



REMOTE VISUALIZATION ON SERVER-CLASS TESLA GPUS

WP-07313-001_v01 | June 2014

White Paper



TABLE OF CONTENTS

Introduction	4
Challenges in Remote and In-Situ Visualization	5
GPU-Accelerated Remote Visualization System	6
General System Setup	6
Graphics Components	7
Graphics Middleware	7
Visualization software	8
Remote Visualization Applications	8
Generic OpenGL Applications	8
Integration of GPU-accelerated Viz into HPC Operations	13
Enabling GPU Graphics Operations	13
Setting up the X server for Headless Operation	15
Testing the Headless X Operation	17
Using Remote Visualization Capabilities	18
Setting Up Visualization Software with Remote Rendering Capabilities	18
Using Applications without Remote Rendering Capabilities - Configuring VirtualGL and TurboVNC ..	19
Queuing system setup to launch X server	21
Prologue script	21
Epilogue script	21
Potential Pitfalls with the X Server Setup	22
Example: Running CUDA Samples Remotely	23
Conclusion	25
Appendix: Xserver Build Instructions on Cray XK/XC Systems	26

LIST OF FIGURES

Figure 1: Typical Scientific Computing Setup	7
Figure 2: Components of an X application with local rendering	9
Figure 3: Remote rendering using X indirect rendering mode, also known as “X forwarding”. The dashed line represents the network, separating the remote compute server (left) from the user’s workstation (right).	10
Figure 4: Remote Rendering with Rendering Application.	11
Figure 5: Setup for Remote Visualization with Interposer Library.	12

INTRODUCTION

Current and future generation graphics processing unit (GPU)-accelerated supercomputers enable numerical simulations of unprecedented fidelity and scale. These models produce vast amounts of output data with sizes measured in petabytes (and soon exabytes), requiring sophisticated analysis and visualization methods to extract the relevant scientific information. Powerful tools are therefore required to analyze and interactively visualize data sets without moving them over long distances.

Today, computational scientists often follow a post-processing workflow: During the course of the simulation, raw data is written to disk, and at a later time, this data is read back into a dedicated system for analysis and visualization. This workflow puts a huge strain on the file system and requires a visualization system capable of dealing with these massive data sets. In addition, this workflow doesn't allow the user to adjust parameters while the simulation progresses. Using the graphics capabilities of the GPUs installed in the HPC system enables scientists to analyze and visualize the simulation data while it progresses.

CHALLENGES IN REMOTE AND IN-SITU VISUALIZATION

In the conventional scientific workflow, users run their models at a remote high-performance computing (HPC) center, write their results to disk, and transfer those results to a visualization platform for analysis. Historically, this approach made sense for various reasons:

- ▶ The hardware of the compute platform was considerably different from the visualization platform, thus two systems were necessary for the different tasks.
- ▶ The throughput discrepancy between computing hardware and the file system was sufficiently small such that writing to disk was affordable.
- ▶ The demand on visualizations was sufficiently low so that high latency in the visualization system was acceptable.

With an increasing amount of data being generated in simulations, and with significantly more complex visualizations, this traditional workflow starts to show its limitations. However, the widespread use of GPUs on the compute node of HPC systems allows homogenizing the hardware between compute and visualization systems, using the same hardware for both compute and visualization/analysis tasks.

This dual use of the compute node GPUs allows efficient “in-situ” processing of the simulation data, enabling the analysis and visualization of the simulation data while the simulation progresses. This not only reduces the dependency on the I/O subsystem, but also enables new forms of interaction with the simulations like simulation steering.

This white paper describes tools and procedures for utilizing the GPU installed in HPC systems for remote and in-situ visualization. Due to the diversity of HPC configurations, the paper focuses on general concepts rather than detailed recipes. The audience for this paper consists of system administrators responsible for HPC installations, scientific software developers, visualization software developers, and decision makers responsible for procurements related to HPC installations.

GPU-ACCELERATED REMOTE VISUALIZATION SYSTEM

Remote and in-situ visualization consists of a range of different components, all working closely together. In this chapter, we provide an overview of the different components and provide some background on the necessity and contribution of each one.

GENERAL SYSTEM SETUP

In the following we will assume a general system setup as shown in Figure 1. The HPC center is remote to the user, connected through a wide area network link. Rather than getting direct access to the high performance computing resource, the user has to connect through some gateway node. From there he connects to the head-node of the HPC cluster. Rather than starting the jobs directly on the compute nodes, the user needs to submit jobs through a queuing system from the head node.

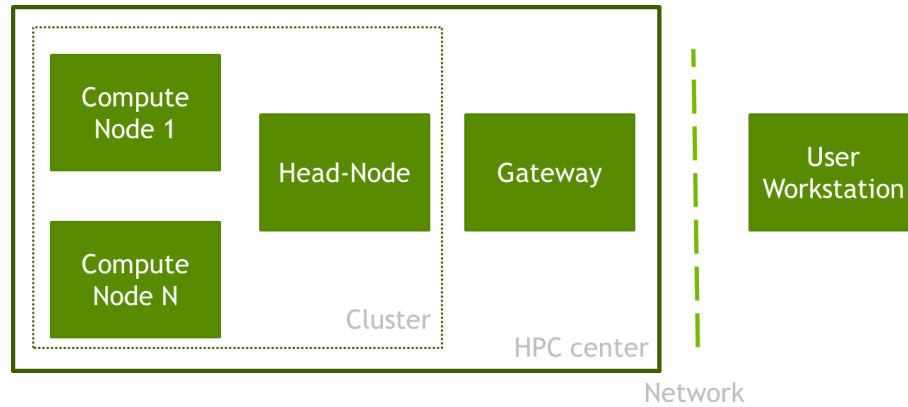


Figure 1: Typical Scientific Computing Setup

Graphics Components

Here we assume that the workstation and the compute nodes are equipped with NVIDIA® GPUs. The compute nodes are running server-class NVIDIA Tesla™ computing solutions (for example, a Tesla K20X), whereas the user is running a consumer-grade NVIDIA GeForce® GPU on a laptop.

In order to use the GPUs installed on the compute node for hardware-accelerated OpenGL, the rendering features need to be turned on. OpenGL support has only recently been added to the server-class GPUs. For Tesla K20X and K20 devices, OpenGL support is an opt-in feature via the GPU Operation Mode. Details on setting the GPU Operation Mode are provided below. On Tesla K40 and newer devices, OpenGL capabilities are always enabled.

Graphics Middleware

Taking advantage of GPUs for rendering tasks is typically accomplished using OpenGL. OpenGL is a state-based API and third party software is needed to manage the OpenGL state, the so-called OpenGL context. At the time of this writing, the only software capable of managing the OpenGL context is the server component of the X windowing system - the so-called X server. Therefore, in order to utilize the GPUs in your HPC cluster for visualization, each compute node needs to be running the X server application.

VISUALIZATION SOFTWARE

A broad range of general purpose visualization tools and individual visualization solutions have been developed over the past decades and many are still in use today. Broadly speaking, two categories of tools can be distinguished, depending on whether they are set up for distributed/remote visualization or not. In the following, we will first describe the commonly used tools already set up for remote visualization, and then go into details for tools that are not set up for remote visualization.

Remote Visualization Applications

Paraview and VisIt are among the most popular massively scalable open source visualization tools. While they can be used stand-alone on a workstation, their main power arises when used in a massively parallel, remote environment. In order to use them in a remote setting, two main challenges need to be addressed:

- ▶ Accessing Paraview or VisIt through the different network layers
- ▶ Utilizing the GPU hardware on each node

All other aspects of parallel, remote rendering are taken care of by the tools which are transparent to the user. Here we will focus primarily on enabling use of the GPU hardware from within these applications. For accessing the tools through multiple network layers and launching them through a queuing system, please refer to the Paraview and VisIt web sites.

Generic OpenGL Applications

In case an application has not been designed for remote visualization, or if it assumes that one of the nodes is able to connect to the user's screen, some additional steps and considerations are necessary for remote visualization.

Local Rendering

In order to understand the steps necessary to obtain hardware accelerated rendering on remote nodes, it is illustrative to look at the individual components of an OpenGL application. Figure 2 shows the basic components of such an application.

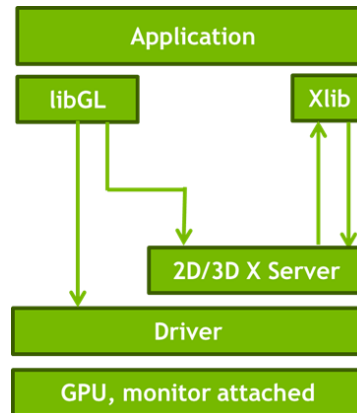


Figure 2: Components of an X application with local rendering

An X windowing system is set up in a client-server fashion, using various forms of inter-process communication between the application and the graphics hardware. The GPU, in this case with an attached monitor, is controlled by the GPU driver. The X server's responsibility is to manage the hardware resources and provide interfaces to the applications. Interaction between the application and the X server is accomplished via the Xlib library, encapsulating the details of the X protocol. If an application wants to use OpenGL, it first queries the X server for creating an OpenGL context. Once the context is available, OpenGL can talk directly to the GPU, bypassing the X server.

Remote Rendering with X Forwarding

The client-server architecture of the X system allows for a seamless extension to a remote configuration. Figure 3 shows a typical scenario with X forwarding: When the application tries to communicate with the X server, it will connect to the remote X server running on the user's desktop, rather than to a local X server. It is therefore not necessary to run an X server on the remote node. In case an application tries to use OpenGL features, it will again query the X server, now running on the remote machine, for an OpenGL context. The OpenGL context provided to the application is therefore one running on the user's GPU. All OpenGL calls will therefore need to go through the network, which can incur a significant performance penalty.

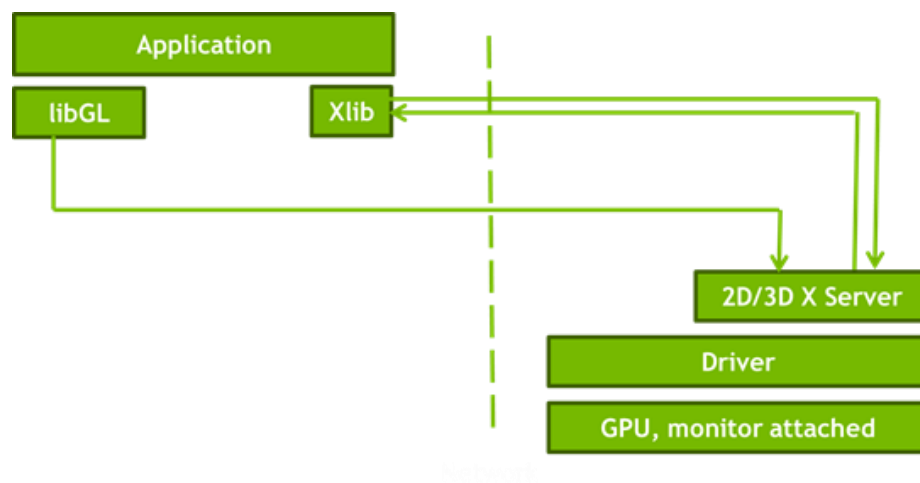


Figure 3: Remote rendering using X indirect rendering mode, also known as “X forwarding”. The dashed line represents the network, separating the remote compute server (left) from the user’s workstation (right).

Remote Rendering with Rendering Application

The impact of remote X can be mitigated in some cases by using X compression techniques, reducing the “chattiness” of the X protocol. For some applications this might be sufficient to provide a reasonably responsive user experience. However, for highly demanding visualizations, this approach is not sufficiently powerful. In addition, X compression approaches (like NoMachine’s NX) do not take advantage of the remote GPU.

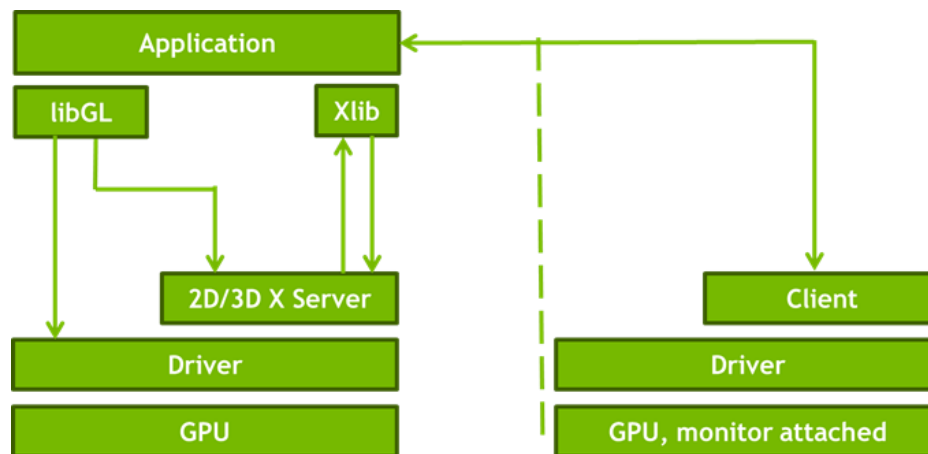


Figure 4: Remote Rendering with Rendering Application.

In order to overcome the low performance of transferring X across the network, rendering on the remote node must be enabled. Figure 4 shows the configuration with a remote visualization application. The application communicates with the X server on the compute node via Xlib. OpenGL contexts, windowing and user interaction are all accomplished by the X server on the compute node. The application on the compute node is in full control, when the rendered frames are captured from the local GPU and transported to the application client running on the user’s workstation. Given that the visualization application is in full control of both the client and server side, compression protocols and the like can be implemented, leading to a high-performance solution for image transport. This is the situation described in the section “Remote Visualization”.

Remote Visualization with Interposer Library

The previous configuration relied on the user's application to capture the remotely rendered frames and transport them to the end user's workstation. In some cases, however, the application's architecture does not foresee any remote visualization capabilities. But even in that case, the GPU installed on the remote compute node can be used for rendering. The key technology in that case are so-called interposer libraries, for example VirtualGL (www.virtualgl.org). Similar to the other cases, an X server needs to be running on the compute node in order to manage the OpenGL context and the GPU's resources. And similar to the previous case, the application uses Xlib to communicate with the X server, and once an OpenGL context is created the application can directly use libGL to communicate with the GPU. However, all management operations are routed through the VirtualGL library. This is accomplished by a so-called interposer library, which is being loaded prior to the regular libGL library. The interposer library inspects the individual OpenGL calls and either acts itself on those calls, or delegates to the system libGL to perform the actual OpenGL operation.

This way, virtualGL knows whenever the application tries to draw the frame buffer to the screen. But rather than drawing to the screen, it captures the buffer content and continues to process it. In the setup shown in Figure 5, VirtualGL hands the content off to a second X server, a so-called Proxy X server, running on the compute node. This is a typical setup for e.g. TurboVNC. In this case, the proxy X server manages compression of the resulting content and transmits the buffer content to the end user. With this configuration, very thin client applications can be used, as they don't need to provide any X server capabilities. All they need to do is display the compressed images and send commands to the proxy X server, allowing even for web-based clients.

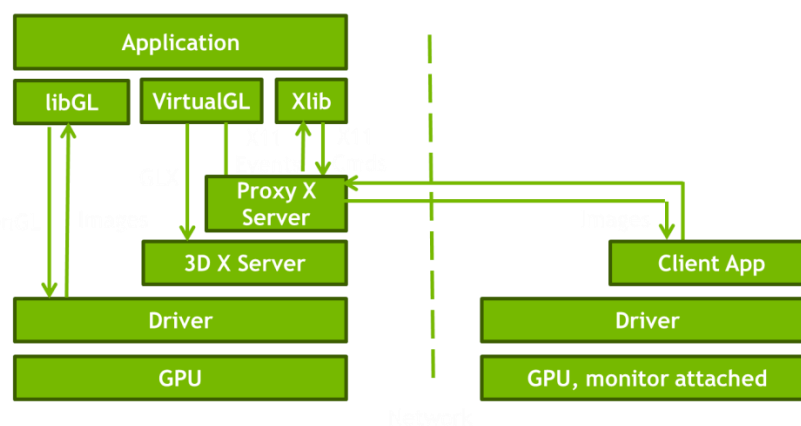


Figure 5: Setup for Remote Visualization with Interposer Library.

INTEGRATION OF GPU-ACCELERATED VIZ INTO HPC OPERATIONS

In the previous example we have described the individual components of a remote visualization system, as well as described different visualization scenarios. In the following, we present the individual steps necessary to make the components work together as a visualization solution.

ENABLING GPU GRAPHICS OPERATIONS

NVIDIA GPUs feature dedicated hardware to perform graphics related operations. For some devices, most notably the Tesla K20X and K20, these graphics capabilities can be enabled on demand. On newer generation devices, starting with the Tesla K40, graphics capabilities are always available.

The GPU Operation Mode (GOM) enables different features of the GPU to meet specific user needs:

- ▶ **0 - All_On**

In this mode everything is enabled and running at full speed.

- ▶ **1 - Compute**

This mode is designed for running only compute tasks. Graphics operations are not allowed.

- ▶ **2 – Low DP (Low Double Precision)**

This mode is designed for running graphics applications not requiring high bandwidth double precision.

The GPU Operation Mode is supported on GK110 M-class and X-class Tesla server products such as the Tesla K20 and K20X from the Kepler family. It is not supported on Quadro and Tesla C-class workstation products. The Tesla K40 has all modes turned on by default. There is no need to switch to a special GPU Operation Mode.

With the GPU Operation Mode switched to “All_On”, all graphics operations supported on Quadro GPUs are supported on Tesla GPUs, except for operations which need a directly connected monitor.

In order to utilize Tesla GPUs for both compute and visualization, the “All On” GOM must be turned on. The most convenient way to accomplish this is via the **nvidia-smi** tool.

To query the state of the GPU Operation Mode of the GPUs in a system, use the following nvidia-smi command:

```
nvidia-smi --query-gpu=gom.current --format=csv
```

The resulting states are either “All_on”, “Compute”, or “low DP”. The following is an example nvidia-smi command output:

```
gom.current
Compute
```

In order to enable hardware-accelerated OpenGL, the GPU needs to be put into the “All on” mode. This can be accomplished by the following command:

```
nvidia-smi --gom=0
```

Changing the GPU operations mode requires root privileges and a system reboot. But once a GPU Operations Mode has been set, it persists through system reboots. For details, refer to the nvidia-smi help using the following command:

```
nvidia-smi -h
```

Switching the GPU Operations Mode to “All On” may lead to a slight increase in power consumption, but generally in the percent range.

SETTING UP THE X SERVER FOR HEADLESS OPERATION

Taking advantage of a server-class Tesla GPU to accelerate OpenGL calls requires an X server to manage the OpenGL contexts. In this section we describe how to configure the X server for headless operation. This assumes that the X server binary, usually located at `/usr/bin/Xorg`, is available on the system.

A broad range of tools and scripts exist to launch the X server on a particular Linux configuration, usually at system startup time, ranging from the simple `xinit` and `startx` to site-specific scripts.

Irrespective of the X server's launch mechanism, the binary will parse at startup a configuration file, by default called `xorg.conf`. This file describes the properties of the individual components of the system.

The simplest mechanism to configure the X server for headless operation is to use the `nvidia-xconfig` tool, shipped with the NVIDIA driver.

When run without any options,

```
nvidia-xconfig
```

the tool will look for an existing `xorg.conf` file and add the necessary options to take advantage of installed GPUs to accelerate X.

Running with the “`--query-gpu-info`” option, `nvidia-xconfig` will print information about all recognized NVIDIA GPUs in the system, including their ID on the PCIe bus.

```
nvidia-xconfig --query-gpu-info
```

```
Number of GPUs: 2
```

```
GPU #0:
```

```
Name       : Tesla K20Xm
UUID       : GPU-e792bc85-1fea-41a1-996e-825917e372cc
PCI BusID  : PCI:4:0:0
Number of Display Devices: 0
```

```
GPU #1:
```

```
Name       : Tesla K20Xm
UUID       : GPU-06163e8f-1f0b-391e-4f8f-22a6c90941d3
PCI BusID  : PCI:5:0:0
Number of Display Devices: 0
```

The following command creates an entry for the GPU with the related PCI BusID into the `xorg.conf` configuration file.

```
nvidia-xconfig --busid=PCI:4:0:0 --use-display-device=none
```

For details, refer to the `nvidia-xconfig` help,

```
nvidia-xconfig -h
```

The alternative approach is to provide a manually created `xorg.conf` file. The file below shows the minimum configuration necessary for headless operation of a GPU:

```
Section "ServerFlags"
    Option "IgnoreABI" "True"
    Option "nolisten" "True"
    Option "AutoAddDevices" "False"
EndSection

Section "ServerLayout"
    Identifier      "layout"
    Screen          0 "nvidia" 0 0
EndSection

Section "Device"
    Identifier      "nvidia"
    Driver          "nvidia"
    BusID           "1:0:0"
EndSection

Section "Screen"
    Identifier      "nvidia"
    Device          "nvidia"
    Option          "UseDisplayDevice" "none"
EndSection
```

The critical option is `"UseDisplayDevice" "none"` under the `"Screens"` section, indicating the headless operation. The other relevant option is the bus ID under the `"Device"` section, which indicates the PCIExpress bus ID for the GPU.

In some cases, when using hardware-accelerated remote visualization by VirtualGL in combination with NVIDIA drivers under various Linux distributions, setting the following `"ConstantFrameRateHint"` option under the `"Devices"` section leads to better stability:

```
Option "ConstantFrameRateHint" "True"
```


When launching the X server, it needs to have access to all the necessary modules and libraries. In particular, make sure that the shared library search path includes the location of the NVIDIA driver if they are not installed in a standard location. For example, on a GPU-accelerated Cray system, you will likely need to include a path to the libGL.so provided by the NVIDIA driver:

```
export LD_LIBRARY_PATH=/opt/cray/nvidia/default/lib64:$LD_LIBRARY_PATH
```

TESTING THE HEADLESS X OPERATION

In order to test the installation of the newly installed X server, it is advisable to run one of the standard OpenGL tests. A particularly helpful one is **glxgears**. After launching the X server on display :0, one can run the benchmark application via

```
export DISPLAY=:0  
glxgears
```

No graphical output will be visible to the user, as rendering will be directed to display :0. However, the application will start reporting the achieved framerate after a few seconds. In case of a failed X installation, glxgears will abort with a “missing glx extension” message or something similar. In those cases, kill your X server, set the correct paths, and then inspect the shared libraries by issuing the following command:

```
ldd `which glxgears`
```

The relevant libraries libGL.so and libglx.so should be pulled from the NVIDIA driver location.

USING REMOTE VISUALIZATION CAPABILITIES

Once OpenGL is enabled on the GPUs and an X server is running, the GPUs installed on the compute node can be used for accelerated rendering. The details of using the GPUs for rendering depends on the visualization software. Here we focus on the two main scenarios: Visualization software with built-in remote rendering capabilities and OpenGL applications without remote rendering capabilities.

Setting Up Visualization Software with Remote Rendering Capabilities

Scalable visualization applications such as Paraview or VisIt offer built-in capabilities for parallel remote data reduction, rendering, and compositing. Both tools offer a client-server architecture where the client runs on the user's workstation and communicates with the server running on a remote visualization cluster.

Using this remote visualization software therefore consists of two parts:

- ▶ Run a parallel server on the visualization cluster, taking advantage of the GPUs per node
- ▶ Connect the client application to the server

Neither Paraview nor VisIt offer pre-built binaries for their parallel servers. It is therefore necessary to build the packages from source.

- ▶ See http://www.paraview.org/Wiki/ParaView:Build_And_Install for build and installation instructions for Paraview.
- ▶ See <http://www.visitusers.org/index.php?title=ParallelPorting> for build and installation instructions for VisIt.

When building these tools, make sure the NVIDIA OpenGL libraries can be detected, either by setting the LD_LIBRARY_PATH environment variables or by specifying their location when configuring the code via cmake.

Once the parallel server is built, it can be run according to tools specific run instruction. See http://www.paraview.org/Wiki/Setting_up_a_ParaView_Server for an example using Paraview.

Using Applications without Remote Rendering Capabilities - Configuring VirtualGL and TurboVNC

Applications not designed for remote rendering often assume that the user has interactive access to the X server. In the case of remote rendering, an extra software layer is needed to transfer data between the user's workstation and the GPU.

One of the possibilities is using an interposer library such as VirtualGL together with the TurboVNC proxy X server. Properly configured, this setup will render the frames on the compute node's GPUs, and then transport them to the end user, with possible compression.

In order to set up VirtualGL/TurboVNC on the compute nodes, the following steps are necessary:

- ▶ X server running on the compute nodes, managing the GPU
- ▶ Proxy X server running on the compute node, responsible for capturing the rendered frames

The TurboVNC proxy X server can run entirely in user space, and it can therefore be left to the user to launch it. An alternative is to start the VNC server in the prologue script. However, the proxy X server will need access to the X server managing the GPU, so it has to be launched after the X server has started. In a typical scenario, with only a single GPU attached to the device, the display of the proxy server can be arbitrarily set, for example, to display :1. Subsequently, all rendering operations on that node should be directed to display :1 in order to take advantage of the VirtualGL mechanism.

```
xvnc :1 -fp /usr/share/fonts/misc &
export DISPLAY=:1
vglrun glxgears
```

Note the use of the `vglrun` launcher script, which sets up the environment to load the VirtualGL library, intercepting some of the OpenGL and GLX functions.

With the above example, the GPU on the compute node will be used for rendering, and the rendered frames will be captured. In order to transfer the frames to the user's workstation, it is necessary to connect the VNC client to the server on the compute node. In the simplest case, this can be accomplished by directly pointing the VNC client to the port on the compute node:

```
vncviewer computeNode01:1
```

In most situations, the compute nodes are not directly visible to the remote user. In that case it's necessary to establish an SSH tunnel from the user's workstation to the compute node. For example, the following establishes a tunnel between the user's workstation (port 5903) and the HPC center's gateway (port 5902), and then a tunnel from the gateway to port 5901 on the compute node (established via the headnode)..

```
ssh -L 5903:localhost:5902 gateway  
ssh -L 5902:computenode01:5901 headnode
```

In this example, the user will be able to connect to the VNC server on the compute node with a

```
vncviewer localhost:3
```

Note that in the above example we needed to establish the tunnel to the compute node. This required knowledge of the node assigned to our allocation via the queuing system. One possibility is to print out this information at simulation setup. An alternative possibility is to establish the tunnel as part of the simulation startup.

QUEUING SYSTEM SETUP TO LAUNCH X SERVER

Many HPC centers enforce a policy that does not allow compute node processes to remain active between subsequent allocations. It is therefore advisable to start/stop an X server for each allocation if selected by the user. In the following sections, we present sample prologue/epilogue scripts for SLURM.

Prologue script

The following script allows the user to start the X server on demand when scheduling the node allocation by specifying the option `--constraint startx`, resp. the `-C startx` option.

In this case, we assume a Cray based system, with the slurm prologue script located in `/opt/slurm/default/etc/prologslurmctld.sh`

```
# parse for the startx constraint
if [[ $SLURM_JOB_CONSTRAINTS =~ .*startx.* ]] then

# convert nodelist provided by SLURM to nodelist useful to
# the parallel remote control launcher, pcmd.
NODELIST=`echo $SLURM_JOB_NODELIST | sed 's/^nid.*\[//'| \
sed 's/\]//'\`
    cd /

# launch the Xserver on all the nodes.
    /opt/cray/nodehealth/default/bin/pcmd -u 2 -r -s -n $NODELIST
    "/opt/xserver/usr/bin/Xorg :0 &">/tmp/startx_log.txt
# wait a bit to ensure X is up before moving forward
    sleep 2
end
```

The above script enables the user to launch e.g. a 10 node job, with an X server running on each node by:

```
salloc -n 10 -C startx
```

Epilogue script

Upon completion of a node's allocation, the X server needs to be terminated. This can be accomplished with the following epilogue script. On a Cray system, a typical slurm epilogue script at `/opt/slurm/default/etc/epilogslurmctld.sh` would be

```
# Kill Xorg
NODELIST=`echo $SLURM_JOB_NODELIST | sed 's/^nid.*\[//'| sed 's/\]//'\`
/opt/cray/nodehealth/default/bin/pcmd -r -s -n $NODELIST
"/dsl/root/bin/killme &"
```

POTENTIAL PITFALLS WITH THE X SERVER SETUP

In a cluster environment, it may seem reasonable to install the X server on a file system shared across all the nodes in the cluster. While this makes perfect sense from a space conservation point of view, the several files usually installed in the X tree are modified by the X server itself. Storing those files in a shared location can lead to race conditions between the multiple X servers. In particular, compiled keymap files, usually stored in

```
/usr/X11R6/xkb/compiled/server-0.xkm
```

and the server log files, usually stored in

```
/usr/var/log
```

must be stored in a node-local directory. This can be accomplished by either specifying an output directory on the command line, or alternately by soft-mounting the corresponding directories to node-local file systems (e.g. /tmp).

EXAMPLE: RUNNING CUDA SAMPLES REMOTELY

In this section we give step-by-step instructions for running one of the CUDA samples with graphics output on a remote system. We assume a system that is set up similar to Figure 1 - A client with a workstation, a remote HPC system with a gateway server, and an attached GPU-accelerated HPC system.

1. Log onto the remote system.

We assume that compilation needs to occur on the head node of the cluster, rather than on the gateway node. The user therefore has to first connect from the workstation to the gateway:

```
ssh gateway
```

and then from the gateway to the cluster's head node:

```
ssh headnode
```

Note that no X forwarding is necessary.

2. Compile the sample code.

Obtain the sample code from the NVIDIA web site. When configuring the code, make sure it picks up the OpenGL library, `libGL.so`, distributed with the NVIDIA driver.

3. Launch X server on compute node if not already running.

Make sure the X server is running on the compute node and that it picks up the configuration for headless operation. In a typical environment, the X server should be launched by the resource management system.

4. Launch TurboVNC on compute node.

Similar to the X server managing the compute nodes' GPU, the TurboVNC proxy X server should be launched by the resource management system. In a typical HPC environment where compute nodes are not shared amongst multiple users, it is safe to assign one specific display to the VNC server. Here we will assume TurboVNC is in charge of display :1, so that it is listening for connections on port 5901.

5. Establish tunnel to compute node.

See the section [Using Applications without Remote Rendering Capabilities](#) for examples on establishing a tunnel from your workstation to port 5901 on the compute node.

6. Connect client to compute node.

At this point, you can connect the TurboVNC client on your workstation to the proxy X server running on the compute node.

7. Run CUDA sample application.

On the compute node, launch the OpenGL application using `vglrun`. Make sure the application uses display :1 for the output in order to get the data through VNC.

CONCLUSION

The conventional scientific computing workflow involves running simulations at a remote HPC center, writing results to a file system and then analyzing and visualizing them in a post-processing manner. With increasing amounts of data generated by the simulations and with increasingly complex analysis algorithms, this approach becomes intractable and new ways for processing the simulation data need to be found.

The widespread use of GPUs on the compute nodes of HPC systems allows not only accelerated simulations, but also use of the GPU's graphics legacy for visualization and analysis tasks. The dual use of the GPUs in the compute nodes allows for in-situ processing of the simulation data and enables scientific visualizations while the simulation progresses. This results in a simplified workflow, reduced dependency on the HPC center's file system, and an overall shortened path to scientific discovery.

Further Reading

Many of the topics described here are highly site specific and the recipes are meant as general guidelines. The following pointers can help to obtain more detailed information to adjust for a specific center's needs.

- ▶ Nvidia-smi: <https://developer.nvidia.com/nvidia-system-management-interface>
- ▶ X server: <http://www.x.org/wiki/XServer/>
- ▶ VirtualGL: <http://www.virtualgl.org>
- ▶ TurboVNC: <http://www.turbovnc.org>
- ▶ SSH tunneling: ssh man page

APPENDIX: XSERVER BUILD INSTRUCTIONS ON CRAY XK/XC SYSTEMS

Some systems, most notable the Cray XC/XK series, do not come with a pre-installed build of the X server. Building and installing this server is left to the user.

In order to build the X server from scratch, a build script can be requested from NVIDIA by sending an email to hpcviz@nvidia.com.

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and other changes to this specification, at any time and/or to discontinue any product or service without notice. Customer should obtain the latest relevant specification before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer. NVIDIA hereby expressly objects to applying any customer general terms and conditions with regard to the purchase of the NVIDIA product referenced in this specification.

NVIDIA products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on these specifications will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this specification. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this specification, or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this specification. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA. Reproduction of information in this specification is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

Trademarks

NVIDIA, the NVIDIA logo, GeForce, and Tesla are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2014 NVIDIA Corporation. All rights reserved.

WP-07313-001_v01