Toward Efficient and Realizable Hardware Virtualization

AMOGH AKSHINTALA

Abstract

The proposed dissertation will focus on developer effort and compatibility in software virtualization of CPU ISAs, and software virtualization of specialized compute devices (e.g., GPUs, TPUs) that are programmed through an API.

Although binary translation is a well-established software ISA virtualization technique, given the size and complexity of today's dominant ISAs, developers are routinely forced to adopt ad-hoc techniques to prioritize development effort. The proposed dissertation will present a principled approach to determine priority among different parts of the ISA. We believe this data will be useful to designers of virtual ISAs and **AA: what was the other thing?** as well.

Specialized compute accelerators, such as GPUs and TPUs, are usually controlled through a user-space API. The proposed dissertation will show that unlike with CPUs, where the ISA is the canonical interface provided to the programmer, ISA virtualization is untenable for specialized compute accelerators. Further, the proposed dissertation will present a present a novel taxonomy, *IEMTS* for cleanly understanding the design space for virtualizing compute accelerators. Based on insights from this taxonomy, the proposed dissertation will present a novel virtualization technique, *hypervisor-mediated API-remoting*, that is at once realizable and performant.

1 Introduction

Virtualization has a long and tumultuous history. Virtual memory was first described by German physicist Fritz-Rudolf Güntsch in his doctoral dissertation in 1956 [39] and commercialized [45] in the Cambridge University/Ferranti Inc. Atlas computer. Virtually all computers since then support virtual memory, with most providing hardware units—*Memory Management Unit (MMU)*—to accelerate the virtualization of memory.

Hardware virtualization [9]—the idea of virtualizing the entire computer to enable the simultaneous execution of multiple Operating Systems (OS)—was invented in 1962, and commercialized as the IBM VM-370 [28] hypervisor for the IBM 370 computer. Virtualization was briefly forgotten through the 1980s and 1990s, as the mainframe computer became all but obsolete during the Personal Computer (PC) revolution. Intel's x86 Instruction Set Architecture (ISA), which came to dominate the PCs that transplanted the mainframes, was not designed to be traditionally virtualizable [53], and was widely considered unvirtualizable **AA: track down Pat Gelsinger note.**. Multiple vendors introduced *software emulation* based solutions to enable the execution of one OS on top of another (e.g., Insignia SoftPC, Connectix VirtualPC, VMware Workstation). Over time, better techniques have been devised to the problem of virtualizing the x86 ISA, including ISA extensions (e.g., AMD-V and Intel VT-x) to enable the execution of virtualized applications at native speed, only trapping to the hypervisor when the application attempts to perform sensitive operations.

Several other forms of virtualization have also been considered. Sun Microsystems popularized *application virtualization* in the 1990s with the Java programming language: applications are written to an abstract machine—the *Java Virtual Machine (JVM)*—which is backed by a runtime system that ensures the program can execute on any platform. This scheme eschews *compatibility*—the ability to execute unmodified legacy applications—for *portability*. *Operating system-level virtualization* (e.g., Library OSes, Containers) virtualizes yet another layer in the software stack: the operating system's interfaces (e.g., system calls, kernel name-spaces). This style of virtualization preserves compatibility by transparently modifying the interfaces the application uses to access system resources, and results in low overhead execution.

The proposed dissertation is primarily concerned with hardware virtualization. Hardware virtualization is vital to high utilization of available physical resources in large computing installations, e.g., hardware virtualization is foundational to *cloud computing*. There have been many attempts to define hardware virtualization, from Popek and Goldberg's classical virtualization properties—*equivalence, performance, and safety* to Bugnion, Nieh and Tsafrir's [26] definition of virtualization—*"the application of the layering principle with enforced modularity such that the exposed resource is identical to the underlying resource"*. While technically correct, all of these defintions are too contrite to be useful. Instead, for the purposes of this dissertation, we concern ourselves with the following overarching goal—*realizable, fair, isolated, and efficient sharing of hardware resources among mutually distrustful entities.*

Hardware virtualization typically involves mediating access to the shared resource either by exposing an interface that is identical to that of the physical resource (*full-virtualization*), or by exposing an alternative interface, operations on which are in-turn synthesized to the native interface (*para-virtualization*). The exposed interface is *virtual*, in that it is not directly exposed by the physical underlying hardware, and instead is entirely under the control of supervisory virtualization software, the *hypervisor* (also known as the *Virtual Machine Monitor*). While operations in the resulting *virtual machine* may be directly executed on the physical hardware for performance reasons, as in the case of hardware-assisted virtualization schemes like AMD-V and Intel VT-x, all privileged operations still trap to the hypervisor. The interface interposed may be a hardware interface (ISA, Memory, I/O Protocols, etc.) or a software interface (Syscalls, APIs, etc.).

1.1 CPU virtualization

Four decades of attention from both the academic community and industry has given rise to a large body of techniques that enable efficient virtualization of CPUs: software techniques such as binary translation and device emulation, are well established. While dominant ISAs, such as x86 and ARM, even provide extensions to enable low-overhead virtualization, binary translation results in lower overhead for sequences of sensitive instructions that need to be emulated [12].

When implementing a new binary translator [33] or developing a secure virtual instruction set, **AA:**

what was 3rd?, the developer is left to their own devices to answer questions regarding *realizability*, e.g., to prioritize different parts of the ISA during development or to understand the relative value of different parts of the ISA to backwards compatibility. Predictably, developers have typically adopt ad-hoc methodologies to overcome this challenge [25]. We hypothesize that a principled approach to answering these questions lies in understanding the distribution of importance, to users, in the ISA being virtualized. Chapter 1 will present a methodology for determining user preference, and the resulting dataset. Briefly, we estimate the importance of an instruction in the ISA by measuring its frequency of occurrence in applications, and then weighting the frequency data with the likelihood of users installing those applications. This is completed work [13].

1.2 Accelerator virtualization

Compute heavy and data parallel workloads such as graph processing and machine learning have precipitated a Cambrian explosion of specialized processors. These emerging compute devices (e.g., GPGPUs, TPUs, IPUs, IO accelerators), however, pose a challenge to virtualization developers, who once again find themselves balancing the essential characteristics of a virtualization scheme—compatibility, interposition, sharing, isolation—with the need to preserve the raw performance these processors provide. Virtualization techniques developed for CPUs (ISA virtualization) are not applicable to these specialized accelerators: their control interfaces are closer to those of I/O devices than the ISAs of CPUs. Techniques developed for I/O devices, such as NICs, are also untenable for specialized compute devices as they result in the sacrifice of one or more of the essential characteristics listed above. Full-virtualization based schemes, such as GPUvm [60], suffer from massive overheads that essentially negate the speedup that makes the specialized compute unit attractive in the first place. Para-virtual systems, such as SVGA [30] that interpose on low-level interfaces, such as the kernel driver, introduce much lower overhead than full-virtualization based schemes but have poor compatibility, i.e., the introduction of an artificial abstract interface constructed expressly for the purpose of interposition necessitates massive engineering effort to support new hardware in the host and new software frameworks in the guest. User-space API-remoting solutions [68, 32, 54] interpose on the user-space API in the guest and forward the interposed operation to the host as an RPC. This approach introduces very low overhead and can evolve with the hardware easily, but has traditionally eschewed hypervisor interposition, thereby making it difficult to enforce safety and isolation among guests.

Virtualizing a Graphics Processing Unit (GPU) for the purposes of graphics rendering is a well studied problem, with existing commercial solutions, e.g., VMware's SVGA [31]. Over the last decade, GPUs have been re-purposed for parallel general purpose compute (commonly known as GPGPU). Chapter 2 will present our findings from attempting to extend the SVGA model of GPU virtualization to cover GPGPU virtualization as well. We find that the tight coupling between ISA virtualization and device virtualization in SVGA leads to poor performance for GPGPU compute. We propose a new virtualization scheme, Trillium, that doesn't rely on ISA virtualization and show that Trillium outperforms all other traditional virtualization schemes while retaining hypervisor interposition. Material presented in Chapter 2 will be drawn from a published paper [14].

Specialized compute units (e.g., Google TPU, Intel QAT, etc.) are typically exposed to developers via a user-space API. The API is typically implemented by a combination of proprietary software that interacts with the hardware through opaque interfaces. Chapters 3, 4 and 5 will explore the performance implications of virtualizing the user-space API for specialized compute accelerators. Chapter 3 will present an overview of AvA, a framework that enables automated virtualization of accelerator APIs. Chapter 4 will focus on the performance implications of API-remoting based virtualization of a single specialized accelerator. Chapters 3 and 4 will draw on material that appeared in a HotOS workshop paper [73] and a full paper that is currently under submission. Chapter 5 (proposed work) will explore performance issues that arise when an application uses multiple API-remoted virtual accelerators in a pipelined fashion.

Virtualization schemes are traditionally taxonomized according to the core techniques employed (e.g. emulation, full- or para-virtualization, API remoting, etc.), and evaluated in a property trade-off space comprising performance, compatibility, interposition, and isolation. We argue that both the de facto taxonomy and the property trade-off space are illustrative but not informative for GPGPU virtualization: there is a large body of research that has had little influence on practice. We suggest an alternative framework called `IEMTS` that teases apart design axes that are implicitly and unnecessarily intertwined in much of the literature. By focusing on the **I**nterface interposed, the interposition **E**ndpoints, the **M**echanism of interposition, the **T**ransport used to move the interposed operations between the guest and the host, and the mechanism used to **S**ynthesize the interposed interface, `IEMTS` enables a clearer understanding of trade-offs in prior designs and provides a model for comparison of alternative designs. `IEMTS` will be presented in Chapter 6, along with analysis of traditional virtualization techniques in the context of GPGPUs.

Concretely, the proposed dissertation will evaluate the following hypotheses:

H 1: Priority among instructions in an ISA, in the context of binary translation, can be automatically inferred from user preferences. (Chapter 1)

H 2: ISA virtualization is untenable for performant virtualization of compute accelerators. (Chapter 2)

H 3: Hypervisor-mediated API-remoting is a low-overhead virtualization scheme for API-controlled compute accelerators. (Chapters 3, 4, and 5)

H 4: The characteristics of a virtualization technique can be succinctly described by a scheme that explicitly captures the *Interface* interposed, the *Endpoints* interposed on, the *Mechanism* of interposition, the *Transport* used to connect the interposed endpoints, and the mechanism used to *Synthesize* the interposed operation on the host. (Chapter 6)

2 CPU virtualization

In order to understand CPU Virtualization as it is today, it is illuminating to consider the history of the technique, even though a complete treatment of this subject is out of the scope of this proposal (interested readers are instead referred to Bugnion, Nieh, and Tsafrir's book on this topic—*Hardware and software support for virtualization* [26].)

CPU virtualization as a technique was first considered for the IBM 360 in 1970 [49]. The idea then, as it is today is, was to provide each user with the illusion of having a dedicated machine to themselves, by simultaneously running multiple operating systems on the same machine. The IBM 370 was specifically designed to be *virtualizable* [28], while a concurrent machine, the PDP-10, wasn't. The inability to virtualize the PDP-10 led Popek and Goldberg [53] to formalize the requirements of a Virtual Machine Monitor—*equivalence, safety, and performance*—and three theorems about the virtualizability of an Instruction Set Architecture (ISA). To summarize briefly, a VMM or a hypervisor must meet the following criteria: the virtual machine constructed by the VMM must be indistinguishable from the native machine (equivalence), must not be able to access resources not allocated to it or influence the way these resources are used by other VMs or the hypervisor (safety), and the performance of software executing in the VM must be comparable to native (performance). The first theorem defined what it means for an ISA to be virtualizable. The second theorem was concerned with recursive or nested virtualization. The third theorem presents the necessary conditions for a *Hybrid Virtual Machine*—one that combines direct execution of non-sensitive instructions with emulation for all sensitive instructions—-to be designed for ISAs that violate the first theorem.

While virtualization was well understood and in production in the 1970s, all the lessons learned during that time were seemingly lost or ignored by later computer architects. The new ISAs that established their dominance in computing as the mainframe computers of the 1970s were rendered mostly obsolete by the Personal Computer (Intel x86, DEC ALPHA, SUN SPARC, IBM POWER, MIPS, ARM, etc.) were all unvirtualizable. Even though some of them initially supported virtualizable modes, (e.g., Intel x86's 16-bit virtual mode— `v8086`—allowed the execution of 16-bit software in 32-bit mode.), this support was left on the wayside as the ISA evolved. Virtualization was considered a quirky bad idea from the 1970s.

Hardware virtualization made a come back in the late 1990s and early 2000s [24, 71, 20] as researchers noticed that virtualization was a promising alternative approach to the problem of multi-core scalability. Despite the non-virtualizable nature of the dominant Intel x86 ISA, researchers realized that they could leverage Popek and Goldberg's third theorem to build a Hybrid Virtual Machine using techniques like *binary translation*—the hypervisor inspects the executing binary and replaces any sensitive instructions with one or more non-sensitive instructions that provide equivalent behavior—and shadow paging.

2.1 Determining priority among instructions for Binary Translation

CPU vendors added support for trap-and-emulate based virtualization as extensions to their CPUs in the mid 2000s (e.g., Intel VT-x, AMD-V). These extensions introduced a new execution mode with support for nested paging, and support for automatically trapping to the hypervisor when the processor attempts to execute a sensitive instruction. This newly introduced hardware support for virtualization, was a mixed blessing for the virtualization software.

Researchers at VMware found that a combination of hardware virtualization and binary translation was essential to achieve optimal performance for the application [12]. On the other hand, naive application of the hardware support of virtualization led to worse performance than when executing the application a software-emulated virtual machine [10].

Binary translation of a gargantuan ISA like Intel x86-64, which has ~3800 instructions, requires prioritizing development effort on the most "important" instructions—trading completeness for simplicity and a quicker development cycle. For instance, the authors of VMware workstation describe an "on-demand implementation" process, where the x86 binary translator focused on just the instructions needed for a target OS; the entire ISA was never supported, and guest OSes such as OS/2 did not work [25]. Similarly, Amit et al. [18] showed that KVM cannot correctly implement certain obscure x86 behaviors in a guest OS. Prioritizing instruction support is a natural and ubiquitous engineering trade-off. Some instructions appear in program binaries more frequently than others, e.g., the `MOV` instruction (used to move data) is the most common `x86-64` instruction. On the contrary, the `VFMADDSD` instruction, used to express a fused multiplication and addition operation, is relatively rare. Further, many instructions perform similar operations, albeit with subtle distinctions.

What, then, is the basis for assigning priority to instructions? Common approaches include analyzing benchmark suites [23, 40, 21], or execution traces collected in target environments [42]. The ad-hoc nature of this approach leaves many useful questions unanswered: Is the chosen test suite actually representative? What is the path of least effort to support a new ISA in a software tool? What minimum set of instructions must be implemented to run at least one application? What instruction sub-set is sufficient to run the majority of deployed applications?

To paraphrase Hennessy and Patterson [51], the best thing to measure is what actually runs on the user's system. This chapter will present and analyze a dataset collected from static analysis of all x86-64 ELF binaries in the Ubuntu 16.04 GNU/Linux distribution. We leverage package installation frequency, an approximation of a package's importance to users, from Ubuntu and Debian popularity contest data [62, 52], to infer the relative importance of an instruction from the percentage of binaries on a given system that contain that instruction. We adapt metrics from a prior study of OS API compatibility [65], specifically, *instruction importance* — the relative importance of a given instruction, and *weighted completeness* — the completeness of a system that implements a subset of the ISA.

This chapter will present:

- An instruction occurrence dataset gathered using static analysis of 9,337 open-source applications in the Ubuntu 16.04 repositories.
- Evaluation of conventional wisdom about ISA usage.
- An iterative plan for developing new tools that use the `x86-64` ISA.
- Empirical validation of standard benchmarks.
- An instruction occurrence data visualization tool, and the analysis framework used in this study are available at `http://x86instructionpop.com/`.

This chapter will be drawn from joint work [13] with Bhushan Jain, Chia-che Tsai, Michael Ferdman and Donald E. Porter.**AA: Add some results here from this work.**

3 ISA virtualization is untenable for GPUs

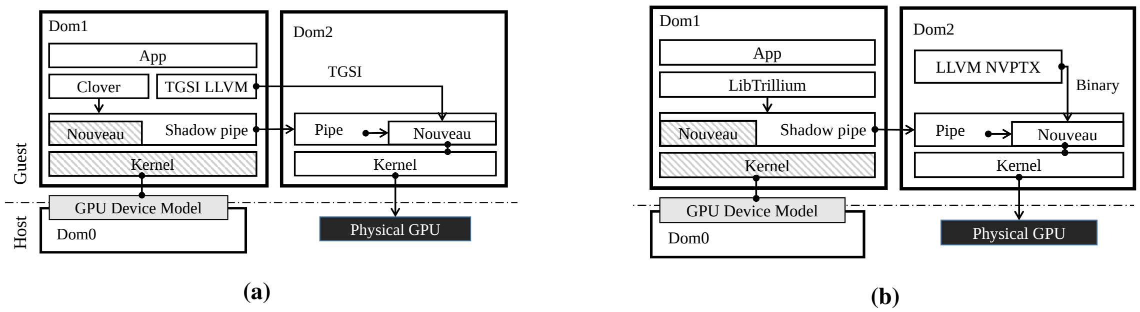

Figure 1: *Xen-SVGA and Trillium designs. (a) The Trillium stack. (b) Xen-SVGA approximates the SVGA model extended to support GPU Compute. (c) The design of Trillium with shadow pipe.*

In many parallel computing domains, compute density and programmability [5, 59, 37] have made GPUs the clear choice for efficiency and performance [3]. Popular machine learning frameworks such as Caffe [44], Tensorflow [8], CNTK [72], and Torch7 [27] 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 [7].

Despite much prior research [69, 41, 11, 67] 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, 63], 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 [22, 46, 54, 68, 32] 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 [56].

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 [31] design in Xen (shown in Figure 1a), 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. Instead, we found that it harms performance, as shown in Figure 2, 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. Figure 1b visually presents the Trillium design.

Trillium is an existence proof of a viable alternative design—hypervisor-mediated API-remoting—

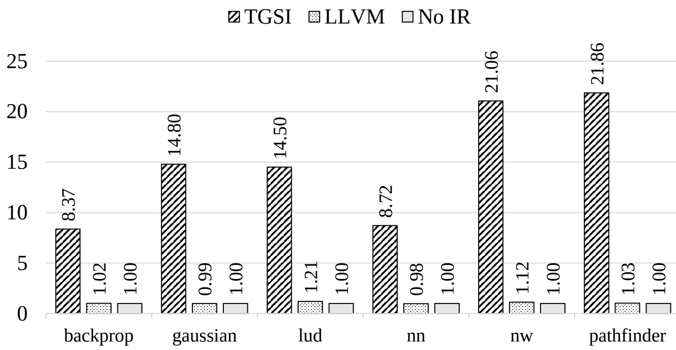

Figure 2: *Kernel execution slowdown due to virtual ISAs.* ***TGSI****: the LLVM TGSI back-end compiler used in Xen-SVGA.* ***LLVM****: LLVM NVPTX back-end used in Trillium.* ***No IR****: native NVIDIA compiler.*

that preserves desirable virtualization properties such as consolidation, hypervisor interposition, isolation, encapsulation, etc., without requiring full hardware virtualization. While Trillium outperforms GPUvm [60], a full virtualization system, by up to 14× (5.5× on average) and the para-virtual SVGA-like design by as much as 7.3× (5.4× on average), it performs worse than a userspace API-remoting framework on average. We believe this is because of a poor choice of API to forward: Trillium forwards the nuoveau kernel graphics driver API, which is low enough in the stack that each each userspace API function is broken into multiple RPC calls. A better approach would be to forward the userspace API itself (presented in the next chapter).

Concretely, this chapter will show that ISA virtualization is harmful for GPU virtualization, and will lay the groundwork for a new hypothesis—Hypervisor-mediated API-remoting (of the userspace programming framework API) is a realizable, performant, safe and composable virtualization scheme for API-controlled accelerators.

The proposed chapter will draw material from joint work [14] with Hangchen Yu, Arthur M. Peters, and Christopher J. Rossbach.

4 Hypervisor-mediated API-remoting

Practical virtualization must support sharing and isolation under flexible policy with minimal overhead. The structure of current accelerator stacks makes this extremely difficult to achieve. Accelerator stacks are *silos* (Figure 3) comprising proprietary layers communicating through memory mapped interfaces. This opaque organization makes it *impossible* to interpose intermediate layers cleanly to form a virtualization boundary. Practically interposable alternatives leave designers with a Hobson's choice between critical virtualization properties such as interposition and compatibility.

We present AvA, a system that addresses the fundamental limitations of existing accelerator virtualization techniques. AvA combines API-agnostic para-virtual I/O stack components with a Domain-Specific Language (DSL) and toolchain to automate construction and deployment of guest

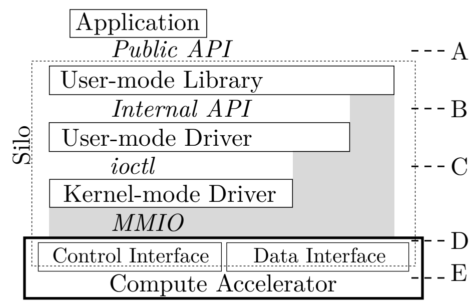

Figure 3: *An accelerator silo. The public API and the interfaces with striped backgrounds are interposition candidates. All interfaces with backgrounds are proprietary and subject to change.*

libraries and API servers. AvA uses an abstract para-virtual device to serve as a transport endpoint for forwarding the public APIs of vendor-provided frameworks (e.g. CUDA or TensorFlow). Unlike currently popular user-space API remoting solutions [1, 43, 68, 32, 55], AvA preserves hypervisor-level resource management and strong isolation using a novel technique called *Hypervisor Interposed Remote Acceleration*. AvA forwards API calls over hypervisor-managed communication channels, inserting automatically-generated resource management components between traditional front- and back-ends to enforce policies described in the DSL specification. Critically, *automation* from AvA enables hypervisors to keep up with fast accelerator evolution: automatic generation of components minimizes engineering effort.

AvA supports a broad range of currently-shipping compute accelerators: We virtualized ten accelerators including NVIDIA and AMD GPUs, Google TPUs, and Intel QuickAssist. Virtualizing an API framework using AvA requires modest developer effort: a single developer virtualized OpenCL in a handful of days, a stark contrast to the person-years of developer effort for VMware's SVGA II or Bitfusion's FlexDirect [1]. Experiments show that AvA provides near-native performance (e.g., 2.4% slowdown for TensorFlow and 5.6% for CUDA), enforces isolation and fair sharing across guests, and supports live migration.

The proposed chapter will make the following arguments:

- The chapter demonstrates feasibility of automatically constructed virtual accelerator support, showing that a single technique can deal with many architectures, APIs, versions, and policies.
- We introduce Hypervisor Interposed Remote Acceleration (HIRA) to enable hypervisor-enforced isolation and sharing policies unachievable with current SR-IOV and API remoting systems.
- We utilize a novel DSL, LAPIS, for describing API functions, resources, and policies to enable automatic construction of virtual stacks from native header files.
- Our evaluation shows low developer effort, strong isolation, and good performance.

This chapter will draw from joint work with Hangchen Yu, Arthur Peters and Christopher J Rossbach. Part of this work was published as a workshop paper [73], and a longer paper is under submission. As with any big system building effort, it was a team effort. My contributions to the project were **AA: what were they?**.

5 IEMTS — A new accelerator virtualization taxonomy

Traditionally, virtualization designs have been taxonomized according to the core techniques employed (e.g. emulation, full- or para-virtualization, API remoting, etc.), and evaluated in a property trade-off space comprising performance, compatibility, interposition, and isolation. *Isolation* ensures that mutually distrustful guests cannot access each other's data or harm each other's performance. *Compatibility*, characterizes how well a design preserves the freedom of hardware and software components to evolve independently: e.g. changes in the hypervisor should not force changes to guest software. Virtualization provides an indirection layer between logical and physical resources by *interposing* a well-defined interface. The quality of interposition determines the nature of benefits (e.g. extent of consolidation) afforded by a virtualized system [70].

Virtualization techniques are well explored, yielding conventional wisdom about their fundamental trade-offs. For example, *full virtualization* interposes the software-hardware interface to provide a virtual view of the underlying hardware. This enables guests to run unmodified OS and application binaries, yielding high compatibility. However, hardware interfaces for GPUs rely heavily on MMIO and communication through memory, which necessitates page-fault-based interposition [64, 4, 47, 50] techniques that cripple performance. *Para-virtual* designs export an abstract device to the guest, but require hypervisor-specific drivers and runtime libraries in the guest, trading compatibility for improved performance. *API remoting (or forwarding)* [38, 31, 35, 58] aggregates high-level API calls issued in VMs, running them on the host or in a dedicated appliance VM. This technique can provide near native performance because API calls are infrequent, but has poor compatibility because it requires changes in guest applications or libraries.

We argue that the current *de facto* taxonomy and property trade-off space are illustrative but not informative for GPUs: there is a large body of research that has had little influence on practice. First, Classifying virtualization designs as API-remoting vs. full vs. para-virtual captures important concepts, and emergent properties compactly, but doesn't explain their correlation to properties like performance. Second, virtualization properties such as compatibility, isolation, and interposition have highly context-dependent meaning and their relative value to system designers can be hard to quantify. Consider compatibility: there are many dimensions to compatibility (library, hardware, OS, etc.), and each of those are commonly achieved by separate technical, and non-technical means (e.g., TGSI is the common vISA for both the VMware and GNU/Linux graphics stacks; this is *not a lucky coincidence*).

We argue that practical design goals, such as providing a virtualization layer with specific characteristics, get obscured when these properties are considered as a set of constraints that must be preserved, without first refining for context. Further, production systems, such as VMware SVGA [31], compose multiple virtualization techniques in order to leverage the best properties of each technique, especially in the presence of multiple interfaces.

To enable a cleaner separation of concerns, we draw on the observation that *all* virtualization relies on encapsulation and interposition, and note that a design can be clearly understood by identifying:

- the **I**nterface that is interposed,

	GPUvm	VMware SVGA Control Interface	VMware SVGA GPU ABI	rCUDA
Interposed Interface	MMIO/BAR	DirectX APIs	Device ISA	Userspace API
Interposition Source	Trap handler	Guest driver/libs	Guest Driver	Guest Library
Interposition Destination	Host driver	Host framework	Host Driver	Host/Server Daemon
Interposition Mechanism	Trap	Guest library	Compilation to vISA	Guest Library Shim
Transport	Fault	Hypervisor FIFOs	Hypervisor FIFOs	RPC
Synthesis	Emulation	Call host API	Binary translation	Call Server API

Table 1: *Comparing virtualization designs using the* `IEMTS` *framework.*

- the **E**nd-points (source and destination) the interposed event is transported between,
- the **M**echanism used to interpose,
- the **T**ransport mechanism used to communicate between endpoints,
- the mechanisms used to **S**ynthesize or implement the desired functionality at the destination. We call this the **`IEMTS`** framework.

AA: Add discussion of the Table.

6 Proposed work — vTask

The previous chapters in the proposed dissertation showed that hypervisor-mediation API-remoting is the only effective mechanism for sharing API-controlled compute devices among mutually distrustful tenants, e.g., in a cloud computing environment. Virtualization vendors, such as VMware, have begun adopting API-remoting based solutions for accelerator virtualization **AA: cite BitFusion acquisition**.

API-remoting works by interposing on API calls invoked by the application in the guest OS, and executing them in a surrogate, the API-server, in the host. Typically, API-servers are associated with a single API framework (for modularity and failure isolation between APIs/accelerators, and in order to be able to use remote resources) and each API-server is a surrogate for a single guest (to preserve memory isolation between guests). Applications that use multiple accelerator API frameworks will be associated with multiple API-servers, one per framework.

Under a typical API-remoting system, applications that pipeline disparate accelerator frameworks are burdened with redundant data movement. All inter-accelerator data movement must take place in the guest application as that is where the accelerators are in the same logical address space. Figure 4 illustrates this scenario: when an *API-1* function is invoked, associated data is copied from the *guest application* to *API-server-1*, and then to *Device-1*'s memory. Once the function finishes executing on *Device-1*, the result is copied back to the *guest application*. When a function from *API-2* is invoked, the same data (i.e., the output of the *API-1 function*) is copied from the *guest application* to *API-server-2* and then to *Device-2* to be processed.

In order to eliminate redundant data movement when an application uses multiple accelerators via API-remoting, the hypervisor must track the data passed to these API calls. The hypervisor must

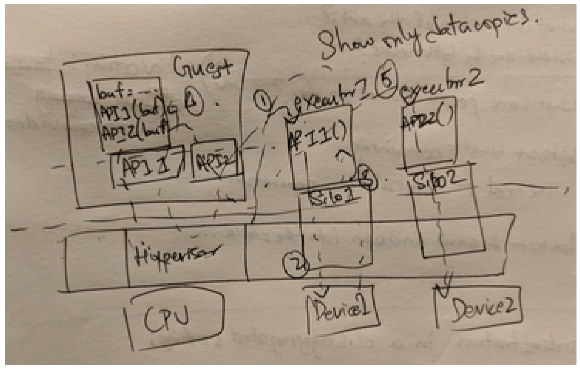

Figure 4: *Data processed by two API stacks must pass through the guest application*

keep track of where the data flowed from and to, the validity of different copies of the data (e.g., if the data is modified on the accelerator, but hasn't been copied back to the accelerator silo, or the guest application), and eliminate redundant data movement. As an example, if a guest application were to invoke the `cudaMempyDtoH()` function to copy data back from an Nvidia GPU, and then invoke the Intel QAT compression function `cpaDCCompressData2()` on the same data without modifying it in any way, the hypervisor should be able to detect this and elide the copying of data to and from the guest application. Further optimization may also be possible: peer-to-peer data copy between the devices if they are on the same machine, or by directly copying the data from the first API-server on one remote machine to the second API-server on another remote machine.

We propose to build vTask, an application-transparent data orchestration system that optimizes data movement among accelerators virtualized via API-remoting. vTask will leverage information from API annotations [73] to track data buffers across the guest application, the API-servers servicing API calls made by the guest, and the accelerator hardware. vTask will optimize data movement across these components while ensuring that a coherent view of the data buffer is presented to anyone attempting to read the data. Ideally, vTask will require no changes to the guest application or extra annotations of any kind from the application programmer. We hypothesize that annotations provided to virtualize the API (by the device or virtualization vendor) will be sufficient to infer the semantics of the data buffers managed.

We will prototype vTask in AvA, a state-of-the-art para-virtual API-remoting system for KVM. vTask will rely on device-side buffer allocation and deallocation API calls, and special annotations provided by LAPIS, AvA's API description language, to determine buffer lifetime. Further, vTask will implement a simple MESI-style coherence protocol to track spatial validity of data (i.e., to track where the latest data is present). vTask will leverage optimizations such as shared memory, Unified Virtual Memory, and PCIe Peer-to-Peer (P2P) data transfer where available, but does not make assumptions about their universal availability.

vTask can handle data movement between both local and remote devices. When API-remoting to a remote system, the devices used by the guest application may be present on separate machines. We hypothesize that vTask will be able to eliminate costly data transfers over the network by adhering to the principle of lazy loading wherever possible, i.e., data is not moved until a demand fault occurs.

AA: needed: a deeper explanation of design (breaking it down into the pieces that need to be built along with an estimation of how long each will take), a better evaluation strategy (with applications that we intend to run and how the machines will be setup, etc.)

7 Plan of Work

Task	Deadline
Dissertation proposal and oral exam	Nov. 2019
Implement vTask in AvA (submit to ATC'20)	15 Jan. 2020
Dissertation draft to committee	1 Mar. 2020
Dissertation defense	late Mar. 2020
Submit dissertation to graduate school	12 Apr. 2020

Table 2: *Proposed timeline.*

8 Related Work

Technique	System	lib unmod	OS unmod	lib-compat	hw-compat	sharing	isolation	migration	sched. policy	graphics	GPGPU	I/D	benchmark	slowdown[b]	native speedup	virtual speedup
Full-virtual	**GPUvm [61]**	✓		✓		✓	✓		XC, BAND		✓	*D*	Rodinia	141×	11.4×	0.08×[c]
	gVirt [63]	✓		✓		✓	✓	✓	QoS	✓		*I*	2D [6], 3D [2]	1.6×	N/A	N/A
PCIe Pass-thru	**AWS GPU [15]**	✓	✓							✓	✓	*D*	Any	1×		
API remoting	**GViM [38]**				✓	✓	✓		RR, XC		✓	*D*	CUDA 1.1 SDK	1.16×	22×	19×
	gVirtuS [34]				✓	✓	✓		RR		✓	*D*	CUDA 2.3 MM	3.1×	11.1×	3.6×
	vCUDA [58]		✓		✓		✓	✓	HW		✓	*D*	CUDA 4.0 SDK	1.91×	6×	3.1×
	vmCUDA [68]		✓		✓	✓	✓		HW		✓	*D*	CUDA 5.0 SDK	1.04×	33×	31.7×
Distributed API remoting	**rCUDA [32, 54]**		✓		✓	✓	✓		RR		✓	*D*	CUDA 3.1 SDK	1.83×	49.8×	27.2×
	GridCuda [48]		✓		✓	✓	✓		FIFO		✓	*D*	CUDA MM, SOR	1.23×		
	SnuCL [46]		✓			✓	✓				✓	*D*	SNU NPB [57]			
	VCL [19]		✓		✓	✓	✓				✓	*D*	Stencil2D [29]			
Para-virtual	**GPUvm [61]**					✓	✓		XC, BAND		✓	*D*	Rodinia	5.9×	11.4×	1.9×
	HSA-KVM [41]	✓				✓	✓		HW		✓	*I*	AMD OCL SDK	1.1×		
	LoGV [36]	✓		✓		✓	✓	✓	RR		✓	*D*	Rodinia	1.01×	11.4×	11.3×
	SVGA2 [31]	✓				✓	✓	✓		✓		*D*	2D, gaming	3.9×		
	Paradice [17]	✓		✓		✓	✓		HW, QoS	✓	✓	*D*	OpenGL, OpenCL	1.1×		
	VGVM [66]				✓	✓	✓		HW		✓	*D*	CUDA 5.0 SDK	1.02×	33×	32.3×

Table 3: *Existing GPU virtualization proposals, grouped by approach. Drawn from published work [14]*

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

References

[1] Bitfusion: The elastic AI infrastructure for multi-cloud. `https://bitfusion.io`. Accessed: 2019-04.

[2] Cairo-perf-trace. `http://www.cairographics.org`. Jan. 2018.

[3] GPU Applications Catalog. `https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/`. Jan. 2018.

[4] KVMGT - the implementation of intel gvt-g(full gpu virtualization) for KVM. `https://lwn.net/Articles/624516/`. 2014.

[5] NVIDIA CUDA 4.0. `http://developer.nvidia.com/cuda-toolkit-40`. 2011.

[6] Phoronix test suites. `http://phoronix-test-suite.com`. Jan. 2018.

[7] TOP500 Supercomputer Sites. `https://www.top500.org/lists/2018/11/`, 2019.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[9] R.J. Adair. *A Virtual Machine System for the 360/40*. IBM Cambridge Scientific Center report. International Business Machines Corporation, Cambridge Scientific Center, 1966.

[10] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. *SIGARCH Comput. Archit. News*, 34(5):2–13, October 2006.

[11] Neha Agarwal, David Nellans, Mike O'Connor, Stephen W Keckler, and Thomas F Wenisch. Unlocking bandwidth for GPUs in CC-NUMA systems. In *HPCA*, 2015.

[12] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 373–385, Boston, MA, 2012. USENIX.

[13] Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman, and Donald E Porter. x86-64 instruction usage among c/c++ applications. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 68–79. ACM, 2019.

[14] Amogh Akshintala, Hangchen Yu, Arthur Peters, and Christopher J Rossbach. Trillium: The code is the ir. In *The Second Special Session on Virtualization in High Performance Computing and Simulation (VIRT 2019), Dublin, Ireland*, 2019.

[15] Amazon. *Amazon Elastic Compute Cloud*, 2015.

[16] Inc or Its Affiliates Amazon Web Services. Amazon EC2 P3 Instances. `https://aws.amazon.com/ec2/instance-types/p3/`. Accessed: 2018-2-6.

[17] Ardalan Amiri Sani, Kevin Boos, Shaopu Qin, and Lin Zhong. I/o paravirtualization at the device file boundary. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 319–332, New York, NY, USA, 2014. ACM.

[18] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. Virtual cpu validation. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2015.

[19] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for opencl based heterogeneous computing on clusters with many gpu devices. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–7, Sept 2010.

[20] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[21] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[22] BitFusion Inc. Bitfusion FlexDirect Virtualization Technology White Paper. `http://bitfusion.io/wp-content/uploads/2017/11/bitfusion-flexdirect-virtualization.pdf`, 2019. Accessed: 2019-2-28.

[23] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 41–42, New York, NY, USA, 2018. ACM.

[24] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[25] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Transactions on Computer Systems (TOCS)*, 30(4):12, 2012.

[26] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture*, 12(1):1–206, 2017.

[27] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[28] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM J. Res. Dev.*, 25(5):483–490, September 1981.

[29] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[30] Micah Dowty and Jeremy Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *SIGOPS Oper. Syst. Rev.*, 43(3):73–82, July 2009.

[31] Micah Dowty and Jeremy Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[32] Jose Duato, Antonio J. Pena, Federico Silla, Juan C. Fernandez, Rafael Mayo, and Enrique S. Quintana-Orti. Enabling CUDA acceleration within virtual machines using rCUDA. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[33] Alex Fishman, Mike Rapoport, Evgeny Budilovsky, and Izik Eidus. HVX: Virtualizing the cloud. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, San Jose, CA, 2013. USENIX.

[34] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A gpgpu transparent virtualization component for high performance computing clouds. *Euro-Par 2010-Parallel Processing*, page 379–391, 2010.

[35] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In *European Conference on Parallel Processing*, pages 379–391. Springer, 2010.

[36] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa. Logv: Low-overhead gpgpu virtualization. In *High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCCEUC), 2013 IEEE 10th International Conference on*, pages 1721–1726, Nov 2013.

[37] Kate Gregory and Ade Miller. C++ amp: accelerated massive parallelism with microsoft visual c++. 2014.

[38] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24. ACM, 2009.

[39] Fritz-Rudolf Güntsch. *Logical Design of a Digital Computer with Multiple Asynchronous Rotating Drums and Automatic High Speed Memory Operation*. Doctoral dissertation, Technische Universität Berlin, 1956.

[40] John L Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[41] Yu-Ju Huang, Hsuan-Heng Wu, Yeh-Ching Chung, and Wei-Chung Hsu. Building a kvm-based hypervisor for a heterogeneous system architecture compliant system. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pages 3–15, New York, NY, USA, 2016. ACM.

[42] R. Jagtap, S. Diestelhorst, A. Hansson, M. Jung, and N. When. Exploring system performance using elastic traces: Fast, accurate and portable. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 96–105, July 2016.

[43] JAIN Jayant, Anirban Sengupta, Rick Lund, Raju Koganty, Xinhua Hong, and Mohan Parthasarathy. Configuring and operating a XaaS model in a datacenter, November 13 2018. US Patent App. 10/129,077.

[44] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[45] Tom Kilburn, David BG Edwards, Michael J Lanigan, and Frank H Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, (2):223–235, 1962.

[46] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, page 341–352. ACM, 2012.

[47] Volodymyr V Kindratenko, Jeremy J Enos, Guochun Shi, Michael T Showerman, Galen W Arnold, John E Stone, James C Phillips, and Wen-mei Hwu. Gpu clusters for high-performance computing. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–8. IEEE, 2009.

[48] Tyng-Yeu Liang and Yu-Wei Chang. Gridcuda: A grid-enabled cuda programming toolkit. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 141–146, March 2011.

[49] R. A. Meyer and L. H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 9(3):199–218, Sep 1970.

[50] Raffaele Montella, Giuseppe Coviello, Giulio Giunta, Giuliano Laccetti, Florin Isaila, and Javier Blas. A general-purpose virtualization service for hpc on cloud computing: an application to gpus. *Parallel Processing and Applied Mathematics*, pages 740–749, 2012.

[51] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[52] Avery Pennarun, Bill Allombert, and Petter Reinholdtsen. Debian Popularity Contest. `http://popcon.debian.org`, 2018.

[53] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[54] C. Reano, A. J. Pena, F. Silla, J. Duato, R. Mayo, and E. S. Quintana-Orti. Cu2rcu: Towards the complete rcuda remote gpu virtualization and sharing solution. *20th Annual International Conference on High Performance Computing*, 0:1–10, 2012.

[55] Carlos Reaño, Antonio J Peña, Federico Silla, José Duato, Rafael Mayo, and Enrique S Quintana-Ortí. Cu2rcu: Towards the complete rcuda remote gpu virtualization and sharing solution. In *2012 19th International Conference on High Performance Computing*, pages 1–10. IEEE, 2012.

[56] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM, 2011.

[57] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the nas parallel benchmarks in opencl. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 137–148. IEEE, 2011.

[58] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vcuda: Gpu-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, 61(6):804–816, 2012.

[59] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[60] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. Gpuvm: Why not virtualizing gpus at the hypervisor? In *USENIX Annual Technical Conference*, pages 109–120, 2014.

[61] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. Gpuvm: Why not virtualizing gpus at the hypervisor? In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 109–120, Berkeley, CA, USA, 2014. USENIX Association.

[62] The Ubuntu Web Team, Avery Pennarun, Bill Allombert, and Petter Reinholdtsen. Ubuntu Popularity Contest. `http://popcon.ubuntu.com`, 2018.

[63] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 121–132, Berkeley, CA, USA, 2014. USENIX Association.

[64] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *USENIX Annual Technical Conference*, pages 121–132, 2014.

[65] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E Porter. A Study of Modern Linux API Usage and Compatibility: What to Support when You'Re Supporting. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, London, United Kingdom, 2016.

[66] Dimitrios Vasilas, Stefanos Gerangelos, and Nectarios Koziris. VGVM: efficient GPU capabilities in virtual machines. In *International Conference on High Performance Computing & Simulation, HPCS 2016, Innsbruck, Austria, July 18-22, 2016*, pages 637–644, 2016.

[67] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *ISPASS*, 2016.

[68] Lan Vu, Hari Sivaraman, and Rishi Bidarkar. Gpu virtualization for high performance general purpose computing on the esx hypervisor. In *Proceedings of the High Performance Computing Symposium*, HPC '14, pages 2:1–2:8, San Diego, CA, USA, 2014. Society for Computer Simulation International.

[69] Carl Waldspurger, Emery Berger, Abhishek Bhattacharjee, Kevin Pedretti, Simon Peter, and Chris Rossbach. Sweet spots and limits for virtualization. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pages 177–177, New York, NY, USA, 2016. ACM.

[70] Carl Waldspurger and Mendel Rosenblum. I/o virtualization. *Commun. ACM*, 55(1):66–73, January 2012.

[71] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, December 2002.

[72] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014–112*, 2014.

[73] Hangchen Yu, Arthur M Peters, Amogh Akshintala, and Christopher J Rossbach. Automatic virtualization of accelerators. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 58–65. ACM, 2019.