GPU VIRTUALIZATION

Rakesh Tirupathi
Department of Computer Science, Arkansas State University, Jonesboro, Arkansas, United States

Abstract – *Graphics Processing Units (GPUs) are widely used in HPC applications due to its high computation power and parallelism. Consequently, need of virtualization of GPU resources have been increased tremendously to provide efficient resource sharing and reducing the overall costs at the same time. In this we provided a detailed review of various GPU virtualization techniques at both hardware and software level. We present vCUDA, a framework for HPC applications that uses hardware acceleration provided by GPUs to address the performance issues associated with system-level virtualization technology.*

1 Introduction

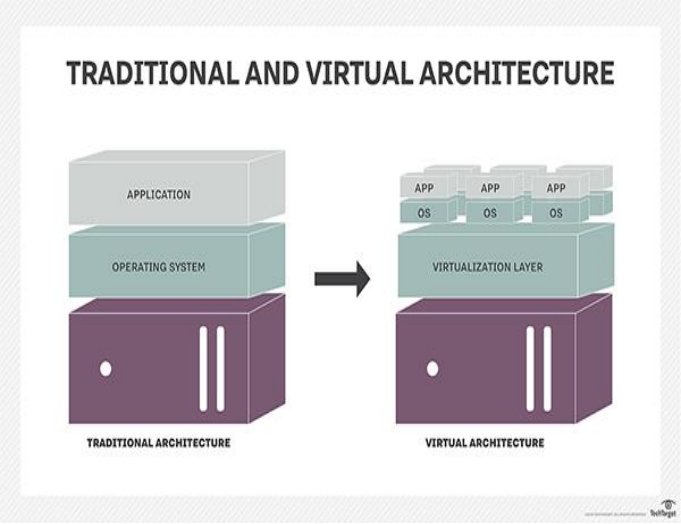
Virtual machines (VMs) have become increasingly popular as a technology for multiplexing both desktop and server commodity x86 computers. Over that time, several critical challenges in CPU virtualization have been overcome, and there are now both software and hardware techniques for virtualizing CPUs with very low overheads. Virtualization machines (VMs) to coexist in a physical machine under the management of a virtual machine monitor (VMM). Technology allows multiple virtual VMM has been applied to many areas including intrusion detection, high-performance computing (HPC), device driver reuse and so on.

2 Virtualization

In computing, virtualization means to create a virtual version of a device or resource, such as a server, storage device, network or even an operating system where the framework divides the resource into one or more execution environments. Even something as simple as partitioning a hard drive is considered virtualization because you take one drive and partition it to create two separate hard drives. Devices, applications and human users are able to interact with the virtual resource as if it were a real single logical resource.

2.1 How Virtualization Works

Virtualization describes a technology in which an application, guest operating system or data storage is abstracted away from the true underlying hardware or software. A key use of virtualization technology is server virtualization, which uses a software layer called a hypervisor to emulate the underlying hardware. This often includes the CPU's memory, I/O and network traffic. The guest operating system, normally interacting with true hardware, is now doing so with a software emulation of that hardware, and often the guest operating system has no idea it's on virtualized hardware. While the performance of this virtual system is not equal to the performance of the operating system running on true hardware, the concept of virtualization works because most guest operating systems and applications don't need the full use of the underlying hardware. This allows for greater flexibility, control and isolation by removing the dependency on a given hardware platform. The diagrammatic representation of the traditional and Virtual Architecture is as given below:



2.1.1 Types of Virtualization

There are several types of Virtualization, they are as given below:

- Network Virtualization
- Storage Virtualization
- Server Virtualization
- Data Virtualization
- Desktop Virtualization
- GPU Virtualization
- Application Virtualization

2.1.2 Nested Virtualization

Nested virtualization is the act of running a hypervisor inside a virtual machine – effectively nesting a hypervisor within a hypervisor. Nested virtualization can be useful for running multiple hypervisors on the same host server. It is also an approach to learning about software products, experimenting with server setups or testing configurations. And, not all hypervisors and operating system versions can nest successfully within all other hypervisors. When talking about nested virtualization, the hypervisor running on

physical hardware is known as the host hypervisor, while the VM running on that hypervisor is called the outer guest. The hypervisor running within the VM is known as the guest hypervisor, while the inner guest or nested guest is the VM running within the other VM.

2.1.3 Why Virtualization

1. Virtual machines can be used to create operating systems, or execution environments with resource limits, and given the right schedulers, resource guarantees. Partitioning usually goes hand-in-hand with quality of service in the creation of QoS-enabled operating systems.
2. Virtual machines can provide the illusion of hardware, or hardware configuration that you do not have (such as SCSI devices, multiple processors) Virtualization can also be used to simulate networks of independent computers.
3. Virtual machines can be used to run multiple operating systems simultaneously: different versions, or even entirely different systems, which can be on hot standby. Some such systems may be hard or impossible to run on newer real hardware.
4. Virtualization can enable existing operating systems to run on shared memory multiprocessors.
5. Virtualization can make tasks such as system migration, backup, and recovery easier and more manageable.
6. Virtualization can be an effective means of providing binary compatibility.
7. Virtualization on commodity hardware has been popular in co-located hosting. Many of the above benefits make such hosting secure, cost-effective, and appealing in general.

2.1.4 Why and when to use vGPUs

GPU virtualization can help make sure virtual desktop performance is top notch. But you must know when and where vGPU cards are necessary before investing in the technology. If you are mostly dealing with task workers, then you can probably skip out on vGPU, because they can get along just fine with plain old VDI. But if you have employees, such as video editors, who work with graphics-intensive applications then GPU acceleration is for you. Once you have determined that, you also need to know what type of GPU technology to adopt -- dedicated hardware, shared hardware or GPU virtualization.

3. GPU Virtualization

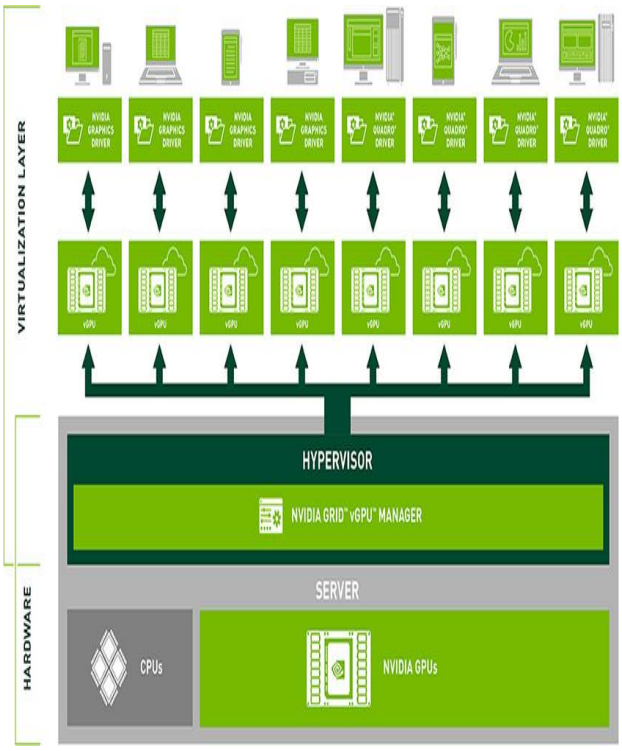
A graphics processing unit (GPU) performs calculations to quickly render images. Even more advanced, a virtualized GPU provides graphics to remote users without doing the actual rendering on the physical device. A virtual graphics processing unit (GPU) is a computer processor that renders graphics on a virtual machine's (VM's) host server rather than on a physical endpoint device. In virtual and remote desktop environments, it is particularly difficult to render and deliver complex graphics to endpoints with adequate performance. The standard GPU was originally developed to offload processing calculations from the CPU for graphics-intensive applications. NVIDIA introduced the first virtual GPU in 2012 to help solve that problem, reducing lag time when delivering graphics to remote users and providing the same performance they would get from a PC. This is especially useful for users that require computer-aided design or 3-D graphics applications. The idea is to provide access to remote users who need complex graphics rendering work done. It does this by allowing the VM's server to interact with a local graphics resource and rendering the graphics quickly so that users can take advantage of them without having to render complex graphics on their own remote devices.

Virtualized GPU technology is particularly compelling for individuals who remote into a VM from a low-powered PC, but even more so for tablet or smartphone users. One way to accomplish this is to hook up the plumbing inside a Windows-based VM environment. Or you could use VMware Fusion for Mac OS users, special-purpose software and APIs for Linux, Android and other runtime environments. NVIDIA's GRID boards make it possible for multiple remote users to share one or more high-end GPUs, to virtualize graphics rendering within the virtual machine. The technology is designed to minimize lag for remote users, even when rendering complex graphics. The Kepler architecture in those boards supports high-performance H.264 encoding for video streaming and can make numerous CUDA cores and physical GPUs available to a pool of remote users.

3.1 NVIDIA Grid Virtual GPU Technology

NVIDIA GRID is the industry's most advanced technology for sharing virtual GPUs (vGPUs) across multiple virtual desktop and applications instances. You can now leverage the full power of NVIDIA data center GPUs to deliver a superior virtual graphics experience to any device, anywhere. The NVIDIA GRID platform offers the highest levels of performance, flexibility, manageability, and security offering the right level of user experience for any virtual workflow.

NVIDIA GRID lets up to sixteen users share each physical GPU, so the graphics resources of the available GPUs can be assigned to virtual machines in an optimally balanced way.



3.2 GPU Virtualization Terms:

3.2.1 Device Pass-Through: This is the simplest virtualization model where the entire GPU is presented to the VM as if directly connected. The virtual GPU is usable by only one VM. The CPU equivalent is assigning a single core for exclusive use by a VM. VMware calls this mode virtual Direct Graphics Accelerator (vDGA).

3.2.2 Partitioning: A GPU is split into virtual GPUs that are used independently by a VM.

3.2.3 Timesharing: Timesharing involves sharing the GPU or portion of between multiple VMs. Also known as oversubscription or multiplexing, the technology for timesharing CPUs is mature while GPU timesharing is being introduced.

3.2.4 Live Migration: The ability to move a running VM from one VM host to another without downtime.

3.3 Approaches to virtualized GPUs

In general, there are three fundamental types of virtualized GPUs: API Intercept, Virtualized GPU, and Pass-through.

3.3.1 API Intercept

The oldest of these, API Intercept, works at the OpenGL and DirectX level. It intercepts commands via an API, sends them to the GPU, then gets them back and shows the results to the user. Since this is all done in software, no GPU features are exposed. This also means that the software capabilities tend to lag behind the GPU in terms of what APIs are supported.

API Intercept typically has good performance when it works, but doesn't have great application compatibility for mid-range to intense 3D apps. As of now it is the only method that supports vMotion.

3.3.2 Virtualized GPU:

This is the hottest spot in desktop virtualization today, spare maybe storage and hyper converged infrastructure. With Virtualized GPU, users get direct access to a part of the GPU. This is preferable to API Intercept because the OS uses the real AMD/NVIDIA/Intel drivers, which means applications can use native graphics calls as opposed to a genericized subset of them.

Virtualized GPU has better performance than API Intercept. Though it gives applications direct access to the CPU, the users are only getting a portion of the CPU, so it can still be limited in certain situations. That said, the application compatibility is good, but vMotion is not supported.

3.3.3 Pass-through:

Pass-through, which if memory serves has been around for longer than Virtualized GPU, connects virtual machines directly to a GPU. If you have two cards in your server, then you get to connect two VMs to GPUs while everyone else gets nothing. This is great for the highest-end workloads, since VMs get access to all of the GPU and its features and application compatibility is great. Pass-through is the most expensive by far.

4. Types of GPU

There are three different companies making virtual GPUs: Intel (GVT-g), AMD (MxGPU), and NVIDIA (vGPU). They each have a different term, but they're really just product names. What sets them apart is how they do their virtualization.

	AMD MxGPU	Intel GVT-g	NVIDIA vGPU
Video RAM	Physical slice*	Physical slice	Physical slice
Shader engines	Physical slice	Time slice	Time slice*
Video decode	No	Time slice	Time slice*
Video encode	No	Time slice	Time slice*
GPU Compute	OpenCL	OpenCL	Passthru only
Hypervisor requirements	SR-IOV	SW Manager	SW Manager
Hypervisors supported	ESX	KVM/Xen	ESX/Xen

(source: Randy Groves, "Virtualized GPUs are now an Option for VDI and DaaS!" BriForum Boston 2016)

4.1.1 Video RAM

Though they do all give a physical slice of VRAM to each virtual machine. The key difference with how each handles VRAM is that since AMD's MxGPU is 100% hardware-based, the individual virtual machines framebuffers (which is what lives in the VRAM) are physically isolated from one another, whereas with NVIDIA and Intel, the isolation is done by software. This is probably not an issue for most people, but could be important in situations where framebuffer security is a requirement.

4.1.2 Shader Engines

AMD differs from the others in how it slices up the shader engines, too. With MxGPU, virtual machines get a dedicated, physical slice of the shaders, whereas Intel and NVIDIA time slice VMs across all of their shaders. The difference is that time slicing gives the users 100% of the GPU for a proportional amount of time, whereas AMD gives users a proportional amount of GPU 100% of the time.

If users on Intel or NVIDIA GPUs aren't using the GPU, that frees up more longer slices of time for users that are using the GPU, so the fewer the users the better the performance. With AMD, if a user isn't using the GPU, those dedicated shaders go unused. That said, since the slicing NVIDIA and Intel do is done in software, the GPU can't be turned back over to the pool until the last command you sent to it has finished executing. That means that misbehaving applications could starve other VMs of GPU resources.

4.1.3 Video Encoding/Decoding

Here's a situation where everyone is different. AMD doesn't have any video encoding or decoding capabilities in MxGPU. Intel and NVIDIA both time slice their encoding/decoding hardware, but NVIDIA has an advantage over Intel in that they use a different time slicer for each component (shaders, video encoding, and video decoding). This adds some flexibility, for example, because a user who is only encoding a video is not affecting the GPU or video decoding users.

4.1.4 GPU Compute

The differences here are basically that AMD and Intel both support OpenCL APIs through virtual machines, while NVIDIA only supports GPGPU use cases in pass-through mode.

4.1.5 Hypervisor Requirements and Hypervisor Support

Both Intel and NVIDIA require a software manager to be installed into the hypervisor. This isn't a big deal since both GPUs are certified to run on certain platforms (more on that in a minute), but it is an extra step. AMD utilizes SR-IOV, which essentially means that they designed their card to present itself to the BIOS in such a way that the BIOS treats it as if it's several cards, which means you don't need a software component in the hypervisor itself.

4.1.6 Virtualization Support for Cuda

CUDA support from five virtualization technology vendors accounting for most of the virtualization market was examined. The five major vendors are VMWare, Microsoft, Oracle, Citrix and Red Hat. A summary is shown in the table below.

	VMware vSphere	Microsoft Hyper-V	Oracle Virtual Box	Citrix XenServer	Red Hat Enterprise Linux
Pass-Through	✓	X	✓ (Linux host server only)	✓	✓
Partitioning	X	X	X	✓	X
Timesharing/ Oversubscription	X	X	X	X	X
Live Migration	X	X	X	X	X

GPU Pass-Through is the common way to support CUDA on VMs. It is hardware intensive option with minimum of one physical GPU per VM required. There is no virtualization technology that supports GPU timesharing or oversubscription for CUDA applications.

5. Virtualized GPU Architecture

GPUs are integrated devices in the form of cards that are attached to a server with a general-purpose processor via a PCI- Express (PCIe) bus. To exploit the GPU computing power, part of the program has to be written as a kernel which

at runtime is sent and executed on the GPU. The GPU driver is in charge of transferring the program, initiating its execution, and handling its completion.

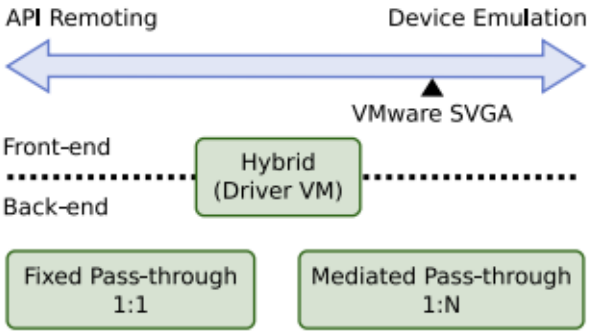
Both the general-purpose server and the GPU feature separate memory maps. Transferring the data required by the kernel and later retrieving back the results is explicitly addressed by the user. Therefore, in an HPC system where a node without a local GPU is running an application that invokes a GPU kernel, we have to provide support for transferring the kernel and data, and dealing with the initiation/completion of the kernel execution on a remote GPU. In particular, our virtualization middleware consists of two parts: the client middleware is installed as shared library on all nodes of the HPC cluster which have no local GPU; and the server middleware is executed in the node(S) equipped with GPU(s). We name these nodes hereafter as clients and server(s), respectively.

6. Virtualization Approaches

There are two approaches for GPU virtualization:

- Low-level abstraction: Virtual GPU (device emulation)
- High-level abstraction: API remoting

The visual representation of GPU virtualization is shown below:



The differences between the virtual GPUs and API remoting is as given below:

	Virtual GPU	API remoting
Method	Virtualzation at GPU device level	Virtualiztion at API level (e.g., OpenGL, DirectX)
Pros	Library-independent	VMM-independent GPU-independent
Cons	VMM-dependent GPU-dependent → Most GPUs are closed and rapidly evolving, so virtualization is difficult	Library-dependent → But, a few libraries (e.g., OpenGL, Direct3D) are prevalently used (# of libraries < # of GPUs)
Use case	Base emulation-based virtualization (e.g., Cirrus, VESA)	Guest extensions used by most VMMs (Xen, KVM, VMware)

At high level, we group GPU virtualization techniques into two categories: front-end (application facing) and back-end (hardware facing)

6.1 Front-end Virtualization

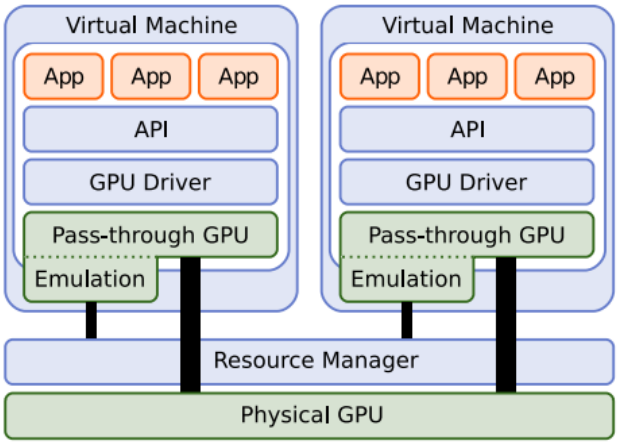
Front-end virtualization introduces a virtualization boundary at a relatively high level in the stack, and runs the graphics driver in the host/hypervisor. This approach does not rely on any GPU vendor- or model-specific details. Access to the GPU is entirely mediated through the vendor provided APIs and drivers on the host while the guest only interacts with software. Current GPUs allow applications many independent "contexts" so multiplexing is easy.

Front-end techniques exist on a continuum between two extremes: API remoting, in which graphics API calls are blindly forwarded from the guest to the external graphics stack via remote procedure call, and device emulation, in which a virtual GPU is emulated and the emulation synthesizes host graphics operations in response to actions by the guest device drivers. These extremes have serious disadvantages that can be overcome by intermediate solutions. Pure API remoting is simple to implement, but completely sacrifices interposition and involves wrapping and forwarding an extremely broad collection of entry points. Pure emulation of a modern GPU delivers excellent interposition and implements a narrower interface, but a highly complicated and under-documented one.

6.2 Back-end Virtualization

Back-end techniques run the graphics driver stack inside the virtual machine with the virtualization boundary between the stack and physical GPU hardware. These techniques have the potential for high performance and fidelity, but multiplexing and especially interposition can be serious challenges. The most obvious back-end virtualization technique is fixed pass-through: the permanent association of a virtual machine with full exclusive access to a physical GPU. Fixed pass-through is not a general solution. It completely forgoes any multiplexing or interposition, and packing machines with one GPU per virtual machine (plus one for the host) is not feasible. One extension of fixed pass-through is mediated pass-through. As mentioned, GPUs already support multiple independent contexts and mediated pass-through proposes dedicating just a context, or set of contexts, to a virtual machine rather than an entire GPU. In this approach, high bandwidth operations (command buffer submission, vertex and texture DMA) would be performed using memory or MMIO resources which are mapped directly to the physical GPU. Low-bandwidth operations (resource allocation, legacy features) may be implemented using fully virtualized resources. This allows multiplexing, but incurs additional costs: the GPU hardware must implement multiple isolated contexts in a way that they can be mapped to different virtual machines efficiently and securely. The host/hypervisor must have enough of a hardware driver to

allocate and manage GPU resources such as memory and contexts. Also, the logical GPUs which appear in each VM may or may not have the same hardware interface which would be exposed by an equivalent physical GPU. This means that mediated pass-through may require changes to the guest device drivers.
The mediated pass-through architecture is as given below:



The design trade-offs for each virtualized graphics techniques is as given below:

Technique	Performance	Fidelity	Multiplexing	Interposition
Software Rendering	very low	high	yes	perfect
Front-end	medium	medium	yes	good
Fixed pass-through	high	high	no	none
Mediated pass-through	high	high	yes	some

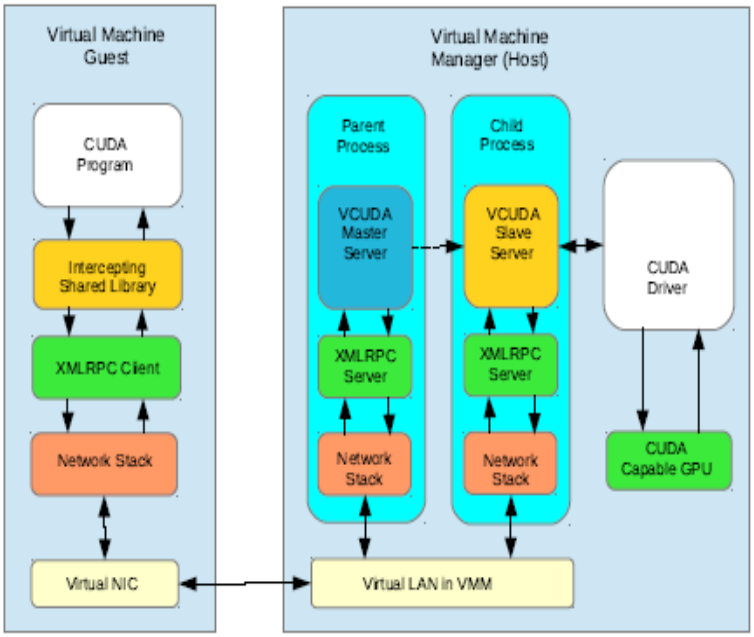
6.3 Hybrid Implementations

By combining front-end virtualization and back-end virtualization, it is possible to use front-end virtualization in environments without a native graphics stack, such as a bare metal hypervisor or a security-enhanced operating system. Pass-through becomes a mechanism for securely deploying an off-the-shelf graphics driver in this environment. Frontend virtualization is then used to multiplex the GPU(s) among several virtual machines while maintaining high degrees of VM portability and isolation. We call this the Driver VM approach. The bulk of the GPU driver stack runs in an isolated low-privilege VM. The physical GPU is securely exposed to the driver VM using fixed pass-through, implemented with the use of I/O virtualization support in the chipset. This requires no special knowledge of the GPU's programming model, and it can be compatible with unmodified GPU drivers. To multiplex the GPU and expose it to other VMs, any form of front-end virtualization may be used.

7. Implementation using vCUDA

The VCUDA system has a client server architecture that virtualizes the CUDA run time API. The client, which resides in the guest VM, intercepts the CUDA API and redirects them to the server. The VCUDA server that is part of the VMM (host), manages the physical GPU device and also ensures resource sharing between multiple clients. For successful remote execution of CUDA programs, certain ordering semantics need to be strictly followed. CUDA has a strictly ordered execution model and does not guarantee data integrity if ordering is violated. Also, the VCUDA system needs to keep track of certain states and proper mapping of these states from client to server is necessary. These are tracked in special map tables by both the client and the server.

The vCUDA virtualization Architecture is as shown below:



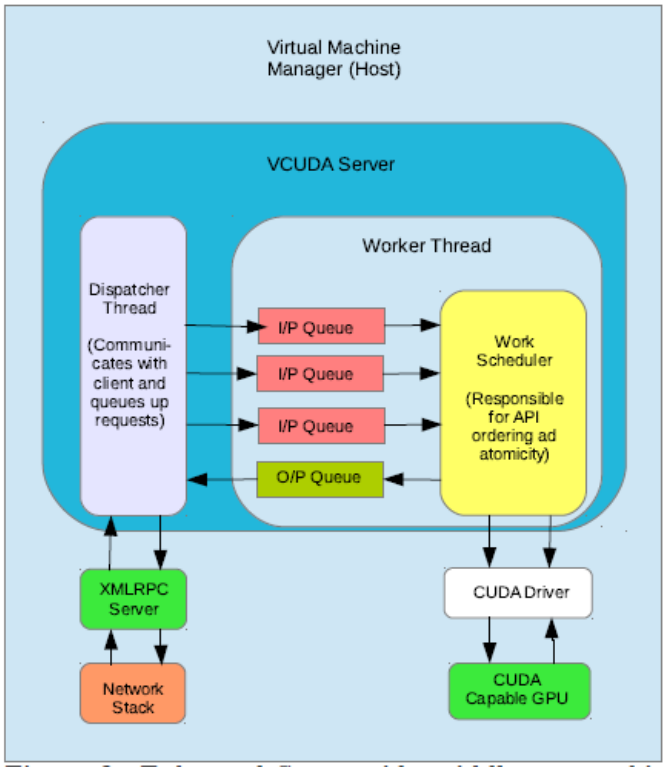
7.1 vCUDA Client Library:

The client side consists of a custom shared library and a XMLRPC client stack. The library replaces the regular CUDA runtime library (libcudart.so) on the guest system. This library implements the CUDA runtime API and redirects the calls and data to the GPU server using appropriate remote function calls. The XMLRPC client stack enables remote function calls and provides XML encoding and XMLRPC protocol. On program start, the client sends a connect request to the master server and if the connection is successful, it receives the port number for a specific server instance (described below). Rest of the program is executed using this server instance. On program exit, the client sends a disconnect request which also ends the corresponding server instance.

7.2 vCUDA Server:

The server has two main components, the XMLRPC server stack and the VCUDA server API implementation. The XMLRPC stack includes the web server and also the RPC handlers. These are responsible for communicating with the client using the XMLRPC protocol, XML decoding and passing on the arguments to the appropriate function. The RPCs, implemented in the VCUDA server, translate to appropriate CUDA runtime API calls. The CUDA run time library interfaces with the CUDA driver which manages execution on the GPU.

The server side middle ware architecture is as given below:



7.3 Multiplexing Guests:

One important aspect of virtualization is resource sharing between multiple guest VMs. VCUDA enables sharing of a single physical GPU among multiple VMs. It achieves this by using different execution contexts for different clients in the CUDA driver. CUDA driver is capable of managing multiple contexts and efficiently switching between them, although preemption is not possible with current GPU technology. Multiple CUDA programs that run as independent processes can share the GPU resources. In our current implementation Figure 1, VCUDA server exploits this by spawning multiple server instances for each client. Each instance is a separate process and hence has a separate CUDA context. When the Master VCUDA instance receives a connect request from a new client, it spawns a child process that listens on a new TCP port, and sends back the port number to the client. Thereafter, the client only communicates with the new VCUDA instance at the new TCP port. A server process ends when the client sends a disconnect request. The CUDA driver efficiently multiplexes work from all contexts, blocking them when GPU resources are busy. This is not the best implementation and a multi-threaded implementation of the server Figure 2 will have better overlapping of tasks from multiple clients as opposed to the multi process implementation, although not without added complexity. In case of the multi-threaded server, a dispatcher thread pushes work into the input queues and the scheduler thread is responsible for maintaining ordering and atomicity (certain API sequences). Since the entire server is a single process, and there is only one worker thread, everything runs in a single CUDA context and API calls from different clients are not blocked except in cases when an atomic API sequence is being run. For example, calls like cudaMalloc() from different clients can be overlapped whereas, a kernel launch sequence (cudaConfigureCall, cudaSetupArgument, cudaLaunch) must be atomic.

8. Conclusion:

Pass-Through is the most common mode for GPU virtualization today. It is supported by VMware, Oracle, Red Hat and Citrix. Citrix is by far the leader in GPU virtualization technology with the ability to take advantage of the NVIDIA Grid platform.

9. References

IEEE Papers on

[1] Micah Dowty, Jeremy Sugerman. "GPU Virtualization on VMware's Hosted I/O Architecture".

[2] Lin Shi, Hao Chen, Jianhua Sun, Kenli Li. "vCUDA : GPU-Accelerated High-Performance Computing in Virtual Machines" ; IEEE Transactions on Computers, VOL. 61, NO. 6, June 2012.

[3] Rangeen Basu, Sidharth Sharma. "Virtualizing GPUs for CUDA based HPC applications".

[4] Wei Huang, Jiuxing Liu, Bulent Abali, Dhabaleswar K. Panda. "A case for High Performance Computing with Virtual Machines".

[5] Mark F. Mergen, Volkmar Uhlig, Orran Krieger, Jimi Xenidis. "Virtualization for High Performance Computing".

[6] Wei Huang, Matthew J. Koop, Qi Gao, Dhabaleswar K. Panda. "Virtual Machine Aware Communication Libraries for High Performance Computing".

Several Online materials from

1) Nvidia
2) Google Scholar
3) YouTube
4) Quora and many other sites.