

Toward Efficient and Realizable Virtualization of Domain Specific Accelerators

Amogh Akshintala

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2020

Approved by:

Michael Ferdman

Kevin Jeffay

Fabian N. Monroe

Donald E. Porter

Christopher J. Rossbach

©2020
Amogh Akshintala
ALL RIGHTS RESERVED

ABSTRACT

Amogh Akshintala: Toward Efficient and Realizable Virtualization of Compute Accelerators
(Under the direction of Donald E. Porter, and Christopher J. Rossbach)

This dissertation is concerned with software techniques for fair, efficient and isolated sharing of Domain Specific Accelerators (DSAs), such as GPUs and TPUs, that are programmed through an API, e.g., CUDA, Tensorflow. Specifically, we explore the following hypothesis: the vendor-provided userspace API is akin to the ISA for CPU virtualization, and is, therefore, the right interface to interpose to virtualize DSAs.

We first present empirical analysis of canonical virtualization techniques, in order to understand their inefficacies. We find that most canonical techniques introduce high performance overhead, which is a non-starter for DSA virtualization: Accelerators should accelerate, after all. The only canonical technique that is able to provide low-overhead virtualization interposes the user-space API, but does so in a manner that precludes hypervisor interposition. Precluding hypervisor interposition results in the inability to enforce key virtualization properties like fairness and isolation.

In order to arrive at a virtualization technique that satisfies the needs of DSA virtualization, we develop a novel analysis framework: **IEMTS**. **IEMTS** characterizes virtualization designs based on the **I**nterface interposed, the **E**nd-points (in the guest and the host) involved in the interposition, the **M**echanism of interposition, the **T**ransport used to connect the endpoints, and the way the interposed functionality is **S**ynthesized in the guest.

Insights from analyzing prior techniques using the **IEMTS** framework enable a novel virtualization scheme specifically designed for API-controlled DSAs: **Hypervisor Interposed Remote Acceleration (HIRA)**. **HIRA** interposes the user-space API, and transports the interposed API calls to the host via a hypervisor managed transport. Use of a hypervisor managed channel, coupled with semantic information of the interposed API, enables the hypervisor to exercise control over the DSA's resources. Once the access has been checked and scheduled by the hypervisor,

the interposed functionality is synthesized in the host by calling the vendor framework for the given API.

Empirical analysis of AVA, a prototype realization of HIRA for KVM, shows that HIRA introduces low overhead (15% on average) while also enabling the hypervisor to enforce fairness and isolation. We wrap up by characterizing adverse scenarios for HIRA, and propose solutions for the common case.

ACKNOWLEDGEMENTS

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet,

TABLE OF CONTENTS

LIST OF TABLES	x
LIST OF FIGURES	xii
LIST OF ABBREVIATIONS	xiv
1 Introduction	1
1.1 History of Virtualization	2
1.2 Virtual hardware	3
1.3 Virtualizing Domain Specific Accelerators.....	4
2 Background	8
2.1 Virtualization Properties	8
2.2 Domain Specific Accelerators.....	9
2.2.1 DSA Design.....	9
2.2.2 DSA Software stacks are silos	10
2.3 DSA Virtualization	11
2.3.1 Inefficacy of Traditional GPGPU Virtualization Techniques.....	12
2.3.1.1 Pass-through	12
2.3.1.2 Device emulation	12
2.3.1.3 Full virtualization.....	12
2.3.1.4 Mediated pass-through	13
2.3.1.5 Para-virtualization	13
2.3.1.6 API Remoting	13
2.3.1.7 Hardware virtualization support	14

2.4	Summary	16
3	ISA virtualization is untenable for GPUs	17
3.1	Implementing representatives of each virtualization scheme	19
3.1.1	GPUvm	19
3.1.2	User-space API remoting	20
3.1.3	SVGA	21
3.1.4	XEN-SVGA and TRILLIUM	22
3.1.4.1	Mesa3D OpenCL Support	24
3.1.4.2	LLVM TGSI Back-end	24
3.1.5	GPU ISAs and IRs	25
3.1.6	Optimizations	26
3.2	Methodology	27
3.2.1	Benchmarks	27
3.2.2	Control Experiments	28
3.3	Evaluation	31
3.3.1	End-to-End	31
3.3.2	Impact of vISA choice	32
3.4	Conclusion	33
4	IEMTS — A new accelerator virtualization taxonomy	34
4.1	Understanding the sources of overhead in TRILLIUM	34
4.2	Where to interpose?	35
4.3	IEMTS: A new analysis framework	35
4.4	Conclusion	38
5	Hypervisor Interposed Remote Acceleration	39
5.1	Accelerator Silos	40
5.2	Design	41
5.3	AvA Components	43

5.4	Developer Work-flow	44
5.4.1	Communication Transport	45
5.4.2	Sharing and Protection	45
5.4.3	Scheduling and Resource Allocation	46
5.4.4	Memory Management	46
5.5	CAVA and LAPIS	46
5.5.1	Resource Management and Policy.	47
5.5.2	Shadow Buffering	48
5.5.3	Mapped memory	48
5.6	Implementation	49
5.6.1	Transport	49
5.6.2	Hypervisor Interposition and Mediation	49
5.6.2.1	Policies in eBPF	50
5.6.2.2	Scheduling	50
5.6.3	Shadow Resources	51
5.6.4	Callbacks	51
5.7	Evaluation	51
5.7.1	Development Effort	52
5.7.2	End-to-end Performance	53
5.7.3	Micro-benchmarks	55
5.7.3.1	Asynchrony Optimizations	56
5.7.4	Scalability	56
5.7.5	Guaranteeing fairness by Rate Limiting APIs	57
5.7.6	Live Migration	58
5.8	Related Work	59
5.9	Conclusion	60
6	Optimized data movement in HIRA	61

7	Related Work	64
7.1	GPU Virtualization	64
7.1.1	Comparison methodology	66
7.1.2	Dominant Trends	66
7.1.3	Additional Considerations	67
7.1.4	Full Virtualization	67
7.1.5	API Remoting	69
7.1.6	Para-virtualization	71
7.2	Language-level Virtualization	72
8	Conclusion	73
	BIBLIOGRAPHY	74

LIST OF TABLES

1.1	The trade-offs canonical virtualization designs make when applied to DSA virtualization.	5
3.1	Benchmarks used in our evaluation grouped into in three categories: workloads where the cost of interposition Dominates , workloads with Moderate amounts of events that must be interposed, and workloads that Rarely exhibit interposition events.	28
3.2	Possible sources of performance differences between kernels generated using LLVM+PTXAS (comparable to NVCC) and Clover+Nouveau.	29
4.1	Comparing virtualization designs using the IEMTS framework.	37
4.2	A possible “sweet spot” in the GPGPU virtualization design space?	38
5.1	Development effort for forwarding different APIs, along with the benchmarks [144, 135, 48] and hardware used to evaluate them. The # column indicates the number of API functions supported. The Python APIs are forwarded dynamically, making # inapplicable. Gen indicates whether the API forwarding was generated by CAVA or was written by hand. LoC is the number of lines of code (including blank lines and comments) in the CAVA specification or C/Python code. Churn is the total number of lines modified in commits.	52

7.1 Existing GPU virtualization proposals, grouped by approach. Previously published in the Trillium paper [33]. The **lib unmod** and **OS unmod** columns indicate ability to support unmodified guest libraries and OS/driver. The **lib-compatible** and **hw-compatible** 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 also 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. Values in this column were computed by dividing the expected speedup from using a GPU by the slowdown introduced by virtualization. Entries where overheads eclipse GPU-based performance gains are marked in **red**. Performance profitable entries are **blue**. Greyed out cells indicate the metric is meaningless for that design. Light grey cells indicate that the data was not available. 65

LIST OF FIGURES

2.1	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.	10
2.2	Unfairness in slowdown between <code>needle</code> and <code>hotspot</code> applications in separate VMs running GPU kernels iteratively with BitFusion FlexDirect. When running alone, <code>hotspot</code> has throughput of 126.3 ms/kernel. Fairness is calculated by $ s_1 - s_2 / (s_1 + s_2)$, where s_i is the slowdown of application i when running concurrently. [REMOVE AVA FROM THIS FIGURE.]	14
2.3	Throughput achieved by three instances of QATzip (running in VMs with SR-IOV pass-through) with different block sizes, running separately (Uncontended) and concurrently (Contended). Slowdown during concurrent execution is dependent on block size, i.e., the QAT HW scheduler cannot guarantee fairness.	15
3.1	Xen-based virtualization designs. (a) GPUvm. (b) User-space API remoting over RPC—dashed arrows indicate API-REMOTE-CPU, while solid ones indicate API-REMOTE-GPU.	20
3.2	Stack diagram of the SVGA virtualization scheme.	21
3.3	XEN-SVGA and TRILLIUM designs. (a) XEN-SVGA approximates the SVGA model extended to support GPU Compute. (c) The design of TRILLIUM with shadow pipe.	22
3.4	End-to-end execution times of benchmarks on virtualization prototypes, relative to end-to-end execution time on the NVIDIA CUDA runtime in a native setting. gRPC overhead is removed from the reported measurements, which is up to 10% of the total execution time for API remoting, and 40% for TRILLIUM.	30
3.5	Kernel execution slowdown due to virtual ISAs. TGSI: the LLVM TGSI back-end compiler used in XEN-SVGA. LLVM: LLVM NVIDIA PTX (NVPTX) back-end used in TRILLIUM. No IR: native NVIDIA compiler.	30
4.1	Possible points of interposition.	35
5.1	Overview of Hypervisor Interposed Remote Acceleration. Components with striped backgrounds are API-specific and are generated from an API specification. Components with solid backgrounds are API-agnostic and only need to be implemented once per hypervisor (or per hypervisor-OS combination.	41

5.2	Overview of AVA.....	43
5.3	End-to-end execution time on virtualized APIs or accelerators normalized to native execution time. τ_{f_py} is the handwritten TensorFlow Python API remoting with AVA API-agnostic components.	54
5.4	End-to-end execution time on virtualized CUDART and CUDA-accelerated TensorFlow APIs normalized to native execution time.	54
5.5	Overhead introduced by AVA for a micro-benchmark with varying work per call and data per call. The plot is log-log and the trend is linear. Runtime is relative to running the same micro-benchmark natively on the NVIDIA GTX 1080 GPU.	55
5.6	End-to-end runtime of CUDA benchmarks (relative to native) using synchronous and asynchronous specifications.	56
5.7	Scalability of AVA when supporting multiple VMs running a single application each (VM bars in figure), and multiple applications in a single VM (App bars in figure). Runtime is relative to running a single application natively (native scalability is shown with the Native bars.	56
5.8	Unfairness of the fixed and adaptive scheduling algorithms with two different measurement periods. The width of the shaded areas show the probability of the bias (unfairness) being a specific value in any given measurement window. The horizontal bar shows the median and the vertical line runs from the minimum to the maximum.	57
5.9	Live migration downtime for single-threaded OpenCL benchmarks on NVIDIA GTX 1080. This downtime is in addition to the ~ 75 ms of downtime of the VM migration itself. Migration downtime does not include time spent waiting for executing kernels to complete (accounted as latency), as the application is still performing useful computation on the accelerator during that time. The width of the shaded areas show the the probability of a migration taking that length of time. The horizontal bar shows the median and the vertical line shows the range from minimum to maximum.	58
6.1	All data processed by multiple DSA API frameworks must pass through the guest application, leading to redundant data movement.	62

LIST OF ABBREVIATIONS

[THIS IS NOT DONE. IGNORE FOR NOW PLEASE]

DSA	Domain Specific Architectures
DSL	Domain Specific Language
I/O	Input/Output
IPC	Inter-Process Communication
IPI	Inter-Processor Interrupt
AvA	Accelerated Virtualization of Accelerators
AYO	Add Your Own in alphabetic order. . .

CHAPTER 1: INTRODUCTION

Domain Specific Accelerators (DSAs) are programmable compute units that are specialized to a particular class of computation in order to improve performance, to optimize energy usage, or both. DSAs are seeing wide adoption in data centers due to their high return on investment. A 2013 projection by Google showed that they would need to double the installed CPU compute capacity in order to support just three minutes of Google Voice Search per user per day, using speech recognition DNNs [82]. This led Google to design and adopt the Tensorflow Processing Unit (TPU), a DSA specialized for the Tensor-based computation popular in Neural Networks. The very first generation of TPUs, which were deployed to Google data centers in 2015, were empirically found to have $200\times$ and $79\times$ higher Performance/Watt respectively over the CPUs and Nvidia k80 GPUs that were prevalent in their data centers at the time [83].

Data centers themselves are proliferating rapidly: a recent report by Synergy Research [143] found that there were more than 500 hyperscale data centers operational across the world at the end of 2019. Hyperscale data centers are typically very large installations (10000+ servers) of compute, storage, and the networks required to connect them. These gargantuan data centers are needed to provide the digital services that have permeated into every facet of our lives, e.g., Google Maps, video streaming, digital communication. Hyperscale data centers typically leverage the cost efficiency and ease of maintenance of centralized computing, while also reaping the flexibility and scalability of distributed computing via virtualization.

Virtualization is a cornerstone technology in the current computing landscape: Virtualization of compute resources is fundamental to high utilization, by securely multiplexing the installed compute capacity among many users, Virtualization is also critical for elasticity: when the demand for a digital service is high, the customer may request and utilize a large number of machines; when the spike in demand subsides, the customer can release the excess compute resources, thereby only paying for what they are actively using.

Despite a decade of attention from the academic community [154, 78, 31, 150, 42, 90, 120, 153, 61, 140], DSA virtualization remains an open problem: prior work either imposes high overhead that effectively offsets the speedup from the DSA, or sacrifices one or more key virtualization properties: compatibility, interposition, sharing, isolation. Cloud providers such as Amazon expose Graphical Processing Units (GPU) and Field Programmable Gate Arrays (FPGA) to individual VMs via PCIe passthrough [35, 147]. Using PCIe passthrough means bypassing the hypervisor, thereby giving up the consolidation and fault tolerance benefits of virtualization.

This dissertation empirically explores the trade-offs in the design space of software virtualization schemes for DSAs in Chapter 3, introduces a novel analytical framework to characterize DSA virtualization schemes in Chapter 4, and based on this analysis, proposes a novel low-overhead virtualization scheme that interposes the user-facing Application Programmer’s Interface (API) while using automation to compensate for the resulting development complexity in Chapter 5.

1.1 History of Virtualization

The story of virtualization is long and tumultuous, stretching from the very early days of computing, right to today. Memory was the first resource to be virtualized: German physicist Fritz-Rudolf Güntsch described the basic idea of virtual memory in his doctoral dissertation in 1956 [72]. The Cambridge University/Ferranti Inc. Atlas [89] computer was the first to openly commercialize virtual memory. Virtually all computers since then have supported virtual memory, with most providing hardware units—*Memory Management Unit (MMU)*—to accelerate the virtualization of memory.

Next came the virtualization of individual machines’ processors and peripherals. *Hardware virtualization* [30]—the idea of virtualizing the entire computer to enable the simultaneous execution of multiple Operating Systems (OS)—was invented in 1962, and commercialized in the IBM VM-370 [52] 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 [117], and was widely considered unvirtualizable [65, 46]. 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, the community devised techniques such as virtualization-specific ISA extensions (e.g., AMD-V and Intel VT-x), to natively execute virtualized applications, only trapping to the hypervisor when the application attempts to perform sensitive operations.

Hardware interfaces were not the only ones virtualized, and in fact, most computing environments today rely on hardware virtualization and one or more of the following *software virtualization* schemes working in tandem. 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.

1.2 Virtual hardware

This dissertation is primarily concerned with virtualization of DSA hardware. Virtualization of hardware (as opposed to virtualization of an abstract machine or of an OS interface) is vital to high utilization of available physical resources in large computing installations, such as in cloud computing. Popek and Goldberg describe hardware virtualization as a scheme that upholds three properties—*equivalence, performance, and safety*, while Bugnion, Nieh and Tsafirir [47] provide a more succinct definition—“*the application of the layering principle with enforced modularity such that the exposed resource is identical to the underlying resource*”. In lay terms, hardware virtualization involves mediating access to a 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 considered *virtual*, as it is not controlled by the physical underlying hardware, and instead is entirely under the control of supervisory virtualization software, the hypervisor or Virtual Machine Monitor (VMM). While operations in the resulting virtual machine may be directly executed on the

physical hardware for improved performance, as in the case of ISA extensions for virtualization support, e.g., AMD-V and Intel VT-x, all privileged operations must still trap to the hypervisor.

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, and are still used in practice due to lower overhead for sequences of sensitive instructions that need to be emulated [32]. Dominant ISAs, such as x86 and ARM, provide extensions to enable low-overhead virtualization in the common case. Both are foundational building blocks for cloud computing [2].

1.3 Virtualizing Domain Specific Accelerators

Virtualizing Domain Specific Accelerators is a delicate act of 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) can not be applied to these 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 DSAs as they sacrifice of one or more essential virtualization characteristics.

Full-virtualization based schemes, e.g., GPUvm [140] suffer from massive overheads that essentially negate the speedup that makes the domain specific accelerator attractive in the first place. Para-virtual systems that interpose on low-level interfaces such as the kernel driver, e.g., SVGA [59], introduce much lower overhead than full-virtualization based schemes but have poor compatibility. 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 [153, 61, 120] 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 [59]). Over the

	Sharing	Interposition	Isolation	Compatibility	Slowdown
Passthrough	✗	✗	✗	✗	1 ×
Full-virtualization	✓	✓	✓	✗	> 100 ×
User-mode API Remoting	✓	✗	✓/✗	✗	1.5 ×
Para-virtualization	✓	✓	✓	✗	10 ×

Table 1.1: The trade-offs canonical virtualization designs make when applied to DSA virtualization.

last decade, GPUs have also been re-purposed for general purpose compute (commonly known as GPGPU). This dissertation is solely concerned with GPGPU virtualization; GPU virtualization is out of scope. GPGPUs embody the hardware interface and software stack design that are commonly used when building DSAs, and are currently the most widely adopted DSA.

Chapter 2 of this dissertation presents background on domain specific accelerators, how their software stacks are different from those of CPUs and I/O devices, and how and why previous software virtualization solutions do not fare well for DSAs. In order to understand the trade-offs with each of the canonical virtualization techniques when applied to GPGPU virtualization, we present an end-to-end evaluation of representative systems. Chapter 2 also considers the notion of the modern DSA stack being a proprietary silo, i.e., that its only stable and publicly available interfaces are at the top and the bottom. Later chapters build on this notion of silo-ed DSA software stacks to design an effective virtualization scheme.

GPGPU virtualization has concretely been studied for the last decade, and yet no viable virtualization technique has emerged. While the drawbacks of these previously considered virtualization techniques are presented in Chapter 2, Chapter 3 presents empirical analysis of representatives of each of the canonical techniques previously considered.

Chapter 3 also contains findings from our attempt to extend VMware’s SVGA model of GPU virtualization to virtualize GPGPUs. We prototyped this extension of the SVGA design for the Xen Hypervisor (as that was the common platform that the rest of the systems evaluated ran on), and hence call this prototype XEN-SVGA. Briefly, we find that while SVGA worked really well for graphical rendering workloads, a naive extension of the same model performs poorly for GPU compute workloads. There are two sources of inefficiencies in this design. This chapter highlights the first: the tight coupling between ISA virtualization and device virtualization.

Eliding ISA virtualization in the XEN-SVGA design, enables the resulting prototype, TRILLIUM [33], to outperform all other traditional virtualization schemes that retain hypervisor interposition. However, TRILLIUM remains 2—3 \times slower than user-space API remoting. This overhead stemmed from our choice of interposition point in TRILLIUM: our implementation of TRILLIUM interposed the “pipe-driver”, the interface between the front and back ends of the GNU/Linux graphics driver system. Interposing this interface meant that a single user-space API call made by the compute application in the VM resulted in multiple interposition events, each of which contributed a fixed remoting overhead. On the other hand, only one such interposition event (with no hypervisor involvement) occurs in a user-space API remoting, leading to the vastly lower overhead observed.

Chapter 4 presents an alternative analysis framework, IEMTS that teases apart design axes that are implicitly and unnecessarily intertwined in much of the literature. Analyzing a virtualization design using IEMTS involves focusing on the **I**nterface the design interposes, the **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. Analyzing canonical virtualization techniques as applied to DSA virtualization, led us to design HIRA a novel virtualization scheme that is specifically suited to DSAs. HIRA is presented in Chapter 5 and Chapter 6.

Domain Specific Accelerators (e.g., Google TPU, Intel QAT, etc.) are typically exposed to developers via a user-space API. The API is typically implemented by proprietary software that interacts with the hardware through opaque interfaces. Chapters 5 and 6 generalize the lessons learned for GPGPUs to all API-controlled domain specific accelerators by interposing user-space APIs.

Chapter 5 presents an overview of AvA, a framework that enables automated virtualization of accelerator APIs. AvA combines on a novel virtualization scheme called Hypervisor Interposed Remote Acceleration (HIRA), with automation based on a Domain Specific Language, LAPIS, which is used to capture semantic information of the interposed APIs. This dissertation is primarily concerned with the HIRA virtualization scheme. Chapters 5 draw on material that appeared in a HotOS workshop paper [163] and an ASPLOS’20 paper [164].

While chapter 5 focuses on the performance implications of API-remoting based virtualization of a single specialized accelerator, Chapter 6 explores performance issues that arise when an application uses multiple API-remoted virtual accelerators in a pipelined fashion. We close the dissertation out by covering related work in Chapter 7.

CHAPTER 2: BACKGROUND

Bugnion, Nieh and Tsafirir [47] define virtualization as “*the application of the layering principle with enforced modularity such that the exposed resource is identical to the underlying resource*”. To put it in simpler words, virtualization is about controlling the interface to a hardware resource (*interposition*) in order to multiplex it among multiple users in a safe manner (*isolation*), without any of them being the wiser (*compatibility*). Virtualization is a huge area with a long and storied history, as alluded to in the introduction; see *Bugnion, Nieh and Tsafirir’s Hardware and software support for virtualization* [47] for comprehensive treatment.

2.1 Virtualization Properties

Given our focus accelerator virtualization, let us consider the following key properties: *interposition*, *compatibility*, and *isolation*.

Interposition. Virtualization decouples a logical resource from a physical one through an indirection layer, intercepting guest interactions with a virtual resource and providing the requested functionality using a combination of software and the underlying physical resource. Thus, virtualization works by *interposing* an interface, and changing or adding to its behavior. Interposition is fundamental to virtualization and provides well-known benefits [155]. The choice of interface and the mechanism for interposing it profoundly impacts the resulting system’s practicality. *Inefficient* interposition of an interface (e.g. trapping frequent MMIO access) undermines performance [140, 165]; *incomplete* interposition compromises the hypervisor’s ability to enforce isolation.

Compatibility as applied to virtualization captures multiple related dimensions, from robustness to evolution of interposed interfaces and adjacent stack layers, to applicability across multiple platforms or related devices. For example, full virtualization of a accelerators’s hardware interface has *poor* compatibility in that it works only with that device. However, it has *good* compatibility with guest software, which will work without modification, assuming the operating system has appropriate

drivers for the device. Current accelerator virtualization techniques reflect a compromise between these two forms of compatibility.

Isolation. Cross-VM isolation is a critical requirement for multi-tenancy: when a resource is multiplexed among mutually distrustful tenants, tenants must not be able to see/alter each other’s data (*data isolation*), or adversely affect each other’s performance (*performance isolation*). A poor choice of interposition mechanism and/or interface limits the system’s ability to provide these guarantees: e.g., API remotng [4, 61, 16] has poor isolation in the common case, as the hypervisor is bypassed. Using separate servers for each protection provides isolation.

2.2 Domain Specific Accelerators

Domain Specific Accelerators (DSAs) are programmable compute units that are specialized to a particular class of computation in order to improve performance, to optimize energy usage for that class of computation, or frequently both. The slowing down of Moore’s law coupled with the rise of Dark Silicon [63] has made Domain Specific Accelerators extremely attractive as they exhibit high computation/Watt efficiency in the computation domain they are specialized to. For example, consider Google’s Tensorflow Processing Unit (TPU) [82], a DSA for the Tensor-based computation popular in Neural Networks. The first generation of TPUs were empirically found to have $200\times$ and $79\times$ higher Performance/Watt respectively over the CPUs and Nvidia k80 GPUs that were prevalent in Google’s data centers at the time [83].

2.2.1 DSA Design

Domain Specific Accelerators are mini-computers that are attached to and controlled by a CPU. DSAs tend to have everything a normal ‘computer’ does (computation units, control logic, memory, and a programming interface), except access to I/O, for which they typically rely on the CPU. Their computation units are typically specialized to a specific domain or type of computation: e.g., GPUs are DSAs that are specially suited to graphics rendering operations (tessellation, occlusion detection, culling, etc.). GPUs have also found wider adoption in other domains (scientific computing, Artificial Intelligence, etc.) due to their inclusion of Single Instruction Multiple Thread (SIMT) style ‘shader’ cores, which efficiently perform the same simple computation in parallel on hundreds (or even

thousands) of threads. DSAs typically have their own memory: while some have small amounts of memory that are just enough to process limited operations (e.g., Intel QAT), most have a lot of internal memory, as memory bandwidth and size are key parameters in tuning the architecture for the given domain. DSAs typically expose an I/O device-like hardware interface, i.e., a command queue, a DMA engine, and some number of memory-mapped control registers.

DSAs look like I/O devices to the host CPU, and many DSA designers take advantage of this to build a software stack that is opaque to the host OS. Exposing an I/O device-like hardware interface enables the vendor to raise the level of abstraction of the end user’s interface to a user-space software API, enabling quick evolution of both the underlying software and hardware. The user-space API is the only interface that needs to be stable (or at least backwards compatible). The ISA of the compute units on the DSA are typically not exposed to the programmer.

2.2.2 DSA Software stacks are silos

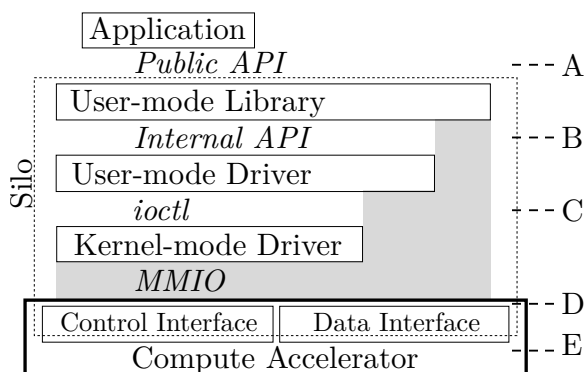


Figure 2.1: An accelerator silo. The public API and the interfaces with striped backgrounds are interposition candidates. All interfaces with backgrounds are proprietary and subject to change.

Domain Specific Accelerator stacks are composed of layered components that include a user-mode library to support an API framework and a driver to manage the device. Vendors are incentivized to use proprietary interfaces and protocols between layers to preserve forward compatibility, and to use kernel-bypass communication techniques to eliminate OS overheads.

Scheduling and resource allocation on DSAs are typically not managed by the CPU-based Operating System. Instead resource allocation is primarily under the control of a combination of the proprietary CPU-based runtime (user-space and kernel), and the controller on the DSA itself.

DSAs typically contain an entire software stack on the hardware module, which is hidden from the programmer. This stack is used to implement a command-based programming interface that enables the vendor’s CPU-based control software for scheduling and resource allocation.

End users work with the DSA almost entirely in higher level languages, typically the language the DSL is embedded in (e.g., Python for Google’s TPU, C/C++ for Nvidia and AMD GPGPUs). The DSL/API is supported by a compiler (which converts the user’s program to the the ISA of the computation units on the DSA), a user-space runtime and a kernel module (which work together to implement the user-space API, and control the DSA).

2.3 DSA Virtualization

Despite mounting evidence of accelerator under-utilization [21, 156, 108, 111, 19], and abundant prior research into multiplexing Domain Specific Accelerators (DSAs) [111, 19, 168, 156, 108, 161], DSAs remain dedicated exclusively to a single guest in shared computing environments. This section provides background on the well studied, but mostly unresolved problem of GPGPU virtualization to explain this trend.

Existing GPU virtualization solutions [59, 96] support graphics frameworks like Direct3D [43], OpenGL [129]. In principle, there should be no fundamental difference between GPU virtualization for graphics versus *compute* workloads, as “compute shaders” are implemented by the hardware as an additional stage in the graphics pipeline [6]. In practice, they have significantly different goals: for graphics, virtualization designs target an interactive frame rate (18-30 fps [24]); 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 [105], research attention to OS abstractions for GPGPUs [122, 124, 134, 84, 85, 91] has yielded little consensus. Persistent vendor-specificity of programming frameworks further impedes both interposition and compatibility.

2.3.1 Inefficacy of Traditional GPGPU 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). We briefly describe each technique, and look at their strengths and shortcomings in this section. Refer to Related Work (§ ??) for details on individual prior work under each technique.

2.3.1.1 Pass-through

PCIe pass-through, the current *de facto* standard technique for GPGPU virtualization, provides a VM with full exclusive access to a physical GPU. The GPU’s hardware interface is directly exposed to the guest OS, and therefore can’t be multiplexed as the hypervisor does not interpose *any* interface. Virtualization hardware extensions (e.g., Intel VT-d [29]) are device-agnostic, making PCIe pass-through easily adaptable to any DSA. Pass-through provides native performance at the cost of *sharing, interposition, compatibility and isolation*.

2.3.1.2 Device emulation

Device Emulation [41] 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.

2.3.1.3 Full virtualization

The hypervisor interposes GPU’s hardware interface to provides a virtual environment in which unmodified GPGPU programs run on unmodified guest software stacks. For DSAs, this interface tends to be memory mapped I/O (MMIO), necessitating trap-based interposition (e.g. using memory protection or de-privileging), leading to devastating performance slowdowns (e.g., 100× slowdown with GPUvm [140, 165]). DSA hardware interfaces tend to be proprietary and device-specific, so full virtualization based solutions have poor compatibility, even across different devices of the same type

(e.g. AMD vs NVIDIA GPUs). Full virtualization solutions also typically rely on reverse engineering of proprietary control interfaces, rendering them extremely tedious to build, maintain and evolve.

2.3.1.4 Mediated pass-through

A hybridization of pass-through and full virtualization, Mediated pass-through [147, 112, 159] uses pass-through for data plane operations, and provides a privileged control plane interface for sensitive operations. Mediated pass-through can preserve some of the raw speedup of acceleration and allows guests to use native drivers and libraries. However, limited interposition limits a hypervisor’s ability to effectively manage resource sharing. More importantly, hardware support is required. To our knowledge, Intel integrated GPUs are the only accelerators with such support.

2.3.1.5 Para-virtualization

Rather than interposing an existing interface in the stack, para-virtualization [140, 59, 149, 97, 74, 68, 102, 118, 139, 157, 36] *creates* an efficiently interposable interface in software and adjusts adjacent stack layers to use it. The driver and runtime libraries in every supported OS must be modified to work in concert with the virtualization layer. Para-virtualization enables encapsulation of diverse hardware behind a single interposable interface, but compromises compatibility. Guest software must be modified, and the para-virtual device interface must be maintained as interfaces evolve. For example, VMware’s SVGA II [59] encapsulates multiple GPU programming frameworks, but keeping up with the evolution of those frameworks has proved untenable: SVGA remains multiple versions behind current frameworks [23, 22].

2.3.1.6 API Remoting

User-space API Remoting based virtualization designs interpose application-level APIs (e.g. CUDA, OpenCL) by shimming a dynamic library and remote them to the corresponding framework in the host [131, 71, 66], on a dedicated appliance VM [153], or on a remote server [61, 120, 101, 90, 39, 60, 100]. API Remoting is similar to RPC [127, 106] or system call interposition [36, 142, 37, 95]. Limited interposition frequency, batching opportunities [61] and high-speed networks [8, 4] reduce overheads, making this class of designs appealing to industry. Dell XaaS [80], BitFusion FlexDirect [4], and Google Cloud TPUs [12] currently use it to support GPUs, FPGAs, and TPUs.

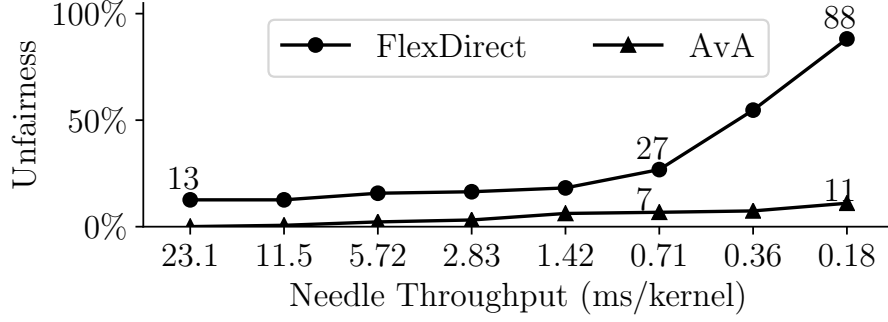


Figure 2.2: Unfairness in slowdown between `needle` and `hotspot` applications in separate VMs running GPU kernels iteratively with BitFusion FlexDirect. When running alone, `hotspot` has throughput of 126.3 ms/kernel. Fairness is calculated by $|s_1 - s_2| / (s_1 + s_2)$, where s_i is the slowdown of application i when running concurrently. [REMOVE AVA FROM THIS FIGURE.]

However, API remotng compromises compatibility if multiple APIs or API versions must be supported. Moreover the technique bypasses the hypervisor, giving up the interposition required for hypervisor-enforced resource management. Our experiments with commercial systems like BitFusion FlexDirect [4] show vulnerability to massive unfairness pathologies. Figure 2.2 shows the problem on an NVIDIA GTX 1080: FlexDirect is unfair (up to 88.1%) when running two applications with different kernel run-lengths (126.3 ms/kernel vs 0.18 ms/ kernel in the worst case).

Deferring enforcement to a trusted surrogate in the host or remote machine is a tenable alternative. However, the co-ordination required to integrate with hypervisor-level resource management means that current solutions do not support it, and the engineering effort required would be substantial. Existing accelerator API remotng systems are by themselves massive undertakings without any hypervisor integration: systems like Bitfusion FlexDirect [4], and rCUDA [61] reflect multi-year system-building efforts.

2.3.1.7 Hardware virtualization support

Hardware support for virtualization (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. NVIDIA and AMD both ship GPU cards targeted at the VDI market that use SR-IOV to export multiple virtual GPUs from the hardware.

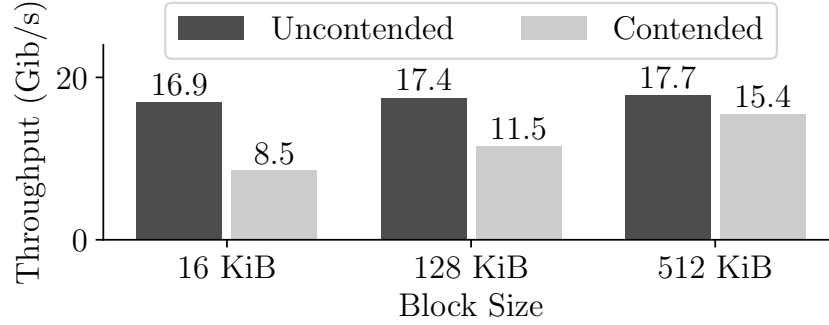


Figure 2.3: Throughput achieved by three instances of QATzip (running in VMs with SR-IOV pass-through) with different block sizes, running separately (**Uncontended**) and concurrently (**Contended**). Slowdown during concurrent execution is dependent on block size, i.e., the QAT HW scheduler cannot guarantee fairness.

SR-IOV exhibits close to native performance [57], 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. SR-IOV provides an interface and protocol for managing VFs, but the device vendor must *implement* any cross-VF sharing support *in silicon*. The technique can provide strong virtualization guarantees [56, 58], but hardware-level resource management is inflexible and slow to evolve: current implementations are trivially vulnerable to fragmentation and unfairness pathologies that cannot be changed.

Hardware designers tend to favor simple resource management policy implementations, easily leading to pathologies. To illustrate the problem, we measured compression throughput when three VMs contend on an Intel QuickAssist [14] (with SR-IOV). The three VMs configured the accelerator to compress at different chunk sizes. Figure 2.3 shows the results of this experiment, with and without contention. Each VM was assigned a PCIe Virtual Function (VF) exposed by the same Physical Function (PF), causing the hardware to schedule requests round-robin. When there is no contention, each application achieves a similar throughput. However, when the 3 applications were executed concurrently, the throughput achieved was a function of offload chunk size used, *yielding unfairness that cannot be fixed without changing the hardware*.

Further, evidence is scant that broad SR-IOV support will emerge for accelerators: only two current GPUs support it [3, 75], none of the TPUs we evaluate support it; and SR-IOV *interface* IP blocks from FPGA vendors (used by [152, 166, 115, 76]) do not implement resource management.

2.4 Summary

Interposing opaque, frequently-changing interfaces communicating with memory mapped command rings is *impractical* because it requires inefficient techniques and yields solutions that sacrifice compatibility. Accelerator stacks are effectively *silos* (Figure 2.1), whose intermediate layers *cannot be practically separated* to virtualize the device. Most current hardware accelerators feature some hardware support for virtualization: primarily for process-level address-space separation, and in a small handful of cases, SR-IOV. A central premise of this dissertation is that hardware support for process-level isolation *could* suffice to support hypervisor-level virtualization as well, but the silo-ed structure of current accelerator stacks prevents it. While hardware support for virtualization is the desired end game, we do not expect a better standard for such support to emerge soon: incentives for investing in the significant engineering effort required for such a standard are scarce. This dissertation focuses on virtualization schemes that can be purely implemented in software so as to sidestep this challenge.

CHAPTER 3: ISA VIRTUALIZATION IS UNTENABLE FOR GPUS

Compute density and programmability [17, 138, 70] have made GPUs the clear choice for efficiency and performance: Popular machine learning frameworks such as Caffe [81], Tensorflow [28], Microsoft CNTK [162], and Torch7 [51] 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 [27].

Despite much prior research [154, 78, 31, 150] 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 [35, 147], 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 [42, 90, 120, 153, 61] 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 [122].

In order to understand why a viable virtualization scheme for GPGPUs hasn't emerged, we set out to empirically analyze representatives of each virtualization scheme adopted from prior work on a single platform. We chose the Xen Virtual Machine Monitor as our empirical platform as that was the only platform that GPUvm [140] ran on. GPUvm was selected as a representative of both full-virtual and para-virtual schemes. We built a custom user-space API-remoting scheme that traps and forwards OpenCL API calls using gRPC and protobuf over a local socket. This API remoting system was then used to forward OpenCL calls to the native GPU stack provided by Nvidia, and to an OpenCL implementation provided by Intel for their CPUs. Finally, we retrofitted support for OpenCL to an implementation of the SVGA [59] design in Xen. We were specifically interested in SVGA as it realizes a hybrid virtualization scheme: SVGA effectively remotes API

calls through a hypervisor controlled channel. SVGA multiplexes multiple rendering frameworks through a hypercall interface that corresponds to the DirectX API, and then translates these DirectX APIs back to whatever framework is available in the host. Our XenSVGA implementation relied on the open source Mesa GPU library, and the Nouveau GPU driver on the GNU/Linux platform. These are the same components used by VMware’s implementation of SVGA. Enabling support for OpenCL in Mesa and Nouveau involved implementing a compiler for SVGA’s TGSI virtual ISA. These empirical measurements are presented in section § 3.3 of this chapter.

Implementing the TGSI compiler for XenSVGA led to questions about the benefits of having presence of a virtual ISA (vISA) for DSAs: GPUs (and most DSAs) already support vendor-specific virtual ISAs (vISAs). A vISA like TGSI introduces several additional steps during virtualization: the program to be accelerated must first be translated to the hypervisor-supported vISA, and subsequently re-translated to the ISA of the hardware vendor’s vISA (before invoking the vendor’s software runtime to yet again translate it to the ISA of the hardware). Further, the compilers used to translate to and from this vISA must be competitive with those from the hardware vendor, and we risk obscuring important semantic information in the original program from the hardware vendor provided compiler. The guest compiler cannot target the native GPU architecture, and valuable semantic information is lost to the host compiler. Further, while incorporating a vISA 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 GNU/Linux Nouveau driver understandably results in code that is significantly less performant than that produced by the proprietary stack.

Flexible interposition and strong isolation mechanisms are critical for device management: a virtualization layer’s primary goal is to enable isolated sharing across VMs. However, a vISA in the case of DSAs serves only compatibility, and often does so redundantly as DSA frameworks typically subsume compilers into the device driver. In order to test this hypothesis, we built a variant of XenSVGA, TRILLIUM. TRILLIUM 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 relegates the translation from GPU source code to physical GPU ISA to the host-resident driver. This vastly reduces complexity, eliminates a redundant translation layer, and ensures that the GPU compiler has a high-fidelity view

of the target hardware, restoring optimization opportunities sacrificed by a design that relies on multiple translations.

We found that the additional vISA provides little benefit; in fact, it harms performance by necessitating a translation layer that obscures the program’s semantic information from the final vendor-provided compiler. TRILLIUM outperforms GPUvm (a full virtualization system) by up to $14\times$ ($5.5\times$ on average) and outperforms XenSVGA by as much as $7.3\times$ ($5.4\times$ on average).

While TRILLIUM ultimately fails to compete with the low overheads available from user-space API-remoting, it serves as existence proof of a viable alternative design that preserves desirable virtualization properties such as consolidation, hypervisor interposition, isolation, and encapsulation, without

3.1 Implementing representatives of each virtualization scheme

Existing GPU virtualization solutions [59, 96] support graphics frameworks like Direct3D [43], OpenGL [129]. In principle, there should be no fundamental difference between GPU virtualization for graphics versus *compute* workloads: “compute shaders” are implemented by the hardware as an additional stage in the graphics pipeline [6]. In practice, they have significantly different goals: For graphics, virtualization designs target an interactive frame rate (18-30 fps [24]). 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 [105], research attention to OS abstractions for GPGPUs [122, 124, 134, 84, 85, 91] has yielded little consensus. This section describes each of the systems that we chose to represent each of the canonical virtualization schemes in our empirical analysis, and how we modified or implemented them.

3.1.1 GPUvm

As a representative of full and para-virtual schemes, we chose to study GPUvm [140], a Xen-based virtualization scheme for NVIDIA’s Kepler and Fermi GPUs. A simplified block-diagram representation is shown in Figure 3.1a). GPUvm presents each VM with a GPU device model,

which is emulated in the privileged domain (Dom 0). Attempts to access the GPU from all VMs are interposed via traps and are routed through a GPU Aggregator. The aggregator maintains shadow page tables, shadow channels, implements a “fair share scheduler”, and modifies requests to enforce isolation. GPUvm interposes on communication between guest device driver and the GPU device model, by trapping and forwarding MMIO writes. The authors also explore a number of optimizations: lazy shadowing, bar remap, para-virtualization, and multi-call batching. GPUvm has not been maintained: The last release, in 2012, is based on Xen 4.2.0 and runs on Fedora 16 [165]. In order to compare all of the representatives on the same modern platform, we ported GPUvm to Ubuntu 16.04 with Xen 4.8.2.

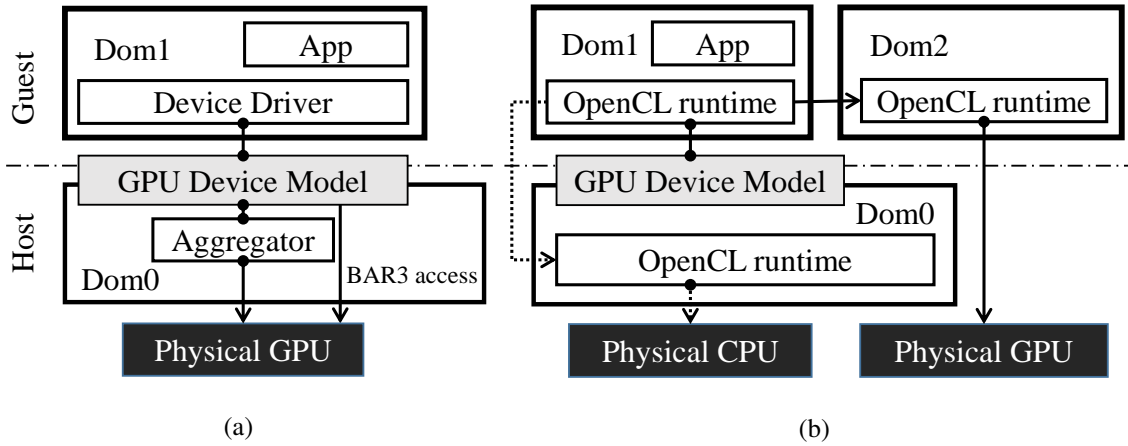


Figure 3.1: Xen-based virtualization designs. (a) GPUvm. (b) User-space API remoting over RPC—dashed arrows indicate API-REMOTE-CPU, while solid ones indicate API-REMOTE-GPU.

3.1.2 User-space API remoting

In order to faithfully mimic user-space API-remoting systems [61, 90, 42], we implemented a system on Xen that trapped OpenCL API calls using a user-space shim library. These trapped calls were then forwarded, via RPC, from one appliance VM (the “client”) to another appliance VM (the “server”). Figure 3.1b 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 ran 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 were executed on a multi-core CPU (Intel CPU Xeon E5-2643) using the Intel OpenCL SRB 5.0

framework. The remoting itself was accomplished using gRPC 1.6 (ProtocolBuffers 3.4.0) and inter-service communications were 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, but were not considered in our implementations.

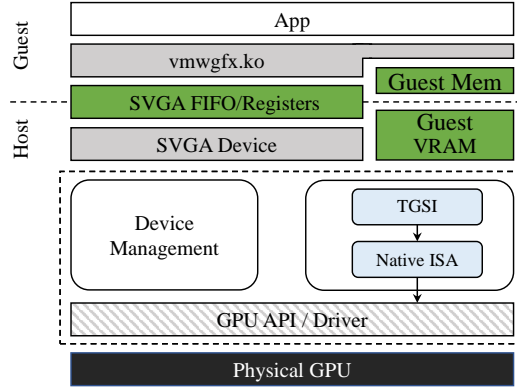


Figure 3.2: Stack diagram of the SVGA virtualization scheme.

3.1.3 SVGA

SVGA [59] 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 3.2 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. As in a physical GPU, SVGA processes commands from a memory mapped command queue; unlike in a physical GPU, the command queue functions as a transport layer for APIs between the guest graphics stack and the hypervisor.

SVGA uses the DirectX [43] 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 (released in 2006) in 2015!).

SVGA also relies on a virtual GPU ISA called TGSI [151]. 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. Further, mapping TGSI to all possible physical GPU ISAs is a herculean task that was doomed from the outset.

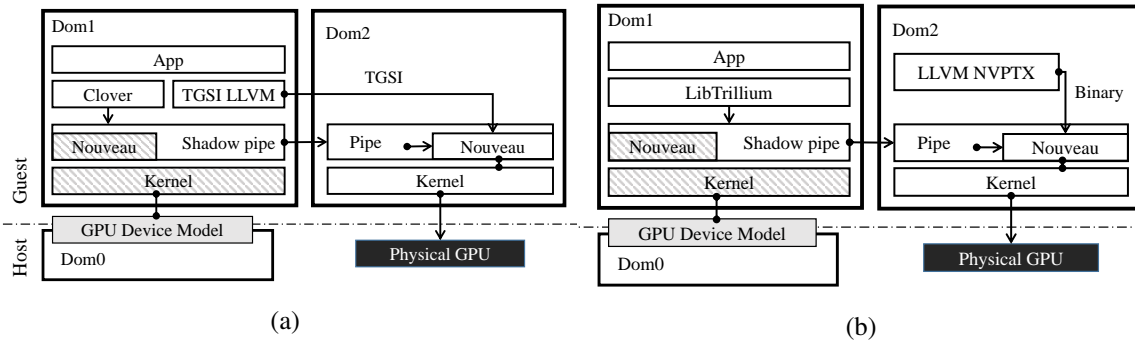


Figure 3.3: XEN-SVGA and TRILLIUM designs. (a) XEN-SVGA approximates the SVGA model extended to support GPU Compute. (c) The design of TRILLIUM with shadow pipe.

3.1.4 XEN-SVGA and TRILLIUM

We initially implemented the SVGA [59] model on Xen strictly keeping with the original design: we implemented OpenCL support in a virtual device and extended the Mesa stack with TGSI support (see Section 3.1.4.2 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. This faithful implementation, hereafter called XEN-SVGA, is used in our study as a representative of the original SVGA design. XEN-SVGA is shown in Figure 3.3a.

In order to test our hypothesis about vISAs, we modified XEN-SVGA to elide the TGSI compiler, thus arriving at TRILLIUM. 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 3.3b shows the Trillium design in the Xen hypervisor stack. The OpenCL API is forwarded from the driver similar to XEN-SVGA. The OpenCL compute kernel (to be run on the GPU) is passed through to the host via hypercalls in the driver, without being translated to TGSI, where it will be translated and optimized for the physical GPU in a virtual appliance (Dom 2 in Figure 3.3b).

XEN-SVGA and TRILLIUM export an abstract virtual device and a para-virtual 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, XEN-SVGA and TRILLIUM forward 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.

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 and TRILLIUM manage user-level contexts, command queues and memory objects. While XEN-SVGA relies on our TGSI compiler to translate the input OpenCL GPGPU kernel to TGSI in the application domain, TRILLIUM skips the compilation phase. Instead, 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 [50, 145] and hypercall [119] 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 3.3. 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. XEN-SVGA and TRILLIUM do not currently guarantee performance isolation, although this can easily be implemented via a rate-limiting API scheduler in the hypervisor, as in GPUvm [141].

3.1.4.1 Mesa3D OpenCL Support

The Mesa3D Graphics Library [26] is an open-source graphics framework that implements graphics runtime libraries (e.g., OpenGL [129], Vulkan [88], Direct3D [43], and OpenCL [138]) on most GNU/Linux installations. It also includes official device drivers, written in a common framework, Gallium3D [25], for Intel and AMD GPUs. Support for NVIDIA GPUs is provided via reverse-engineered open-source driver, Nouveau. 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 graphics pipeline (e.g. translate the state, shaders, and primitives into something that the hardware understands). Effort is underway to replace TGSI with SPIR-V and LLVM IR, but it wasn't mature when we undertook this project.

OpenCL support was first introduced in Mesa3D 9.0 with the release of the Clover state tracker. Clover supports OpenCL 1.1 and was mainly contributed by AMD developers. It was envisioned that Clover would leverage the LLVM [99] compiler to lower the OpenCL source to TGSI. Despite much effort by the open-source community [9, 13], 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.

3.1.4.2 LLVM TGSI Back-end

While Clover provides the library for the OpenCL application to link against, most of the compilation is handled by invoking the OpenCL and C++ front-ends of the LLVM [99] compiler framework. Clover provides much of the front-end infrastructure required to support GPGPU computing in XEN-SVGA and TRILLIUM. However, LLVM lacks a working TGSI back-end, which presented a challenge for XEN-SVGA.

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 added support for a majority of the 32-bit integer and floating

point operations, intrinsics, memory barriers, and control flow. Using this backend we were able to compile and run 10 out of the 12 Rodinia benchmarks [49] used to benchmark GPUvm. Because the compiler was built using the LLVM framework, it enjoyed 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 backend 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.

3.1.5 GPU ISAs and IRs

IRs are of great interest to the realm of virtualization because an IR that is expressive enough to be able to take advantage of new HW developments, while also being universally accepted by all the competing parties will make a wonderful virtualization primitive.

Intermediate Representations are incredibly useful tools to hardware vendors as well, enabling them to simultaneously:

- preserve backward compatibility without compromising innovation at the ISA. The publicly available vISA can be held constant, while the physical ISA is free to change across generations of hardware,
- have the ability to re-optimize legacy code for new HW without having a dependency on the high level toolkit that generated the code in the first place,
- have the freedom to optimize their hardware any way they see fit without having to worry about the effect of said optimizations on the ISA,
- simplify their tool-chain building process by having to only modify one piece—the software that translates from IR to physical ISA) with each new generation of HW,
- and the ability to leverage open source frameworks like LLVM without having to give up their secret sauce.

Given these properties, it comes as no surprise that both AMD and Nvidia both have a public vISA that is stable across generations (i.e. IL and PTX respectively), and a physical ISA that is free to evolve with each generation of hardware (i.e., GCN and SASS respectively). AMD and Nvidia’s front-end compilers generate code in their proprietary vISAs (NVIDIA PTX and LLVM IR for AMD), and then subsequently finalize this code to the native ISA (SASS and GCN) using JIT compilers in the GPU driver . 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 in the graphics realm—enabling interoperability between graphics frameworks and GPUs from different vendors.

As is often the case in a space where competition is fierce, standardization is hard to come by. Despite efforts by standards organizations [88] to convince competing parties to find a middle ground, so as to give tool writers some semblance of sanity, no clear standard IR has emerged in the GPGPU realm. SPIR-V ¹ is the latest challenger to walk this gauntlet.

We observe that LLVM IR is in a unique position to become a standard IR. 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. The framework supports a wide array of front-end languages including CUDA and OpenCL among others, and a wide array of back-ends as well, including other IRs like SPIR-V.

3.1.6 Optimizations

TRILLIUM interposes at the pipe-driver API yielding fine-grained interposition, and therefore fine-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.

¹<https://www.khronos.org/spir/>

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.

3.2 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 [85] an open source CUDA runtime (released in 2012) implemented using Nouveau [10] 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. This restricted all of our evaluation to the Quadro 6000. Experiments to control for hardware versions are reported in §3.2.2.

3.2.1 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 the GDev CUDA runtime, and can correctly execute at least the same 10 benchmarks that run on XEN-SVGA. 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 3.2.2.

The 10 Rodinia benchmarks that our TGSI compiler could compile were categorized based on frequency of interposition: **I**nterposition-**D**ominant workloads run kernels hundreds or thousands of times requiring frequent interposition to set arguments, etc. **I**nterposition-**R**are workloads run a small number of long-running kernels, requiring very little interposition. **M**oderate-interposition workloads

Benchmark	Description	Type
backprop	Back propagation (pattern recognition)	R
gaussian	256x256 matrix Gaussian elimination	D
lud	256x256 matrix LU decomposition	M
nn	k -nearest neighbors classification	D
nw	Needleman-Wunsh (DNA-seq alignment)	M
pathfinder	Search shortest paths through 2-D maps	R

Table 3.1: Benchmarks used in our evaluation grouped into in three categories: workloads where the cost of interposition **D**ominates, workloads with **M**oderate amounts of events that must be interposed, and workloads that **R**arely exhibit interposition events.

lie somewhere in between the other two. Two benchmarks were selected from each category to be used in the evaluation (the optimizations described in § 3.1.6 take significant manual effort).

3.2.2 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.

Code generated by open source compiler vs Nvidia compiler We manually inspected the SASS binaries produced by the Nvidia and the ‘Clover + Nouveau’ frameworks to understand the performance differences. While an in depth of analysis is beyond the scope of this dissertation, we make the following observations:

- Both of the binaries benefit from common optimizations such as loop unrolling, and constant propagation, courtesy of LLVM.
- SASS code produced through the TGSI has substantially more convergence points (SYNC and SSY) instructions, which represent additional opportunity for control flow divergence. Table 3.2 presents the number of control flow instructions in the two binaries, as a proxy for diverging control flow.
- NVIDIA-produced SASS produces very different instruction sequences in several cases, e.g. XMAC (16bit Multiply Add) vs FFMA in TGSI (32-bit Fused Multiply Add). Our conjecture, in keeping with our hypothesis, is that the NVIDIA compiler has better information about which instructions more efficient on a particular architecture. It may be possible to reproduce some of these optimizations in the TGSI to SASS transformation, but since production of the TGSI code cannot rely on knowledge of the architecture, some optimizations may be impossible.

	LLVM+PTXAS				Clover+Nouveua			
	SYNC	SSY	BRA	BB	SYNC	SSY	BRA	BB
bfs	0	0	15	12	8	4	5	18
gaussian	0	0	23	18	6	3	3	9
nn	0	0	9	7	0	0	0	4
nw	10	5	18	23	8	4	8	18
pathfinder	9	5	18	23	12	6	7	32
lud	31	13	67	76	20	10	14	35

Table 3.2: Possible sources of performance differences between kernels generated using LLVM+PTXAS (comparable to NVCC) and Clover+Nouveau.

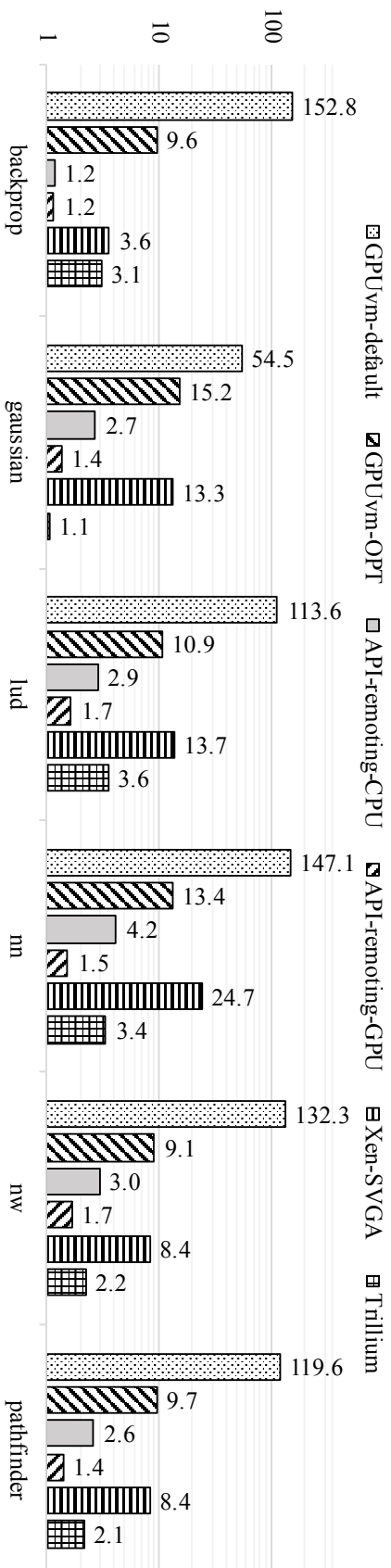


Figure 3.4: End-to-end execution times of benchmarks on virtualization prototypes, relative to end-to-end execution time on the NVIDIA CUDA runtime in a native setting. gRPC overhead is removed from the reported measurements, which is up to 10% of the total execution time for API remoting, and 40% for Trillium.

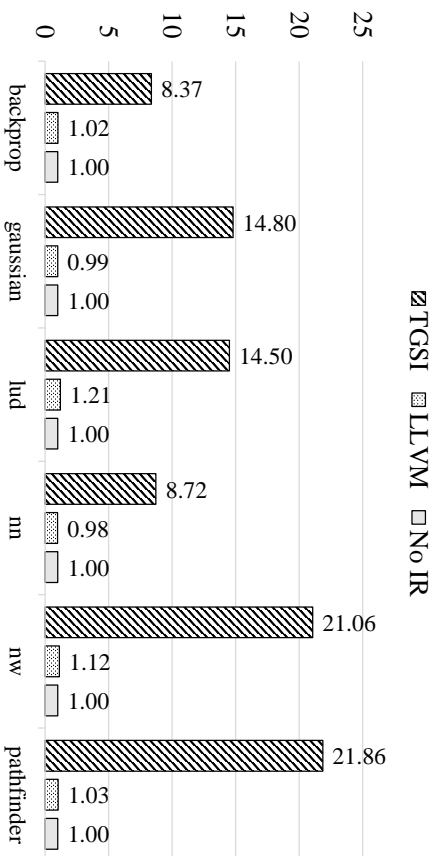


Figure 3.5: Kernel execution slowdown due to virtual ISAs. TGSI: the LLVM TGSI back-end compiler used in Xen-SVGA. LLVM: LLVM NVIDIA PTX (NVPTX) back-end used in Trillium. No IR: native NVIDIA compiler.

3.3 Evaluation

3.3.1 End-to-End

Figure 3.4 depicts one of the major results from this chapter: an end to end empirical analysis of systems representative of all of prior canonical virtualization schemes. GPUvm [141] stands in for full-virtualization designs, in its default configuration (GPUVM-DEFAULT), and in its fully optimized configuration (GPUVM-OPT). API-REMOTE-GPU and API-REMOTE-CPU represent user-space API remoting schemes, while XEN-SVGA represents the SVGA design. TRILLIUM is very similar to XEN-SVGA, other than that it elides ISA virtualization (the use of the TGSI vISA).

Figure 3.4 shows the end-to-end execution time (relative to native GPU execution) for the six chosen benchmarks for all the systems evaluated. As expected, traditional API remoting designs incur the lowest overhead, which is achieved by giving up hypervisor interposition. GPUVM-OPT exhibits about $9.1\times$ slowdown for applications with short-lived kernels (e.g. Needleman-Wunsh algorithm); the overhead can be as high as $15.2\times$ when the workload has long-running kernels (e.g. Gaussian Elimination), proving that trap-based virtualization schemes are doomed to squander the raw performance that makes accelerators desirable in the first place. XEN-SVGA fares better than GPUVM-DEFAULT and GPUVM-OPT, but performs worse than API-REMOTE-GPU, TRILLIUM, and in some cases even API-REMOTE-CPU. XEN-SVGA is sensitive to the performance lost in GPU kernel code resulting from redundant compilation through TGSI (See Figure 3.5 for deeper analysis).

We find that remoting calls 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 ($1.6\times$ 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.

Overall, it appears that user-space API remoting introduces the lowest overhead when multiplexing Domain Specific Accelerators. However, the lack of a viable interposition point means that the

hypervisor is out of the loop and can't make scheduling and resource allocation decisions, or ensure isolation.

3.3.2 Impact of vISA choice

Our hypothesis that ISA virtualization is redundant for GPGPUs (and DSAs generally) is confirmed by the fact that TRILLIUM out performs XEN-SVGA in the empirical results presented in Figure 3.4. 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 used has a high-fidelity view of the physical hardware.

Typically, DSA execution/compilation frameworks are 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 for NVIDIA SASS kernels generated in 3 ways:

- using the Mesa3d OpenCL stack (OpenCL→ TGSI→ SASS),
- using the LLVM OpenCL stack (OpenCL→ LLVM IR→ SASS),
- and using the native NVIDIA OpenCL compiler (OpenCL→ SASS).

These measurements are reported in Figure 3.5 relative to kernel execution time in a native setting. Code generated from TGSI IR is dramatically slower in all cases than code generated by the NVIDIA OpenCL framework. We observe slowdowns of up to $22\times$, with a harmonic mean of $13\times$ across the 6 benchmarks that were optimized for evaluation.

While we predicted the basic trend these experiments show, we were surprised by the magnitude of the difference. We found quality of the kernel generated by the LLVM NVPTX compiler to be comparable to native, at least in terms of execution time. This is unsurprising given recent efforts [158] to optimize the LLVM tool-chain for NVIDIA GPUs.

3.4 Conclusion

Virtualizing GPGPUs is a balancing act: there are multiple design points each offering a different trade-off of key virtualization properties. This chapter provides the first (to our knowledge) comprehensive empirical and qualitative comparison of a wide range of fundamental virtualization techniques from the literature. We implemented 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 [11] project. We proposed an improved design called TRILLIUM that removes the necessity for the vISA defined by SVGA resulting in dramatic performance improvements. 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.

While TRILLIUM exhibits greater overhead than the API-remoting schemes evaluated, it presents existence proof of a an unexplored point in the DSA virtualization design space: hypervisor-interposed API-remoting.

CHAPTER 4: IEMTS — A NEW ACCELERATOR VIRTUALIZATION TAXONOMY

The qualitative and empirical analysis presented in prior chapters implies that no virtualization technique preserves performance, compatibility, interposition, and isolation for GPGPUs. User-space API remoting based designs exhibit low overhead, but eschew hypervisor interposition entirely. The best performing design that retained hypervisor interposition, TRILLIUM, consistently introduced a $2\text{--}3\times$ overhead over the user-space API remoting design. It would appear that there exists no coveted “sweet spot” in the GPGPU virtualization design/property space.

This chapter hypothesizes the existence of a “sweet spot” in the GPGPU virtualization design space, and makes the case for this position based on empirical analysis of TRILLIUM, and a novel taxonomy: IEMTS.

4.1 Understanding the sources of overhead in TRILLIUM

The empirical results presented in Chapter 3, show that TRILLIUM is $2\text{--}3\times$ slower than API-REMOTE-GPU, the user-space API remoting system. The analysis presented in earlier, however, doesn’t offer any insight into the source of this overhead. While TRILLIUM eliminates one source of overhead in the XEN-SVGA design, it appears that there is at least one other design decision that is of questionable merit. Let us revisit the TRILLIUM design in order to identify this questionable design decision.

Figure 3.3b presents a block diagram overview of TRILLIUM as prototyped on the Xen hypervisor. Notice the “shadow pipe” that acts as a connection point between the driver in Dom1 (Application VM) and the driver in Dom2 (service VM). The pipe driver is the back-end of the GNU/Linux Gallium3D graphics driver framework. We chose to interpose on all calls between the front- and back-ends of the Gallium3D framework primarily because this split-driver design is how many prior systems (including SVGA) are implemented.

Our decision to keep the split-driver design, i.e., to interpose on the interaction between the front and back-ends of the GNU/Linux kernel graphics driver, led to a simple performance pitfall:

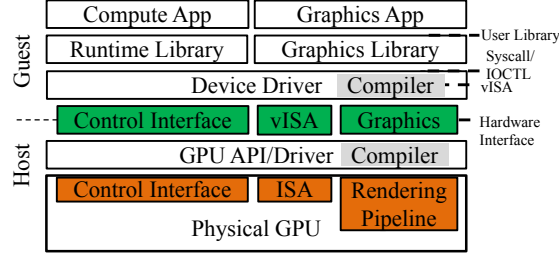


Figure 4.1: Possible points of interposition.

The design of the Gallium3D driver is such that each CUDA or OpenCL API call made by the user program gets decomposed into multiple pipe-driver (Gallium3D back-end API) calls. This is commensurate with the general idea of abstraction: lower level APIs tend to be more granular and less expressive, whereas higher level APIs (such as OpenCL) tend to be more expressive. Anecdotally, we found that most OpenCL calls were being decomposed into at least 3 pipe-driver calls, which is commensurate with the performance difference we observed in our empirical analysis.

4.2 Where to interpose?

GPGPUs (and generally DSAs) are controlled through vendor-specific programming frameworks (often more than one), key parts of which are often proprietary. Research to-date on first-class OS abstractions for GPUs [123, 124, 134, 84, 85, 91] has not impacted practice, and GPU software stacks actively avoid or bypass system software. Moreover, in contrast to many other common devices, GPUs have *multiple* interfaces, that may require virtualization: Figure 4.1 shows potential interposition points in a GPGPU’s software stack: the user-space library, the syscall/ioctl interface the user-space API operates on, within the kernel driver, and the hardware interface. Typically, lower layers present finer granularity of interposition for the hypervisor, but also sacrifice performance. What then is the right level of abstraction to interpose?

4.3 IEMTS: A new analysis framework

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 [155]. In order to select the level of abstraction to interposition for virtualizing a GPGPU, we need a framework to characterize the nature of interposition in canonical techniques.

We posit that the current *de facto* taxonomy and property trade-off space are illustrative but not informative for indentifying the right abstraction to interpose for DSA virtualization. First, lassifying 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*). It is common to see systems compared to other systems intended as exemplars for a technique or design pattern. Ironically, the end-to-end comparison presented in the prior chapter on TRILLIUM is guilty of exactly this: we compared TRILLIUM against other systems intended to represent “full virtualization”, “API remoting”, and so on. Methodologically, such a comparison is laudable: evaluating a design exhaustively against plausible alternatives in an end-to-end setting is fundamental to good science. However, its value for drawing out fundamental insights is scant because “full virtualization” only talks about the quality of the guest-visible abstraction, while findings involving “API remoting” are really about the particular API in question. “Para-virtualization” is often cast as a design dimension, ultimately grouping designs that share no core techniques, such as SVGA and GPUvm. As a framework within which to seek higher-level insights about design, a rough taxonomy that fails to cleanly separate most concerns is in unacceptable. Further, production systems typically compose multiple virtualization techniques in order to leverage the best properties of each technique, especially in the presence of multiple interfaces (e.g., VMware

	GPUvm	VMware SVGA Control Interface	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 4.1: Comparing virtualization designs using the **IEMTS** framework.

SVGA [59] exposes a para-virtual device abstraction, which it then uses to “remote” DirectX calls from the VM to the host. SVGA also features ISA virtualization.). 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.

In order to determine the right layer of abstraction to interpose for DSA virtualization, it is instructive to consider prior art through the following interposition related properties, collectively known as the **IEMTS** framework:

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

Table 4.1 presents analysis of three prior GPU virtualization systems under the **IEMTS** framework. The **IEMTS** framework readily offers a number of key insights into the pros and cons of each interposition interface: GPUvm’s [140] performance woes arise from its reliance on *trapping-and-emulation* of guest MMIO accesses. VMware’s SVGA has two entries in the table because there are two interfaces being virtualized: the control interface (the DirectX/OpenGL API) and the shader ISA. Explicitly separating the two interfaces shows that our earlier observation about ISA-virtualization being unnecessary for accelerator virtualization is exactly on point. Further, this separation also makes obvious that control interface virtualization in SVGA and the API-remoting in rCUDA are very similar with two key differences: the source endpoints, and the transport used. SVGA (and in turn TRILLIUM) and rCUDA [153, 61] are both forwarding framework APIs. However, they do

	TRILLIUM	rCUDA	HIRA
Interposed Interface	Pipe-Driver API	User-space API	User-space API
Interposition Source	Guest driver (front-end)	Guest Library	Guest Library
Interposition Destination	Host driver (back-end)	Host/Server Daemon	Host/Server Daemon
Interposition Mechanism	Custom Guest Driver	Guest Library Shim	Guest Library Shim
Transport	Hypervisor FIFOs	RPC	Hypervisor channel
Synthesis	Call host driver API	Invoke API on host	Invoke API on host

Table 4.2: A possible “sweet spot” in the GPGPU virtualization design space?

so over different transports: SVGA implements the transport layer over hypervisor-managed FIFO queues, enabling hypervisor interposition that is impossible with the RPC transport used by rCUDA. SVGA’s design mandates modifications to guest libraries and drivers. This lost compatibility is retrieved through non-technical means: VMware’s SVGA driver is integrated into commodity OSes. VMware also maintains the Linux graphics stack, enabling it to ensure compatibility.

4.4 Conclusion

We posit that a “sweet spot” might exist in the DSA virtualization design space. Table 4.2 presents this novel technique, Hypervisor Interposed Remote Acceleration (HIRA), in terms of the IEMTS framework and compares it to TRILLIUM and rCUDA. HIRA combines properties of both of these designs: HIRA interposes the user-space API as in a user-space API-remoting system like rCUDA, thereby enabling low-overhead interposition; HIRA forwards interposed API calls to the daemon on the host through a hypervisor controlled channel (like the FIFO used in SVGA and TRILLIUM), thereby preserving hypervisor control over the DSA. Chapter 5 considers this design in detail.

CHAPTER 5: HYPERVISOR INTERPOSED REMOTE ACCELERATION

Practical DSA virtualization must support sharing and isolation under flexible policy with minimal overhead. The structure of current accelerator stacks makes this combination extremely difficult to achieve. Accelerator stacks are *silos* (see § 2.2.2) comprising proprietary layers communicating through memory mapped interfaces. This opaque organization makes it *impractical* to interpose intermediate layers to form an efficient and compatible virtualization boundary (§2.1). The remaining interposable interfaces leave designers with untenable alternatives that sacrifice critical virtualization properties such as interposition and compatibility, as we saw in prior chapters.

This chapter explores a novel hypervisor virtualization design, *Hypervisor Interposed Remote Acceleration* (HIRA), which we argued represents a “sweet-spot” in the DSA virtualization design space in the previous chapter (§ 4). We explore how the silo-ization of DSA software and hardware stacks leaves only one viable software interface for interposition: the user-space API (this dissertation is focused on software virtualization techniques for DSAs, and doesn’t explore hardware support for virtualization). We describe how HIRA enables us to combine the low overhead and ease-of-evolution of API-remoting with the isolation, and effective policy enforcement of hypervisor interposition.

The API-agnostic para-virtual stack, API-specific API servers, and hypervisor-level resource management espoused by HIRA, are combined with LAPIS (a Domain-Specific Language used to capture semantic information of the APIs interposed), and a compiler for LAPIS to automate construction and deployment of guest libraries in our prototype: AVA. AVA uses the 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 [4, 80, 153, 61, 121], AVA preserves hypervisor-level resource management and strong isolation by forwarding API calls over hypervisor-managed communication channels, inserting automatically-generated resource management components at the transport layer to enforce policies from a DSL specification. Critically, *automation* from AVA enables hypervisors to keep up with fast

accelerator evolution: automatic generation of components dramatically shortens the development cycle.

AVA supports a broad range of currently-shipping compute offload accelerators. We virtualized nine 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 [59] or BitFusion’s FlexDirect [4]. AVA provides near-native performance (e.g., 2.4% slowdown for TensorFlow and 5.6% for CUDA), enforces isolation and fair sharing (§2.1) across guests, and supports live migration. AVA is available on GitHub [utcs-scea/ava](https://github.com/utcs-scea/ava).

This dissertation describes the core virtualization technique, HIRA, and provides a brief overview of the design, implementation, and evaluation of AVA, as necessary, to discuss the efficacy of HIRA as a virtualization technique. This document elides detailed discussion of AVA’s DSL, LAPIS, and various other automation techniques used to recover compatibility in AVA. For detailed treatment of these topics, please refer to the published papers on AVA [163, 164], and Hangchen Yu and Arthur Peters’ dissertations.

5.1 Accelerator Silos

Domain Specific Accelerator software stacks are effectively *silos*, interposable only at the top (user-space API) and at the bottom (hardware support for virtualization). Intermediate layers of DSA software stacks *cannot be practically separated* to virtualize the device. The software stacks of Domain Specific Accelerators are typically composed of a library that exposes a user-space API, and a runtime and a kernel driver that manage and orchestrate computation on the DSA. These components typically communicate via explicitly unstable (and often undocumented) interfaces and protocols, which enables the vendor to preserve forward compatibility easily. Where possible, DSA software also typically uses kernel-bypass communication techniques to eliminate overheads from the kernel. Scheduling and resource allocation for DSAs are typically under the control of a combination of the proprietary CPU-based runtime (user-space and kernel), and the controller on the DSA itself, not the CPU-based Operating System. DSAs typically contain an entire software stack on the hardware module, which is hidden from the programmer. This software is used to implement

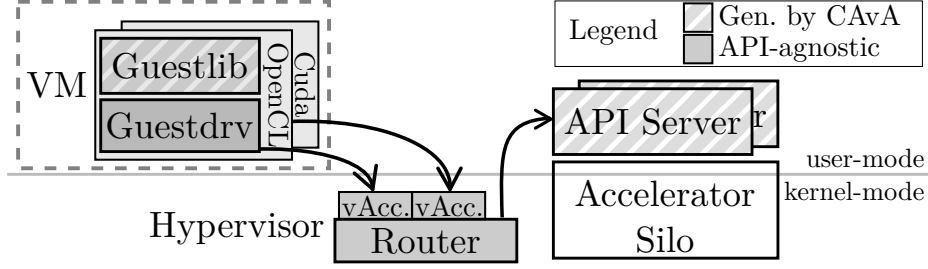


Figure 5.1: Overview of Hypervisor Interposed Remote Acceleration. Components with striped backgrounds are API-specific and are generated from an API specification. Components with solid backgrounds are API-agnostic and only need to be implemented once per hypervisor (or per hypervisor-OS combination).

a command-based programming interface that enables the vendor’s CPU-based control software for scheduling and resource allocation. End users work with the DSA almost entirely in higher level languages, typically the language the DSL is embedded in (e.g., Python for Google’s TPU, C/C++ for Nvidia and AMD GPGPUs). The DSL/API is supported by a compiler (which converts the user’s program to the the ISA of the computation units on the DSA), a user-space runtime and a kernel module (which work together to implement the user-space API, and control the DSA).

Chapter 3 showed that virtualization designs that interpose these opaque and frequently-changing interfaces are *impractical*: these design depend on inefficient interposition techniques and yield solutions that sacrifice *compatibility*.

The user-space API is the *only* stable and efficiently interposable software interface for DSA *silos*. From our analysis of the design space using the IEMTS framework in Chapter 4, we also identified an opportunity to recover or compensate for interposition and compatibility, virtualization properties that are traditionally sacrificed by user-space API remoting. Interposition can be recovered by using a hypervisor-managed transport, instead of a user-space RPC channel. The hypervisor-managed channel also represents a central interface at which to enforce resource management policies.

5.2 Design

The resulting novel design, *Hypervisor Interposed Remote Acceleration (HIRA)* is shown in Figure 5.1. Applications in the guest VM link against a custom “Guestlib” which is a drop-in replacement for the API the application uses (e.g., OpenCL, CUDA). The “guestlib” traps API calls (*the interposition mechanism*) and serializes their parameters and any relevant environment

information to a buffer that is exposed to it by a custom driver (shown as “Guestdrv” in the figure). A single “guestdrv” driver is reusable across all API frameworks in the guest (the same driver can be loaded for each API for improved security), and importantly only one driver needs to be implemented for a given OS-hypervisor combination. This driver interfaces with the “vAcc” virtual accelerator that the hypervisor presents guest VMs with. The “vAcc” is an abstract virtual device that exposes the typical hardware interface (MMIO Base Address Registers, Command Queues, etc.), and is used as *the source endpoint* for routing communication through the hypervisor. Importantly this virtual device is *not a virtual accelerator* in that it isn’t specialized to any one DSA. Like the “guestdrv” driver, only a single implementation of this virtual device is necessary per hypervisor. The virtual accelerator acts as *the destination endpoint* of the hypervisor managed transport for interposed API calls. The method used to communicate between the guestdriver and the virtual accelerator is considered the *interposition transport*: this could be a FIFO backed by shared memory, or a network based transport (TCP/IP). All serialized API calls received by the “vAcc” are then processed by, a hypervisor module (“Router” in the Figure) that enforces resource control and scheduling policies at the granularity of the API calls. Semantic information about the interposed API function is captured in the API specification written in LAPIS. The API calls are then forwarded to a per-VM “API server” that executes the API call on the vendor’s API framework (“Accelerator Silo” in the figure) in the host. Results of the API invocation are captured, serialized, and transported back to the guest application from the host.

Using hypervisor-managed transport recovers interposition, but also complicates compatibility and introduces engineering effort: HIRA requires custom guest libraries, guest drivers, and API servers for each OS and API, and API-specific resource-management code in the hypervisor (policy code for the “Router”). Our prototype implementation, AVA, mitigates this with automated construction (§5.5). Automatically generating code to implement HIRA components presents several challenges which follow from the need to specify API semantics and policies for which existing Interface Description Languages (IDLs) [98, 104] are not applicable. AVA uses a DSL called LAPIS, a compiler called CAVA, and device-agnostic transport components to address these challenges.

AVA targets *compute offload* accelerator APIs, such as OpenCL, CUDA, and TensorFlow, which control an accelerator explicitly through data transfer and task creation interfaces. AVA consists of API-agnostic para-virtual stack components to implement transport and dispatch for API remoting,

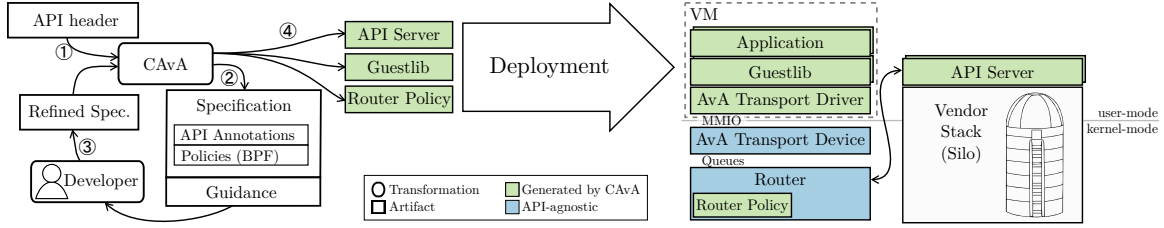


Figure 5.2: Overview of AVA.

API-specific components that interpose and execute the API functions, and a compiler—CAvA, which generates the API-specific components from a specification of the API. The API specification is written in a new high-level specification language, LAPIS, that is used to both capture both the syntax and semantics of the API. The “Router” may be deployed in the hypervisor or in an appliance VM to support type I and II hypervisors [117].

5.3 AVA Components

Figure 5.2 provides an overview of the interaction between the various components in a AVA-generated design, and the work-flow to support a new API with AVA (§5.4) and the AVA stack.

The **guest library** is generated by CAVA from the LAPIS specification. It provides the same symbols as the vendor library for the application to link against. The guest library intercepts all API functions called by guest applications, marshals function arguments and implicit state, and forwards the call through the transport channel for remote execution.

The **guest driver** interacts with the virtual transport device exposed to the VM and provides a transport channel endpoint in the VM. Each guest-driver manages a set of command queues that are used to forward API calls to the API server via the router. CAVA generates a separate transport driver instance for each API framework to preserve cross-framework isolation and guest OS interposition.

The **virtual transport device** is an abstract device exposed to the guest to forward API calls between the guest and the API server. The virtual transport device exposes MMIO BARs for control (Our implementation uses 4 BAR registers: one stores `vm_id`, one stores the physical address of a buffer used to implement zero-copy, one to notify KVM when the `guestdrv` is installed, and the last to notify KVM when an app is spawned.) and a command queue interface. It is API-agnostic, and its purpose is to provide an interposable interface for the hypervisor.

The **router** is an API-agnostic extension to the hypervisor that implements AVA’s interposition logic. The router performs security checks and enforces scheduling policies on forwarded API calls according to the LAPIS specification.

The **API server** is an API-specific user-space process generated by CAVA. It runs either in an appliance VM (with PCIe pass-through access to the physical device) or in the host. The API server executes forwarded calls on behalf of the guest application. A given API server is dedicated to a single guest process, leveraging process-level isolation when the hardware supports it to guarantee fault and device context isolation.

The router, the API server, and vendor device drivers are considered part of the Trusted Computing Base, while the guest library and the guest virtual transport device driver are untrusted. The router may rely on the API server to provide semantic information that may be specific to a given API, e.g., the API server may provide device resource accounting information to the router for resource management.

5.4 Developer Work-flow

AVA’s API-agnostic components must be implemented for each hypervisor, along with the guest drivers needed for each supported guest OS. The development effort to build them is amortized across all of the accelerators and framework APIs supported.

AVA’s API-specific components are generated from LAPIS by CAVA to plug into AVA’s API-agnostic components. LAPIS (§5.5) is used to annotate the functions in an API and provide an overview of how the API controls accelerator resources. This semantic information enables automatic construction of the API remoting system.

Figure 5.2 shows the work-flow to support a new API with AVA. First, CAVA automatically generates a preliminary LAPIS specification from the unmodified API header file. The programmer refines the specification with guidance from CAVA; adding information missing from the header file, e.g., buffer sizes or implicitly referenced state. Once the developer is satisfied with the API specification, she invokes CAVA to generate code for the API-specific components and the customized driver. CAVA also generates deployment scripts. When a new version of an API is released, the same process can be used, starting with the previous specification.

5.4.1 Communication Transport

AVA relies on an abstract communication channel that defines how interposed API calls and their associated data buffers are sent to the host, and results are sent back, validated, and received. The channel provides an interposition point to track resources and invoke control policies. Using an abstract interface allows hypervisor developers to choose the best available communication transport (e.g., shared memory FIFOs vs RDMA). While the channel explicitly requires that all communication between the guest and the API server must take place through the router, no assumptions are made about the actual location of components, which may be disaggregated. The transport handles two types of payloads: *commands* which contain opaque arguments and metadata for calls (e.g., thread ID and function ID) and *data* which contains buffers referenced from the arguments (e.g., bulk data). As communication is bidirectional in a virtualization system, AVA also provides support for function callbacks from the API server to the guest library. Callbacks are fundamental in many frameworks, e.g., TensorFlow, and must be run in the guest VM.

5.4.2 Sharing and Protection

AVA re-purposes process-level isolation mechanisms (e.g. memory protection) provided by the accelerator silo (when available) to simplify supporting cross-guest isolation. We anticipate that emerging accelerators will support process-level isolation, as all GPUs do today. While the lack of hardware virtualization support represents an obstacle for accelerator virtualization, DSA stack structure is just as important. Even if all accelerators were to support process-level virtualization in hardware, a software based API-remoting scheme like AVA will still be necessary unless vendor stacks expose the interfaces required to take advantage of that support.

For accelerators that do not natively support process-level isolation, AVA can still share the device: semantic information from additional LAPIS descriptors enable us to generate code that supports coarse-grain time-sharing. Metadata on the API functions that create and destroy connections to the hardware allows the CAVA to insert additional logic in the device open/close calls to transparently spin until the device becomes available. This solution admits no concurrency between tenants, but enables protected sharing for devices which cannot otherwise be shared.

5.4.3 Scheduling and Resource Allocation

AVA can enforce policies (e.g., rate limiting) on shared resources, e.g., device execution time, by tracking resource consumption and invoking policy callbacks that change how API calls from guests are scheduled. We envision the virtualization developer providing these policy callbacks in the LAPIS specification, and using annotations on the functions of the APIs to identify how resources are consumed. When such an annotation is unavailable, AVA falls back to coarse-grained estimation of resource utilization, for example, using wall-clock time to approximate device execution time.

5.4.4 Memory Management

To support memory sharing on devices with onboard memory, LAPIS resource usage annotations on the functions of an API enable generated code to track the device memory allocated to each guest VM. Resource accounting code is auto-generated from semantic knowledge of the device memory allocation APIs provided by annotations in the API specification (e.g., memory types and how to compute size of buffer). This enables the hypervisor to enforce policies at the granularity of API function calls. For example, If a device memory allocation request would exceed the guest's quota, the hypervisor instructs the API server to return the appropriate Out-of-Memory (OOM) error for the allocation request.

5.5 CAVA and LAPIS

The AVA toolchain comprises a language, **Low-level API Specification language (LAPIS)**, a compiler (CAVA), and a runtime library. CAVA accepts API specifications written in LAPIS, and generates C source for an API-specific remoting stack that implements the HIRA design. LAPIS extends C declarations with descriptors to express a broad range of semantic information necessary to generate that stack. This includes information captured by traditional IDLs (e.g., function parameter semantics and data layout), as well as information required for accelerator virtualization that is not expressible in existing IDLs.

To understand how AVA relates to other IDL-based API-remoting techniques, it is useful to compare it to the Sun Network File System [126]. NFS supports remote access to the the file system using an IDL to specify API semantics and a compiler to automate code generation. NFS and AVA

share a number of key challenges. Both must marshal and transfer function calls and arguments, handle asynchrony, refactor functionality and (potentially implicit) state across newly decoupled components. Both must preserve the resource management and sharing support expected for a resource managed by system software.

However, key differences between AVA and NFS arise around additional virtualization requirements and limitations in the design space. For example, DSA virtualization requires that AVA be able to capture of sufficient API-level state to enable features like live VM migration. Key techniques used by NFS to deal with implicit state and resource management are impractical for AVA. NFS mostly *eliminates* implicit state by altering the API, e.g. replacing functions using implicit seek pointers with stateless read/write functions and offset parameters. To deal with resource management, sharing, and compatibility challenges, NFS *introduced* the VFS layer, providing an application-transparent interposition point at which to centralize or delegate that functionality using code written to run at that layer. AVA cannot alter APIs, so it exposes language-level features for dealing with implicit state. AVA cannot introduce new abstraction layers due to vendor opacity and API diversity: so the resource management and sharing policy must be expressed at the language layer as well. This dissertation elides most details of the LAPIS language and the corresponding runtime. The discussion below only presents topics that are important to understand our implementation of HIRA. For detailed treatment of these topics, please refer to the published papers on AVA [163, 164].

5.5.1 Resource Management and Policy.

LAPIS supports descriptors to express the resources consumed by API functions. Resources may be either *instantaneous* or *continuous*. Instantaneous resources are typically consumed by an API function only while executing. Continuous resources are typically assigned to a client for a period of time (e.g., Memory allocated using CudaMalloc). Instantaneous resource accounting can be implemented by measuring resources used at each function invocation, while accounting of continuous resources requires tracking of resource assignment to each client/VM across API functions. The resource descriptors provided by LAPIS allow the hypervisor to perform book keeping for resources at an API function granularity and enforce fairness and sharing policies.

To specify policies, developers provide functions that schedule API calls from different VMs based on the recorded resource usage of those VMs. In our current implementation, policy functions

are specified as eBPF programs stored in a separate file and referenced from the LAPIS source. We currently use eBPF because it enables unprivileged code to run safely in the hypervisor and is available today, enabling AVA to be used without modifying the hypervisor and without trusting the developer.

To enforce resource sharing requirements, the code generator changes how API calls are handled by inserting accounting code in the router and hooks to call programmer-provided policy functions. For continuous resources, the generated code may need to generate an artificial failure in response to an allocation request. This requires that the compiler know how to fake a failure by constructing return values and/or executing specific code to change the library state. For instantaneous resources, enforcement is implemented by delaying certain calls until other VMs have a chance to perform their instantaneous operations.

5.5.2 Shadow Buffering

The AVA API-agnostic components provide shadow buffer management primitives that the generated code uses to maintain API server-side shadows of application buffers. AVA's shadow buffers function as a caching layer that can buffer updates and apply them in batch. In most cases, copy operations to synchronize shadow and application buffers are required only at API call boundaries, so AVA-controlled buffers are transparent to the guest, work without true shared memory between the guest and API server, and are faster than page-granularity software shared memory. In cases where updates must be made visible in the guest without an API call to serve as a synchronization point, true shared memory between the guest lib and the API server can be specified using LAPIS's zerocopy support.

5.5.3 Mapped memory

AVA does not currently map API server host memory into guest application space by default. However, AVA still supports applications that use device-mapped memory by copying data between the guest and API server. The implementation uses LAPIS descriptors to track mapped buffers and ensure they are always passed as contextual arguments to synchronization functions, e.g., `cuSynchronizeStream`. Importantly, the technique respects the semantics of the API: even without AVA the only way an application can *guarantee* that device writes are visible to the application is to call a synchronization function. However, some GPUs do make writes visible

between synchronization functions and research systems rely on it to implement accelerator-driven communication (e.g. GPUfs [134]), but will not function correctly with AVA.

5.6 Implementation

We prototyped AVA on Linux kernel 4.14.0 with QEMU 3.1.0 and LLVM 7.0. Our resource management modifications to the KVM hypervisor took 1,500 LoC. We modified the QEMU *virtio-vsock* device and the corresponding *vhost-vsock* host driver to enable interposition. The para-virtual transport device, which is used for both interposition and transport, was built as a QEMU display adapter (500 LoC). The guest driver is 500 LoC long, each transport channel is about 400 LoC on average, the CAVA was implemented in 3,200 LoC of Python code. Other libraries accounted for 2,000 LoC.

5.6.1 Transport

AVA supports several interchangeable transports, allowing it to support disaggregated hardware via *sockets*, as well as local execution via guest-API server *shared memory*. The *socket* transport uses either a TCP/IP network socket or an inter-VM socket (VSOCK) to transport commands and data. The socket layer copies data multiple times, and incurs queuing delays. *Shared memory* provides efficient data transfer when the guest and the API server are on the same physical machine. The hypervisor exposes a contiguous virtual buffer to the VM through the virtual transport device PCIe BAR (base address register). The guest para-virtual driver manages the virtual BAR and assigns a partition to each guest application. AVA uses the shared buffer to transport buffers to the API server, but still uses a socket (currently VSOCK) to transport commands to retain hypervisor interposition.

5.6.2 Hypervisor Interposition and Mediation

AVA enables hypervisor mediation by interposing the transport channel. We extended QEMU *virtio-vsock* [125, 73] (a host/guest communication device) to build the virtual device. The corresponding *vhost-vsock* host driver was extended to perform interposition during packet delivery.

When forwarding an API call, the command is always sent on the modified VSOCK channel, while the argument buffer can be transferred via either VSOCK or guest—API server shared memory.

Transferring the command via VSOCK provides a doorbell to the router—the router then schedules the invocation based on resource limits. The API server and guest application have unfettered access to the shared memory, but the API server does not know what the requested operation is or where the buffers are until the hypervisor forwards the command.

5.6.2.1 Policies in eBPF

AVA supports policies written as eBPF(Extended Berkeley Packet Filter [103]) programs. We defined a new eBPF program type that can be loaded into KVM via `ioctl`. AVA reuses the same eBPF instruction set as socket filtering, and leverages the unmodified LLVM compiler to compile the eBPF program. The eBPF verifier had to be modified to verify the memory accesses of the new type of program. We provide helper functions for AVA eBPF programs. Leveraging eBPF allows AVA to take advantage of eBPF program verification at a very low cost (4.3% of AVA’s internal overhead). The current implementation computes resource utilization in the API server and then reports this utilization to the hypervisor.

5.6.2.2 Scheduling

AVA provides a weighted fair queuing (WFQ) scheduler, with two rate control algorithms. Each VM v sharing the device is configured with one share s_v . v ’s average device time usage is $L_v = s_v / \sum_{v \in V} s_v$, where V is the set of running VMs. VM v ’s device utilization time is accumulated into $T_v(t)$ in the time window $[t, t + 1)$. If a VM’s device usage time exceeds its share s_v , API calls from v will be postponed until its utilization proportion becomes lower than the threshold. The scheduling window is 500 ms (or the interval between two adjacent calls), and device utilization is updated upon every API completion. AVA supports the following algorithms:

Fixed-rate polling where the delay is a fixed interval d (usually longer than the time window).

Feedback control where the adaptive delay, $d_v(t)$, is computed by the additive-increase multiplicative-decrease (AIMD) algorithm [1] below ($a = 1$ ms and $b = 1/2$; see §5.7.5).

$$d_v(t+1) = \begin{cases} d_v(t) \times b, & \text{if VM } v \text{ exhausted its share} \\ d_v(t) + a, & \text{otherwise} \end{cases}$$

5.6.3 Shadow Resources

AVA supports threading and long-lived buffers by shadowing them in the API server. The API server spawns a shadow thread when a new guest thread makes its first API call, and reuses it for all future calls from that thread. For synchronous calls, the guest thread will be blocked while the shadow thread executes the call. Shadow threads are destroyed when the original thread is destroyed or when the guest application exits. Similarly, the worker allocates a new shadow buffer when it is first notified of a buffer annotated with a long lifetime and deallocates it when the application calls a function annotated with the corresponding annotation. Reverse shadows, guest library buffers, and threads which shadow an API server resource, are supported in the same way.

5.6.4 Callbacks

When an API registers a callback, the guest library stores both the original application `userdata/tag` value and the function pointer in a buffer. This buffer is then supplied as the `userdata` argument to the API server. The API server registers a generated stub function with the accelerator API. When the API framework calls the stub in the API server, a callback is made to the guest library with the guest library buffer as the `userdata` argument. The guest library finally extracts the original application `userdata` value and function pointer and performs the call back into application code. The call to the guest library uses the same protocol as calls to the API server, so all features of AVA apply to callbacks. For example, callbacks block the API server thread that called them if the callback is synchronous.

5.7 Evaluation

AVA was evaluated on an Intel Xeon E5-2643 CPU with 128 GiB DDR4 RAM, using Ubuntu 18.04 LTS, and Linux 4.14 with modified KVM and vhost modules. Guest VMs were assigned 4 virtual cores, 4 GiB memory, 30 GB disk space, and ran Ubuntu 18.04 LTS with the stock Linux 4.15 kernel. API servers and VMs were co-located on the same server for all experiments except live migration and the FPGA benchmarks. Experiments involving a Google Cloud TPU were carried out on a Google Compute Engine instance with 8 vCPUs (SkyLake), 10 GiB memory, and a disaggregated Cloud TPU v2-8 in the same data center. The guest VM was located on the same instance via nested

API	Gen	#	LoC	Churn	Benchmark	Hardware
OpenCL 1.2	✗	39	7514	14318	Rodinia	NVIDIA GTX 1080
	✓	38	1060	2868		AMD RX 580
CUDA 10 Driver	✓	16	266	410	Rodinia	NVIDIA GTX 1080
CUDA 10 Runtime	✓	93	1358	1973	Rodinia	NVIDIA GTX 1080
TensorFlow 1.12 C	✓	46	501	887	Inception	NVIDIA GTX 1080
TensorFlow 1.13 Py	✗	n/a	3245	5972	VGG-net Inception	Google Cloud TPU v2-8
TensorFlow 1.14 Py	✓	111	1865	2557	Neural networks	NVIDIA GTX 1080
TensorFlow Lite 1.13	✗	n/a	1295	2005	Official examples	Coral Edge TPU
NCSDK v2	✓	26	479	1279	Inception	Movidius NCS v1
GTI SDK 4.4	✓	38	284	568	Official examples	Gyr Falcon 2803 Plai Plug
Custom FPGA on AmorphOS [86]	✓	4	30	40	BitCoin	AWS F1
QuickAssist 1.7	✓	19	444	676	QATzip	Intel QAT 8970
HIP	✓	41	624	990	Galois [114]	AMD Vega 64

Table 5.1: Development effort for forwarding different APIs, along with the benchmarks [144, 135, 48] and hardware used to evaluate them. The # column indicates the number of API functions supported. The Python APIs are forwarded dynamically, making # inapplicable. **Gen** indicates whether the API forwarding was generated by CAVA or was written by hand. **LoC** is the number of lines of code (including blank lines and comments) in the CAVA specification or C/Python code. **Churn** is the total number of lines modified in commits.

virtualization. Experiments for the custom FPGA API were done on an AWS F1 `f1.2xlarge` instance (with 1 Virtex UltraScale+ FPGA, 8 vCPUs, and 122 GiB memory). For live migration, a second similar server was used as the remote machine, and the servers were directly connected by 10 Gigabit Ethernet.

5.7.1 Development Effort

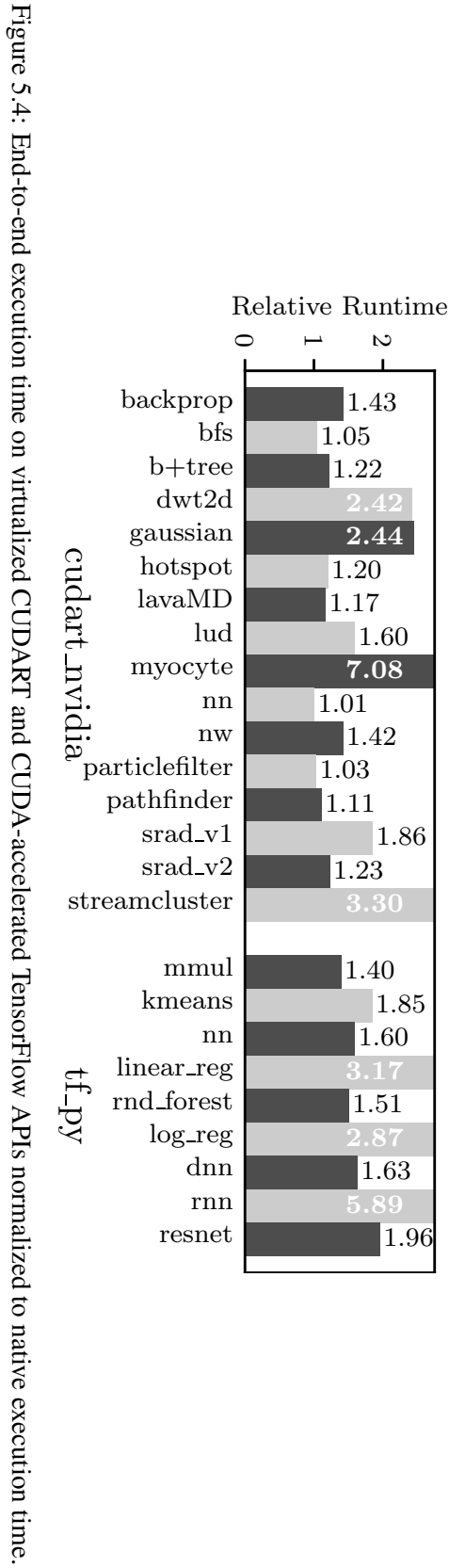
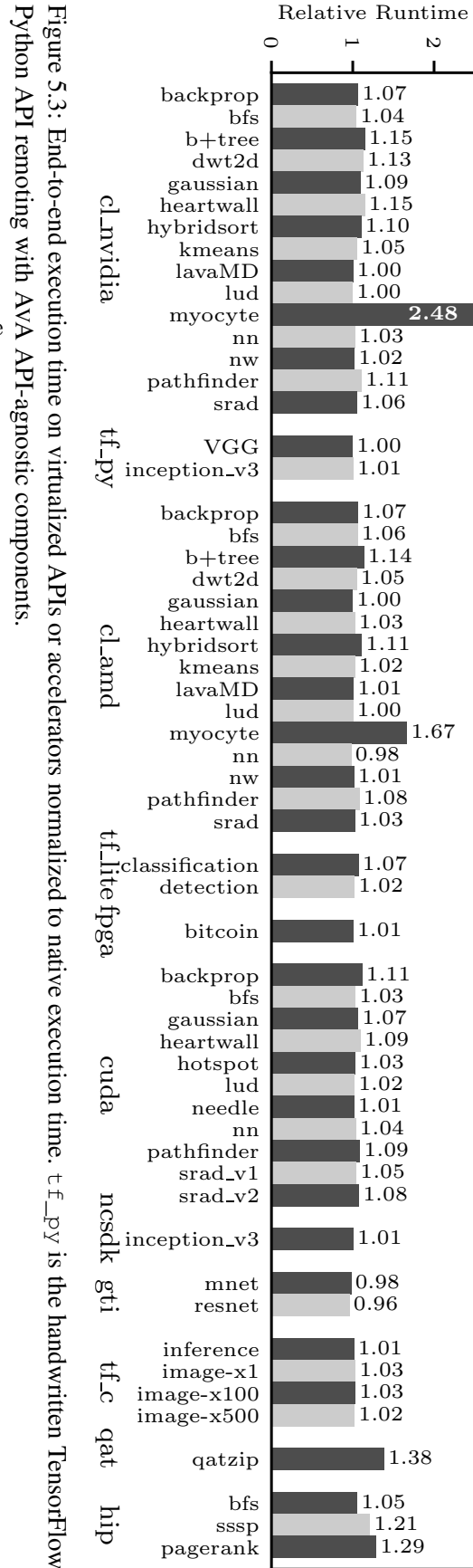
Table 5.1 presents the eleven APIs and nine devices virtualized using HIRA (either by hand or with CAVA). Automation provided by CAVA enables significant reduction in developer effort when virtualizing APIs according to the HIRA design. In order to understand developer effort, we use the lines of LAPIS specification or C/Python code (**LoC** in Table 5.1), and the number of lines of code modified (**Churn** in Table 5.1) during development (counted from commits) as indicative of effort. Larger **LoC** and **Churn** represents higher effort expended to build the virtualization scheme for that API. Overall, we find that the automation afforded by LAPIS and CAVA enable much APIs to be virtualized at much lower effort and in much shorter time frames. For example, our hand-rolled OpenCL API remoting system, which supports the parts of the OpenCL API needed to run the Rodinia [48] benchmarks, took more than 3 developer-months to build, and the resulting artifact

had 7,514 LoC (see row 1 of Table 5.1). Supporting the same subset of OpenCL with LAPIS and CAVA, took a single developer a little over a week, and the resulting API specification was 1,060 LoC long. Even in cases where we couldn't leverage AVA—TensorFlow and TensorFlow Lite Python APIs—leveraging AVA's API-agnostic components enabled us to build a HIRA system with reasonable effort (3,245 lines of Python code and 2 developer weeks for TensorFlow Python).

5.7.2 End-to-end Performance

Figure 5.3–5.4 shows the end-to-end runtime, normalized to native, for all benchmarks, accelerator and API combinations we support (see Table 5.1). AVA introduces modest overhead for most workloads. Excluding the `myocyte` benchmark, the Rodinia OpenCL benchmark suite on NVIDIA GTX 1080 GPU slowed down by 7% on average. The outlier, `myocyte` has over $2\times$ overhead because it is extremely **call-intensive**—it makes over 200,000 calls in 18.5s; most others make between 30 and 3,000 calls. `Myocyte` experienced lower overhead on the AMD Radeon RX 580 GPU, as the kernels executed $3\times$ slower, allowing more of AVA's overheads to be amortized. The benchmarks for CUDA runtime API and CUDA-accelerated TensorFlow are mostly call-intensive. The geometric mean overhead is 79.6%, $4\times$ faster than FlexDirect.

The TensorFlow benchmarks for the handwritten Python API remoting system, and Movidius benchmarks show low overhead—0% overhead on VGG-net running on TensorFlow Python (Cloud TPU), and 7% slowdown for image classification on TensorFlow Lite (Coral Edge TPU)—as they are **compute-intensive**. Each offloaded kernel performs a lot of computation per byte of data transferred, with relatively few API calls. The Gyr Falcon benchmarks enjoy a slight speedup as time spent loading and initializing the library are eliminated by using a pre-spawned API server pool. The QuickAssist accelerator proved challenging to virtualize, as it is a high-data-rate kernel-bypass encryption/compression accelerator. Applications that run on this device are **data-intensive**: computation per transferred byte is very low. We ran the Intel QATzip compression application on the Silesia corpus [55] using synchronous QAT APIs: while the application only experienced a $1.38\times$ end-to-end runtime slowdown, its throughput was $2.2\times$ lower on average. AVA was not able to keep up with the high throughput of the device, due to data transfer and marshalling overheads, as the time spent transferring data between the guest and the host was equivalent to compute time on



the accelerator. Zero-copy between the VM and the host might ameliorate this overhead. We note that QAT on AVA is fair, unlike the onboard SR-IOV support.

Overall, the end-to-end experiments show that, AVA, our realization of the HIRA virtualization design, introduces very low overheads for most workloads and is competitive with user-space API-remoting. This confirms our hypothesis that HIRA represents a “sweet-spot” in the DSA virtualization design space.

5.7.3 Micro-benchmarks

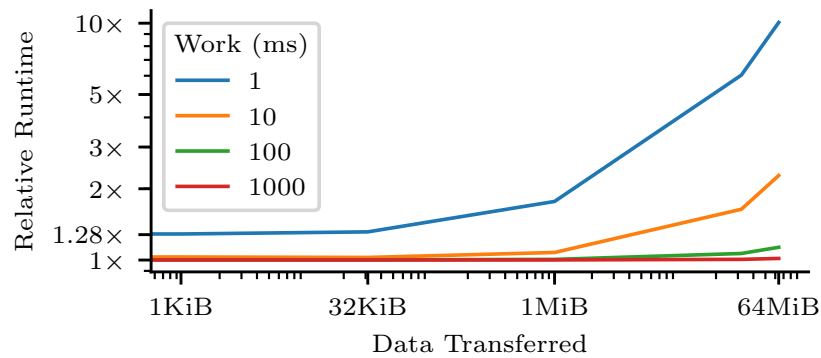


Figure 5.5: Overhead introduced by AVA for a micro-benchmark with varying work per call and data per call. The plot is log-log and the trend is linear. Runtime is relative to running the same micro-benchmark natively on the NVIDIA GTX 1080 GPU.

Given that our benchmark applications can be classified as being compute-intensive, data-intensive, and call-intensive, it would be instructive to understand the impact of the HIRA design for each of these classes of workloads. To that end, we crafted a micro-benchmark that mimicked each of these classes of applications by varying the amounts of data transferred to the GPU per call, and the duration of computation per call (simulated by spinning on the host). Figure 5.5 shows that *compute-intensive* applications (represented by the lines for 100 ms and 1,000 ms of work) suffer the lowest overhead, as data transfer is amortized by time computing on that data. *Data-intensive* applications (represented by the 1 ms and 10 ms lines) experience severe slowdowns as the data transferred per call increases, such as when 64 MiB is moved for only 1 ms of compute. *Call-intensive* applications (represented by the 1 ms line on the left side of the graph) transfer small amounts of data and execute relatively short kernels, so control transfer dominates execution, (e.g. 28% overhead on 1 ms calls with no data).

5.7.3.1 Asynchrony Optimizations

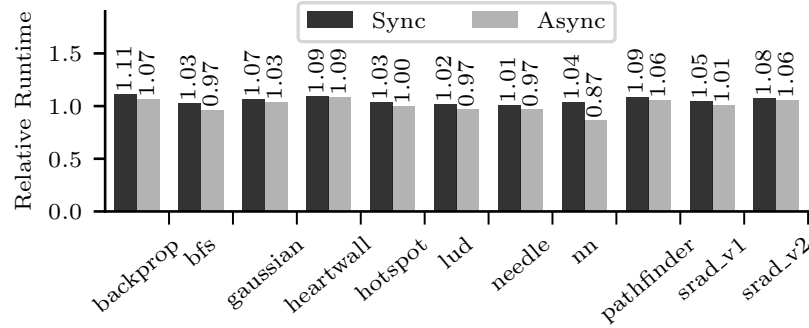


Figure 5.6: End-to-end runtime of CUDA benchmarks (relative to native) using synchronous and asynchronous specifications.

Synchronous APIs calls that have no output of any kind remain semantically correct if executed asynchronously. For example, `clSetKernelArg` is a synchronous OpenCL API, but can be forwarded asynchronously to reduce the overhead of these calls. The application’s execution will not be faithful to native execution, as the library would return immediately after the command is sent to the API server. Any resulting errors will be delivered from a later API call. Similar techniques were applied in vCUDA (lazy RPC) [132] and rCUDA (API batching) [61].

In order to understand the effectiveness of this optimization, We annotated several synchronous APIs—`cuLaunchKernel`, `cuMemcpyHtoD`, and resource free functions—to be asynchronous. Figure 5.6 shows that this optimization results in a 5% speedup on average (geometric mean) in end-to-end runtime (normalized to native) for CUDA Rodinia benchmarks.

5.7.4 Scalability

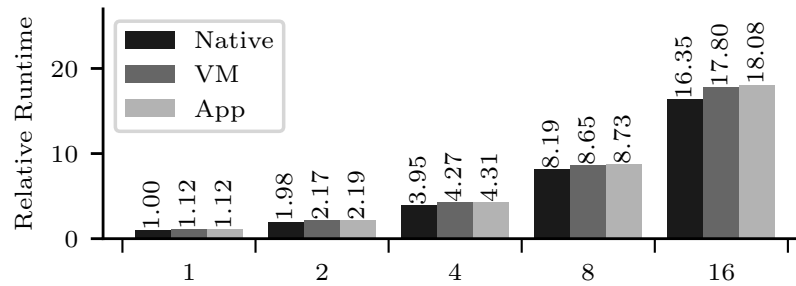


Figure 5.7: Scalability of AVA when supporting multiple VMs running a single application each (VM bars in figure), and multiple applications in a single VM (App bars in figure). Runtime is relative to running a single application natively (native scalability is shown with the Native bars).

To evaluate scalability, we ran multiple instances of the OpenCL `gaussian` benchmark simultaneously in a number of different setups on a NVIDIA GTX 1080 GPU: natively, all applications in one VM, and one VM per application. A single instance of the `gaussian` benchmark fully saturates the GPU. Overhead due to AVA ($\sim 10\%$) does not increase as the number of VMs or applications increases, as shown by the near-perfect scaling in Figure 5.7. The GPU kernel execution has an average 5.7% slowdown each time the number of VMs and applications is doubled. This slowdown is small due better utilization of the physical device and other system resources (e.g., when an instance of *gaussian* in one VM is stalled, other instances can utilize the GPU).

Accelerators without process-level protection or sharing support (e.g., Intel Movidius NCS) do not scale with AVA (or any other virtualization scheme), as multiple applications attempting to use the device have to be serialized. AVA added modest overheads (11%) in a case where 4 VMs were all running inception on the NCS v1. We note that AVA still provides benefit by enabling a hypervisor to expose and share the device across guest VMs.

5.7.5 Guaranteeing fairness by Rate Limiting APIs

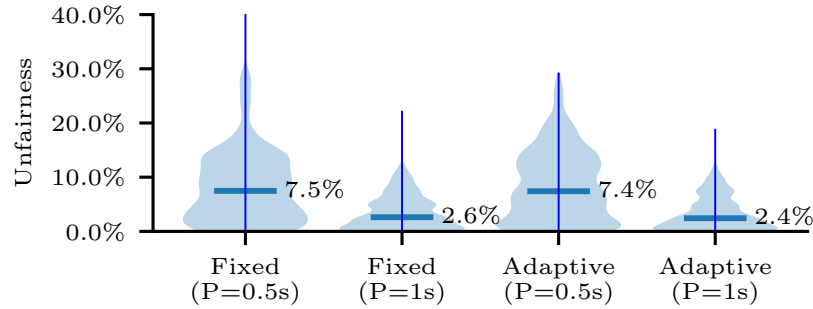


Figure 5.8: Unfairness of the fixed and adaptive scheduling algorithms with two different measurement periods. The width of the shaded areas show the probability of the bias (unfairness) being a specific value in any given measurement window. The horizontal bar shows the median and the vertical line runs from the minimum to the maximum.

To evaluate whether AVA can guarantee fairness, we repeatedly executed kernels drawn from six CUDA OpenCL benchmarks in pairs simultaneously in two VMs on the NVIDIA GTX 1080 GPU. The kernels' execution time ranges from 1 ms to 100 ms. Figure 5.8 shows the fairness of the execution with fixed-rate polling and feedback control method in 500-ms and 1-s measurement windows. We compute unfairness as $|t_1 - t_2| / (t_1 + t_2)$, where t_i is the device time used by VM_i in the time window. For fixed-rate polling ($p = 5$ ms), median unfairness in a 1 s window is 2.6%, and

scheduling overhead was 7%. For feedback control ($a = 1$ ms and $b = 1/2$), median unfairness is 2.4% in a 1 s measurement window, with 15% overhead.

5.7.6 Live Migration

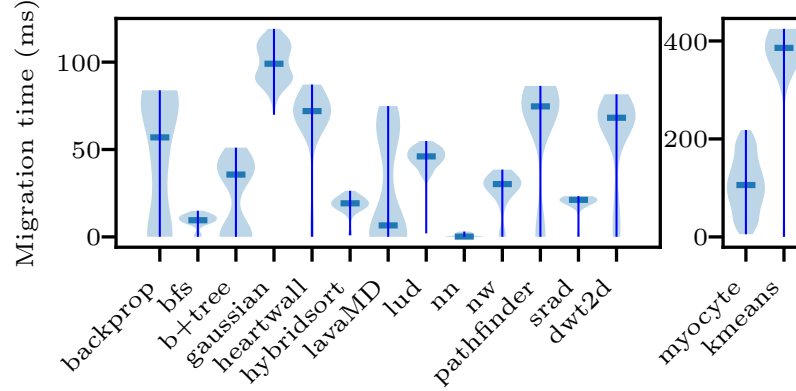


Figure 5.9: Live migration downtime for single-threaded OpenCL benchmarks on NVIDIA GTX 1080. This downtime is in addition to the ~ 75 ms of downtime of the VM migration itself. Migration downtime does not include time spent waiting for executing kernels to complete (accounted as latency), as the application is still performing useful computation on the accelerator during that time. The width of the shaded areas show the the probability of a migration taking that length of time. The horizontal bar shows the median and the vertical line shows the range from minimum to maximum.

AVA’s live-migration scheme uses annotations and selective record-and-replay instead of replaying the entire application on the remote. To understand the efficacy of this scheme, we live-migrated a VM with 4 GB memory, that was running OpenCL applications from the Rodinia benchmark suite, between two servers that were directly connected via a 10 Gib Ethernet link, both equipped with NVIDIA GTX 1080 GPUs. Migration was triggered at random points in each benchmark, and the application could not make API calls for the duration of the migration. Migrating the VM without AVA or GPU usage takes 19 s with a 75 ms downtime on average. Figure 5.9 shows the downtime experienced by applications in the VM, not including downtime for migrating the VM itself. 200 samples were collected for `myocyte`, 150 for `gaussian` and `lud`, and 50 for all others.

The dominant cost is command transfer and replay, but this cost is also affected by the size of the benchmark’s state. Figure 5.9 shows a bimodal distribution of downtime for most benchmarks. This is an artifact of applications allocating device memory before entering a steady execution state, and freeing it at termination. Migrations that occur before device memory allocation do not need to transfer significant state; migrations that occur after device memory allocation do.

5.8 Related Work

Chapter ?? provides detailed related work; this section addresses related work not covered there that are closely related to this chapter.

FPGA virtualization has a long history [113, 116, 54, 107, 93, 77, 44, 92, 137]. Most prior work relies on hardware-specific features, focuses on sharing in a single protection domain [86], or virtualization primitives [128]. HIRA can be combined with any of these techniques to virtualize FPGA accelerators. Our implementation virtualizes a bitcoin workload on an FPGA with AmorphOS [86].

Nooks [142] uses kernel-level interposition mechanisms that are similar in spirit to AVA. AVA's compiler generates components that, like the wrappers and XPC in Nooks, provide transparent control across address space and machine boundaries. Object tracking and shadow copies in Nooks' *NIM* are similar to the object tracking and shadow buffers in AVA.

RPC frameworks [146, 38, 110, 160] provide an interface description language (IDL) and tools to easily implement those interfaces. Unlike LAPIS, these IDLs do not capture all the semantics of *existing* C interfaces required to implement a HIRA API remoting design. CAVA also generates code for controlling remote resources.

Program specification languages [79] allow programmers to specify properties of functions and their behavior, and are generally used to check correctness, either statically (e.g., with model checking) or dynamically (e.g., by inserting checks in the program). While such languages allow (nearly) arbitrary predicates on programs, they are not designed to provide semantic information to other tools. In addition, these languages are not designed to specify how API calls are performed, and do not support features like state tracking. LAPIS is optimized to allow easy and specific descriptions of APIs and how calls should be performed.

Foreign Function Interface tools allow one language to call functions written in another, such as C. Some [20] make use of C headers, but require manual annotations in many common cases. Unlike AVA, language specific DSLs [7, 15], do not support marshalling data structures and encapsulate, rather than export, the C API. Cross-language serialization standards and frameworks [67, 64, 62], only provide serialization for a set of primitives and supported constructs. The user must write code

to translate their data structure into the language of the framework and provide their own transport for the serialized data.

5.9 Conclusion

The *silo*-ed nature of DSA software stacks means that the user-space API is the only viable interface for a virtualization scheme to interpose. While most prior work that interposes this interface have been user-space only, i.e., they elide all hypervisor involvement, analysis of prior techniques using the IEMTS framework showed that it is possible to build a virtualization scheme that interposes the user-space API but maintains hypervisor involvement by utilizing a hypervisor-mediated transport. Hypervisor Interposed Remote Acceleration enables the hypervisor to enforce fairness guarantees and perform resource control, while introducing low overhead on the virtualized workload. AVA, our implementation of the HIRA technique, represents a viable DSA virtualization approach that interposes compute-offload APIs, uses automation to provide agility, retains hypervisor interposition, and shortens development cycles.

CHAPTER 6: OPTIMIZED DATA MOVEMENT IN HIRA

[THIS IS THE UNFINISHED CHAPTER; PROCEED TO PAGE 63 FOR RELATED WORK.]

Hypervisor Interposed Remote Acceleration works by interposing API calls invoked by the application in the guest OS, and forwarding them to an API server in the host, via hypervisor-mediated transport. Typically, API servers are per-API (for modularity and failure isolation between APIs and accelerators, and to support remote resources) and per-guest (to preserve isolation between guests). Applications that use multiple API frameworks will be associated with multiple API-servers, one per framework. Previous chapters in this dissertation established the efficacy of the Hypervisor Interposed Remote Acceleration (HIRA) technique for virtualizing API-controlled Domain Specific Accelerators. This chapter considers a performance issue that arises from the choice of interposing the user-space API: that of redundant data movement in the virtualization stack.

Any virtualization scheme based on interposing the user-space API, burdens applications that pipeline disparate DSA API frameworks with redundant data movement. All inter-framework data movement must take place in the guest application as that the only point where they share the same logical address space. The left-hand side of Figure 6.1 illustrates this scenario. Consider a guest application (Guest Application in the figure) that pipelines functions from two API frameworks: API-1, and API-2. When the first API framework function is invoked, the associated data is copied from the Guest Application to API-Server-1, and then to Device-1's memory. Once the function has finished executing on Device-1, the result is copied first to API-Server-1, and then to the Guest Application. Since we're considering a pipelined scenario, when the Guest Application invokes the function from API-2, the same data (i.e., the output of the API-1 function) will be copied from the guest application to API-server-2 and then to Device-2 to be processed. **[ADD NOTE HERE ABOUT MEASURED OVERHEAD. IS THIS A REAL PROBLEM?]**

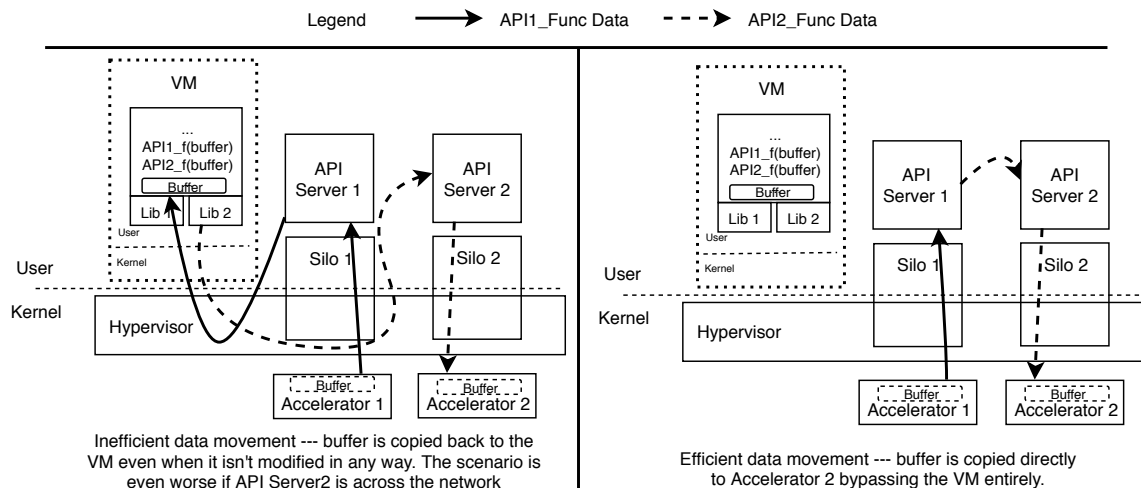


Figure 6.1: All data processed by multiple DSA API frameworks must pass through the guest application, leading to redundant data movement.

The crux of the issue is that the guest application (and the virtualization scheme's interposition library) is the only point in the constructed software stack where the two API frameworks are in the same address space. At all other points, the two API frameworks have no notion of the other's existence. Such a design is typically desirable for maintaining isolation, and to enable independent scaling, migration etc.

The right hand side of Figure 6.1 illustrates the desired behavior in the scenario described before. As before, consider a guest application that invokes functions from two API frameworks in a pipelined fashion, i.e., the output of the function from the API1 framework is consumed by a function from the API2 framework. In such a scenario, eliding movement of data to and from the guest application would lead to improved performance. 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.

Given this overhead arises due to the introduction of virtualization, this chapter explores techniques to eliminate it automatically, i.e., without involving the end user. In order to automatically eliminate the redundant data movement described in the left side of Figure 6.1 when an application uses multiple accelerators via API-remoting, the flow of data among these API calls must be tracked. Keeping 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 guest application), etc., will enable the virtualization system to detect scenarios like the one presented in the left side of Figure 6.1 and transform them to the one shown on the right side of the figure instead.

Automatic detection and tracking of data movement among accelerator API frameworks requires semantic understanding of the interposed API functions. The API function annotations provided by LAPIS allow us to identify parts of the API that deal with the movement of data, the direction of movement, and the actual data itself. This chapter describes vTask, an optimization for AvA that leverages LAPIS annotations to orchestrate data movement among accelerators virtualized with HIRA in an application-transparent manner. vTask tracks data buffers across the guest application, the API-servers servicing API calls made by the guest, and the accelerator hardware, and optimizes data movement across these components while ensuring that a coherent view of the data buffer is presented to anyone attempting to read the data.

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.

With vTask, AvA will be able to 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.

CHAPTER 7: RELATED WORK

Virtualization has such a long and storied history that Attempting to capture the entire story is an exercise in futility. The introduction 1 captures the history of CPU virtualization in broad strokes. This section then focuses on a major theme of the proposed dissertation: accelerator virtualization.

Accelerating specific computation is not a new idea—support for specialized computation is extremely commonplace in CPUs (e.g., Floating Point Units (FPU), Vector Processing Units). These specialized compute units are typically exposed to the programmer as extensions to the Instruction Set Architecture (ISA). Virtualizing these specialized compute units, therefore, is no different from virtualizing the rest of the CPU and ISA virtualization is well explored [30, 52, 117, 45, 47, 46].

Processors specialized for complex computational tasks, such as graphical rendering, largely evolved as discrete devices separate from the CPU (although some CPUs do integrate GPUs). These devices are not typically integrated into the CPU ISA; instead, they appear to system software as I/O devices with memory-mapped command-queues and I/O registers. I/O virtualization is well understood [155, 36, 94, 133, 167, 29], but these techniques aren't enough to virtualize programmable accelerators. Although programmable accelerators look like I/O devices, they are also general computing platforms, i.e., they load binaries, have their own memory, and are typically exposed to the application programmer via an API.

7.1 GPU Virtualization

GPU virtualization has received a lot of attention since the late 2000s. This section presents an overview of all prior work. Table 7.1 presents a comprehensive overview of virtualization systems in the research literature, classifying them (sometimes tenuously) according to traditional virtualization properties. We evaluate the completeness of each solution along several dimensions: fidelity (ability run with unmodified guest libraries and OS), sharing (ability to safely and fairly multiplex GPUs across guest VMs), compatibility (ability to support a GPU device abstraction that is independent of hardware actually present on the host), ability to support VM mobility, and performance.

Technique	System	lib unmod	OS unmod	lib-compat	hw-compat	sharing	isolation	migration	sched. policy	graphics	GPGPU	LD	benchmark	slowdown	native speedup	virtual speedup
Full-virtual	GPUvm [141]	✓		✓		✓	✓		XC, BAND		✓	D	Rodinia	141×	11.4×	0.08×
	gVirt [147]	✓		✓		✓	✓	✓	QoS	✓		I	2D [18], 3D [5]	1.6×	N/A	N/A
PCIe Pass-thru	AWS GPU [34]	✓	✓							✓	✓	D	Any	1×		
API remotng	GVim [71]				✓	✓	✓		RR, XC		✓	D	CUDA 1.1 SDK	1.16×	22×	19×
	gVirtuS [66]				✓	✓	✓		RR		✓	D	CUDA 2.3 MM	3.1×	11.1×	3.6×
	vCUDA [131]		✓		✓			✓	HW		✓	D	CUDA 4.0 SDK	1.91×	6×	3.1×
	vmCUDA [153]		✓		✓	✓			HW		✓	D	CUDA 5.0 SDK	1.04×	33×	31.7×
Distributed API remotng	rCUDA [61, 120]		✓		✓	✓	✓		RR		✓	D	CUDA 3.1 SDK	1.83×	49.8×	27.2×
	GridCuda [101]		✓		✓	✓	✓		FIFO		✓	D	CUDA MM, SOR	1.23×		
	SnuCL [90]		✓			✓	✓				✓	D	SNU NPB [130]			
	VCL [39]		✓		✓	✓	✓				✓	D	Stencil2D [53]			
Para-virtual	GPUvm [141]					✓	✓		XC, BAND		✓	D	Rodinia	5.9×	11.4×	1.9×
	HSA-KVM [78]	✓				✓	✓		HW		✓	I	AMD OCL SDK	1.1×		
	LoGV [69]	✓		✓		✓	✓	✓	RR		✓	D	Rodinia	1.01×	11.4×	11.3×
	SVGA2 [59]	✓				✓	✓	✓		✓		D	2D, gaming	3.9×		
	Paradice [36]	✓		✓		✓	✓		HW, QoS	✓	✓	D	OpenGL, OpenCL	1.1×		
	VGVM [149]				✓	✓	✓		HW		✓	D	CUDA 5.0 SDK	1.02×	33×	32.3×

Table 7.1: Existing GPU virtualization proposals, grouped by approach. Previously published in the Trillium paper [33].

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 also 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. Values in this column were computed by dividing the expected speedup from using a GPU by the slowdown introduced by virtualization. Entries where overheads eclipse GPU-based performance gains are marked in **red**. Performance profitable entries are **blue**. Greyed out cells indicate the metric is meaningless for that design. Light grey cells indicate that the data was not available.

7.1.1 Comparison methodology

Our goal in reducing the performance of a research system to an aggregate number or two, is to arrive at “back-of-the-envelope” estimates that can ultimately inform some intuition of the impact of fundamental design tradeoffs on *expected* performance. Under performance, we include the benchmarks used, and where possible, a report (or estimate) of the geometric mean speedup one should expect for using GPUs over CPUs on the given benchmark suite (called *native speedup*, and *virtual-speedup*—the geometric mean expected speedup one might expect for the given benchmarks in a system whose GPUs are virtualized similarly. It should go without saying, that if the overheads induced by virtualization overwhelm the expected speedup, the case for using GPUs in that context is significantly weakened. The virtual speedup is computed as the expected speedup from GPUs for the given suite of benchmarks divided by the slowdown induced by virtualization. Entries where overheads eclipse GPU-based performance gains are marked in red; performance profitable entries are blue.

An ideal system would leave guest libraries and OS unmodified, provide strong isolation with fairness guarantees, maintain compatibility across diverse GPU hardware, and preserve GPU performance profitability. More succinctly, an ideal system would have all checks in the qualitative columns, with a **virtual speedup** column that differs negligibly from the **native speedup** column. An acceptable system might relax this along any axis, with the caveat that **virtual-speedup** must be blue, indicating the preservation of at least *some* of the expected performance profitability of GPU execution.

7.1.2 Dominant Trends

Two trends are clearly expressed in this table. First, **no proposals address compatibility**, suggesting it is at once a serious challenge and an ideal research opportunity. Second, The performance profitability of GPU acceleration is at risk if full virtualization overheads reported in the literature are to be believed.

7.1.3 Additional Considerations

In many cases, reporting gpu:cpu performance ratios and expected performance after virtualization overheads are accounted is either inappropriate or impossible. For example, gVirt is included in this table because it claims full virtualization, we cannot report expected speedups because the evaluation relies entirely on graphics workloads, for which we have no way to obtain reasonable sequential baselines. The same is true for SVGA2.

Several systems are evaluated on samples from various versions of the CUDA SDK. GVim, vmCUDA, vCUDA, and gVirtus were evaluated on hand-picked samples from the CUDA 1.1, 5.0, 4.0 and 4.0 respectively. Since the papers do not report CPU-GPU speedup baselines, we estimate the expected speedup from GPUs by measuring GPU speedup on corresponding CUDA 7.0 SDK samples on a machine with a Tesla k20 and Intel i7 CPU. Clearly, the hardware differences from the original papers are significant, and we encourage the reader to interpret the numbers with that mind. Generally speaking the evaluations in these papers are sufficiently specious that additional uncertainty induced by using incomparable hardware is likely negligible, as we are primarily concerned with estimating a single bit of information: will performance profitability be preserved.

7.1.4 Full Virtualization

GPUvm [140] virtualizes CUDA on Kepler and Fermi (NVIDIA) GPUs for Xen. GPUvm safely multiplexes the basic GPU physical resources: GPU contexts (analogous to a process), GPU channels (the mechanism by which commands are submitted to a context), GPU page tables, and GPU control registers, which are memory apertures mapped onto PCIe BARs with MMIO. To this end, the design introduces *GPU shadow channels*, *GPU shadow page tables*, and *virtual GPU schedulers*: these abstractions form the interposition boundary for virtualization. GPUvm presents a GPU device model to each VM. Attempts to access the GPU from all VMs are routed through a GPU Aggregator. The aggregator maintains shadow page tables, shadow channels, implements a “fair share scheduler”, and modifies requests to enforce isolation. GPUvm interposes on communication between guest device driver and the GPU device model, by trapping and forwarding MMIO writes. The performance costs of full virtualization are unacceptable, and primarily result from page table management overheads. TLB flushes are required with every GPU page table update. Moreover, GPU page faults are not

forwarded to the host CPU, so GPUvm must scan GPU page tables on every TLB flush to keep GPU shadow page tables current. The authors explore a number of optimizations: lazy shadowing, bar remap, para-virtualization, and multi-call batching. *Lazy Shadowing* reflects guest updates to page tables into shadow page tables *only* when the tables are referenced, rather than on every TLB flush. *Bar Remap* limits BAR interposition and passes through BAR accesses other than those made to GPU channel descriptors. *Para-virtualization* allows a guest GPU driver updates page tables through hypercalls (which can be further optimized with *multi-calls*), and GPUvm validates those updates, eliminating the scan of guest GPU page tables. Naturally, this optimization gives up full virtualization, as guest GPU drivers must be modified. Despite these optimizations, GPUvm remains non-viable due to its high overhead—the most optimal configuration of GPUvm induces a $6 \times$ slowdown on average. GPUvm [141] warrants entries under multiple virtualization techniques in Table 7.1 because the authors (to their credit) built a large number of variants of their system to characterize the performance impact of a large range of fundamental design tradeoffs. Specifically, while full virtualization is the stated goal of the effort, the authors implemented a number of variants using para-virtualization techniques, along with a simple pass-through variant. Indeed, the characterization of the space is sufficiently prolific to challenge summarization. GPUvm under para-virtualization requires the guest to issue hypercalls to make GPU page table updates (similar to direct-paging in Xen [40]) and provides a multi-call interface to batch those hypercalls, again borrowed from Xen.

gVirt [148] is a *graphics*-only virtualization technique for Intel GPUs. gVirt represents an Intel-centric technique for implementing virtualization of *graphics*, but as full-virtualization system the techniques are relevant to GPGPU. The system runs the native graphics driver in the guest OS in dom0, and implements pass-through for access only to performance-critical resources (command and frame buffers), using trap-emulate for resources generally accessed off the critical path (PTEs, I/O Registers). The native driver is present primarily to simplify tasks like initialization and power management. Trap-emulate operations are forwarded to a mediator module in dom0, which implements the vGPU interface and scheduler. Operations are subsequently handled with hypercalls into a stub in Xen. Memory is multiplexed with a combination of partitioning and “ballooning”. Each VM gets 2GB of local graphics memory and 2GB global graphics memory. These partitions are striped across the actual physical memory, and sections not belonging to a particular VM are marked “ballooned” in

the page tables of that VM, which means they are inaccessible. This makes address translations exactly the same for each VM with the caveat that regions belonging to different VMs must be made inaccessible. gVirt handles this through “smart shadowing” and auditing of the command buffer. The primary limitations of this work are that it is geared only toward graphics and not compute, and that partitioning memory space means the full physical memory can never be utilized by each VM. Further, the auditing process seems tenuous: While inspecting register values to ensure they are bounded by regions that should be mapped to a given VM is feasible, it is much harder to determine whether the register value is actually a pointer.

7.1.5 API Remoting

GVIM [71] supports a straightforward split-driver API remoting approach to virtualization of CUDA in Xen. CUDA API calls made by applications in the Guest VM are interposed through a front-end driver (using Xen event channels) and forwarded to a back-end driver in DOM0, which exercises the CUDA driver and runtime. GVIM proposes some memory management optimizations to avoid double buffering, redundant copy, such as using `mmap` to map guest kernel memory into the front end driver to avoid user-to-kernel copies, and using a page-directory structure to avoid copy between VMs from front-end driver to back-end driver.¹ While GVIM’s split-driver design is very similar to AvA’s HIRA, AvA presents an accelerator-agnostic framework that can be used to implement hypervisor-mediated API-remoting for arbitrary devices, can enforce flexible policies via callbacks and tackles the compatibility issues inherent in API-remoting.

vCUDA [132] is another CUDA API-remoting system. CUDA API calls are redirected through an interposer library (“vCUDA”) to a stub in the host OS, which interacts with the device using pass through. RPC turns out to be the primary performance term. The authors explores RPC batching as an optimization. The system has no support for interposition.

vmCUDA [153] observes that while pass-through is performant, it precludes sharing, and VM migration. vmCUDA supports API remoting for CUDA in the the ESX Hypervisor. vmCUDA employs a standard split-driver model with a front-end driver in the guest, and a backend driver

¹Most of these optimizations should be obviated by VM support for cross-VM bulk-transfer mechanisms like VMCI.

in the control domain (called the “appliance VM”) which interacts with the CUDA runtime and driver. The driver in the appliance VM interacts with the GPU hardware using pass-through. CUDA applications in the guest are linked against an interposer library which uses vRDMA, VCMI, TCP to forward calls and data to the appliance VM. The application starts on the client, sends a copy of binary to appliance VM, which modifies the binary and starts a process container for it: the client VM communicates with that process. The authors find that data copy calls must be broken down into smaller fragments to enable multiple VMs to share the GPU. Cross-VM isolation guarantees result from use of per-application child processes in the appliance VM, which effectively level process-level protection mechanisms toward cross-VM protection. The design addresses compatibility challenges with VM mobility (vMotion): the appliance VM needn’t move if the guest VM moves.² vmCUDA performs dynamic binary re-writing of the client application (replacing API calls) to make API forwarding transparent to the developer. As with vCUDA, the system guarantees no isolation among VMs.

gVirtuS [66] is an API remoting framework that claims to provide transparent support for CUDA, OpenCL, and OpenGL on Xen, KVM, and VMware ESXi, using a split-driver design to provide a formal abstraction layer for GPUs that is independent of VMM.

rCUDA [61, 120], *GridCuda* [101], *SnuCL* [90] and *VCL* [39] are all user-mode middle-ware systems for multiplexing GPUs and CUDA/OpenCL across a cluster. *rCUDA* is a middle-ware system for multiplexing NVIDIA GPUs and CUDA across a cluster. A client library encapsulates access to a (potentially) remote GPU. Client applications must be recompiled/re-linked against the *rCUDA* library, which results in feature incompatibilities for a number of undocumented features that are handled transparently by the `nvcc` compiler. While the basic design is isomorphic to the API remoting design, virtual machines need not be present. *GridCuda* [101] is similar: a pure user-mode fabric for tunnelling CUDA API traffic to GPUs on remote machines. *SnuCL* [90] provides an OpenCL [87] programming interface to clusters of CPU/GPU servers. The basic approach is to extend the OpenCL semantics to encapsulate remote resources, which preserves the original programming model which, like CUDA [109], assumes a process model, running in the context of a

²This is a fairly limited form of support for mobility, and is not likely what the user intends when migrating a VM. That said, API remoting is fundamentally resistant to mobility, and this is the best known solution as of this writing.

single operating system image. Like rCUDA [61] and GridCuda [101], SnuCL should be viewed as a distributed runtime that supports GPUs, rather than a general approach to GPU virtualization. VCL [39] is the lower layer in a package called MGP (many-gpu-package) which encapsulates remote compute resources and data management within the local OpenCL API implementation, such that remote GPUs look to the application running on the master node, like a local OpenCL device. We do not estimate expected speedup for rCUDA, GridCUDA, SnuCL, and VCL, in Table 7.1 as evaluations of these systems include network overheads that would not be present in our target setting. Moreover, since these are ultimately API remoting solutions, we consider the performance case to be sufficiently well made by the four systems targeting single machine settings.

7.1.6 Para-virtualization

LoGV [69] describes an approach to GPGPU virtualization that uses GPU protection hardware in the hypervisor to enforce cross-VM isolation. This strategy has two important consequences. First, cross-VM isolation is easy to enforce, and the overheads of virtualization induced by the hypervisor component is minimal. Second, guests are left with no hardware mechanism to enforce cross-process isolation, which pushes the responsibility on the guest driver, which may be forced to use high overhead techniques to provide such guarantees. We classify LoGV as a para-virtualization technique because it ultimately forces change on the guest OS in the form of changes to support process isolation. The virtualization overheads reported in the paper are exceptionally rosy (indeed, the virtualized solution is faster than native in 3 of 4 reported benchmarks). However, the evaluation prototype makes no effort to enforce isolation in guests, which ultimately hides a significant cost and makes the reported numbers meaningless. LoGV is present under para-virtualized systems in Table 7.1 because it leverages GPU protection hardware in the hypervisor to enforce cross-VM isolation, with the side-effect that guest drivers must be modified to implement that displaced functionality. We report no expected speedup first because the four micro-benchmarks used in the evaluation are non-standard, and second because the evaluation prototype used unmodified guest drivers, which means guests could not enforce isolation. Consequently, the numbers reported neglect the (likely high) performance impact of a critical feature.

HSA-KVM [78] is a para-virtual system for Heterogeneous System Architecture (HSA) compliant systems. HSA has the CPU and GPU integrated into the same physical address space. The GPU exposes multiple Architected Queues (Command Queues) that can be allocated to different guests. HSA-KVM comes closest to the flexibility, compatibility and performance of CPU virtualization. However, the design espoused requires high levels of co-operation from the accelerator hardware, and as alluded earlier, this level of hardware support is still missing in most accelerators.

SVGA2 [59] is VMware’s *graphics-only* GPU virtualization scheme. SVGA2 employs a para-virtual split-driver design with a custom GPU ISA for shader programs (TGSI). SVGA2 supports multiple front-end libraries (OpenGL, DirectX, etc.) via a common driver that is shipped with most mainstream operating systems. SVGA2 uses DirectX as an internal transport mechanism, effectively realizing API-remoting through the split-driver design. Trillium attempts to extend the SVGA2 model to GPGPU computing. We find that the SVGA2 model is a poor fit for general purpose accelerators due to drastically different constraints. As an example, consider performance. SVGA2 has a much lower target to hit: frames per second (fps), while GPGPU virtualization must preserve the raw speedup over computing with a CPU. This makes the multiple translations needed to implement the many-to-many multiplexing viable for graphics rendering but not for general purpose computation.

7.2 Language-level Virtualization

Dandelion [124] abstracts accelerators at the language runtime by compiling sequential .Net code to the accelerator (GPU or FPGA). vTask draws inspiration from the buffer management in Dandelion and PTask [122], the underlying accelerator abstraction layer.

HPVM [136] explores the design of a virtual ISA (vISA) for abstracting heterogeneous compute devices. The HPVM vISA serves as a portable compilation target for managed language run-times built on top of the LLVM compiler infrastructure. HPVM can serve as a replacement for LLVM in Trillium. HPVM, however, doesn’t absolve the language run-time implementation from interacting with the accelerator silo. **TornadoVM** is an example of such a managed language runtime implementation (Gaal VM).

CHAPTER 8: CONCLUSION

[WRITE-ME ;-D] [BIBLIOGRAPHY PROBABLY HAS A LOT OF MISTAKES AND DUPLICATES...]

BIBLIOGRAPHY

- [1] Additive Increase/Multiplicative Decrease. https://en.wikipedia.org/wiki/Additive_increase/multiplicative_decrease. Accessed: 2019-08.
- [2] Amazon EC2 instance types. <https://aws.amazon.com/ec2/instance-types/>. Accessed: 2017-04.
- [3] AMD Multi-user GPU. <http://www.amd.com/Documents/Multiuser-GPU-White-Paper.pdf>. Accessed: 2018-07.
- [4] Bitfusion: The elastic AI infrastructure for multi-cloud. <https://bitfusion.io>. Accessed: 2019-04.
- [5] Cairo-perf-trace. <http://www.cairographics.org>. Jan. 2018.
- [6] Compute shader. https://www.khronos.org/opengl/wiki/Compute_Shader. Jan. 2017.
- [7] Cython: C-extensions for Python. <https://cython.org/>. Accessed: 2019-08.
- [8] Deploying rCUDA in cloud computing environments. http://www.rcuda.net/pub/white_paper_cloud_v2.pdf. Published: 2016-12.
- [9] Francisco Jerez's TGSI back-end. <https://github.com/curro/llvm>. Jan. 2018.
- [10] Freedesktop nouveau open-source driver. <http://nouveau.freedesktop.org>. Accessed: 2017-04.
- [11] GalliumCompute. <https://dri.freedesktop.org/wiki/GalliumCompute/>. Accessed: 2018-2-6.
- [12] Google Cloud TPU. <https://cloud.google.com/tpu>. Accessed: 2019-04.
- [13] Hans de Goede's TGSI back-end. <https://cgit.freedesktop.org/~jwrdegoede/llvm>. Jan. 2018.
- [14] Intel QuickAssist Technology. <https://01.org/intel-quickassist-technology>. Accessed: 2019-04.
- [15] Java Native Access (JNA). <https://github.com/java-native-access/jna>. Accessed: 2019-08.
- [16] Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. Accessed: 2019-08.
- [17] NVIDIA CUDA 4.0. <http://developer.nvidia.com/cuda-toolkit-40>. 2011.
- [18] Phoronix test suites. <http://phoronix-test-suite.com>. Jan. 2018.
- [19] Project Fiddle: Fast and Efficient Infrastructure for Distributed Deep Learning. <https://www.microsoft.com/en-us/research/project/fiddle>. Accessed: 2019-04.

- [20] SWIG: Simplified Wrapper and Interface Generator. <http://www.swig.org>. Accessed: 2019-08.
- [21] Underutilizing Cloud Computing Resources. <https://www.gigenet.com/blog/underutilizing-cloud-computing-resources/>. Published: 2017-11.
- [22] VMware SVGA3D Guest Driver. <https://www.mesa3d.org/vmware-guest.html>. Accessed: 2019-08.
- [23] VMware Workstation Version History. https://en.wikipedia.org/wiki/VMware_Workstation#Version_history. Accessed: 2019-08.
- [24] Why frame rate and resolution matter. <https://www.polygon.com/2014/6/5/5761780/frame-rate-resolution-graphics-primer-ps4-xbox-one>. 2011.
- [25] Gallium3d technical overview, 2017.
- [26] The mesa 3d graphics library, 2017.
- [27] TOP500 Supercomputer Sites. <https://www.top500.org/lists/2018/11/>, 2019.
- [28] 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.
- [29] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel virtualization technology for directed i/o. *Intel technology journal*, 10(3), 2006.
- [30] R.J. Adair. *A Virtual Machine System for the 360/40*. IBM Cambridge Scientific Center report. International Business Machines Corporation, Cambridge Scientific Center, 1966.
- [31] 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.
- [32] 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.
- [33] 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.
- [34] Amazon. *Amazon Elastic Compute Cloud*, 2015.
- [35] Inc or Its Affiliates Amazon Web Services. Amazon EC2 P3 Instances. <https://aws.amazon.com/ec2/instance-types/p3/>. Accessed: 2018-2-6.
- [36] 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.

- [37] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. Rio: A system solution for sharing i/o between mobile systems. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 259–272. ACM, 2014.
- [38] Apache Software Foundation. Apache thrift. <https://thrift.apache.org>. Accessed: 2019-04.
- [39] 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.
- [40] 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.
- [41] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.
- [42] 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.
- [43] David Blythe. The direct3d 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.
- [44] Alexander Brant and Guy GF Lemieux. Zuma: An open fpga overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 93–96. IEEE, 2012.
- [45] 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.
- [46] 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.
- [47] Edouard Bugnion, Jason Nieh, and Dan Tsafirir. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture*, 12(1):1–206, 2017.
- [48] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.
- [49] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.
- [50] Hsiao-keng Jerry Chu. Zero-copy tcp in solaris. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 21–21. Usenix Association, 1996.

- [51] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.
- [52] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM J. Res. Dev.*, 25(5):483–490, September 1981.
- [53] 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.
- [54] André DeHon, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, and John Wawrzynek. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems*, 30(6):334–354, 2006.
- [55] Sebastian Deorowicz. *Universal Lossless Data Compression Algorithms*. PhD thesis, Silesian University of Technology, 2003.
- [56] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with SR-IOV. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.
- [57] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: Architecture, design and implementation. In *Proceedings of the First Conference on I/O Virtualization*, WIOV’08, pages 10–10, Berkeley, CA, USA, 2008. USENIX Association.
- [58] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: Architecture, design and implementation. In *Workshop on I/O Virtualization*, 2008.
- [59] Micah Dowty and Jeremy Sugerman. Gpu virtualization on vmware’s hosted i/o architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.
- [60] José Duato, Francisco D. Igual, Rafael Mayo, Antonio J. Peña, Enrique S. Quintana-Ortí, and Federico Silla. An efficient implementation of gpu virtualization in high performance clusters. In *Proceedings of the 2009 International Conference on Parallel Processing*, Euro-Par’09, pages 385–394, Berlin, Heidelberg, 2010. Springer-Verlag.
- [61] 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.
- [62] ECMA. The JSON data interchange syntax, December 2017. ECMA Standard 404, 2nd Edition.
- [63] H Esmailzadeh, E Blem, R St.Amant, K Sankaralingam, and D Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376, June 2011.
- [64] Sadayuki Furuhashi. Messagepack. <https://msgpack.org/>. Accessed: 2019-04.
- [65] Patrick Paul "Pat" Gelsinger. Private Communication, 1998.

- [66] 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.
- [67] Google Inc. Protocol buffers. <https://developers.google.com/protocol-buffers/>. Accessed: 2019-04.
- [68] Abel Gordon, Nadav Har’El, Alex Landau, Muli Ben-Yehuda, and Avishay Traeger. Towards exitless and efficient paravirtual i/o. In *Proceedings of the 5th Annual International Systems and Storage Conference, SYSTOR ’12*, pages 10:1–10:6, New York, NY, USA, 2012. ACM.
- [69] 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.
- [70] Kate Gregory and Ade Miller. C++ amp: accelerated massive parallelism with microsoft visual c++. 2014.
- [71] 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.
- [72] 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.
- [73] Stefan Hajnoczi. Virtio-vsock, zero-configuration host/guest communication. <https://vmsplICE.net/~stefan/stefanha-kvm-forum-2015.pdf>. Accessed: 2019-04.
- [74] Nadav Har’El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. Efficient and scalable paravirtual i/o system. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference, USENIX ATC’13*, pages 231–242, Berkeley, CA, USA, 2013. USENIX Association.
- [75] Alex Herrera. Nvidia grid: Graphics accelerated vdi with the visual performance of a workstation. *Nvidia Corp*, 2014.
- [76] Chun-Hsian Huang and Pao-Ann Hsiung. Hardware resource virtualization for dynamically partially reconfigurable systems. *IEEE Embedded Systems Letters*, 1(1):19–23, 2009.
- [77] Chun-Hsian Huang and Pao-Ann Hsiung. Hardware Resource Virtualization for Dynamically Partially Reconfigurable Systems. *IEEE Embed. Syst. Lett.*, 1(1):19–23, May 2009.
- [78] 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 the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE ’16*, pages 3–15, New York, NY, USA, 2016. ACM.

- [79] Microsoft Inc. Using SAL annotations to reduce C/C++ code defects. <https://docs.microsoft.com/en-us/visualstudio/code-quality/using-sal-annotations-to-reduce-c-cpp-code-defects>, November 2016. Accessed: 2019-11.
- [80] 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.
- [81] 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.
- [82] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. A domain-specific architecture for deep neural networks. *Commun. ACM*, 61(9):50–59, August 2018.
- [83] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, and et al. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017.
- [84] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC’11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.
- [85] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class gpu resource management in the operating system. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC’12, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association.
- [86] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI’18)*, pages 107–127, 2018.
- [87] Khronos Group. *The OpenCL Specification, Version 1.0*, 2009.
- [88] Khronos Group. *Vulkan 1.0.64 - A Specification*, 2017.
- [89] 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.
- [90] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. Snuc1: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, page 341–352. ACM, 2012.
- [91] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. Gpunet: Networking abstractions for gpu programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI’14, pages 201–216, Berkeley, CA, USA, 2014. USENIX Association.

- [92] Robert Kirchgessner, Alan D. George, and Greg Stitt. Low-Overhead FPGA Middleware for Application Portability and Productivity. *ACM Trans. Reconfigurable Technol. Syst.*, 8(4):21:1–21:22, September 2015.
- [93] Robert Kirchgessner, Greg Stitt, Alan George, and Herman Lam. Virtualrc: A virtual fpga platform for applications and tools portability. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 205–208. ACM, 2012.
- [94] Yossi Kuperman, Eyal Moscovici, and Joel Nider. Paravirtual Remote I/O.
- [95] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafir. Paravirtual remote i/o. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 49–65. ACM, 2016.
- [96] H. Andres Lagar-Cavilla, Niraj Tolia, M. Satyanarayanan, and Eyal de Lara. Vmm-independent graphics acceleration. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 33–43, New York, NY, USA, 2007. ACM.
- [97] H. Andres Lagar-Cavilla, Niraj Tolia, M. Satyanarayanan, and Eyal de Lara. Vmm-independent graphics acceleration. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 33–43, New York, NY, USA, 2007. ACM.
- [98] David Alex Lamb and David Alex. IDL: Sharing intermediate representations. *ACM Transactions on Programming Languages and Systems*, 9(3):297–318, jul 1987.
- [99] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.
- [100] Teng Li, Vikram K. Narayana, Esam El-Araby, and Tarek El-Ghazawi. Gpu resource sharing and virtualization on high performance computing systems. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 733–742, Washington, DC, USA, 2011. IEEE Computer Society.
- [101] 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.
- [102] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 3–3, Berkeley, CA, USA, 2006. USENIX Association.
- [103] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, 1993.
- [104] Microsoft. MIDL Compiler. <https://docs.microsoft.com/en-us/windows/win32/com/midl-compiler>. Accessed: 2019-08.
- [105] Microsoft Inc. *Windows GDI*, 2017.
- [106] Sun Microsystems. Rfc1050: Rpc: Remote procedure call protocol specification, 1988.

- [107] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. Tartan: Evaluating Spatial Computation for Whole Program Execution. *SIGOPS Oper. Syst. Rev.*, 40(5):163–174, October 2006.
- [108] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317. ACM, 2011.
- [109] NVidia. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.
- [110] Object Management Group. CORBA Component Model. <https://www.omg.org/spec/CCM/4.0/PDF>, March 2006. Accessed: 2019-04.
- [111] Johns Paul, Jiong He, and Bingsheng He. Gpl: A gpu-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1935–1950. ACM, 2016.
- [112] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. Mdev-nvme: A nvme storage virtualization solution with mediated pass-through. In *2018 USENIX Annual Technical Conference (USENIX ATC’18)*, pages 665–676, 2018.
- [113] K. Dang Pham, A. K. Jain, J. Cui, S. A. Fahmy, and D. L. Maskell. Microkernel Hypervisor for a Hybrid ARM-FPGA Platform. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 219–226, June 2013.
- [114] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtcher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The TAO of parallelism in algorithms. In *ACM Sigplan Notices*, volume 46, pages 12–25. ACM, 2011.
- [115] Sébastien Pinneterre, Spyros Chiotakis, Michele Paolino, and Daniel Raho. vfpgamanager: A virtualization framework for orchestrated fgpa accelerator sharing in 5g cloud environments. In *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–5. IEEE, 2018.
- [116] Christian Plessl and Marco Platzner. Zippy-a coarse-grained reconfigurable array with support for hardware virtualization. In *null*, pages 213–218. IEEE, 2005.
- [117] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.
- [118] Himanshu Raj and Karsten Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC ’07, pages 179–188, New York, NY, USA, 2007. ACM.
- [119] Kaushik Kumar Ram, Jose Renato Santos, and Yoshio Turner. Redesigning xen’s memory sharing mechanism for safe and efficient i/o virtualization. In *Proceedings of the 2nd conference on I/O virtualization*, pages 1–1. USENIX Association, 2010.
- [120] 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.

- [121] 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.
- [122] 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.
- [123] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. Symposium on Operating Systems Principles (SOSP), October 2011.
- [124] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. SOSP’13: The 24th ACM Symposium on Operating Systems Principles, November 2013.
- [125] Rusty Russell. virtio: Towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.
- [126] R Sandberg, D Golgberg, S Kleiman, D Walsh, and B Lyon. Design and implementation of the Sun network filesystem. In C Partridge, editor, *Innovations in Internetworking*, chapter Design and, pages 379–390. Artech House, Inc., Norwood, MA, USA, 1988.
- [127] Russel Sandberg. The sun network file system: Design, implementation and experience. In *in Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, 1986.
- [128] Eric Schkufza, Michael Wei, and Christopher J. Rossbach. Just-in-time compilation for verilog: A new technique for improving the FPGA programming experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 271–286, 2019.
- [129] Mark Segal and Kurt Akeley. The opengl graphics system: A specification. Technical report, Silicon Graphics Inc., December 2006.
- [130] 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.
- [131] 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.
- [132] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vcuda: Gpu-accelerated high-performance computing in virtual machines. *IEEE Trans. Comput.*, 61(6):804–816, June 2012.
- [133] Pci Sig. Single Root I/O Virtualization and Sharing Specification Revision 1.1. Technical report, January 2010.
- [134] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: Integrating a file system with gpus. 2013.

- [135] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [136] Prakalp Srivastava, Maria Kotsifakou, and Vikram S. Adve. HPVM: A portable virtual instruction set for heterogeneous parallel systems. *CoRR*, abs/1611.00860, 2016.
- [137] Greg Stitt and James Coole. Intermediate fabrics: Virtual architectures for near-instant fpga compilation. *IEEE Embedded Systems Letters*, 3(3):81–84, 2011.
- [138] 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.
- [139] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation’s hosted virtual machine monitor. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.
- [140] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. Gpvm: Why not virtualizing gpus at the hypervisor? In *USENIX Annual Technical Conference*, pages 109–120, 2014.
- [141] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. Gpvm: 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.
- [142] Michael M Swift, Brian N Bershad, and Henry M Levy. Improving the reliability of commodity operating systems. In *ACM SIGOPS operating systems review*, volume 37, pages 207–222. ACM, 2003.
- [143] Synergy Research Group, Reno, and NV. Hyperscale Data Center Count Passed the 500 Milestone in Q3. <https://www.srgresearch.com/articles/hyperscale-data-center-count-passed-500-milestone-q3>, October 2019. Accessed: 2020-6-9.
- [144] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.
- [145] Hiroshi Tezuka, Francis O’Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, pages 308–314. IEEE, 1998.
- [146] The gRPC Authors. gRPC. <https://grpc.io>. Accessed: 2019-04.
- [147] 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.
- [148] 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.

- [149] 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.
- [150] 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.
- [151] VMware, X.org, Nouveau. *Tungsten Graphics Shader Infrastructure*, 2012.
- [152] Duy Viet Vu, Oliver Sander, Timo Sandmann, Steffen Baehr, Jan Heidelberger, and Juergen Becker. Enabling Partial Reconfiguration for Coprocessors in Mixed Criticality Multicore Systems using PCI Express Single-Root I/O Virtualization. In *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, pages 1–6. IEEE, 2014.
- [153] 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.
- [154] Carl Waldspurger, Emery Berger, Abhishek Bhattacharjee, Kevin Pedretti, Simon Peter, and Chris Rossbach. Sweet spots and limits for virtualization. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '16*, pages 177–177, New York, NY, USA, 2016. ACM.
- [155] Carl Waldspurger and Mendel Rosenblum. I/o virtualization. *Commun. ACM*, 55(1):66–73, January 2012.
- [156] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 358–369. IEEE, 2016.
- [157] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized i/o devices. In *USENIX 2008 Annual Technical Conference, ATC'08*, pages 15–28, Berkeley, CA, USA, 2008. USENIX Association.
- [158] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Rouné, Rob Springer, Xuétian Weng, and Robert Hundt. Gpucc: An open-source gpgpu compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO '16*, pages 105–116, New York, NY, USA, 2016. ACM.
- [159] Lei Xia, Jack Lange, Peter Dinda, and Chang Bae. Investigating virtual passthrough i/o on commodity devices. *ACM SIGOPS Operating Systems Review*, 43(3):83–94, 2009.
- [160] Zhonghua Yang and Keith Duddy. CORBA: A platform for distributed object computing. *SIGOPS Oper. Syst. Rev.*, 30(2):4–31, April 1996.
- [161] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G Rogers. Pagoda: Fine-grained gpu resource virtualization for narrow tasks. In *ACM SIGPLAN Notices*, volume 52, pages 221–234. ACM, 2017.

- [162] 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.
- [163] 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.
- [164] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. Ava: Accelerated virtualization of accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, page 807–825, New York, NY, USA, 2020. Association for Computing Machinery.
- [165] Hangchen Yu and Christopher J Rossbach. Full virtualization for gpus reconsidered. In *14th Workshop on Duplicating, Deconstructing, and Debunking (WDDD), ISCA*, 2017.
- [166] Jose Fernando Zazo, Sergio Lopez-Buedo, Yury Audzevich, and Andrew W Moore. A PCIe DMA Engine to Support the Virtualization of 40 Gbps FPGA-accelerated Network Appliances. In *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*, pages 1–6. IEEE, 2015.
- [167] Lingfang Zeng, Yang Wang, Wei Shi, and Dan Feng. An improved xen credit scheduler for i/o latency-sensitive applications on multicores. In *Cloud Computing and Big Data (CloudCom-Asia), 2013 International Conference on*, pages 267–274. IEEE, 2013.
- [168] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. G-net: Effective gpu sharing in nfv systems. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 2018.