to GPU, where one thread does hextile encoding for one tile. However, multi-core GPU is still expected to speedup the performance of high resolution. One block works together to implement hextile encoding for one tile in the future, where many threads can speedup hextile encoding. One block for one tile can also utilize shared memory to improve the performance.

6. Conclusion and Future Work

High resolution display wall desktop is compute-intensive. CUDA is very powerful to implement compute-intensive applications parallelly. gTiledVNC is implemented with CUDA in order to improve the performance of display wall desktop on GeForce GTX 295. Zero-copy buffer is used to avoid memory copy between CPU and GPU. It also simplifies the design of gTiledVNC. We have measured and compared the performance under cases using GPU. The initial implementation shows a slow-down compared to the

CPU-only implementation. The results show that hextile encoding is a main bottleneck when the code is ported directly, because hextile encoding has too much pixel comparison and consumes memory buffer. gTiledVNC cannot benefit from GPU in the initial version.

The improvement will continue in the future. The possible future work is listed here. The further investigation of the reason for a low performance using GPU should be made. Instead of porting to GPU from CPU directly, a better algorithm is required to balance work of GPU and CPU for good performance.

References

[1] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. Thinc: a virtual display architecture for thin-client computing. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 277–290, New York, NY, USA, 2005. ACM.

[2] Ian Buck, Greg Humphreys, and Pat Hanrahan. Tracking graphics state for networked rendering. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 87–95, New York, NY, USA, 2000. ACM.

[3] R. E. Faith and K. E. Martin. Xdmx: distributed multi-head x. *http://dmx.sourceforge.net/*.

[4] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[5] FFmpeg. *http://ffmpeg.mplayerhq.hu/*.

[6] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph.*, 21(3):693–702, 2002. 566639.

[7] Byungil Jeong, Luc Renambot, Ratko Jagodic, Rajvikram Singh, Julieta Aguilera, Andrew Johnson, and Jason Leigh. High-performance dynamic graphics streaming for scalable adaptive graphics environment. 2006. 1188568 108.

[8] Yong Liu and Otto J. Anshus. "improving the performance of vnc for high-resolution display walls". In *CTS 2009: The 2009 International Symposium on Collaborative Technologies and Systems*, 2009.

[9] Yong Liu, Otto J. Anshus, Phuong H. Ha, Tore Larsen, and John Markus Bjørndalen. Multistream a cross-platform display sharing system using multiple video streams. *Distributed Computing Systems Workshops, ICDCS '08. 28th International Conference*, pages 90–95, 2008.

[10] Yong Liu, John Markus Bjørndalen, and Otto J. Anshus. "using multi-threading and server update pushing to improve the performance of vnc for a wall-sized tiled display wall". In *InfoScale 2009: The 4th International ICST Conference on Scalable Information Systems*, 2009.

[11] NVIDIA. Nvidia cuda programming guide v2.2. *http://www.nvidia.com/object/cuda_develop.html*.

[12] Xinerama. *http://sourceforge.net/projects/xinerama/*.