Towards GPGPU Assisted Computing in Virtualized Environments

Thilo Schmitt, Alexander Weggerle, Christian Himpel, and Peter Schulthess

Institute of Distributed Systems
Ulm University

James-Franck-Ring O-27/3210, 89069 Ulm, Germany

thilo.schmitt@uni-ulm.de

<http://www-vs.informatik.uni-ulm.de/>

Abstract. General Purpose Computation on Graphics Processing Units (GPGPU) makes it possible to use the massive computing power of modern graphics cards for generic high-performance computing. However, the new virtualization technologies will typically not support high-performance graphics cards and as a consequence GPGPU resources can not be used in typical virtualization setups. In this paper we present an approach to introduce accelerated 3D graphics support as well as GPGPU facilities into virtualized environments. We present our proof-of-concept "VirtGL" and discuss architectural considerations and a lean, straight-forward way of implementation. We then give an outlook on how this approach can easily be adopted for virtualizing GPGPU APIs like CUDA, Stream, OpenCL and alike.

Keywords: GPGPU, virtualization, virtualized 3D acceleration, High Performance Computing.

1 Introduction

Accelerated graphics access is a relatively new feature in the virtualization technology realm. Only a few virtualization solutions currently support full 3D-accelerated graphics. It is therefore difficult to do General Purpose Computation on Graphics Processing Units (GPGPU) in virtualized environments.

We suggest an approach for implementing fully accelerated 3D-graphics support. Our approach relays graphics commands through the virtualization layer to the readily available graphics card interface of the host system. As it avoids thick and multiple software layers the resulting implementation turns out to be fast, lean, robust and straight-forward.

"VirtGL" is a software package which brings accelerated 3D-graphics to virtualized guest operating systems. It reuses existing abstractions provided by the hosting operating system and lets them reappear in the guest environment. In other words: A "tunnel" between the guest's graphics interface abstraction and the pre-existing functionality in the host is created. With VirtGL we are able

to provide high-end High Performance Computing facilities in virtualized environments. VirtGL integrates GPGPU techniques with virtualization technology, with low implementation costs as well as with minimal performance overhead.

2 Environment

In this section we give an overview of the environment, in which we have established VirtGL. We show the virtualization solution chosen (QEMU), the graphics card programming interface we use (OpenGL) and the operating system we targeted for a prototype driver on the virtualization guest side (Rainbow OS).

2.1 Virtualization

Several virtualization solutions are currently available for productive use. To add our new VirtGL device emulation to the virtualization software it needs to be available as source code. As one of the most important goals of our operating system project is a lean system, adding new complexity besides a new driver to our operating system was not an option. Thus, paravirtualization is not viable for our purposes. Accordingly, only the following virtualization solutions are considered to satisfy our requirements: VirtualBox Open Source Edition, Xen (version 3.0 or later), QEMU, Kernel Virtual Machine (KVM), Bochs.

The decision was made in favor of QEMU, because it is viable even without hardware virtualization. Device emulation modules in QEMU can easily be ported to KVM, Xen HVM and VirtualBox [9].

Our VirtGL device emulation uses memory mapped IO (MMIO) as the device's interface for the communication between the guest operating system and the host. QEMU devices can register such MMIO regions and corresponding handlers to be called on MMIO operations. Whenever the guest reads from or writes to a MMIO region, QEMU calls the according registered handlers. As a consequence the execution traps into QEMU. To determine the result of the access, the registered handler is called and the result is passed to the guest. So every read or write access to this region is immediately handled by the emulated device within QEMU.

2.2 Graphics Card Programming

The Open Graphics Library (OpenGL) [5] is a platform and programming language independent API. OpenGL is designed to offer a single and uniform interface to access various different graphics cards. To compensate for the different feature sets of the cards the API is capable to emulate in software missing features of the hardware. OpenGL programs use this API to view 2D- and 3D-data and images. The API does also support the use of shaders. Shaders are streaming processors, specially designed to execute a huge number of floating point operations, which are typically required to render a 3D scene. OpenGL shader language (GLSL) is used to program the shaders.

Shader programming can also be used for general purpose computation on graphics hardware (GPGPU). High-level abstractions were created to alleviate the programmer's job. With, for example, CUDA [8] and OpenCL [4] high-level languages, libraries and toolchains are made available to support the GPU programmer. These tools and interfaces are finally based on the above mentioned shader programming facilities like GLSL (as part of OpenGL).

2.3 Operating System

Rainbow OS is a transactional distributed memory (TDM) operating system for commodity 64-bit multicore architectures. The PCs constitute a cluster connected via Gigabit Ethernet [11]. Rainbow OS is almost completely implemented in a Java-like language. With our locally developed high-speed compiler SJC [1], Rainbow OS compiles into native x86 code, thereby avoiding the need for a Java Virtual Machine (JVM). Objects are accessed natively without interposed middleware layers or wrapper classes. So Rainbow OS offers to system programmers and application programmers the benefits of a strongly type-safe language coupled with fast runtime performance. Code and data structures of all applications as well as the kernel are accessible and shared by all nodes of the cluster (single system image) using a distributed memory abstraction. In order to guarantee a consistent memory perspective for both operating system components and application tasks Rainbow OS implements a unique transactional consistency mechanism [13].

3 VirtGL

3.1 Architecture

The initial and most important idea behind VirtGL was to develop a lean solution with low code complexity. Another important design goal was to develop the QEMU part without modifying the core of QEMU. We needed a module that was easily interchangeable between subsequent versions of QEMU and at a later time easily transferable to KVM.

VirtGL comprises two components: the *VirtGL device* and the *VirtGL driver*. The VirtGL device is a new emulated device, which is added as a module to the virtualization software. From the perspective of the guest system it is a hardware OpenGL interface. This is VirtGL's guest-system independent core. On the guest system's side a device driver for the VirtGL device is needed. That driver is guest-system specific by nature.

Important to note is, that the interface of the VirtGL device essentially is an OpenGL interface "in hardware" and the VirtGL driver making this interface accessible exposes itself as an OpenGL interface, too. The driver's interface can be directly used by the applications. Thus, particularly we *remove* the need for several layers of interfaces and wrappers in the guest system. Usually operating systems have a stack of interface levels with successively more abstract semantics:

- the graphics card exposes an intricate interface, which is vendor specific or even model specific,
- the device driver exposes an interface, which is typical for graphics, drivers and often is vendor specific,
- an abstraction interface that allows uniform access to all drivers,
- often several additional wrappers or filters,
- the actual graphics interface in standardized way (e.g. OpenGL) to be used by applications.

With VirtGL the situation is reduced to (cf. figure 1):

- the VirtGL driver exposes a standard graphics interface (e.g. OpenGL)
- the device driver in the guest OS exposes a standard graphics interface (e.g. OpenGL), which is at the same time the actual graphics interface to be used

Fig. 1. VirtGL architecture

Not only the number of layers and interfaces is reduced. Also the mapping between the layers is much simpler. This is due to the fact that semantics do not change from one layer to the next; merely the way of identifying functions and passing parameters is different. Thus relaying a call from the software side to the hardware side is simple and efficiently implementable.

3.2 Implementation

Few changes were actually necessary in QEMU. From the point of view of QEMU this simply results in the addition of a simple PCI device. The changes essentially consist of: (1) the VirtGL emulation module: a C-file and its header file ("VirtGL core"); (2) a little patch that instructs QEMU to load the module just added; (3) adjusting the Makefile to link against certain libraries (OpenGL, window management).

The VirtGL core registers some callback routines during initialization time. These callback functions handle intercepted accesses to memory mapped I/O regions (see above). It is this piece of software that ultimately issues the OpenGL commands to the host system's OpenGL interface. The intercepted memory accesses tell the VirtGL core which OpenGL functions to call and which parameters to pass.

The guest system - in our case the locally developed Rainbow OS - merely provides a device driver for the VirtGL device. Because the VirtGL device is emulated as a traditional PCI device all the preexisting features in the operating system dealing with PCI device enumeration and configuration can be used. The VirtGL device is configured and commands are issued by reading and writing to MMIO regions. The device's PCI configuration space reports two MMIO regions which are required by the VirtGL device.

The first and most important region is the so called "command region", typically one page (4 KB) in size. All accesses are intercepted by QEMU and delegated to the VirtGL device emulation module. Such accesses result in a state change of the emulated VirtGL device. The second region is called the "direct data region". It has a configurable size, we currently chose to spend 2 MB. There is a corresponding memory block assigned and none of the accesses to this region are intercepted by QEMU. The direct data region is not really physical memory observable by the guest system. Rather it should be perceived as device memory embedded in the emulated PCI device (VirtGL graphics device) - very much like dedicated graphics RAM on real graphics cards.

Before an OpenGL command is issued, the parameters are written to the direct data region and then the command code assigned to the required OpenGL function is written to the command region and suitably intercepted. Thus, the device emulation will remain inactive as long as not all parameters are written. Writing the command code is intercepted to allow an immediate response. Because all parameters are known and the command code specifies the OpenGL function to be called, the command is immediately carried out.

3.3 Measurements

The performance of VirtGL was tested using the well-known GLXgears¹. For a maximum of comparability the native GLXgears implementation was ported to Rainbow OS and QEMU.

In the virtualized QEMU environment of the guest system, the hardware timers and clocks might not provide correct values. However, during the measurements it is essential to have a reliable clock, so the code that determines the frames per second rate (FPS), was moved to the QEMU counterpart. Effectively, the time between two buffer swappings is measured to determine the FPS values. Due to the timing issues we are faced with in the virtualized guest system, the rotation of the gears was done in every frame with a constant angle².

¹ Part of the Mesa 3D Graphics Library 7.7

² The Mesa GLXgears implementation calculates dynamically a rotation angle, but this has no impact on the performance.

The test system was a Lenovo Thinkpad T410 with a Nvidia NVS 3100M graphics card and an Intel Core i7 M620 CPU and 4 GB of RAM. The operating system was a recent Linux kernel (2.6.32.9) with the proprietary Nvidia graphics driver (190.53). The CPU frequency was set to 1.20 GHz in the first test and to 2.67 GHz in a second test run. The Linux host system was booted using the kernel parameter “`nosmp`” to suppress the influence of optimizations that could be achieved with more than one processor.

The tests ran for approximately one minute printing every 5 seconds the current FPS rate. Afterwards the frame rate was averaged over the measuring points. At the CPU frequency of 1.20 GHz we achieved a performance of approximately 93% (Rainbow 5280 FPS and native 5670). With an over 2 times higher CPU frequency (2.67 GHz) the performance of VirtGL was about 99% (Rainbow 5877 FPS and native 5903 FPS). The performance difference can be explained with the overhead of the virtualization. The higher the CPU frequency, the smaller the weighting of the overhead. The outcome shows that the CPU frequency has only a small impact and can be neglected.

4 Perspective

VirtGL currently supports only a tiny subset of OpenGL functions. We plan to extend the set of supported functions extensively in the future. Especially supporting more texture-related functions and integration of shader support is one of the most urgent needed extensions in VirtGL.

We also expect that relaying the shader functions would have only marginal negative effect on overall performance. The shader-related functions will typically only be used in the initialization part of the program and very rarely in the later program runtime. Texture operations might be used more frequently in GPGPU applications and the amount of data to be moved could be much more compared to typical 3D graphics applications. How this behavior impacts the overall performance remains unclear at the moment. Due to the fact, that VirtGL at the moment introduces one additional data copying step might be a downside. Preliminary measurements indicate that the impact is not as big as suspected at first. Thus, some refinements targeted on this issue might be needed and are currently in the evaluation phase.

As soon as all texture- and shader-related functions can be relayed from the guest to the host system, VirtGL will immediately be ready for the GPGPU use. A refined version of VirtGL would enable HPC applications in virtualized environments to use the GPGPU technology. Carrying the thought a bit further, VirtGL could enable GPGPU usage for HPC clouds: Infrastructure-as-a-Service providers could make GPGPU facilities available within their node instances, which are realized via virtualization.

5 Related Work

The research to provide accelerated 3D graphics to virtual machines is going on since several years. There are a number of different solutions. The most common

solution is [7], which describes a way of reusing device drivers from the host in the virtual machine. This is a low-level approach which offers direct access to the host device driver. The authors made no attempt to use this approach to graphics cards. Due to the fact that the device driver has to be exclusively used by the virtual machine this approach is rather inappropriate for graphics cards, unless there are multiple graphics cards installed. This is also true for the approach of allowing a direct access to the graphics card itself from within the guest operating system.

There are a number of publications related to replacing the original library with a stub which forwards the OpenGL command stream. VirtualGL [12] is designed to provide accelerated 3D-graphics for remote display software like VNC. The data is sent over a network connection. This is a by far more indirect solution than the VirtGL approach; latencies can be expected to be dramatically higher and due to the interposed network connection overall performance should be considerably lower. A similar approach has WireGL [3], system for scalable interactive rendering on clusters. It focuses on distributing rendering resources and thus is not geared towards virtualization appliances.

In the past years several programs were published specializing in accelerated graphics for virtual machine guests. VMGL [6] uses the WireGL network protocol for communication between the guest and the host. Apart from displaying 3D data, the GPU can also be used for scientific calculations. Shi et al. [10] proposed a system to use the GPGPU libraries from inside an guest operating system. They use classical XML based remote procedure calls (RPC), which, due to its massive indirections and rather unfortunate data representation, imposes a considerable overhead compared to the custom-tailored VirtGL interface.

The system closest to VirtGL might be GViM [2], which makes the CUDA facilities of the host available inside the guest system. GViM uses queues of CUDA commands which are filled by the guest system's driver, eventually processed in the hypervisor and by then are ultimately issued to the host's CUDA interface. To achieve the functionality GViM introduces additional software layers. In contrast, our VirtGL package *removes* software layers and thus has a leaner design and a smaller footprint. Unlike GViM, VirtGL does not uncouple the processing of the commands from their issuing. VirtGL rather provides a direct interface to work with which entails immediate command execution/relaying.

None of the solutions found has the extremely lean and straight-forward characteristics of the VirtGL package. Most of the listed projects impose a considerable overhead, which VirtGL does not.

6 Conclusion

In this paper we presented an approach of re-using host system facilities in the guest environment. We used this approach to introduce accelerated 3D graphics support for virtualized environments. Our tests show that the approach is viable and that the solution has reasonable overall performance. We have found that our approach lends itself to a lean and simple software design, thus raising

software quality. Compared to the implementation of a native graphics driver our development effort, the resulting code size and the program complexity is reduced by orders of magnitude. The approach also allows us to more easily keep up with fast evolving 3D-graphics technology.

VirtGL in its current state is not ready for production environments. At the moment it is rather a “proof of concept”. Currently, a small subset of OpenGL functions is supported. Clearly we aim to support the complete OpenGL function set. One of the next steps would be to support modern shader functions and then to extend the approach with the OpenCL or CUDA interfaces. In the near future we will extend, improve and optimize VirtGL in several ways. In the more remote future VirtGL might facilitate GPGPU usage for HPC clouds.

References

1. Frenz, S.: Small Java Compiler, <http://www.fam-frenz.de/stefan/compiler.html>
2. Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., Ranganathan, P.: GVim: GPU-accelerated virtual machines. In: HPCVirt 2009: Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing, pp. 17–24. ACM, New York (2009)
3. Humphreys, G., Eldridge, M., Buck, I., Stoll, G., Everett, M., Hanrahan, P.: WireGL: a scalable graphics system for clusters. In: SIGGRAPH 2001: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques. ACM, New York (2001)
4. Khronos Group: OpenCL, <http://www.khronos.org/opencl/>
5. Khronos Group: OpenGL, <http://www.opengl.org/>
6. Lagar-Cavilla, H.A., Tolia, N., Satyanarayanan, M., de Lara, E.: VMM-Independent Graphics Acceleration. In: Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE), San Diego, CA (2007)
7. LeVasseur, J., Uhlig, V., Stoess, J., Gotz, S.: Unmodified device driver reuse and improved system dependability via virtual machines. In: OSDI 2004: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation. USENIX Association, Berkeley (2004)
8. NVIDIA: CUDA, http://www.nvidia.com/object/cuda_home_new.html
9. Richter, J.: Porting of the NE2000 Device Model. Großer beleg, TU Dresden, Germany (2009)
10. Shi, L., Chen, H., Sun, J.: vCUDA: GPU accelerated high performance computing in virtual machines. In: IPDPS 2009: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1–11. IEEE Computer Society, Washington, DC, USA (2009)
11. Ulm University: Rainbow OS, <http://www.rainbow-os.net/>
12. VirtualGL, <http://www.virtualgl.org/>
13. Wende, M.: Communication model of a distributed virtual memory. Ph.D. thesis, Ulm University (2003) (in German)