Trillium: The code is the IR

Amogh Akshintala
The University of North Carolina
Email: aakshintala@cs.unc.edu

Hangchen Yu, Arthur Peters
The University of Texas at Austin
Email: hyu@cs.utexas.edu
Email: amp@cs.utexas.edu

Christopher J. Rossbach
The University of Texas at Austin
and VMware Research Group
Email: rossbach@cs.utexas.edu

Abstract—GPUs are the platform of choice for many general purpose workloads such as machine learning. This is driving demand for better GPGPU support in virtualized environments like the cloud. Despite significant research attention, GPGPU virtualization remains largely an open problem due to the challenge of balancing performance against key virtualization properties: compatibility, isolation, and interposition. Consequently, two different approaches to GPGPU virtualization have been adopted by the industry: Cloud service providers, such as AWS, support GPU-capable VMs using PCIe-pass-through techniques that bypass virtualization entirely, sacrificing its benefits; Virtualization vendors, such as BitFusion and Dell XaaS, support GPGPU virtualization using user-space API-remoting, which retains some of the benefits of virtualization, but elides hypervisor interposition, thereby giving up key virtualization properties.

We hypothesize that while API-remoting may be the only viable software virtualization technique (as it interposes the only practical interface), API-remoting should not be implemented purely in user-space. We revisit VMware’s SVGA in the context of GPGPU computing and find that hypervisor-mediated API-remoting is efficient: Decoupling device virtualization from GPU ISA virtualization is key to preserving the raw speedup from GPGPU acceleration, while also preserving the benefits of hypervisor-mediation: migration, isolation, fairness, etc.

Index Terms—Virtualization, GPGPU, Compute Accelerator

I. INTRODUCTION

In many parallel computing domains, compute density and programmability [8, 60, 34] have made GPUs the clear choice for efficiency and performance [4]. Popular machine learning frameworks such as Caffe [38], Tensorflow [13], Microsoft CNTK [72], and Torch7 [25] rely on GPU acceleration heavily. GPUs have made significant inroads in HPC as well: five of the top seven supercomputers in the world are powered by GPUs [12].

Despite much prior research [69, 37, 14, 66] on GPGPU virtualization, practical options currently available to providers of virtual infrastructure all involve bypassing the hypervisor. The most commonly adopted technique is to dedicate GPUs to single VM instances via PCIe pass-through [16, 64], thereby giving up the consolidation and fault tolerance benefits of virtualization. More recently, industry players such as VMware, Dell and BitFusion have introduced user-space API-remoting [21, 42, 53, 68, 30] based solutions as an alternative to pass-through. API-remoting recovers the consolidation and encapsulation benefits of virtualization but bypasses hypervisor interposition. The absence of hypervisor interposition results in multiple disjoint resource managers (the remote user-space API executor and the hypervisor) with no insight into each others’ decisions, thereby leading to poor decision making, and priority-inversion problems [54].

To recover hypervisor interposition while maintaining low-overhead, we retrofit GPGPU support into a virtual GPU device: We added support for OpenCL to an implementation of the SVGA [28] (see § II-B) design in Xen, by implementing the key missing component—a compiler for SVGA’s TGSi virtual ISA. This effort helped us realize that because GPUs already support vendor-specific virtual ISAs (vISAs), the additional vISA provides little benefit. In fact, we found that it harms performance by necessitating a translation layer that obscures the program’s semantic information from the final vendor-provided compiler. Drawing on this lesson, we adapted Trillium to take a more flexible approach to ISA virtualization: eliding it entirely when the host GPU stack bundles a compiler (most do), and using LLVM IR, when necessary, to provide a common target for GPGPU drivers.

Trillium represents an unexplored point in the GPGPU virtualization design space: hypervisor-mediated API-remoting. Trillium is an existence proof of a viable alternative design that preserves desirable virtualization properties such as consolidation, hypervisor interposition, isolation, encapsulation, etc., without requiring full hardware virtualization. Trillium outperforms a full virtualization system from the literature by up to $14 \times (5.5 \times$ on average) and outperforms the para-virtual SVGA-like design by as much as $7.3 \times (5.4 \times$ on average).

This paper makes the following contributions:

- We show that API-remoting does not have to be done entirely in user-space and that it can be hypervisor-mediated with minimal loss of performance.
- We implement GPGPU support for an SVGA-like design in the Xen hypervisor, by completing a long-missing element—the TGSi compiler—in order to leverage OpenCL support provided by the Mesa/Gallium graphics stack for Linux, via the Clover [3] project.
- We propose an improved design called Trillium that removes the necessity for the vISA defined by SVGA resulting in dramatic performance improvements.
- We provide the first (to our knowledge) comprehensive empirical and qualitative comparison of a wide range of fundamental virtualization techniques from the literature.
TABLE I: COMPARISON OF EXISTING GPU VIRTUALIZATION PROPOSALS, GROUPED BY APPROACHa

<table>
<thead>
<tr>
<th>Technique</th>
<th>System</th>
<th>lib unmod</th>
<th>OS unmod</th>
<th>lib-compat</th>
<th>hw-compat</th>
<th>sharing</th>
<th>isolation</th>
<th>migration</th>
<th>sched policy</th>
<th>graphics</th>
<th>I/D</th>
<th>benchmark</th>
<th>slowdown</th>
<th>native speedup</th>
<th>virtual speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full-virtual</td>
<td>GPUvm [62]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>Rodinia</td>
<td>141×</td>
<td>11.4×</td>
<td>0.08×</td>
</tr>
<tr>
<td>PCIe Pass-thru</td>
<td>AWS GPU [15]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>Any</td>
<td>1×</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>API remoting</td>
<td>GVIS [35]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>CUDA 1.1 SDK</td>
<td>1.16×</td>
<td>19×</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>gVirUS [31]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>CUDA 2.3 MM</td>
<td>3.1×</td>
<td>11.1×</td>
<td>3.6×</td>
</tr>
<tr>
<td></td>
<td>vmCUDA [58]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>CUDA 4.0 SDK</td>
<td>1.91×</td>
<td>6×</td>
<td>3.1×</td>
</tr>
<tr>
<td>Distributed</td>
<td>rCUDA [30, 53]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>CUDA 3.1 SDK</td>
<td>1.83×</td>
<td>31.7×</td>
<td>27.2×</td>
</tr>
<tr>
<td>API remoting</td>
<td>GridCUDA [48]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>SNU NPB</td>
<td>12.3×</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VCL [18]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>Stencil2D</td>
<td>22×</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Para-virtual</td>
<td>GPUvm [62]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>Rodinia</td>
<td>5.9×</td>
<td>11.4×</td>
<td>1.9×</td>
</tr>
<tr>
<td></td>
<td>HSA-KVM [37]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>AMD OCL SDK</td>
<td>1.1×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LoCu [33]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>Rodinia</td>
<td>1.01×</td>
<td>11.4×</td>
<td>11.3×</td>
</tr>
<tr>
<td></td>
<td>SVGA2 [28]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>2D, gaming</td>
<td>3.9×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Paradice [17]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>OpenGl, OpenCl</td>
<td>1.1×</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VGVM [65]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>CUDA 5.0 SDK</td>
<td>1.02×</td>
<td>33×</td>
<td>32.3×</td>
</tr>
<tr>
<td></td>
<td>TRILLIUM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>D</td>
<td>Rodinia</td>
<td>2.4×</td>
<td>11.4×</td>
<td>4.8×</td>
</tr>
</tbody>
</table>

a. The lib unmod and OS unmod columns indicate ability to support unmodified guest libraries and OS driver. The lib-compat and hw-compat columns indicate the ability (compatibility) to support a GPU device abstraction that is independent of framework or hardware actually present on the host. The sharing, isolation and migration columns indicate cross-domain sharing, isolation and some attempt to support fairness or performance isolation (policies such as RR Round-Robin, XC XenonCredit, HW hardware-managed, etc.). The migration shows support for VM migration. I/D indicates it supports either integrated or discrete GPU.

b. The table includes performance entries for each system including the geometric-mean slowdown (execution time relative to native execution) across all reported benchmarks. We additionally include the benchmarks used, and where possible, a report (or estimate) of the geometric-mean speedup one should expect for using GPUs over CPUs using hardware similar to that used in this paper. The final column is the expected geometric-mean speedup for the given benchmarks running in the virtual GPGPU system over running on native GPUs. The column is computed as the expected speedup from GPUs divided by the slowdown induced by virtualization.

c. Entries where overheads eclipse GPU-based performance gains are marked in red; performance profitable entries are blue. The greyed out cells indicate the metric is meaningless for that design. Light grey cells mean the data is unavailable.

II. BACKGROUND

Existing GPU virtualization solutions [28, 44] support graphics frameworks like Direct3D [22], OpenGL [56]. In principle, there should be no fundamental difference between GPU virtualization for graphics versus compute workloads. In practice, they have significantly different goals: For graphics, virtualization designs target an interactive frame rate (18-30 fps) [7]. For GPGPU compute, virtualization designs must preserve the raw speedup achieved by the hand-optimized GPGPU application, which is a considerably harder target to hit. As a result, GPGPU virtualization remains an open problem. While graphics devices have long enjoyed well-defined OS abstractions and interfaces [50], research attention to OS abstractions for GPGPUs [54, 55, 59, 39, 40, 43] has yielded little consensus.

A. Traditional Virtualization Techniques

An ideal GPGPU virtualization design would require no modification of guest applications, libraries and OSes (compatibility), arbitrate fair and isolated sharing of GPU resources between mutually distrustful VMs (sharing and isolation) at the native performance of the hardware (performance), while allowing virtualized software and physical hardware to evolve independently (encapsulation). Table 1 presents designs in the literature characterized by the properties sacrificed or preserved by traditional virtualization techniques.

Pass-through techniques provide a VM with full exclusive access to a physical GPU, yielding native performance at the cost of interposition, compatibility and isolation.

Device emulation [20] provides a full-fidelity software-backed virtual device which yields excellent compatibility, interposition, and isolation. However, device emulation can’t support hardware acceleration making it untenable for virtualizing GPGPUs.

Full virtualization provides a virtual environment in which unmodified GPGPU programs run on unmodified guest software stacks. Full virtualization designs from the literature [64, 62] show that overheads can be staggering due to trap-based interposition of interactions through MMIO and memory-mapped command-queues.

Para-virtualization [62, 28, 65, 45, 36, 32, 49, 51, 61, 70, 17] refers to any design in which guest artifacts are modified to work in concert with the virtualization layer. For example, VMware’s SVGA [28] supports an efficiently interoperable para-virtual device abstraction, but sacrifices compatibility by requiring modified guest drivers and libraries.

API remoting designs interpose application-level API calls (e.g. by shimming a dynamic library) and remote them to a
user-level GPGPU framework (e.g., CUDA, OpenCL) in the host [58, 35, 31], on a dedicated appliance VM [68], or on a remote server [30, 53, 48, 42, 18, 29, 47]. API remoting can easily provide near-native performance, at a loss of interposition for the hypervisor, and poor compatibility — guest libraries or applications must change, and evolve with any changes in the underlying GPGPU framework.

**Hardware virtualization support** (e.g., Single Root I/O Virtualization (SR-IOV)) enables a single physical device to present itself as multiple virtual devices. A hypervisor can manage and distribute these virtual devices to guests, effectively deferring virtualization, scheduling, and resource management to the hardware. SR-IOV exhibits close to native performance [27], but this is achieved at the cost of interposition — the hypervisor can’t interpose on any interactions with the hardware. SR-IOV also suffers from the multiple administrator problem: the hardware controller and the hypervisor/OS may make mutually inconsistent decisions leading to unpredictable behavior.

**B. SVGA**

SVGA [28] remotes DirectX and OpenGL over an emulated (software) PCIe device. The SVGA virtual device behaves like a physical GPU, by exporting virtual resources in the form of registers, extents of guest memory accessible to the virtual device, and a command queue. I/O registers (used for mode switching, IRQs, memory allocation) are mapped in an interposed PCIe Base Address Register (BAR) to enable synchronous emulation. Access to GPU memory is supported through asynchronous DMA. Figure 1 presents an overview of SVGA.

SVGA combines many aspects of full-, para-virtual and API remoting designs. Unmodified guests can transparently use SVGA as a VGA device, making full virtualization possible where necessary. However, access to GPU acceleration requires para-virtualization through VMware’s guest driver. SVGA processes commands from a memory mapped command queue; the command queue functions as a transport layer for protocols between the guest graphics stack and the hypervisor.

SVGA uses the DirectX [22] API as its internal protocol, thereby realizing an API-remoting design. The transport layer and protocol are completely under the control of the hypervisor, enabling many of the benefits of API-remoting while ameliorating its downsides. However, using the DirectX API as a transport protocol requires that the driver and hypervisor translate guest interactions into DirectX whether they are natively expressed in DirectX or not. Coupling the transport layer with a particular version of the DirectX protocol has led to serious complexity and compatibility challenges: supporting each new version of the API takes many person-years (VMware introduced support for DirectX 10 (introduced in 2006) in 2015!). SVGA also supports a virtual GPU ISA called TGSI [67]. TGSI maps naturally to the graphics features of the ISAs it was originally designed to encapsulate, but has failed to keep up with GPU ISAs that have evolved to support general purpose computation primitives.

**C. Mesa3D OpenGL Support**

The Mesa3D Graphics Library [11] is an open-source graphics framework that implements graphics runtime libraries (e.g., OpenGL [56], Vulkan [41], Direct3D [22], and OpenCL [60]) on most GNU/Linux installations. It also includes official device drivers, written in a common framework, Gallium3D [10], for Intel and AMD GPUs. Support for NVIDIA GPUs is provided via reverse-engineered open-source Nouveau driver. Gallium3D imposes TGSI as the common virtual ISA for compute shaders, and decomposes drivers into two components: state trackers, which keep track of the device state, and pipe drivers, which provide an interface for controlling the GPU’s pipelines.

OpenCL support was first introduced in Mesa3D 9.0 with the release of the Clover state tracker. It was envisioned that Clover would leverage the LLVM [46] compiler to lower the OpenCL source to TGSI. Despite much effort by the open-source community [2, 5], an LLVM TGSI back-end has remained incomplete. Clover currently supports an incomplete set of OpenCL 1.1 APIs on AMD GPUs and fails to operate correctly on NVIDIA GPUs.

**D. GPU ISAs and IRs**

GPU front-end compilers produce code in virtual ISAs (NVIDIA PTX and LLVM IR for AMD) which are subsequently finalized using JIT compilers in the GPU driver to the native ISA (SASS and GCN). The vISA remains stable across generations to preserve compatibility, while the physical ISA is free to evolve. TGSI, the virtual ISA used in both the Mesa stack and SVGA, plays a similar role — enabling interoperability between graphics frameworks and GPUs from different vendors. An improved virtual ISA, SPIR-V, has been proposed as a new standard [41] and an effort is under way to replace TGSI with SPIR-V in the Mesa3D stack.

LLVM has become the de-facto standard for building compilers: both NVIDIA and AMD use it to implement their virtual ISA compilers, as do all the compilers in the Mesa stack including the TGSI compiler we implemented. LLVM IR is in a unique position to become a standard IR.
III. Design

TRILLIUM exports an abstract virtual device and a paravirtual guest driver, which we use to interpose and forward the OpenCL and CUDA APIs to the host. Unlike SVGA, which requires translation layers to ensure that all graphics frameworks APIs can be mapped to the SVGA protocol, TRILLIUM forwards the lowest layer in the GNU/Linux Graphics stack: the pipe-driver, effectively remoting OpenCL/CUDA API calls in the guest to the OpenCL/CUDA library in the host.

Our experience implementing the required TGSi vISA support in the Mesa graphics stack led us to believe that the TGSi layer is unnecessary. Not only does this translation introduce additional complexity in the guest stack, it also hurts performance, as we demonstrate in Section VI. The guest OpenCL compiler cannot target the native GPU architecture, and semantic information is lost to the host compiler. Further, while incorporating a TGSi compiler is possible in open frameworks like OpenCL, the task is significantly more daunting for closed frameworks like CUDA. Attempts to translate between TGSi and NVIDIA SASS in the reverse-engineered Nouveau driver understandably results in code that is significantly less performant than that produced by the proprietary stack.

TRILLIUM takes a different approach: TRILLIUM forwards API calls for compiling OpenCL code to the hypervisor. The OpenCL compiler in the host OpenCL framework (optimized for the physical hardware by the hardware vendor) is invoked on the forwarded OpenCL code to lower it directly to the physical device ISA.

Figure 2a shows the TRILLIUM design layers in a generic hypervisor stack. The OpenCL API is forwarded from the driver similar to the SVGA model. The OpenCL compute kernel (to be run on the GPU), can be passed through to the host via hypercalls in the driver, without being translated to any vISA, where it will be translated and optimized for the physical GPU in a virtual appliance (Dom 2 in Figure 2a).

TRILLIUM does not currently guarantee performance isolation and relies on the hardware scheduler. Performance isolation can easily be implemented via a rate-limiting API scheduler in the hypervisor, such as in GPUvm [62].

IV. Implementation

We evaluated the TRILLIUM design against a representative of each traditional virtualization technique: full-virtualization, user-space API-remoting and SVGA. Due to the difficulty of implementing a trap-based virtualization scheme, we chose to evaluate against GPUvm [62], the only existing open-source implementation. GPUvm is tightly coupled with the Xen hypervisor [19]. As a result, all the other prototypes were built on the Xen hypervisor to keep the platform common for fair comparison.

A. Trillium

We initially implemented TRILLIUM on Xen following the SVGA design, by implementing OpenCL support in a virtual device and extending the Mesa stack with TGSi support (see Section IV-A for details). The generated TGSi is sent to the host via RPC, and then finalized to a binary that can be run on the physical NVIDIA GPU using the open source Nouveau driver. Upon empirically finding that TGSi is a performance bottleneck, we revisited the basic design. We preserve the original prototype, hereafter called XEN-SVGA, as a baseline representative of the original SVGA design: this design is shown in Figure 2b. The current TRILLIUM design is shown in Figure 2c.

XEN-SVGA and TRILLIUM, implement API-forwarding in a custom pipe-driver in Gallium3D, that we call shadow-pipe. We chose to forward the pipe-driver as it presents a narrow interposition interface in the graphics driver. However, given that each OpenCL API call is decomposed into many different pipe-driver calls, other APIs higher up in the graphics stack may be better suited for interposition. The shadow-pipe is in the application domain’s graphics stack, and shims the pipe-driver interface as RPC calls to the actual Nouveau pipe-driver in the privileged domain.

XEN-SVGA manages user-level contexts, command queues and memory objects; and translates the input OpenCL GPGPU kernel to TGSi in the application domain. TRILLIUM skips the compilation phase in the application domain. The OpenCL kernel is forwarded to the privileged domain via RPC, where it is parsed and compiled by the LLVM NVPTX back-end in parallel. This binary is then loaded onto the GPU when the pipe-driver hits the binary loading phase. TRILLIUM can also emit LLVM IR if an OpenCL compiler is not available in the host.

Our implementation relies on gRPC as a transport mechanism between the guest and the host, as an implementation convenience. As zero-copy transfer [24, 63] and hypercall [52] mechanisms are well-studied, and a production-ready version of TRILLIUM would rely on these mechanisms, we measure and remove transport overhead from our reported measurements in Section VI. The overheads stem from remoting calls to the privileged domain over the network, which is especially significant since a single OpenCL API call may be decomposed into many pipe-driver APIs, and from the large amount of kernel input data that must be copied between VMs.

LLVM TGSi Back-end The Mesa3D stack implements OpenCL support via a state-tracker called Clover. Clover provides the library for the OpenCL application to link against, while most of the compilation is handled by invoking the OpenCL and C++ front-ends of the LLVM [46] compiler framework. Clover provides much of the front-end infrastructure required to support GPGPU computing in XEN-SVGA and TRILLIUM.

Historically, lack of a working TGSi back-end in LLVM, despite several attempts at building one in the past 5 years [2, 5],
has left OpenCL support for NVIDIA GPUs and SVGA in Mesa3D incomplete. In order to support OpenCL in XEN-SVGA, we implemented an LLVM TGSI back-end. While the TGSI back-end is not yet mature, we have added support for a majority of the 32-bit integer and floating point operations, intrinsics, memory barriers, and control flow. Using this backend we are able to compile and run 10 out of the 12 Rodinia benchmarks [23] used to benchmark GPUvm. Because the compiler is built using the LLVM framework, it enjoys all of the IR-level optimizations in LLVM.

LLVM IR handles control flow by using conditional and unconditional branches to and from Basic Blocks. A majority of the usual optimizations (constant propagation, loop unrolling, etc) are applied on the IR. On the other hand, TGSI assumes a linear control flow through the program, using higher level constructs such as IF-THEN-ELSE, FOR and WHILE loops. To accommodate this difference in control flow techniques, we leveraged a similar implementation in the AMDGPU back-end which calculates a Strongly-Connected-Components (SCC) graph from the Basic Block-based control flow in the LLVM IR, and then duplicates Basic Blocks as necessary. It is a testament to the maturity and flexibility of LLVM that the infrastructure to produce an SCC, and an example of how to use it to raise the control flow abstraction level were readily available.

B. GPUvm

GPUvm [62] is an open-source trap-based interposition design (a simplified block-diagram representation is shown in Figure 3a). The application domain (Dom 1) is presented with a GPU Device Model, which is emulated in the privileged domain (Dom 0). The emulation layer in Dom 0 interposes, validates, and fulfills all attempts to access the GPU. GPUvm has not been maintained: The last release, in 2012, is based on Xen 4.2.0 and runs on Fedora 16 [73]. In order to compare all prototypes on the same modern platform, we ported GPUvm to Ubuntu 16.04 with Xen 4.8.2.

C. User-space API remoting over RPC

In order to faithfully mimic user-level API-remoting-over-RPC systems [30, 42, 21], OpenCL API calls are trapped by a user-space shim library and forwarded via RPC from one appliance VM, which is the OpenCL “client”, to another appliance VM, which acts as the OpenCL “server”. Figure 3b shows the setup of the two API-remoting schemes we considered: API-REMOTE-GPU and API-REMOTE-CPU. The black arrows indicate the workflow of API-REMOTE-GPU, where the OpenCL server runs the OpenCL commands on a physical GPU using the NVIDIA OpenCL framework. The grey arrows show the API-REMOTE-CPU setup, where the OpenCL commands are executed on a multi-core CPU (Intel CPU Xeon E5-2643) using the Intel OpenCL SRB 5.0 framework. RPC is implemented using gRPC 1.6 (based on Google ProtocolBuffers 3.4.0) and inter-service communications are implemented over XML-RPC 1.39. Lower-overhead data-movement techniques, such as zero-copy, can be applied when both the client and the server are on a local machine.

D. Optimizations

TRILLIUM interposes at the pipe-driver API yielding fine-grained interposition, and therefore finer-grained multiplexing of the GPGPU. However, interposing at this layer also results in significant transport overhead. Many pipe-driver functions are responsible for context management and information retrieval—operations that do not result in interaction with the GPU. We reduce communication overhead by batching these types of API-calls, taking care to fall back to synchronous API-forwarding when any pipe-driver API calls that interact with the physical GPU are invoked.
We optimize the API-REMOTE-GPU and API-REMOTE-CPU systems by preinitializing the device and preallocating contexts and command queues on the privileged domain. These contexts are assigned to applications as they execute context creation APIs and are reclaimed asynchronously.

V. METHODOLOGY

All experiments were run on a Dell Precision 3620 workstation with NVIDIA Quadro 6000 GPU and Intel Xeon CPU E5-2643 (3.40GHz) CPU. We implemented or ported all prototypes and benchmarks on Ubuntu 16.04 with Xen 4.8.2. VMs were hardware-accelerated via Xen Hardware Virtual Machines (HVM) with 2 virtual CPUs (pinned) and 4 GB memory.

Of the GPU hardware available to us, the NVIDIA Quadro 6000 GPU was the only one that GPUvm, the full-virtualization baseline ran on. GPUvm depends on GDev [40] an open source CUDA runtime (released in 2012) implemented using Nouveau [9] GPU drivers, and the CUDA 4.2 compiler on Linux Kernel 3.6.5. GDev has not been maintained since 2014, and the effort to update it was too onerous. Experiments to control for hardware versions are reported in V-B.

A. Benchmarks

XEN-SVGA depends on the TGSI back-end compiler that we implemented to leverage the Clover OpenCL runtime in Mesa3D. API-REMOTE-GPU and API-REMOTE-CPU leverage the NVIDIA and Intel OpenCL library respectively and support all of the Rodinia benchmarks. GPUvm is built on top of the GDev CUDA runtime. Care was taken to ensure that the CUDA and OpenCL versions of the benchmarks use the same parameters, datasets, memory barriers, sync points, etc. Experiments to control for the programming framework are reported in V-B.

We optimize the API-REMOTE-GPU and API-REMOTE-CPU systems by preinitializing the device and preallocating contexts and command queues on the privileged domain. These contexts are assigned to applications as they execute context creation APIs and are reclaimed asynchronously.

V. METHODOLOGY

All experiments were run on a Dell Precision 3620 workstation with NVIDIA Quadro 6000 GPU and Intel Xeon CPU E5-2643 (3.40GHz) CPU. We implemented or ported all prototypes and benchmarks on Ubuntu 16.04 with Xen 4.8.2. VMs were hardware-accelerated via Xen Hardware Virtual Machines (HVM) with 2 virtual CPUs (pinned) and 4 GB memory.

Of the GPU hardware available to us, the NVIDIA Quadro 6000 GPU was the only one that GPUvm, the full-virtualization baseline ran on. GPUvm depends on GDev [40], an open source CUDA runtime (released in 2012) implemented using Nouveau [9] GPU drivers, and the CUDA 4.2 compiler on Linux Kernel 3.6.5. GDev has not been maintained since 2014, and the effort to update it was too onerous. Experiments to control for hardware versions are reported in V-B.

TABLE II: EVALUATION BENCHMARKS IN THREE CATEGORIES

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>backprop</td>
<td>Back propagation (pattern recognition)</td>
<td>R</td>
</tr>
<tr>
<td>gaussian</td>
<td>256x256 matrix Gaussian elimination</td>
<td>D</td>
</tr>
<tr>
<td>lud</td>
<td>256x256 matrix LU decomposition</td>
<td>M</td>
</tr>
<tr>
<td>nn</td>
<td>k-nearest neighbors classification</td>
<td>D</td>
</tr>
<tr>
<td>nw</td>
<td>Needleman-Wunsch (DNA-seq alignment)</td>
<td>M</td>
</tr>
<tr>
<td>pathfinder</td>
<td>Search shortest paths through 2-D maps</td>
<td>R</td>
</tr>
</tbody>
</table>

a. Interposition-dominant, interposition-rare, and moderate workloads.

The 10 Rodinia benchmarks that our TGSI compiler could compile were categorized based on frequency of interposition: Interposition-Dominant workloads run kernels hundreds or thousands of times requiring frequent interposition to set arguments, etc. Interposition-Rare workloads run a small number of long-running kernels, requiring very little interposition. Moderate-interposition workloads lie somewhere in between the other two. Two benchmarks were selected from each category to be used in the evaluation (the optimizations described in IV-D take significant manual effort).

B. Control Experiments

Software and platform version dependencies necessitated that our experimental environments vary slightly for the systems under evaluation — different front-end programming languages (CUDA vs. OpenCL), different runtime implementations (GDev CUDA vs. NVIDIA CUDA), or different drivers (Nouveau vs. NVIDIA). Resolving all of these differences would have taken monumental effort, but control experiments showed that these variables had negligible impact on our measurements.

OpenCL vs. CUDA GPUvm relies on the GDev implementation of the CUDA framework, while all the other designs rely on OpenCL. To assess the impact of different front-end languages on performance, we measured execution times for all benchmarks in both CUDA and OpenCL (Rodinia includes both implementations) holding all other variables constant, and found that the front-end language has near negligible impact, and the harmonic mean of differences in kernel execution time across all benchmarks is less than 1%; the worst (maximal) case is 15%. We also found negligible difference in performance between kernels compiled using CUDA 8.0 and the CUDA 4.2 required by GDev.

Hardware Generations. The performance improvements over the span of generations between the Quadro 6000 and modern cards is substantial. To estimate the effect of this variable we ran all benchmarks on both Quadro 6000 and a more recent GPU, Quadro P6000. While overall execution times are improved substantially, and the ratio of time spent on the host to time spent on the GPU changes as a result, the relative speedups are uniform across all benchmarks. This suggests that the trends that we observe on the Quadro 6000 still hold on newer hardware. We re-iterate that software dependencies of the GPUvm baseline prevent us from using more recent hardware. Our evaluation is performed on the newest (several generations older) GPU hardware that all our systems can run on.

VI. EVALUATION

We are interested in understanding the impact of a vISA on end-to-end performance, the effect of interposition frequency on performance, and the effectiveness of our proposed design, TRILLIUM.

A. The impact of vISA choice

Deferring the compilation of front-end code to the host not only eliminates redundant translations, and the need to have a compiler in the guest driver, but also ensures that the compiler has a high-fidelity view of the physical hardware. Typically, the execution/compilation framework is extremely tightly coupled with the vISA used, making the choice of vISA even more tenuous as it leads to the second order effect of having to rely on a particular implementation of the compute framework (e.g., Mesa3D OpenCL vs NVIDIA OpenCL).

To understand the impact of the virtual ISA on the quality of the generated GPU code we measured GPU execution time
We find that remoting calls intended to a CPU is uniformly more performant than full-virtualization of the GPU, and sometimes performs just as well as (backprop) or better than remoting to the GPU (1.6× faster for the bfs benchmark). The performance gain from accelerating the bfs kernel on the GPU is severely dwarfed by the cost of initialization on the GPU. GPGPU compute is only economical when it provides acceleration over the CPU; if overheads make the CPU competitive, the profitability threshold has been crossed. Further, the competitiveness of API-REMOTE-CPU suggests opportunity: systems could back a virtual GPU with CPU if they can detect when it is profitable to do so.

VII. CONCLUSION

TRILLIUM represents a local optima in the GPGPU virtualization space—by decoupling device virtualization from GPU ISA virtualization, it maintains the virtualization benefits of a para-virtual system, while exhibiting the performance of a user-space remoting system.
ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers, the SCEA group, and Donald E. Porter for their helpful feedback. This work was supported by NSF grants CNS-1618563 and CNS-1718491.

REFERENCES


