# Toward Efficient and Realizable Virtualization of Compute 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:

Christopher J. Rossbach

Donald E. Porter

Fabian N. Monrose

Kevin Jeffay

Michael Ferdman

# ABSTRACT

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

The proposed dissertation will focus on software virtualization of specialized compute devices (e.g., GPUs, TPUs) that are programmed through an API. Specialized compute accelerators, such as GPUs and TPUs, are usually controlled through a user-space API. The proposed dissertation will show that unlike with CPUs, where the ISA is the canonical interface exposed to the programmer, ISA virtualization is untenable for specialized compute accelerators. Further, the proposed dissertation will present a present a novel taxonomy, *IEMTS* for cleanly understanding the design space for virtualizing compute accelerators. Based on insights from this taxonomy, the proposed dissertation will present a novel virtualization technique, *Hypervisor-Interposed Remote Acceleration*, that is at once realizable and performant.

# 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 . . . . . . . . . . ix

LIST OF FIGURES . . . . . . . . . . x

LIST OF ABBREVIATIONS . . . . . . . . . . xiii

1 Introduction . . . . . . . . . . 1

1.1 History of Virtualization . . . . . . . . . . 2

1.2 Hardware Virtualization . . . . . . . . . . 3

1.3 Domain Specific Accelerator Virtualization . . . . . . . . . . 4

2 Background . . . . . . . . . . 7

2.1 Virtualization Properties . . . . . . . . . . 7

2.2 Domain Specific Accelerators . . . . . . . . . . 8

2.2.1 DSA Design . . . . . . . . . . 8

2.2.2 DSA Software stacks are silos . . . . . . . . . . 9

2.3 DSA Virtualization . . . . . . . . . . 10

2.3.1 Inefficacy of Traditional GPGPU Virtualization Techniques . . . . . . . . . . 11

2.3.1.1 Pass-through . . . . . . . . . . 11

2.3.1.2 Device emulation . . . . . . . . . . 11

2.3.1.3 Full virtualization . . . . . . . . . . 11

2.3.1.4 Mediated pass-through . . . . . . . . . . 12

2.3.1.5 Para-virtualization . . . . . . . . . . 12

2.3.1.6 API Remoting . . . . . . . . . . 12

2.3.1.7 Hardware virtualization support . . . . . . . . . . 13

2.4 Summary ........ 15

3 ISA virtualization is untenable for GPUs ........ 16

3.1 Background ........ 17

3.1.1 SVGA ........ 18

3.1.2 Mesa3D OpenCL Support ........ 18

3.1.3 GPU ISAs and IRs ........ 19

3.2 Design ........ 19

3.3 Implementation ........ 20

3.3.1 Trillium ........ 22

3.3.2 GPUvm ........ 24

3.3.3 User-space API remoting over RPC ........ 24

3.3.4 Optimizations ........ 24

3.4 Methodology ........ 25

3.4.1 Benchmarks ........ 25

3.4.2 Control Experiments ........ 26

3.5 Evaluation ........ 27

3.5.1 The impact of vISA choice ........ 27

3.5.2 End-to-End ........ 28

3.6 Conclusion ........ 29

4 IEMTS — A new accelerator virtualization taxonomy ........ 30

5 Hypervisor-mediated API-remoting ........ 33

5.1 Introduction ........ 33

5.2 Accelerator Silos ........ 34

5.3 Design ........ 36

5.3.1 AvA Components ........ 37

5.3.2 Developer Work-flow ........ 38

5.3.3 Communication Transport ........ 38

5.3.4 Sharing and Protection ........ 39
5.3.5 Scheduling and Resource Allocation ........ 39
5.3.6 Memory Management ........ 39
5.4 CAvA and LAPIS ........ 39
5.4.1 LAPIS ........ 40
5.4.2 Code Generation ........ 49
5.4.3 Mapped memory ........ 51
5.4.4 Resource accounting and scheduling ........ 52
5.4.5 VM Migration ........ 53
5.4.6 Memory Over-subscription ........ 53
5.4.7 Limitations ........ 54
5.5 Implementation ........ 54
5.5.1 Transport ........ 55
5.5.2 Hypervisor Interposition and Mediation ........ 55
5.5.2.1 Policies in eBPF ........ 55
5.5.2.2 Scheduling ........ 56
5.5.3 Shadow Resources ........ 56
5.5.4 Callbacks ........ 57
5.6 Evaluation ........ 57
5.6.1 Development Effort ........ 58
5.6.2 End-to-end Performance ........ 59
5.6.3 Micro-benchmarks ........ 60
5.6.3.1 Asynchrony Optimizations ........ 61
5.6.4 Scalability ........ 61
5.6.5 API Rate Limiting ........ 62
5.6.6 Live Migration ........ 62
5.7 Related Work ........ 64
5.8 Conclusion ........ 65

6 Proposed work — vTask ........ 66

7 Related Work ........ 69

7.1 GPU Virtualization ........ 69

7.1.1 Full Virtualization ........ 70

7.1.2 API Remoting ........ 71

7.1.3 Para-virtualization ........ 72

7.2 Language-level Virtualization ........ 73

8 Conclusion ........ 74

BIBLIOGRAPHY ........ 75

# LIST OF TABLES

3.1 EVALUATION BENCHMARKS IN THREE CATEGORIES[a] . . . . . 26

4.1 Comparing virtualization designs using the IEMTS framework. . . . . . 31

5.1 LAPIS syntax which values descriptors apply to and when those descriptors apply. . . . . . 43

5.2 LAPIS descriptors for specifying values and objects. . . . . . 44

5.3 LAPIS descriptors for functions. . . . . . 46

5.4 An except from the LAPIS standard library for use in expressions and utility functions. . . . . . 47

5.5 Development effort for forwarding different APIs, along with the benchmarks [**?** **?** 44] 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. . . . . . 59

7.1 Existing GPU virtualization proposals, grouped by approach. Previously published in the Trillium paper [31]. . . . . . 70

## 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. . . . . . . . . . 9

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.]** . . . . . . . . . . 13

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. . . . . . . . . . . 14

3.1 The design of SVGA. . . . . . . . . . 17

3.2 XEN-SVGA and TRILLIUM designs. (a) The TRILLIUM stack. (b) XEN-SVGA approximates the SVGA model extended to support GPU Compute. (c) The design of TRILLIUM with shadow pipe. . . . . . . . . . 21

3.3 Xen-based virtualizaton designs. (a) Trap-based virtualization: GPUvm. (b) User-space API remoting over RPC—dashed arrows indicate API-REMOTE-CPU, while solid ones indicate API-REMOTE-GPU. . . . . . . . . . 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. The gRPC transport overhead is removed from the reported measurements, which is up to 10% of the total execution time for API remoting, and 40% for TRILLIUM. . . . . . . . . . 27

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. . . . . . . . . . 27

5.1 The number of accelerators (discrete GPUs and AI accelerators) and APIs released since 2010 compared to the number of accelerators officially supported by production hypervisors (VMware ESX, Citrix XenServer, and Microsoft Hyper-V). This data was drawn from release notes and specification sheets. . . . . . . . . . 33

5.2 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. . . . . . . . . . 35

5.3 Overview of AvA. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

5.4 An example LAPIS description for the CUDA driver API. Code elements in **bold blue** are LAPIS keywords, elements in *italic green* are runtime library calls, elements in gray are function prototypes incorporated by CAvA from the original CUDA header file, and the remaining code is programmer-provided LAPIS. The variable **`index`** is a LAPIS builtin which provides the index of the current element of a buffer. The specification of the argument `extra` to `cuLaunchKernel` is complex due to alignment and padding, and is elided from this example for clarity. The file `policy_kern.c` is shown in Figure 5.7. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

5.5 An outline of the generated guestlib code for `cuStreamSynchronize` and `cuMemcpyDtoHAsync`. The real code manages a number of additional details, e.g., threads. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

5.6 An outline of the generated API server code for `cuStreamSynchronize` and `cuMemcpyDtoHAsync`. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

5.7 An example eBPF policy program (simplified for clarity). This is referenced from Figure 5.4 as `policy_kern.c`. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

5.8 End-to-end execution time on virtualized APIs or accelerators normalized to native execution time. `tf_py` is the handwritten TensorFlow Python API remoting with AvA API-agnostic components. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

5.9 End-to-end execution time on virtualized CUDART and CUDA-accelerated TensorFlow APIs normalized to native execution time. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

5.10 Overhead on a micro-benchmark with varying work per call and data per call. The plot is log-log and the trend is linear. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

5.11 End-to-end runtime of CUDA benchmarks (relative to native) using synchronous and asynchronous specifications. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

5.12 Scalability of multiple VMs running a single application each, and multiple applications in a single VM, with AvA. Runtime is relative to running the same number of applications natively. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

5.13 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. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

5.14 Live migration downtime for single-threaded OpenCL benchmarks on NVIDIA GTX 1080. This downtime is in addition to the $\sim 75\,\mathrm{ms}$ 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. . . . . 63

6.1 Data processed by two API stacks must pass through the guest application . . . . . 66

# LIST OF ABBREVIATIONS

| | |
|---|---|
| DSA | Domain Specific Architectures |
| 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

This dissertation is concerned with fair, efficient and safe sharing of domain specific accelerators between mutually distrustful users.

Domain Specific Accelerators (DSAs) are programmable compute units that are specialized to a particular class of computation in order to improve performance, or to optimize energy usage (or both) for that class of computation. Domain Specific Accelerators are seeing wide adoption in data centers as they afford the data center provider a very low cost/performance on the specific computation they are specialized for. For example, a 2013 projection by Google showed that they would need to double the installed CPU compute capacity in in order to support just three minutes of Google Voice Search per user per day, using speech recognition DNNs [71]. This realization 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× and 79× higher Performance/Watt respectively over the CPUs and Nvidia k80 GPUs that were prevalent in their data centers at the time [72].

Data centers themselves are proliferating rapidly: a recent report by Synergy Research [116] found that there were more than 500 hyperscale data centers operational across the world at the end of 2019. This proliferation of data centers is primarily motivated by the increased need for computation resulting from the unquestionable permeation of digital services into every facet of our lives (e.g., Google Maps, video streaming, digital communication). Data centers leverage the cost efficiency of centralized computing, while using techniques such as virtualization to provide the flexibility of dedicated/distributed computing.

Virtualization is a cornerstone technology in our current computing landscape. Unlike physical compute, virtualized compute resources can be securely multiplexed among many users, which allows for high utilization of the installed compute capacity. Further, virtualization provides customers with the ability to scale elastically: 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 just release the excess compute resources, thereby only paying for what they are actively using.

Virtualization of domain specific accelerators remains an open problem: DSAs are dedicated to single users instead of being multiplexed in today's data centers. Cloud providers such as Amazon expose Graphical Processing Units (GPU) and Field Programmable Gate Arrays (FPGA) to individual VMs via PCIe-passthrough, thereby bypassing the hypervisor, and giving up the consolidation and fault tolerance benefits of virtualization. This dissertation explores the trade-offs in the design space for software-based virtualization schemes for DSAs and proposes a low-overhead novel virtualization scheme that interposes the user-facing Application Programmer's Interface (API) while using automation to compensate for the resulting development complexity.

## 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 [64]. The Cambridge University/Ferranti Inc. Atlas [76] 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.

The next era dealt with the virtualization of the individual machine with it's processor and peripherals. *Hardware virtualization* [28]—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 [48] 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 [96], and was widely considered unvirtualizable [57, 42]. Multiple vendors introduced *software emulation* based solutions to enable the execution of one OS on top of another (e.g., Insignia SoftPC, Connectix VirtualPC, VMware Workstation). Over time, better techniques were devised to virtualize the x86 ISA, including ISA extensions (e.g., AMD-V and Intel VT-x) to enable native execution of virtualized applications, only trapping to the hypervisor when the application attempts to perform sensitive operations.

Hardware interfaces weren't 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 Hardware Virtualization

This dissertation is primarily concerned with hardware virtualization of domain specific accelerators. Hardware virtualization is vital to high utilization of available physical resources in large computing installations, e.g., hardware virtualization is foundational to *cloud computing*. There have been many attempts to define the fundamental characteristics of hardware virtualization: Popek and Goldberg came up with three properties that a virtualization scheme must exhibit—*equivalence, performance, and safety*, while Bugnion, Nieh and Tsafrir [43] provided a more succinct definition—*"the application of the layering principle with enforced modularity such that the exposed resource is identical to the underlying resource"*. For the purposes of this dissertation, we concern ourselves with *realizable, fair, isolated, and efficient sharing of domain specific accelerators among mutually distrustful entities.*

Hardware virtualization typically involves mediating access to the shared resource either by exposing an interface that is identical to that of the physical resource (*full-virtualization*), or by exposing an alternative interface, operations on which are in-turn synthesized to the native interface (*para-virtualization*). The exposed interface is 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* (also known as the *Virtual Machine Monitor*). While operations in the resulting *virtual machine* may be directly executed on the physical hardware for improved performance, as in the case of hardware-assisted virtualization schemes like 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 [30]. Dominant ISAs (e.g., x86 and ARM) provide extensions to enable low-overhead virtualization (e.g.,Intel VT-x, AMD-V). Both of these are foundational building blocks for cloud computing [1].

### 1.3 Domain Specific Accelerator Virtualization

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 [113]) 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 [53]), 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 [124, 55, 99] 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 [53]). Over the last decade, GPUs have also been re-purposed for general purpose compute (commonly known as GPGPU). 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 it's 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.

Chapter 3 of this dissertation presents our findings from attempting to extend the SVGA model of GPU virtualization to virtualize GPGPUs. We find that the tight coupling between ISA virtualization and device virtualization in SVGA leads to poor performance for GPGPU compute. Eliding ISA virtualization enables the resulting prototype, Trillium [31], to outperform all other traditional virtualization schemes that retain hypervisor interposition. However, Trillium is still 2—3× slower than RPC-style API forwarding.

Virtualization schemes are traditionally taxonomized according to the core techniques employed (e.g. emulation, full- or para-virtualization, API remoting, etc.), and evaluated in a property trade-off space comprising performance, compatibility, interposition, and isolation. We argue that both the de facto taxonomy and the property trade-off space are illustrative but not informative for GPGPU virtualization: there is a large body of research that has had little influence on practice. We suggest an alternative framework called `IEMTS` that teases apart design axes that are implicitly and unnecessarily intertwined in much of the literature. By focusing on the **I**nterface interposed, the interposition **E**ndpoints, the **M**echanism of interposition, the **T**ransport used to move the interposed operations between the guest and the host, and the mechanism used to **S**ynthesize the interposed interface, `IEMTS` enables a clearer understanding of trade-offs in prior designs and provides a model for comparison of alternative designs. Chapter 4 presents `IEMTS`, along with analysis of traditional virtualization techniques in the context of GPGPUs.

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 [133] and an ASPLOS'20 paper [134]. 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.

# CHAPTER 2: BACKGROUND

Bugnion, Nieh and Tsafrir [43] 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 Tsafrir*'s *Hardware and software support for virtualization* [43] 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 [126]. 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 [113, 135]; *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 remoting [3, 55, 14] 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 [56] 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) [71], a DSA for the Tensor-based computation popular in Neural Networks. The first generation of TPUs were empirically found to have 200× and 79× higher Performance/Watt respectively over the CPUs and Nvidia k80 GPUs that were prevalent in Google's data centers at the time [72].

### 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 (tesselation, 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

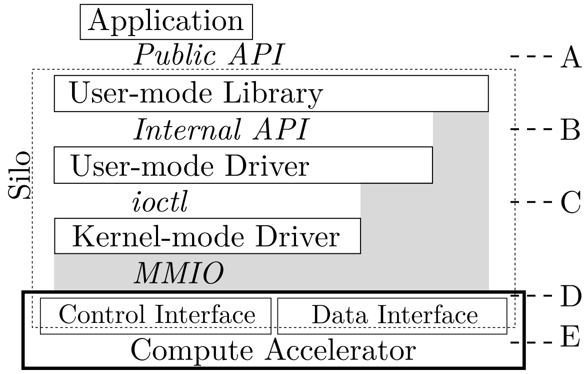


Figure 2.1: An accelerator silo. The public API and the interfaces with striped backgrounds are interposition candidates. All interfaces with backgrounds are proprietary and subject to change.

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 [18, 127, 92, 93, 17], and abundant prior research into multiplexing Domain Specific Accelerators (DSAs) [93, 17, 138, 127, 92, 131], 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 [53, 82] support graphics frameworks like Direct3D [40], OpenGL [104]. 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 [5]. In practice, they have significantly different goals: for graphics, virtualization designs target an interactive frame rate (18-30 fps [22]); 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 [89], research attention to OS abstractions for GPGPUs [101, 102, 109, 73, 74, 78] 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 (§ 7) 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 [27]) 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 [38] 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 [113, 135]). 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 [118, 94, 130] 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 [113, 53, 120, 83, 65, 60, 87, 97, 112, 128, 34] *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 [53] encapsulates multiple GPU programming frameworks, but keeping up with the evolution of those frameworks has proved untenable: SVGA remains multiple versions behind current frameworks [21, 19].

#### 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 [106, 63, 58], on a dedicated appliance VM [124], or on a remote server [55, 99, 86, 77, 36, 54, 85]. API Remoting is similar to RPC [103, 90] or system call interposition [34, 115, 35, 81]. Limited interposition frequency, batching opportunities [55] and high-speed networks [6, 3] reduce overheads, making this class of designs appealing to industry. Dell XaaS [69], BitFusion FlexDirect [3], and Google Cloud TPUs [9] currently use it to support GPUs, FPGAs, and TPUs. However, API

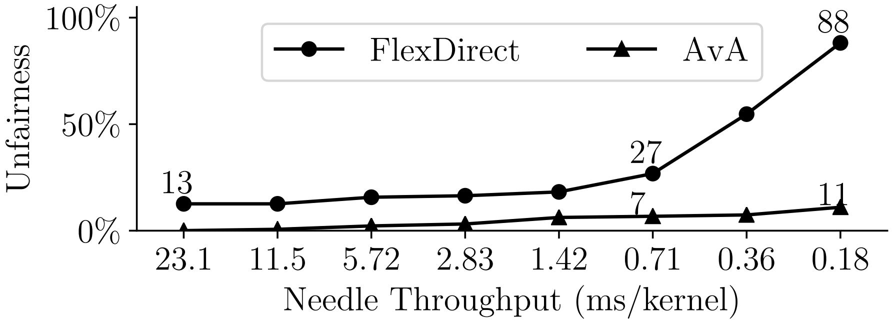


Figure 2.2: Unfairness in slowdown between `needle` and `hotspot` applications in separate VMs running GPU kernels iteratively with BitFusion FlexDirect. When running alone, `hotspot` has throughput of 126.3 ms/kernel. Fairness is calculated by $|s_1 - s_2| \ / \ (s_1 + s_2)$, where $s_i$ is the slowdown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

remoting 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 [3] 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 remoting systems are by themselves massive undertakings without any hypervisor integration: systems like Bitfusion FlexDirect [3], and rCUDA [55] 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.

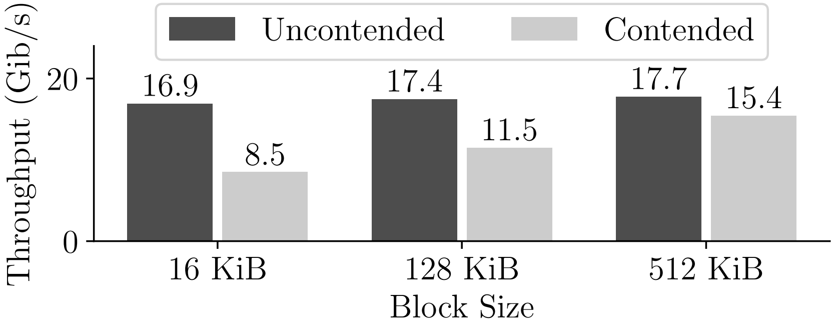


Figure 2.3: Throughput achieved by three instances of QATzip (running in VMs with SR-IOV pass-through) with different block sizes, running separately (**`Uncontended`**) and concurrently (**`Contended`**). Slowdown during concurrent execution is dependent on block size, i.e., the QAT HW scheduler cannot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

SR-IOV exhibits close to native performance [51], 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 [50, 52], 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 [12] (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 [2, 66], none of the TPUs we evaluate support it; and SR-IOV *interface* IP blocks from FPGA vendors (used by [123, 136, 95, 67]) 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 5.2), 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

In many parallel computing domains, compute density and programmability [15, 111, 62] have made GPUs the clear choice for efficiency and performance [10]. Popular machine learning frameworks such as Caffe [70], Tensorflow [26], Microsoft CNTK [132], and Torch7 [47] 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 [25].

Despite much prior research [125, 68, 29, 121] 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 [33, 118], 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 [39, 77, 99, 124, 55] 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 [101].

To recover hypervisor interposition while maintaining low-overhead, we explore retrofitting GPGPU support into a virtual GPU device: We added support for OpenCL to an implementation of the SVGA [53] design in Xen, by implementing the key missing component—a compiler for SVGA's TGSI virtual ISA. This effort helped us realize that because GPUs already support vendor-specific virtual ISAs (vISAs), the additional vISA provides little benefit. In fact, we found that it harms performance by necessitating a translation layer that obscures the program's semantic information from the final vendor-provided compiler. Drawing on this lesson, we adapted TRILLIUM to take a more flexible approach to ISA virtualization: eliding it entirely when the host GPU stack bundles a compiler (most do), and using LLVM IR, when necessary, to provide a common target for GPGPU drivers.

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

### 3.1 Background

Existing GPU virtualization solutions [53, 82] support graphics frameworks like Direct3D [40], OpenGL [104]. In principle, there should be no fundamental difference between GPU virtualization for graphics versus *compute* workloads. In practice, they have significantly different goals: For graphics, virtualization designs target an interactive frame rate (18-30 fps [22]). 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 [89], research attention to OS abstractions for GPGPUs [101, 102, 109, 73, 74, 78] has yielded little consensus.

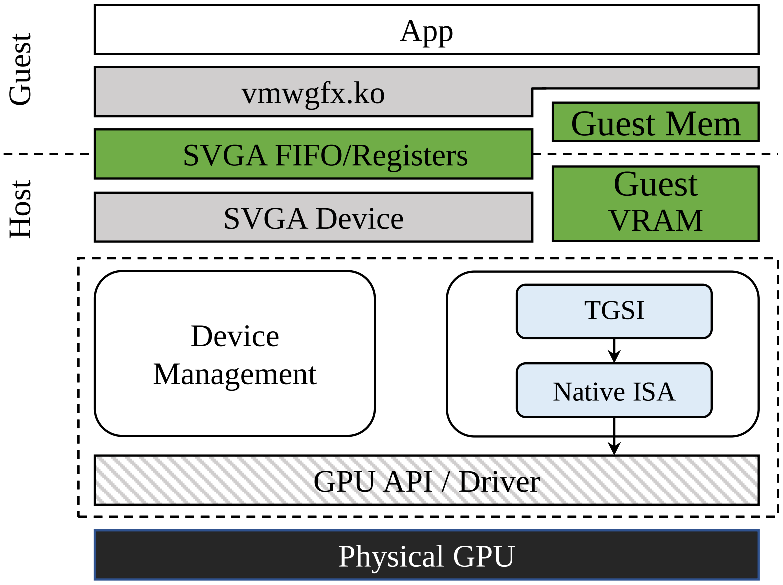


Figure 3.1: The design of SVGA.

### 3.1.1 SVGA

SVGA [53] 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 3.1 presents an overview of SVGA.

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

SVGA uses the DirectX [40] API as its internal protocol, thereby realizing an API-remoting design. The transport layer and protocol are completely under the control of the hypervisor, enabling many of the benefits of API-remoting while ameliorating its downsides. However, using the DirectX API as a transport protocol requires that the driver and hypervisor translate guest interactions into DirectX whether they are natively expressed in DirectX or not. Coupling the transport layer with a particular version of the DirectX protocol has led to serious complexity and compatibility *challenges*: supporting each new version of the API takes many person-years (VMware introduced support for DirectX 10 (introduced in 2006) in 2015!). SVGA also supports a virtual GPU ISA called TGSI [122]. 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.

### 3.1.2 Mesa3D OpenCL Support

The Mesa3D Graphics Library [24] is an open-source graphics framework that implements graphics runtime libraries (e.g., OpenGL [104], Vulkan [75], Direct3D [40], and OpenCL [111]) on most GNU/Linux installations. It also includes official device drivers, written in a common framework, Gallium3D [23], for Intel and AMD GPUs. Support for NVIDIA GPUs is provided via reverse-engineered open-source Nouveau driver. Gallium3D imposes TGSI as the common virtual ISA for compute shaders, and decomposes drivers into two components: *state trackers*, which keep track of the device state, and *pipe drivers*, which provide an interface for controlling the GPU's graphics pipeline.

OpenCL support was first introduced in Mesa3D 9.0 with the release of the Clover state tracker. It was envisioned that Clover would leverage the LLVM [84] compiler to lower the OpenCL source to TGSI. Despite much effort by the open-source community [7, 11], 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.

### 3.1.3 GPU ISAs and IRs

GPU front-end compilers produce code in virtual ISAs (NVIDIA PTX and LLVM IR for AMD) which are subsequently finalized using JIT compilers in the GPU driver to the native ISA (SASS and GCN). The vISA remains stable across generations to preserve compatibility, while the physical ISA is free to evolve. TGSI, the virtual ISA used in both the Mesa stack and SVGA, plays a similar role—enabling interoperability between graphics frameworks and GPUs from different vendors. An improved virtual ISA, SPIR-V, has been proposed as a new standard [75] and an effort is under way to replace TGSI with SPIR-V in the Mesa3D stack.

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

## 3.2 Design

TRILLIUM exports 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, Trillium forwards the lowest layer in the GNU/Linux Graphics stack: the pipe-driver, effectively remoting OpenCL/CUDA API calls in the guest to the OpenCL/CUDA library in the host.

Our experience implementing the required TGSI vISA support in the Mesa graphics stack led us to believe that the TGSI layer is unnecessary. Not only does this translation introduce additional complexity in the guest stack, it also hurts performance, as we demonstrate in Section 3.5. The guest OpenCL compiler cannot target the native GPU architecture, and semantic information is lost to the host compiler. Further, while incorporating a TGSI compiler is possible in open frameworks like OpenCL, the task is significantly more daunting for closed frameworks like CUDA. Attempts to translate between TGSI and NVIDIA SASS in the reverse-engineered Nouveau driver understandably results in code that is significantly less performant than that produced by the proprietary stack.

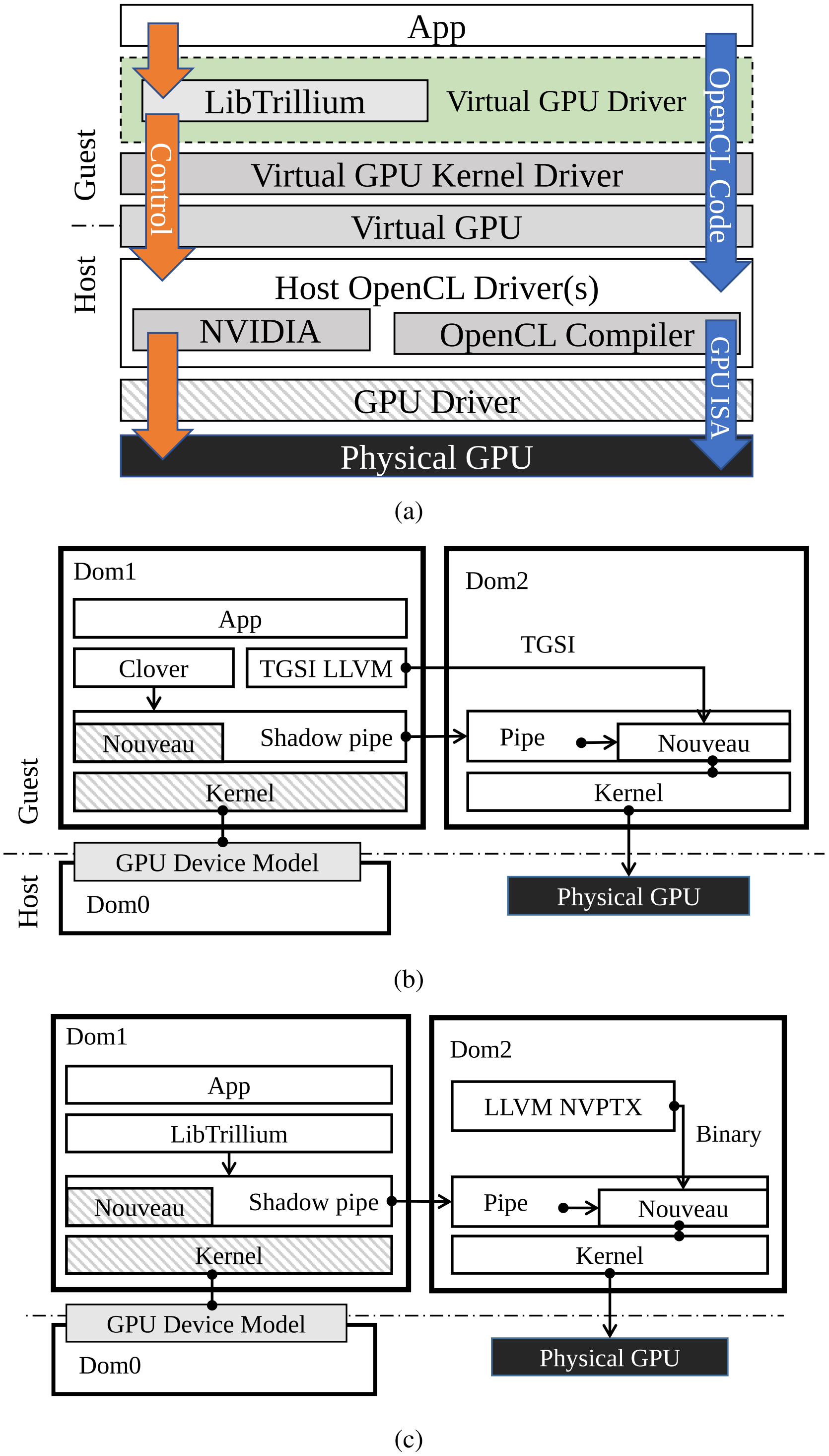
TRILLIUM takes a different approach: TRILLIUM forwards API calls for compiling OpenCL code to the hypervisor. The OpenCL compiler in the host OpenCL framework (optimized for the physical hardware by the hardware vendor) is invoked on the forwarded OpenCL code to lower it directly to the physical device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

Figure 3.2a shows the Trillium design layers in a generic hypervisor stack. The OpenCL API is forwarded from the driver similar to the SVGA model. The OpenCL compute kernel (to be run on the GPU), can be passed through to the host via hypercalls in the driver, without being translated to any vISA, where it will be translated and optimized for the physical GPU in a virtual appliance (Dom 2 in Figure 3.2a).

TRILLIUM does not currently guarantee performance isolation and relies on the hardware scheduler. Performance isolation can easily be implemented via a rate-limiting API scheduler in the hypervisor, such as in GPUvm [114].

### 3.3 Implementation

We evaluated the TRILLIUM design against a representative of each traditional virtualization technique: full-virtualization, user-space API-remoting and SVGA. Due to the difficulty of implementing a trap-based virtualization scheme, we chose to evaluate against GPUvm [114], the only existing open-source implementation. GPUvm is tightly coupled with the Xen hypervisor [37]. As a result, all the other prototypes were built on the Xen hypervisor to keep the platform common for fair comparison.



(a)

(b)

(c)

Figure 3.2: XEN-SVGA and TRILLIUM designs. (a) The TRILLIUM stack. (b) XEN-SVGA approximates the SVGA model extended to support GPU Compute. (c) The design of TRILLIUM with shadow pipe.

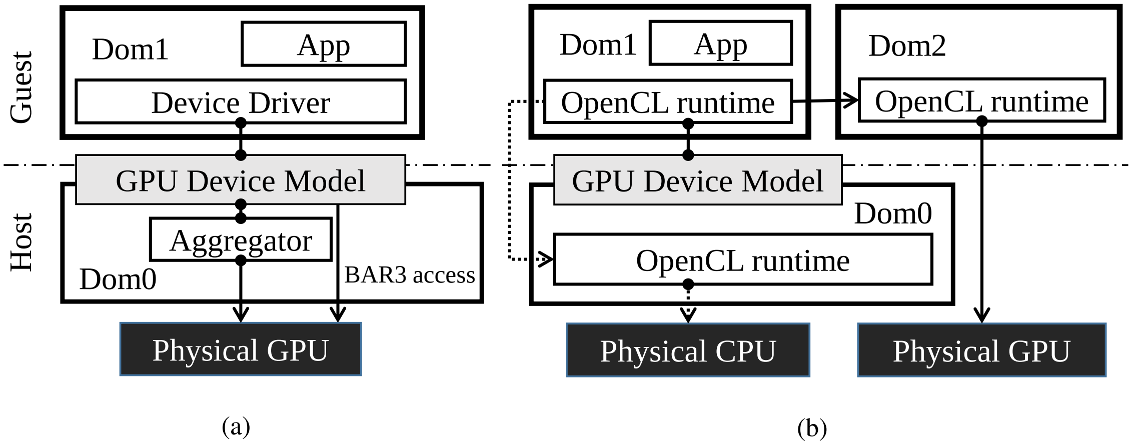


Figure 3.3: Xen-based virtualizaton designs. (a) Trap-based virtualization: GPUvm. (b) User-space API remoting over RPC—dashed arrows indicate API-REMOTE-CPU, while solid ones indicate API-REMOTE-GPU.

### 3.3.1 Trillium

We initially implemented TRILLIUM on Xen following the SVGA design, by implementing OpenCL support in a virtual device and extending the Mesa stack with TGSI support (see Section 3.3.1 for details). The generated TGSI is sent to the host via RPC, and then finalized to a binary that can be run on the physical NVIDIA GPU using the open source Nouveau driver. Upon empirically finding that TGSI is a performance bottleneck, we revisited the basic design. We preserve the original prototype, hereafter called XEN-SVGA, as a baseline representative of the original SVGA design: this design is shown in Figure 3.2b. The current TRILLIUM design is shown in Figure 3.2c.

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

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

Our implementation relies on gRPC as a transport mechanism between the guest and the host, as an implementation convenience. As zero-copy transfer [46, 117] and hypercall [98] 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 3.5. The overheads stem from remoting calls to the privileged domain over the network, which is especially significant since a single OpenCL API call may be decomposed into many pipe-driver APIs, and from the large amount of kernel input data that must be copied between VMs.

***LLVM TGSI Back-end*** The Mesa3D stack implements OpenCL support via a state-tracker called Clover. Clover provides the library for the OpenCL application to link against, while most of the compilation is handled by invoking the OpenCL and C++ front-ends of the LLVM [84] compiler framework. Clover provides much of the front-end infrastructure required to support GPGPU computing in XEN-SVGA and TRILLIUM.

Historically, lack of a working TGSI back-end in LLVM, despite several attempts at building one in the past 5 years [7, 11], has left OpenCL support for NVIDIA GPUs and SVGA in Mesa3D incomplete. In order to support OpenCL in XEN-SVGA, we implemented an LLVM TGSI back-end. While the TGSI back-end is not yet mature, we have added support for a majority of the 32-bit integer and floating point operations, intrinsics, memory barriers, and control flow. Using this backend we are able to compile and run 10 out of the 12 Rodinia benchmarks [45] used to benchmark GPUvm. Because the compiler is built using the LLVM framework, it enjoys all of the IR-level optimizations in LLVM.

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

### 3.3.2 GPUvm

GPUvm [114] is an open-source trap-based interposition design (a simplified block-diagram representation is shown in Figure 3.3a). The application domain (Dom 1) is presented with a GPU Device Model, which is emulated in the privileged domain (Dom 0). The emulation layer in Dom 0 interposes, validates, and fulfills all attempts to access the GPU. GPUvm has not been maintained: The last release, in 2012, is based on Xen 4.2.0 and runs on Fedora 16 [135]. In order to compare all prototypes on the same modern platform, we ported GPUvm to Ubuntu 16.04 with Xen 4.8.2.

### 3.3.3 User-space API remoting over RPC

In order to faithfully mimic user-level API-remoting-over-RPC systems [55, 77, 39], OpenCL API calls are trapped by a user-space shim library and forwarded via RPC from one appliance VM, which is the OpenCL "client", to another appliance VM, which acts as the OpenCL "server". Figure 3.3b shows the setup of the two API-remoting schemes we considered: API-REMOTE-GPU and API-REMOTE-CPU. The black arrows indicate the workflow of API-REMOTE-GPU, where the OpenCL server runs the OpenCL commands on a physical GPU using the NVIDIA OpenCL framework. The grey arrows show the API-REMOTE-CPU setup, where the OpenCL commands are executed on a multi-core CPU (Intel CPU Xeon E5-2643) using the Intel OpenCL SRB 5.0 framework. RPC is implemented using gRPC 1.6 (based on Google ProtocolBuffers 3.4.0) and inter-service communications are implemented over XML-RPC 1.39. Lower-overhead data-movement techniques, such as zero-copy, can be applied when both the client and the server are on a local machine.

### 3.3.4 Optimizations

TRILLIUM interposes at the pipe-driver API yielding fine-grained interposition, and therefore finer-grained multiplexing of the GPGPU. However, interposing at this layer also results in significant transport overhead. Many pipe-driver functions are responsible for context management and information retrieval—operations that do not result in interaction with the GPU. We reduce communication overhead by batching these types of API-calls, taking care to fall back to synchronous API-forwarding when any pipe-driver API calls that interact with the physical GPU are invoked.

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

## 3.4 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 [74] an open source CUDA runtime (released in 2012) implemented using Nouveau [8] GPU drivers, and the CUDA 4.2 compiler on Linux Kernel 3.6.5. GDev has not been maintained since 2014, and the effort to update it was too onerous. Experiments to control for hardware versions are reported in 3.4.2.

### 3.4.1 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 top of the GDev CUDA runtime. Care was taken to ensure that the CUDA and OpenCL versions of the benchmarks use the same parameters, datasets, memory barriers, sync points, etc. Experiments to control for the programming framework are reported in 3.4.2.

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

Table 3.1: EVALUATION BENCHMARKS IN THREE CATEGORIES[a]

| 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 |

[a]. Interposition-**d**ominant, interposition-**r**are, and **m**oderate workloads.

lie somewhere in between the other two. Two benchmarks were selected from each category to be used in the evaluation (the optimizations described in 3.3.4 take significant manual effort).

### 3.4.2 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.

## 3.5 Evaluation

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

### 3.5.1 The impact of vISA choice

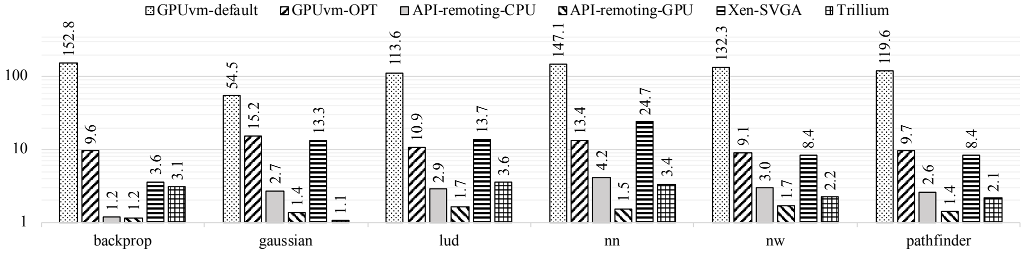


Figure 3.4: End-to-end execution times of benchmarks on virtualization prototypes, relative to end-to-end execution time on the NVIDIA CUDA runtime in a native setting. The gRPC transport overhead is removed from the reported measurements, which is up to 10% of the total execution time for API remoting,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

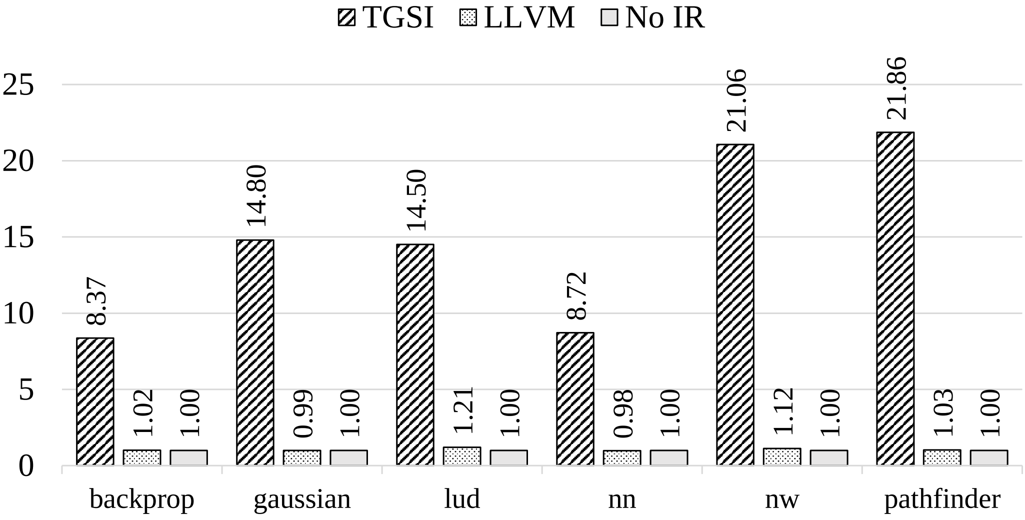


Figure 3.5: Kernel execution slowdown due to virtual ISAs. TGSI: the LLVM TGSI back-end compiler used in XEN-SVGA. LLVM: LLVM NVIDIA PTX (NVPTX) back-end used in TRILLIUM. No IR: native NVIDIA compiler.

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

To understand the impact of the virtual ISA on the quality of the generated GPU code we measured GPU execution time for NVIDIA SASS kernels generated in 3 ways: a) using the Mesa3d OpenCL stack (OpenCL→ TGSI→ SASS), b) using the LLVM OpenCL stack (OpenCL→ LLVM IR→ SASS), and c) 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×, with a harmonic mean of 13× 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 [129] to optimize the LLVM tool-chain for NVIDIA GPUs.

The TRILLIUM design uses LLVM IR as the common virtual ISA for GPGPU applications, where necessary: OpenCL code is compiled to PTX using the LLVM NVPTX back-end in the guest, and then finalized and executed in the host using the NVIDIA CUDA framework.

### 3.5.2 End-to-End

We compare TRILLIUM against full-virtual (GPUVM-DEFAULT and GPUVM-OPT), API remoting (API-REMOTE-GPU and API-REMOTE-CPU) and para-virtual (XEN-SVGA) systems. XEN-SVGA approximates an SVGA-like design in Xen (Mesa3D with TGSI). TRILLIUM bypasses translation from OpenCL to TGSI. To characterize the behavior a full-virtualization design, we measure GPUvm [114] in its default configuration (worst case) shown as GPUVM-DEFAULT, and in its fully optimized configuration, labeled GPUVM-OPT.

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. TRILLIUM fares well with best case performance of just 1.1× over native, and within 3.6× at worst. XEN-SVGA is sensitive to the performance lost in GPU kernel code resulting from redundant compilation through TGSI (which adds significant overheads as previously shown in Figure 3.5). GPUVM-OPT exhibits about 9.1× slowdown for applications with short-lived kernels (e.g. Needleman-Wunsh algorithm); the overhead can be as high as 15.2× when the workload has long-running kernels (e.g. Gaussian Elimination).

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

## 3.6 Conclusion

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

# CHAPTER 4: IEMTS — A NEW ACCELERATOR VIRTUALIZATION TAXONOMY

Traditionally, virtualization designs have been taxonomized according to the core techniques employed (e.g. emulation, full- or para-virtualization, API remoting, etc.), and evaluated in a property trade-off space comprising performance, compatibility, interposition, and isolation. *Isolation* ensures that mutually distrustful guests cannot access each other's data or harm each other's performance. *Compatibility*, characterizes how well a design preserves the freedom of hardware and software components to evolve independently: e.g. changes in the hypervisor should not force changes to guest software. Virtualization provides an indirection layer between logical and physical resources by *interposing* a well-defined interface. The quality of interposition determines the nature of benefits (e.g. extent of consolidation) afforded by a virtualized system [126].

Virtualization techniques are well explored, yielding conventional wisdom about their fundamental trade-offs. For example, *full virtualization* interposes the software-hardware interface to provide a virtual view of the underlying hardware. This enables guests to run unmodified OS and application binaries, yielding high compatibility. However, hardware interfaces for GPUs rely heavily on MMIO and communication through memory, which necessitates page-fault-based interposition [119, 13, 79, 91] techniques that cripple performance. *Para-virtual* designs export an abstract device to the guest, but require hypervisor-specific drivers and runtime libraries in the guest, trading compatibility for improved performance. *API remoting (or forwarding)* [63, 53, 59, 106] aggregates high-level API calls issued in VMs, running them on the host or in a dedicated appliance VM. This technique can provide near native performance because API calls are infrequent, but has poor compatibility because it requires changes in guest applications or libraries.

We argue that the current *de facto* taxonomy and property trade-off space are illustrative but not informative for GPUs: there is a large body of research that has had little influence on practice. First, Classifying virtualization designs as API-remoting vs. full vs. para-virtual captures important concepts, and emergent properties compactly, but doesn't explain their correlation to properties like performance. Second, virtualization properties such as compatibility, isolation, and interposition

|  | GPUvm | VMware SVGA |  | rCUDA |
|---|---|---|---|---|
|  |  | Control Interface | GPU ABI |  |
| 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.

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. “Paravirtualization” 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.

We argue that practical design goals, such as providing a virtualization layer with specific characteristics, get obscured when these properties are considered as a set of constraints that must be preserved, without first refining for context. Further, production systems, such as VMware SVGA [53], compose multiple virtualization techniques in order to leverage the best properties of each technique, especially in the presence of multiple interfaces.

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

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

Table 4.1 presents analysis of three prior GPU virtualization systems under the IEMTS framework. A quick glance at the table tells us that GPUvm's [113] performance woes arise from its realiance on trapping and emulating the guest's 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 helped us realize that ISA-virtualization is not necessary for accelerator virtualization in the Trillium project. Further, it became obvious to us that the control interface virtualization in SVGA and the API-remoting in rCUDA look almost the same under IEMTS with the exception of the transport. This observation led us to design AvA's hypervisor-mediated API-remoting scheme, and shift our attention to solving the compatibility effort via automation.

# CHAPTER 5: HYPERVISOR-MEDIATED API-REMOTING

## 5.1 Introduction

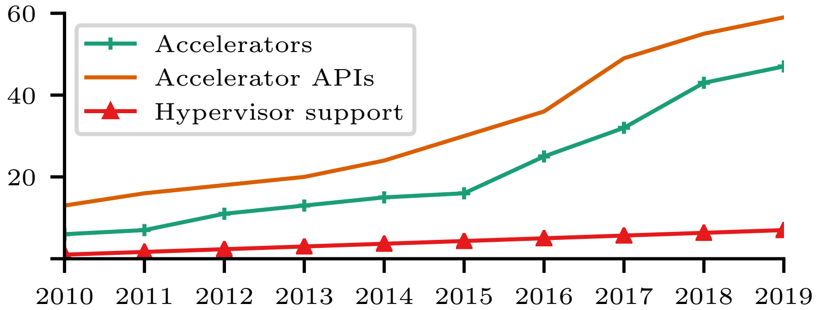


Figure 5.1: The number of accelerators (discrete GPUs and AI accelerators) and APIs released since 2010 compared to the number of accelerators officially supported by production hypervisors (VMware ESX, Citrix XenServer, and Microsoft Hyper-V). This data was drawn from release notes and specification sheets.

Hypervisors have not kept pace with accelerator innovation. Figure **??** shows the evolution of accelerator framework APIs, accelerator architectures, and hypervisor support for them over the last decade. Specialized hardware and frameworks emerge far faster than hypervisors support them and the gap is widening. Many factors contribute to this trend, but lack of demand is *not* among them, evinced by the wide variety of accelerators currently available from cloud providers [1**? ? ? ?** , 88, 9]. The challenge is technical: hypervisor-level accelerator virtualization requires substantial engineering effort and the design space features multiple fundamental trade-offs for which a "sweet spot" has remained elusive.

Practical 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* (Figure 5.2) 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.

This chapter describes AvA, which addresses the fundamental limitations of existing accelerator virtualization techniques. AvA combines API-agnostic para-virtual stack components with a DSL (Domain-Specific Language) and a compiler to automate construction and deployment of guest libraries, hypervisor-level resource management, and API servers. AvA uses an abstract para-virtual device to serve as a transport endpoint for forwarding the public APIs of vendor-provided frameworks (e.g. CUDA or TensorFlow). Unlike currently popular user-space API remoting solutions [3, 69, 124, 55, 100], AvA preserves hypervisor-level resource management and strong isolation using a novel technique called *Hypervisor Interposed Remote Acceleration (HIRA)*. HIRA forwards API calls over hypervisor-managed communication channels, inserting automatically-generated resource management components at the transport layer to enforce policies from the DSL specification. Critically, *automation* from AvA enables hypervisors to keep up with fast accelerator evolution: automatic generation of components dramatically shortens the development cycle. As Figure 5.1 suggests, a solution that tracks API framework evolution can track hardware evolution as well.

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 [53] or BitFusion's FlexDirect [3]. 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.

## 5.2 Accelerator Silos

Accelerator stacks compose layered components that include a user-mode library to support an API framework and a driver to manage the device. Vendors are incentived to use proprietary interfaces and protocols between layers to preserve forward compatibility, and to use kernel-bypass communication techniques to eliminate OS overheads. However, interposing opaque, frequently-changing interfaces

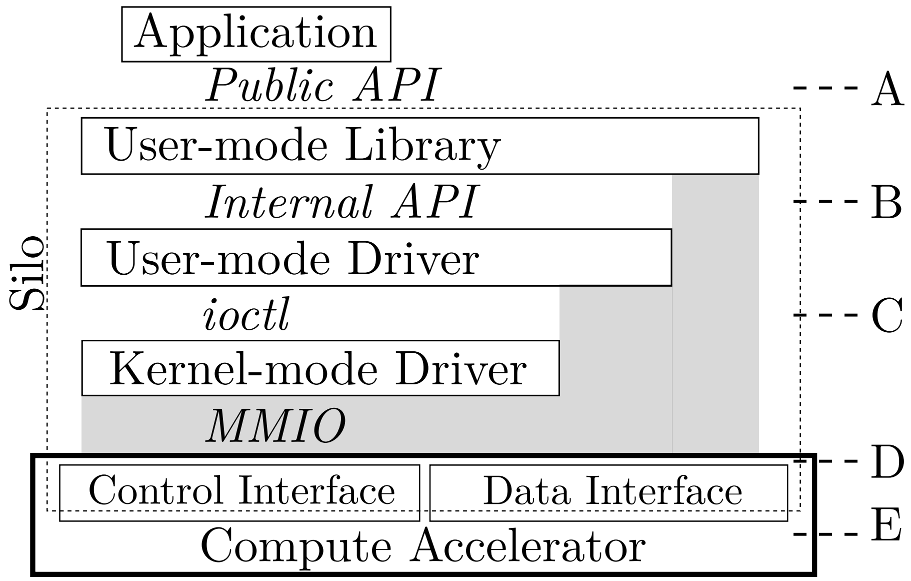


Figure 5.2: An accelerator silo. The public API and the interfaces with striped backgrounds are interposition candidates. All interfaces with backgrounds are proprietary and subject to change.

communicating with memory mapped command rings is *impractical* because it requires inefficient techniques and yields solutions that sacrifice compatibility. Consequently, accelerator stacks are effectively *silos* (Figure 5.2), whose intermediate layers *cannot be practically separated* to virtualize the device.

***Current support.*** 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 paper is that hardware support for process-level isolation *could* suffice to support hypervisor-level virtualization as well, but the siloed structure of current accelerator stacks prevents it.

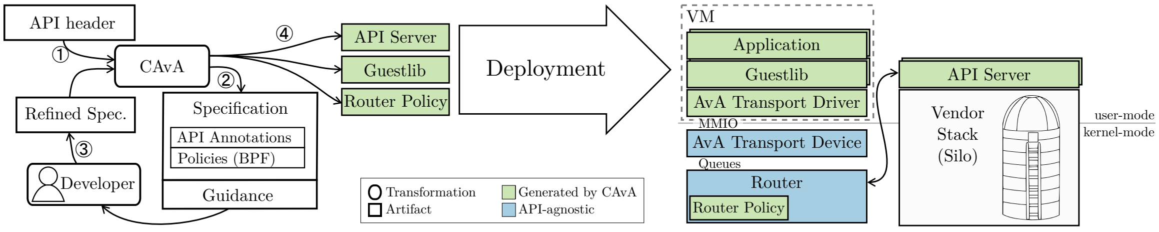


Figure 5.3: Overview of AvA.

For accelerator silos, the *only* stable and efficiently interposable interface is the framework API, so we focus on techniques to recover or compensate virtualization properties lost by API remoting:

interposition and compatibility. Interposition can be recovered by using hypervisor-managed forwarding transport, creating a central interface at which to enforce resource management policies. AvA uses a novel technique called *Hypervisor Interposed Remote Acceleration (HIRA)* to achieve this. HIRA presents guest VMs with an abstract virtual device with MMIO Base Address Registers (BAR), but this device is *not a virtual accelerator*, but an endpoint that routes communication through the hypervisor.

Using hypervisor-managed transport recovers interposition, but 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. AvA mitigates this with automated construction (§5.4). 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) [**? ?** ] are not applicable. AvA uses a new DSL called Lapis, a compiler called CAvA, and device-agnostic transport components to address these challenges.

### 5.3 Design

For accelerator silos, the *only* stable and efficiently interposable interface is the framework API, so we focus on techniques to recover or compensate virtualization properties lost by API remoting: interposition and compatibility. Interposition can be recovered by using hypervisor-managed forwarding transport, creating an interface at which to enforce resource management policies. AvA uses a novel technique called *Hypervisor Interposed Remote Acceleration (HIRA)* to achieve this. HIRA presents guest VMs with an abstract virtual device with MMIO Base Address Registers (BARs), but this device is *not a virtual accelerator*, but an endpoint that routes communication through the hypervisor.

Using hypervisor-managed transport recovers interposition, but 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. AvA mitigates this by automatically generating code to implement HIRA components (§5.4), which introduces the need to specify API semantics and policies beyond the ability of existing Interface Description Languages (IDLs) [**?** **?** ].

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, 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. Figure 6.1 shows the work-flow to support a new API with AvA (§5.3.2) and the AvA stack. The API-agnostic components may be deployed in the hypervisor or in an appliance VM to support type I and II hypervisors [**?** ].

### 5.3.1 AvA Components

Figure 6.1 provides an overview of the interaction between the various components in a AvA-generated design.

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. CAvA generates a separate transport driver instance for each API framework.

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 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 and 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.

### 5.3.2 Developer Work-flow

AvA's API-agnostic components (§5.3.1) 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. Figure 6.1 shows the work-flow to support a new API with AvA. First, CAvA automatically generates a preliminary LAPIS specification from the unmodified 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.

### 5.3.3 Communication Transport

AvA relies on an abstract communication channel that defines how calls, and their associated data buffers, and results are sent, validated, and received. The channel provides an interposition point to track resources and invoke policies from the LAPIS specification. 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.

AvA communication is bidirectional: it must support function callbacks from the API server to the guest library, because callbacks are fundamental in many frameworks, e.g., TensorFlow, and must be run in the guest VM. In AvA, 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).

### 5.3.4 Sharing and Protection

AvA re-purposes process-level isolation mechanisms (e.g. memory protection) provided by the accelerator silo when it is available to simplify supporting cross-guest isolation. We anticipate that emerging accelerators will support process-level isolation, as all GPUs do today. For accelerators that do not natively support process-level isolation, AvA can still share device, relying on semantic information from additional LAPIS descriptors to generate code that supports coarse-grain time-sharing, leveraging the same state capture techniques we use for VM migration (§5.4.5). This solution admits no concurrency between tenants, but enables protected sharing for devices which cannot otherwise be shared.

### 5.3.5 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 (§5.5.2.2) are scheduled. The developer provides policy callbacks in the LAPIS specification, and uses descriptors to identify how resources are consumed. When such a descriptor is unavailable, AvA falls back to coarse-grained estimation of resource utilization, for example, using wall-clock time to approximate device execution time.

### 5.3.6 Memory Management

To support memory sharing on devices with onboard memory, LAPIS resource usage descriptors 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 (such as, memory types and how to compute size of buffer) provided by descriptors in the API specification.

## 5.4 CAvA and LAPIS

The AvA toolchain comprises a language (LAPIS), a compiler (CAvA), and a runtime library. CAvA accepts code written in LAPIS (**L**ow-level **API S**pecification), 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 [**?** ]. 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, 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 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 are expressed at the language layer as well.

### 5.4.1 LAPIS

A developer writing a LAPIS specification must be concerned with capturing and expressing API semantics that fall roughly into four categories: argument and data marshalling, asynchrony, explicit and implicit state, and resource management policy. To illustrate how CAvA, LAPIS, and the runtime library, work together to address these concerns, we will use a running example that uses the CUDA driver API in an idiomatic scenario. The example allocates GPU memory (`cuMemAlloc`), makes asynchronous calls using CUDA streams to transfer data before (`cuMemcpyHtoDAsync`) and after (`cuMemcpyDtoHAsync`) launching a kernel (`cuLaunchKernel`), and synchronizes

```
continuous_rc device_memory;
instantaneous_rc kernel_invocations;
policy("policy_kern.c");
struct fn_arg_info {
  // CUfunction type information, filled by other LAPIS code
  int fn_argc;
  char fn_arg_is_handle[64];
  size_t fn_arg_size[64];
};
declare_metadata(struct fn_arg_info);
buf_registry async_DtoH;
type(CUdeviceptr) { handle; }
cuMemAlloc(CUdeviceptr *pp, size_t size) {
  allocates_rc(device_memory, size);
  argument(pp) { out; element { obj_record; allocate; } } }
cuMemcpyDtoHAsync(void* dst, CUdeviceptr src,
               size_t size, CUstream stream) {
  async; success(CUDA_SUCCESS);
  argument(dst) { no_copy; buffer(size); lifetime_manual; }
  register_buf(async_DtoH, stream, dst, size); }
cuLaunchKernel(CUfunction f,
    uint gridDimX, uint gridDimY, uint gridDimZ,
    uint blockDimX, uint blockDimY, uint blockDimZ,
    uint sharedMemBytes, CUstream hStream,
    void **kernelParams, void **extra) {
  async; consumes_rc(kernel_invocation, 1);
  argument(kernelParams) {
    in; buffer(metadata(f)->fn_argc);
    element {
      if (metadata(f)->fn_arg_is_handle[index]) {
        type_cast(CUdeviceptr*); buffer(1);
      } else {
        type_cast(void*); buffer(metadata(f)->fn_arg_size[index]);
      } } }
  argument(extra) { in; ... } }
cuStreamSynchronize(CUstream stream) {
  contextual_argument void** bufs = get_bufs(async_DtoH, stream);
  argument(bufs) {
    in; buffer(get_n_bufs(async_DtoH, stream));
    element {
      buffer(get_buf_size(async_DtoH, stream, index));
      out; deallocate; } } }
```

Figure 5.4: An example LAPIS description for the CUDA driver API. Code elements in **bold blue** are LAPIS keywords, elements in *italic green* are runtime library calls, elements in gray are function prototypes incorporated by CAvA from the original CUDA header file, and the remaining code is programmer-provided LAPIS. The variable **`index`** is a LAPIS builtin which provides the index of the current element of a buffer. The specification of the argument `extra` to `cuLaunchKernel` is complex due to alignment and padding, and is elided from this example for clarity. The file `policy_kern.c` is shown in Figure 5.7.

the stream (`cuSynchronizeStream`). The LAPIS source for the four of the five CUDA API functions required is shown in Figure 5.4. We elide source for `cuMemcpyHtoDAsync` because it is conceptually similar to `cuMemcpyDtoHAsync`.

LAPIS extends C types and function prototypes with information to serialize arguments and return values, and manage user-defined metadata on API-level objects such as GPU kernels or memory buffers. The latter can be used to specify the higher-level semantics and behaviors required for virtualization such as how to capture, transfer, and synchronize implicit state. Descriptors are embedded in LAPIS function declarations, can be flexibly scoped (Table 5.1), and applied to values (Table 5.2) or functions (Table 5.3). Descriptors can be conditional using an `if` statement. In the following discussion of LAPIS descriptors, we use the term "this function" to refer the function being described or the function whose argument is being described.

***Marshalling and Managing Values and Objects.*** LAPIS value descriptor usage is illustrated by the CUDA function `cuMemcpyDtoHAsync` defined in Figure 5.4 (Line 16). The argument `src` is only used by the application through API calls (AvA does not currently support UVM, see §5.4.7 for details). This is true of the type `CUdeviceptr` in general, so it is expressed on line 12 with **`type`**`(CUdeviceptr){ `**`handle`**`; }`. The **`handle`** descriptor enables AvA to generate code that implements an indirection layer for accessing it. The argument `dst` is a pointer to an application buffer of length `size`; this is expressed on line 19 with **`argument`**`(dst){ `**`no_copy`**`; `**`buffer`**`(size); }` which selects the argument `src` and specifies it as a **`buffer`** of `size` length. The buffer will be copied back to the application later so it is **`no_copy`** here. Because the function is asynchronous (see below), the buffer for `src` in the API server must exist at least until the copy has completed: the descriptor **`lifetime_manual`** on line 19 states that the specification will indicate when the buffer is no longer needed using **`deallocate`** (line 42).

***Asynchrony and Implicit State.*** LAPIS supports descriptors for expressing semantics that arise due to asynchrony. Most commonly, these are expressed by applying descriptors to functions. For example, `cuMemcpyDtoHAsync` in Figure 5.4 is asynchronous and can return `CUDA_SUCCESS` before the copy completes; this is expressed using **`async`** and **`success`** on line 18. Asynchrony can also create implicit API-level state whose semantics must be expressed. `cuStreamSynchronize` must copy buffers back to the guest if there are outstanding asynchronous copies. Those buffers may

| Scope | `descriptors` in this scope apply to ... |
|---|---|
| **`argument`**`(x) {`<br>`descriptors }` | the argument `x`, called the described argument. |
| **`element`** `{`<br>`descriptors }` | the referenced value, called the described value; e.g., if applied to an argument `x`, the descriptors inside apply to `*x`. |
| **`type`**`(ty) {`<br>`descriptors }` | all values of type `ty`. |
| `if (pred) {`<br>`descriptors1`<br>`} else {`<br>`descriptors2 }` | `descriptors1` apply only if `pred` is true; `descriptors2` apply otherwise. |

Table 5.1: LAPIS syntax which values descriptors apply to and when those descriptors apply.

| Type | The value is ... |
|---|---|
| `type_cast(ty)` | type `ty` and should be cast in the generated code. |
| `handle` | an API object handle. |
| `buffer(size)` | a buffer of `size` elements. |
| `in` | an input and should be copied from the application to the API server before the call. |
| `out` | an output and should be copied from the API server to the application after the call. |
| `no_copy` | a buffer which should not be copied. Useful when combined with lifetimes. |
| `lifetime_call` | a buffer which need only exist in the API server for the length of the call. (default) |
| `lifetime_manual` | a buffer which should exist in the API server until it is explicitly deallocated. |
| `allocate` | a buffer which calls to this function allocate. |
| `deallocate` | a buffer which calls to this function deallocate. |
| `zerocopy` | a buffer which should be in zero-copy memory. |
| API Object Recording | |
| `obj_record` | This call should be recorded and associated with the described value. The recorded calls capture the implicit state of the value. |
| `obj_depends_on(obj)` | Add a dependency between the described value and `obj`, enabling `obj` to be recreated if the described value is needed. |
| `obj_state_cbs(write, read)` | Attach state serialization and deserialization (`write` and `read`, resp.) functions to the described value. The serialized data captures the explicit state of the value and may be used in combination with implicit state. |

Table 5.2: LAPIS descriptors for specifying values and objects.

not be updated in the API server until the call completes, so copy-back must be delayed to this point. This requirement is expressed by passing dependent buffers using **`contextual_argument`** along with the normal explicit arguments. The LAPIS code on line 37–42 uses this technique along with runtime library functions to track buffers dependent on ongoing asynchronous copies. AvA supports zero-copy transport of buffers: if a buffer allocation is described with **`zerocopy`**, CAvA allocates zero-copy memory for the application-side buffer.

***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 consumed by an API function implementation only once, e.g., by executing a GPU kernel upon request. Accounting works by measuring resources used at each function invocation. In general, instantaneous resources are used to control throughput in some way; e.g., limiting the amount of compute resource a client is allowed to use in a fixed interval of time. Continuous resources capture the ability an API implementation to assign a resource to a client for a period of time. Accounting for continuous resources tracks resources assigned to each client/VM. For example, GPU memory is a continuous resource limited by available physical memory, which needs to be allocated according to a sharing policy that manages cross-VM contention for it.

Figure 5.4 uses descriptors to track kernel calls (`cuLaunchKernel`) and GPU memory allocation. In LAPIS, this is expressed using two resources and descriptors on functions to specify how much of each resource is consumed. Line 2 declares a resource representing function invocation rate. Line 3 specifies the custom policy used to schedule function invocations. Line 26 specifies that a call to `cuLaunchKernel` counts as one unit of the `kernel_calls` instantaneous resource. Line 14 specifies that a call to `cuMemAlloc` allocates `size` bytes of the continuous resource `gpu_mem`.

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 using **`policy`**`("policy_kern.c")` at the top level. In future work, we plan to extend LAPIS to express these policies directly. 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. However, in principle, the same properties can be achieved using LAPIS and will provide a significant complexity reduction for the developer.

| Function | The function ... |
|---|---|
| **sync** | is synchronous: application calls wait for completion in the API server. |
| **async** | can be asynchronous: calls will return immediately after call dispatch. |
| **success**(v) | should return v when called with **async**. |
| **contextual_argument** | has additional context that should be transported along with the normal arguments. This is used to copy values or buffers which are not explicitly arguments to the call, e.g., errno. |
| **allocates_rc**(r, x) | allocates x units of the continuous resource r. |
| **deallocates_rc**(r, x) | deallocates x units of the continuous resource r. |
| **consumes_rc**(r, x) | consumes x units of the instantaneous resource r during the call. |
| Declarations | The statement declares ... |
| **continuous_rc** r; | a continuous resource r. |
| **instantaneous_rc** r; | an instantaneous resource r. |
| **utility** *C declaration;* | a global utility function or variable that is not part of the API, but is used in the specification. |

Table 5.3: LAPIS descriptors for functions.

| | |
|---|---|
| Metadata | |
| `declare_metadata(ty)` | Specify the type of value to store as metadata to be `ty`. |
| `metadata(key)` | Get, creating if needed, the metadata object associated with `key`. |
| In-flight buffers | |
| `buf_registry r;` | Declare `r` to be a registry of buffers. |
| `register_buf(r, k, buf, size)` | registers `buf` which is `size` elements attached to key `k` (e.g., a CUDA stream) in registry `r`. |
| `get_bufs(r, k)` | gets an array of the buffers attached to key `k` in registry `r`. |
| `get_n_bufs(r, k)` | gets the number of buffers return from `get_bufs`. |
| `get_buf_size(r, k, i)` | gets the size of a specific registered buffer. |
| Zero-copy | |
| `zerocopy_alloc(n)` | Allocate `n` bytes of zero-copy memory. |
| `zerocopy_free(p)` | Free the zero-copy memory pointed to by `p`. |

Table 5.4: An except from the LAPIS standard library for use in expressions and utility functions.

***State Capture.*** To support VM migration, AvA must capture the state of API objects and recreate that state at the destination. This requires visibility into the relationships between API calls and device state: AvA cannot simply serialize, transfer, and deserialize device state to effect a migration because AvA's view of device state is through the high level API and through semantics specified by the programmer. AvA splits object state into two categories based on descriptors from the AvA programmer. *Explicit* state is serialized into AvA-managed storage by programmer-defined functions; *implicit* state is captured by recording calls which mutate an *object* (e.g. a buffer) based on **`obj_record`** descriptors. In Figure 5.4, **`obj_record`** descriptor is used in `cuMemAlloc` to expose implicit state, in this case a device-side buffer which must be allocated to recreate state at the destination.

***Runtime Library.*** The LAPIS standard library is illustrated (in excerpted form) in Table 5.4, and is designed to support common features we encountered across multiple APIs. State tracking often requires storing metadata about an API object, so the library provides a system to do that. Many APIs support asynchronous functions for which buffers tracking is required, so the library provides a set of function to track buffers. For APIs that require zero-copy data transfer for performance or because they expose hardware doorbells, the library provides memory management functions that expose AvA's support for zero-copy buffers. The library also provides common utilities such as *`min`*, *`max`*, and *`strlen`*.

***Workflow.*** In the AvA workflow, CAvA is initially invoked on the API header files to generate a provisional LAPIS specification for all functions in the API; the developer then refines that specification. For perspective, the actual provisional specification generated by CAvA for the functions in Figure 5.4 totals 107 lines, of which 29 lines are deleted, 9 lines are changed, and 21 lines are added to reach the final specification used in our evaluation. CAvA can infer the semantics of functions and arguments in simple cases and provides safe defaults when possible (e.g., by default, functions are synchronous and numeric types are opaque). For cases that do not admit a conservative guess, CAvA emits code that will cause an error, e.g., for "direction" (in/out parameters), which provides programmer guidance about what additional information is required and where it should be expressed.

```
 1 CUresult cuStreamSynchronize(CUstream stream) {
 2   // Allocate and initialize the command
 3   cu_stream_synchronize_call *cmd = allocate;
 4   cmd->command_id = CALL_CU_STREAM_SYNCHRONIZE;
 5   cmd->call_id = generate_call_id(); // Unique call ID
 6   cmd->stream = stream;          // Copy simple value
 7   // Compute and attach contextual parameter bufs
 8   void **bufs = get_bufs(async_DtoH, stream);
 9   size_t nbufs = get_n_bufs(async_DtoH, stream);
10   // Attached buffers referenced from bufs
11   void **tmp_bufs = allocate and fill with output buffer sentinal;
12   cmd->bufs = attach_buffer(cmd, tmp_bufs,
13     get_n_bufs(async_DtoH, stream) * sizeof(void *));
14   // Create and register a local record of the call
15   cu_stream_synchronize_record *record = alloc();
16   record->stream = stream; record->bufs = bufs;
17   record->call_complete = 0; register_call(record);
18   // Send the command and wait for and return response
19   send_command(cmd);
20   wait_until(record->call_complete);
21   return record->ret; }
22 CUresult cuMemcpyDtoHAsync(void *dst, CUdeviceptr src,
23                        size_t size, CUstream stream) {
24   ...Allocate and initialize the command as above...
25   // Execute state-tracking code from spec
26   register_buf(async_DtoH, stream, dst, size);
27   ...Store and attach arguments similar to above...
28   ...Create and register a local record of the call, and send...
29   // Return the success value from LAPIS specification
30   return CUDA_SUCCESS; }
```

Figure 5.5: An outline of the generated guestlib code for `cuStreamSynchronize` and `cuMemcpyDtoHAsync`. The real code manages a number of additional details, e.g., threads.

Writing a LAPIS specification requires user-level knowledge of the API. The developer must understand the API function semantics but does not need to know how to implement the API or understand details of AVA or API remoting. Our experience is that most APIs (e.g., OpenCL [**?** ]) provide documentation sufficient to achieve this. However one API (the CUDA runtime API) required LAPIS specifications for undocumented functions which required more developer effort to produce. The real target users of AVA are hypervisor developers, cloud providers, and accelerator vendor software engineers, for whom the required knowledge can be safely assumed, and for whom the reduction in development effort is compelling even for complex APIs.

### 5.4.2 Code Generation

CAVA generates several separate components for each API function, including a stub function, a call handler, a return handler, and a replay handler for VM migration.

The generated stubs, in Figure 5.5, construct a command from the arguments (including handling lists of asynchronous output buffers on lines 11–13) and then send it, via the hypervisor, to the

```
void api_server_handle_command(call) {
  switch (call->command_id) {
  case CALL_CU_STREAM_SYNCHRONIZE: {
    // For handles, get the real address
    CUstream stream = handle_deref(call->stream);
    // Get pointers to shadows of the attached buffers
    void **bufs = get_buffer(call, call->bufs);
    size_t nbufs = get_n_bufs(async_DtoH, stream);
    for (size_t i = 0; i < nbufs; i++)
      bufs[i] = get_shadow_buffer(call, bufs[i]);
    // Perform call
    CUresult stat = cuStreamSynchronize(stream);
    // Construct and send the return command
    cu_stream_synchronize_ret *ret_cmd = alloc();
    ret_cmd->command_id = RET_CU_STREAM_SYNCHRONIZE;
    ret_cmd->call_id = call->call_id; ret_cmd->ret = stat;
    // Attach sync buffers to return command
    void **tmp_bufs = alloc();
    for (size_t i = 0; i < nbufs; i++)
      tmp_bufs[i] = attach_buffer(ret_cmd, bufs[i],
          get_buf_size(async_DtoH, stream, i));
    ret_cmd->bufs = attach_buffer(ret_cmd, tmp_bufs,
       nbufs * sizeof(void *));
    send_command(ret_cmd);
    break; }
  case CALL_CU_MEMCPY_DTOH_ASYNC: {
    ...Extract dst, stream, and size from call...
    // Get or allocate the shadow buffer
    void *dst = get_shadow_buffer(call, call->dst);
    // Execute state-tracking code from spec
    register_buf(async_DtoH, stream, dst, size);
    // Perform call
    CUresult stat = cuMemcpyDtoHAsync(dst, src, size, stream);
    ...Construct and send the return command...
    break; } ... } }
```

Figure 5.6: An outline of the generated API server code for `cuStreamSynchronize` and `cuMemcpyDtoHAsync`.

API server. The hypervisor enforces resource sharing policy at the router based on the resource accounting and policy descriptors. The API server, in Figure 5.6, deserializes the arguments from the command and then performs the real call. The results of the call are passed back to the guest lib in the same was as the call is made. The API server also transfers (lines 18–23) the shadow buffers registered during calls to `cuMemcpyDtoHAsync` (line 31) back to the guest. The guest lib handles the response by completing record-and-replay tracking, looking up the API call record, copying transferred shadow buffers into application buffers, and marking the call complete (code elided for brevity).

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.

Currently, CAvA only supports C as an output language, but this is not a fundamental limitation of AvA. We expect implementations for C++ and Python to be straightforward.

### 5.4.3 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

```
map_def cmd_cnt, priority;
kernel_calls = ava_get_rc_id("kernel_calls");
tot_id = 0; // ID of total in cmd_cnt
int consume(__sk_buff *skb) {
  vm_id = load_ava_vm_id(skb)
  amount = load_ava_rc_amount(skb, kernel_calls);
  vm_cnt = map_lookup_elem(&cmd_cnt, &vm_id);
  tot_cnt = map_lookup_elem(&cmd_cnt, &tot_id);
  fetch_and_add(vm_cnt, amount);
  fetch_and_add(tot_cnt, amount); }
int schedule(__sk_buff *skb) {
  ...Load vm_id, vm_cnt, vm_pri, tot_cnt, and tot_pri...
  if((*vm_cnt) *(*tot_pri) <= (*vm_pri) *(*tot_cnt))
    return HIGH_PRIO;
  else return LOW_PRIO; }
```

Figure 5.7: An example eBPF policy program (simplified for clarity). This is referenced from Figure 5.4 as `policy_kern.c`.

(e.g. GPUfs [**?** ]), but will not function correctly with AvA. The limitation is not fundamental, and we plan to address it in future work.

### 5.4.4 Resource accounting and scheduling

CAvA supports resource accounting using LAPIS descriptors which specify the type and quantity of resources each API function consumes. 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.

CAvA generates code to compute resource usage information in the hypervisor from call arguments. This code makes the call arguments available, similarly to Figure 5.6 lines 27–29. Then executes the expression to compute the used resources (e.g., `size` bytes of memory for `cuMemAlloc`) and records the usage via an API-agnostic component of AvA.

In Figure 5.4, the specification of `cuLaunchKernel` includes resource usage descriptors and a reference to a custom scheduler. Figure 5.7 shows code for an eBPF based scheduler. The `consume` function is called when the API server reports resource utilization to the hypervisor, which in this case, is every time an API call is made. The `schedule` function is called whenever a command reaches the head of its VM's queue to compute the priority for the command. The router dispatches commands in priority order (highest to lowest). The AvA policy functions take an __sk_buff argument containing the AvA command information. This allows AvA to use the existing eBPF infrastructure directly.

The algorithm in Figure 5.7 is simplified for clarity. It counts the number of commands each VM sends (consume, lines 9–10) and prioritizes VMs which have have sent less than their share of the total commands (schedule, lines 13–15). A real policy would periodically reset or slowly reduce the counts and use additional information to properly handler varied command costs [**?** 101**?** ].

### 5.4.5 VM Migration

AvA supports VM migration using *record-and-replay* [**?** ], augmented with explicit serialization to reduce the number of calls that must be replayed. Recreating explicitly managed state is more efficient and predictable because the developer has full control over it, making tracking of mutations unnecessary. In many cases, the programmer-defined serialization functions are thin wrappers around API functions. For example, for CUDA device buffers, serialization functions wrap cuMemcpyDtoH and cuMemcpyHtoD. Implicit state requires less developer effort and can capture state that cannot be obtained through the API (e.g., the size of a buffer). CAvA automatically generates all the record and replay code required to migrate objects that have been annotated. CAvA also provides the obj_depends_on descriptor to specify dependencies between API objects (e.g., a memory object may depend on a configuration object used to set its attributes). AvA remaps resource handles that change due to replay.

When a migration is triggered, the router invalidates the transport channel to the guest so that no new API calls are transmitted. Once all in-flight API calls have been completed, the router quiesces the device, invoking API synchronization functions (e.g., clFinish), and begins transferring and recreating device state on the target device. Explicit state is recreated by invoking developer-provided code; implicit state is recreated by replaying recorded API invocations.

### 5.4.6 Memory Over-subscription

While memory over-subscription is uncommon for main memory in virtual environments, it remains important for accelerators because on-board memory capacity is typically small [**? ? ?** ]. To swap out a victim object, the API server extracts and stores its implicit and explicit state. To swap an object back in, the API server replays the recorded APIs to recreate implicit state, and calls the developer-provided function for explicit state. This swapping is at *buffer object* granularity, instead of at page granularity [**? ? ?** ].

### 5.4.7 Limitations

AvA relies on a translation layer in which guests access API-level objects in the API server through handles. This introduces some limitations on support for full unified virtual memory, device-side memory allocation, and application polling for device-side writes (discussed in §5.4.3), most of which manifest only in combination with VM migration. AvA is able to preserve pointer-is-a-pointer semantics for the memory translation layer by using an identity mapping between handle-space and the API server virtual address space. However, AvA's techniques for supporting VM migration cannot be guaranteed to recreate the API server virtual address space with exactly the same virtual mappings at the migration target, so when state is recreated, it will be isomorphic to the source state, but that identity mapping may not be preserved. Supporting GPU mapped memory by mapping API server memory into the guest address space has a similar limitation: mappings cannot be reliably preserved across the migration, so AvA cannot support migration for applications that use zero-copy. AvA could fix this if accelerator memory management APIs provided more control over the virtual address space. Device-side memory allocation is not visible to AvA through framework APIs, so migration for such workloads is not yet supported. AvA currently does not support demand-paging between accelerators and guest memory: this limitation is not fundamental, and we plan to implement support for it in the near future. AvA does not support migration in the presence of application-level non-determinism.

## 5.5 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 (§5.5.2). 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.

### 5.5.1 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.

### 5.5.2 Hypervisor Interposition and Mediation

AVA enables hypervisor mediation by interposing the transport channel. We extended QEMU *virtio-vsock* [**?** **?** ] (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.

#### 5.5.2.1 Policies in eBPF

AVA supports policies written as eBPF (Extended Berkeley Packet Filter [**?** ]) 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. This simplifies the implementation somewhat, but will be changed in the future so that resources tracking can be performed fully in the hypervisor.

#### 5.5.2.2 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 [**?** ] below ($a = 1$ ms and $b = 1/2$; see §5.6.5).

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

### 5.5.3 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

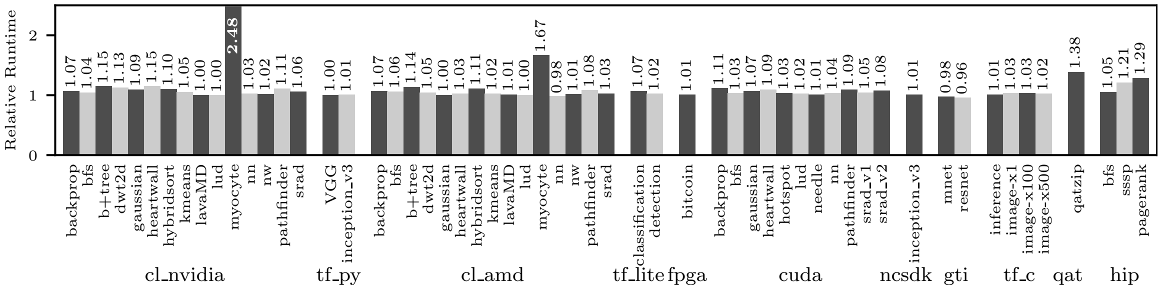


Figure 5.8: End-to-end execution time on virtualized APIs or accelerators normalized to native execution time. `tf_py` is the handwritten TensorFlow Python API remoting with AvA API-agnostic components.

function annotated with `deallocates`. Reverse shadows, guest library buffers, and threads which shadow an API server resource, are supported in the same way.

### 5.5.4 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.

## 5.6 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

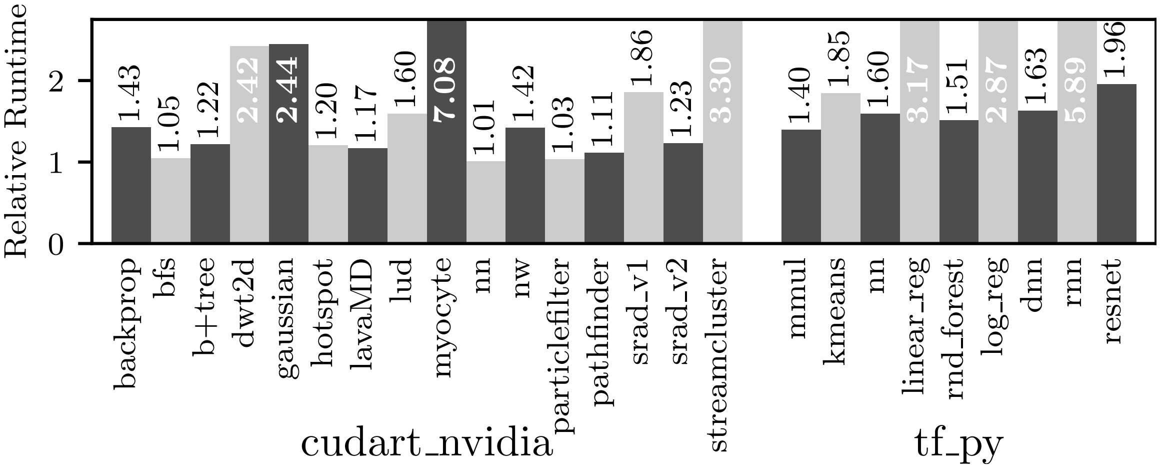


Figure 5.9: End-to-end execution time on virtualized CUDART and CUDA-accelerated TensorFlow APIs normalized to native execution time.

Cloud TPU v2-8 in the same data center. The guest VM was located on the same instance via nested 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.

### 5.6.1 Development Effort

We present our experience building API remoting systems by hand and with CAvA as evidence of the significant reduction in developer effort CAvA provides. We characterize developer effort to virtualize eleven APIs and nine devices in terms of the lines of code in the CAvA specifications or C/Python code (`LoC` in Table 5.5), and the number of lines of code modified (**Churn** in Table 5.5) during development (counted from commits). Building an API remoting system for OpenCL, which supports all the APIs needed to run the Rodinia [44] benchmarks, by hand took more than 3 developer-months, and spanned 7,514 LoC (see row 1 of Table 5.5). Supporting the same subset of OpenCL with AvA, 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).

| 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 | Gyrfalcon 2803 Plai Plug |
| Custom FPGA on AmorphOS [? ] | ✓ | 4 | 30 | 40 | BitCoin | AWS F1 |
| QuickAssist 1.7 | ✓ | 19 | 444 | 676 | QATzip | Intel QAT 8970 |
| HIP | ✓ | 41 | 624 | 990 | Galois [? ] | AMD Vega 64 |

Table 5.5: Development effort for forwarding different APIs, along with the benchmarks [? ? 44] 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.

### 5.6.2 End-to-end Performance

Figure 5.8–5.9 shows the end-to-end runtime, normalized to native, for all benchmarks, accelerator and API combinations we support (see Table 5.5). AvA introduces modest overhead for most workloads. Excluding `myocyte`, the Rodinia OpenCL benchmarks on NVIDIA GTX 1080 GPU slowed down by 7% on average. The outlier, `myocyte` has over 2× overhead because it is extremely **call-intensive**—it makes over 200,000 calls in 18.5 s; most others make between 30 and 3,000 calls. For comparison, FlexDirect sees 3.3× slowdown for `myocyte`; GPUvm sees 100× or more for call-intensive workloads [? ]. `Myocyte` experienced lower overhead on the AMD Radeon RX 580 GPU, as the kernels executed 3× 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× 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 Gyrfalcon benchmarks enjoy a slight speedup as

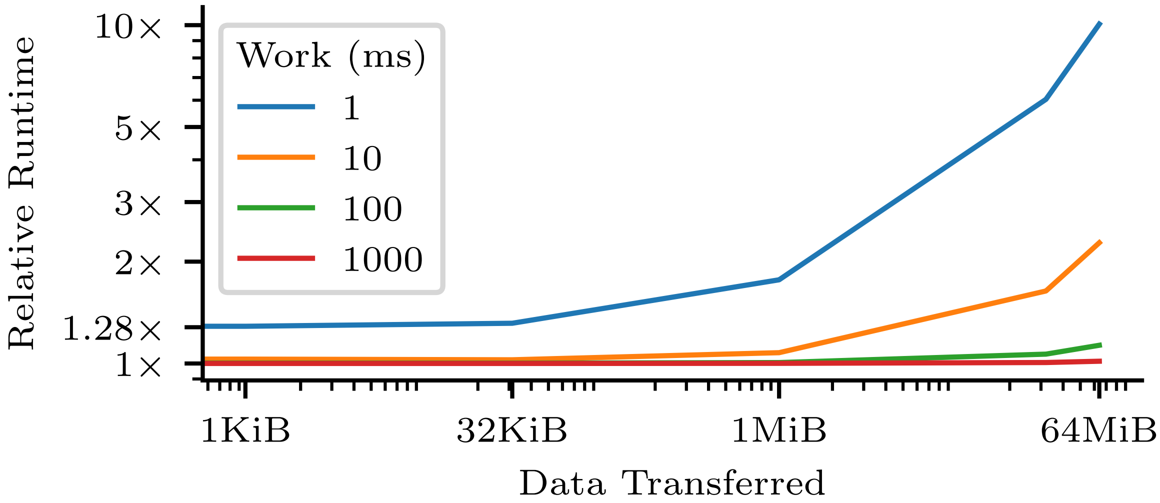


Figure 5.10: Overhead on a micro-benchmark with varying work per call and data per call. The plot is log-log and the trend is linear.

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 [**?** ] using synchronous QAT APIs: while the application only experienced a 1.38× end-to-end runtime slowdown, its throughput was 2.2× 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. We are exploring zero-copy techniques to ameliorate this. We note that QAT on AvA is fair, unlike the onboard SR-IOV support.

### 5.6.3 Micro-benchmarks

To understand performance trade-offs for AvA, we ran a micro-benchmark that transferred different amounts of data per call and simulated accelerator computation for different lengths of time by spinning on the host. Figure 5.10 shows 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 transfer little data and have short kernels, so control transfer dominates execution, (e.g. 28% overhead on 1 ms calls with no data).

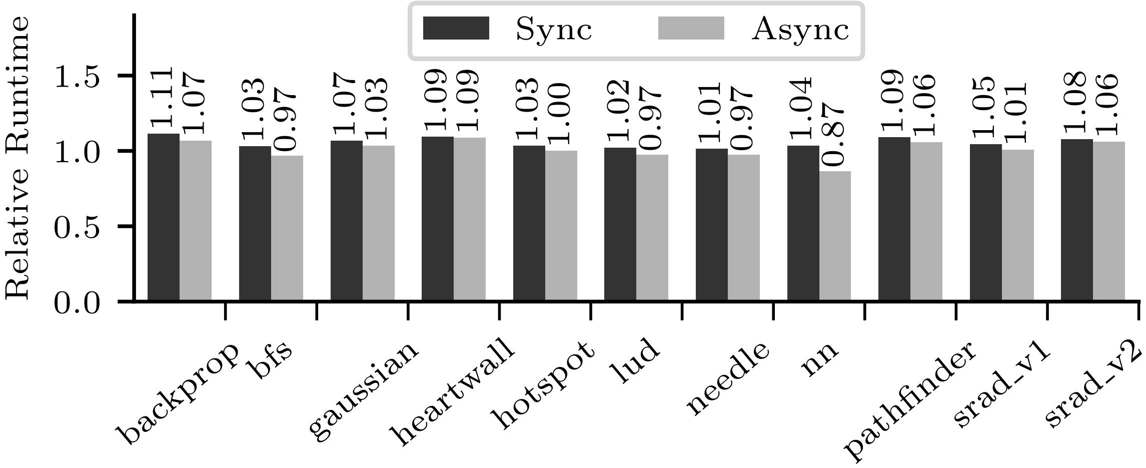


Figure 5.11: End-to-end runtime of CUDA benchmarks (relative to native) using synchronous and asynchronous specifications.

#### 5.6.3.1 Asynchrony Optimizations

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) [107] and rCUDA (API batching) [55].

We annotated several synchronous APIs—`cuLaunchKernel`, `cuMemcpyHtoD`, and resource free functions—as asynchronous. Figure 5.11 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.

### 5.6.4 Scalability

To evaluate scalability, we ran multiple instances of the OpenCL `gaussian` benchmark simultaneously in a varying number of VMs with a NVIDIA GTX 1080 GPU. The `gaussian` benchmark fully saturates the GPU. The AvA overhead (∼10%) does not increase as the number of VMs or applications increases, as shown by the near-perfect scaling in Figure 5.12. The GPU kernel execution

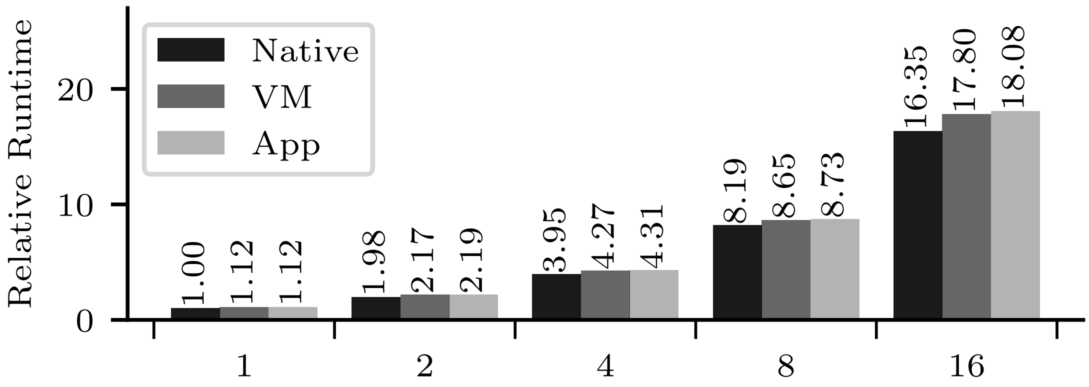


Figure 5.12: Scalability of multiple VMs running a single application each, and multiple applications in a single VM, with AvA. Runtime is relative to running the same number of applications natively.

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.

Accelerators without process-level protection or sharing support (e.g., Intel Movidius NCS) do not scale will with AvA, 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.

### 5.6.5 API Rate Limiting

To measure the fairness achieved by AvA, we repeatedly executed kernels drawn from six CUDA OpenCL benchmarks in pairs simultaneously in two VMs. The kernels last from 1 ms to 100 ms. Figure 5.13 shows the fairness of the execution with fixed-rate polling and feedback control method in 500-ms and 1-s measurement windows. We compute the 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.

### 5.6.6 Live Migration

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,

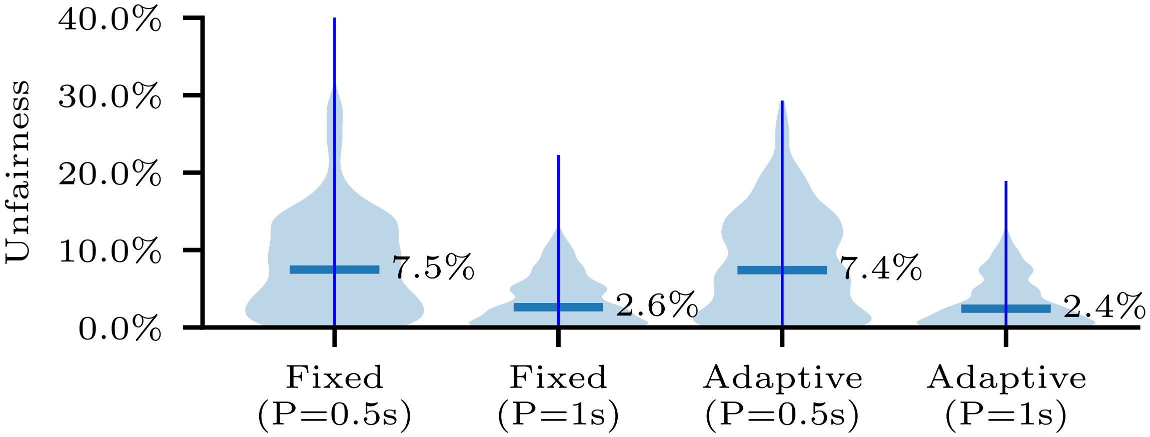


Figure 5.13: Unfairness of the fixed and adaptive scheduling algorithms with two different measurement periods. The width of the shaded areas show the probability of the bias (unfairness) being a specific value in any given measurement window. The horizontal bar shows the median and the vertical line runs from the minimum to the maximum.

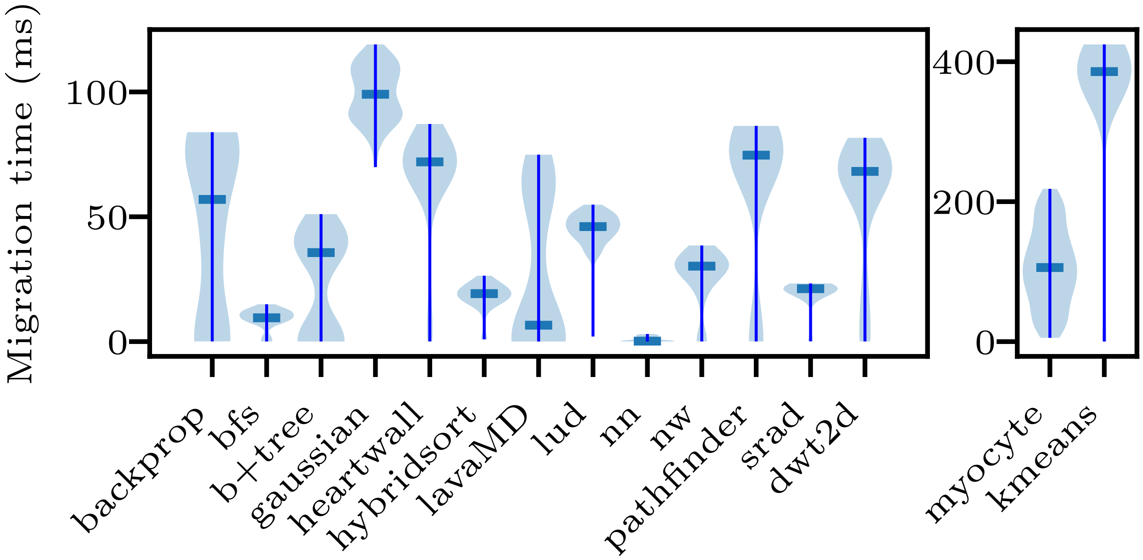


Figure 5.14: Live migration downtime for single-threaded OpenCL benchmarks on NVIDIA GTX 1080. This downtime is in addition to the ~75 ms of downtime of the VM migration itself. Migration downtime does not include time spent waiting for executing kernels to complete (accounted as latency), as the application is still performing useful computation on the accelerator during that time.

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 5.14 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 5.14 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.

### 5.7 Related Work

Chapter 7 provides detailed related work; this section addresses related work not covered there. Our previous work [**?** ] proposed many key ideas in AvA; this paper fleshes out and evaluates those ideas.

***FPGA virtualization*** has a long history [**? ? ? ? ? ? ? ? ?** ]. Most prior work relies on hardware-specific features, focuses on sharing in a single protection domain [**?** ], or virtualization primitives [**?** ]. AvA can be combined with any of these techniques to virtualize FPGA accelerators.

***Nooks*** [115] 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*** [**? ? ? ?** ] provide an interface description language (IDL) and tools to easily implement those interfaces. Unlike the CAvA language, 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*** [**?** ] 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 [**?** ] make use of C headers, but require manual annotations in many common cases. Unlike AvA, language specific DSLs [**? ?** ], do not support marshalling data structures and encapsulate rather than export the C API. Cross-language serialization standards and frameworks [**? ? ?** ], 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.

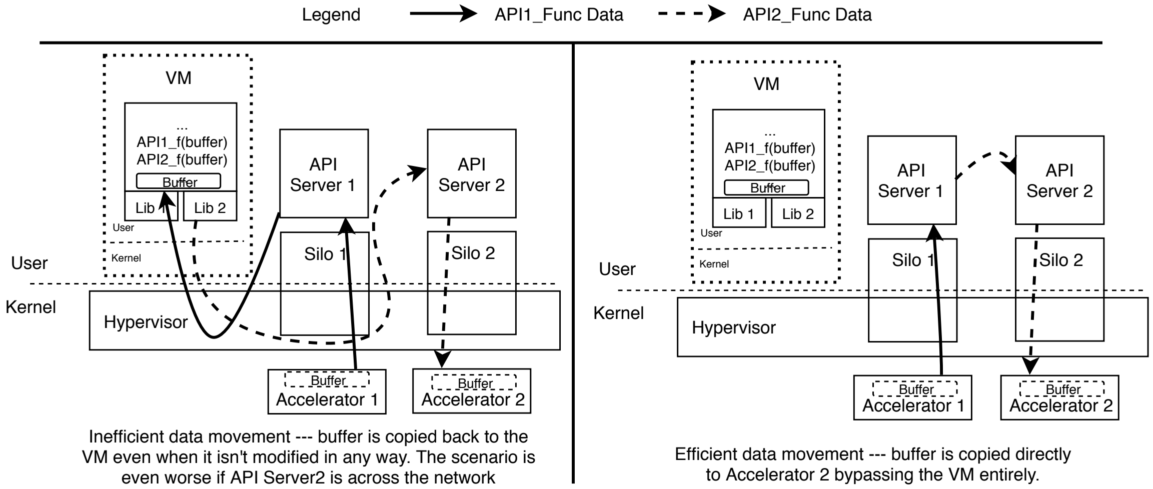
## 5.8 Conclusion

Virtualization techniques that rely on clean separation of software layers are untenable for accelerator *silos*. AvA is an alternative approach that interposes compute-offload APIs, uses automation to provide agility, recover hypervisor interposition, and shorten development cycles.

# CHAPTER 6: PROPOSED WORK — VTASK

The previous chapters in the proposed dissertation showed that hypervisor-mediation API-remoting is an effective mechanism for sharing API-controlled compute devices among mutually distrustful tenants, e.g., in a cloud computing environment. Virtualization vendors, such as VMware, have begun adopting API-remoting based solutions for accelerator virtualization [20].

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



Inefficient data movement --- buffer is copied back to the VM even when it isn't modified in any way. The scenario is even worse if API Server2 is across the network

Efficient data movement --- buffer is copied directly to Accelerator 2 bypassing the VM entirely.

Figure 6.1: Data processed by two API stacks must pass through the guest application

Under a typical API-remoting system, applications that pipeline disparate accelerator frameworks are burdened with redundant data movement. All inter-accelerator data movement must take place in the guest application as that is where the accelerators are in the same logical address space. Figure 6.1 illustrates this scenario: when an *API-1* function is invoked, associated data is copied from the *guest* *application* to *API-server-1*, and then to *Device-1*'s memory. Once the function finishes executing on *Device-1*, the result is copied back to the *guest application*. When a function from *API-2* is invoked, the same data (i.e., the output of the *API-1 function*) is copied from the *guest application* to *API-server-2* and then to *Device-2* to be processed.

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

We propose to build vTask, an application-transparent data orchestration system that optimizes data movement among accelerators virtualized via API-remoting. vTask will leverage information from API annotations [133] to track data buffers across the guest application, the API-servers servicing API calls made by the guest, and the accelerator hardware. vTask will optimize data movement across these components while ensuring that a coherent view of the data buffer is presented to anyone attempting to read the data. Ideally, vTask will require no changes to the guest application or extra annotations of any kind from the application programmer. We hypothesize that annotations provided to virtualize the API (by the device or virtualization vendor) will be sufficient to infer the semantics of the data buffers managed.

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

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 virtulaization is well explored [28, 48, 96, 41, 43, 42].

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 [126, 34, 80, 108, 137, 27], 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 prior accelerator virtualization techniques in terms of traditional virtualization properties. 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

| Technique | System | lib unmod | OS unmod | lib-compat | hw-compat | sharing | isolation | migration | sched. policy | graphics | GPGPU | I/D | benchmark | slowdown | native speedup | virtual speedup |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Full-virtual | **GPUvm** [114] | ✓ | | ✓ | | ✓ | ✓ | | XC, BAND | | ✓ | *D* | Rodinia | 141× | 11.4× | 0.08× |
| | **gVirt** [118] | ✓ | | ✓ | | ✓ | ✓ | ✓ | QoS | ✓ | | *I* | 2D [16], 3D [4] | 1.6× | N/A | N/A |
| PCIe Pass-thru | **AWS GPU** [32] | ✓ | ✓ | | | | | | | ✓ | ✓ | *D* | Any | 1× | | |
| API remoting | **GViM** [63] | | | | ✓ | ✓ | ✓ | | RR, XC | | ✓ | *D* | CUDA 1.1 SDK | 1.16× | 22× | 19× |
| | **gVirtuS** [58] | | | | ✓ | ✓ | ✓ | | RR | | ✓ | *D* | CUDA 2.3 MM | 3.1× | 11.1× | 3.6× |
| | **vCUDA** [106] | | ✓ | | ✓ | | | ✓ | HW | | ✓ | *D* | CUDA 4.0 SDK | 1.91× | 6× | 3.1× |
| | **vmCUDA** [124] | | ✓ | | ✓ | ✓ | | | HW | | ✓ | *D* | CUDA 5.0 SDK | 1.04× | 33× | 31.7× |
| Distributed API remoting | **rCUDA** [55, 99] | | ✓ | | ✓ | ✓ | ✓ | | RR | | ✓ | *D* | CUDA 3.1 SDK | 1.83× | 49.8× | 27.2× |
| | **GridCuda** [86] | | ✓ | | ✓ | ✓ | ✓ | | FIFO | | ✓ | *D* | CUDA MM, SOR | 1.23× | | |
| | **SnuCL** [77] | | ✓ | | | ✓ | ✓ | | | | ✓ | *D* | SNU NPB [105] | | | |
| | **VCL** [36] | | ✓ | | ✓ | ✓ | ✓ | | | | ✓ | *D* | Stencil2D [49] | | | |
| Para-virtual | **GPUvm** [114] | | | | | ✓ | ✓ | | XC, BAND | | ✓ | *D* | Rodinia | 5.9× | 11.4× | 1.9× |
| | **HSA-KVM** [68] | ✓ | | | | ✓ | ✓ | | HW | | ✓ | *I* | AMD OCL SDK | 1.1× | | |
| | **LoGV** [61] | ✓ | | ✓ | | ✓ | ✓ | ✓ | RR | | ✓ | *D* | Rodinia | 1.01× | 11.4× | 11.3× |
| | **SVGA2** [53] | ✓ | | | | ✓ | ✓ | ✓ | | ✓ | | *D* | 2D, gaming | 3.9× | | |
| | **Paradice** [34] | ✓ | | ✓ | | ✓ | ✓ | | HW, QoS | ✓ | ✓ | *D* | OpenGL, OpenCL | 1.1× | | |
| | **VGVM** [120] | | | | ✓ | ✓ | ✓ | | 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 [31].

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 Full Virtualization

***GPUvm*** [113] virtualizes CUDA on Kepler and Fermi (NVIDIA) GPUs for Xen. 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 authors also explore a number of optimizations: lazy shadowing, bar remap, para-virtualization, and multi-call batching. Despite these optimizations, GPUvm remains non-viable due to its high overhead—the most optimal configuration of GPUvm induces a 6 × slowdown on average.

***gVirt*** [119] is a *graphics*-only virtualization technique for Intel GPUs. The GPU is multiplexed among multiple VMs via pass-through for access to performance critical resources (command buffer and frame buffer), and trap-emulate for resources generally accessed off the critical path (PTEs, I/O Registers). Initialization and power management are done by the native driver in DOM0. Memory is multiplexed with a combination of partitioning and " ballooning". gVirtus is geared toward graphics and can't be easily extened to support GPGPU computing.

### 7.1.2 API Remoting

***GViM*** [63] 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. 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*** [107] 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*** [124] observes that while pass-through is performant, it precludes sharing, and VM migration. vmCUDA employs a split-driver model with a front-end driver in the guest that provides an interposition point, and a back-end driver in the control domain which interacts with the CUDA runtime and driver. CUDA applications in the guest are linked against an interposer library which forward calls and data to the appliance VM. As with vCUDA, the system guarantees no isolation among VMs.

***gVirtuS*** [58] 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*** [55, 99], ***GridCuda*** [86], ***SnuCL*** [77] and ***VCL*** [36] are all user-mode middle-ware systems for multiplexing GPUs and CUDA/OpenCL across a cluster. A client library encapsulates access to a (potentially) remote GPU. While the basic design is isomorphic to the API remoting design, virtual machines need not be present.

### 7.1.3 Para-virtualization

***LoGV*** [61] 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.

***HSA-KVM*** [68] 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*** [53] 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*** [102] 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 [101], the underlying accelerator abstraction layer.

***HPVM*** [110] 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 runtimes 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 (Graal VM).

# CHAPTER 8: CONCLUSION

# BIBLIOGRAPHY

[1] Amazon EC2 instance types. https://aws.amazon.com/ec2/instance-types/. Accessed: 2017-04.

[2] AMD Multi-user GPU. http://www.amd.com/Documents/Multiuser-GPU-White-Paper.pdf. Accessed: 2018-07.

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

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

[5] Compute shader. https://www.khronos.org/opengl/wiki/Compute_Shader. Jan. 2017.

[6] Deploying rCUDA in cloud computing environments. http://www.rcuda.net/pub/white_paper_cloud_v2.pdf. Published: 2016-12.

[7] Francisco Jerez's TGSI back-end. https://github.com/curro/llvm. Jan. 2018.

[8] Freedesktop nouveau open-source driver. http://nouveau.freedesktop.org. Accessed: 2017-04.

[9] Google Cloud TPU. https://cloud.google.com/tpu. Accessed: 2019-04.

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

[11] Hans de Goede's TGSI back-end. https://cgit.freedesktop.org/~jwrdegoede/llvm. Jan. 2018.

[12] Intel QuickAssist Technology. https://01.org/intel-quickassist-technology. Accessed: 2019-04.

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

[14] Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. Accessed: 2019-08.

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

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

[17] Project Fiddle: Fast and Efficient Infrastructure for Distributed Deep Learning. https://www.microsoft.com/en-us/research/project/fiddle. Accessed: 2019-04.

[18] Underutilizing Cloud Computing Resources. https://www.gigenet.com/blog/underutilizing-cloud-computing-resources/. Published: 2017-11.

[19] VMware SVGA3D Guest Driver. https://www.mesa3d.org/vmware-guest.html. Accessed: 2019-08.

[20] Vmware to acquire bitfusion. https://blogs.vmware.com/vsphere/2019/07/vmware-to-acquire-bitfusion.html. Accessed: 2019-10.

[21] VMware Workstation Version History. https://en.wikipedia.org/wiki/VMware_Workstation#Version_history. Accessed: 2019-08.

[22] Why frame rate and resolution matter. https://www.polygon.com/2014/6/5/5761780/frame-rate-resolution-graphics-primer-ps4-xbox-one. 2011.

[23] Gallium3d technical overview, 2017.

[24] The mesa 3d graphics library, 2017.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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 *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[38] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[39] 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.

[40] David Blythe. The direct3d 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.

[46] 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.

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

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

[49] 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.

[50] 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.

[51] 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.

[52] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: Architecture, design and implementation. In *Workshop on I/O Virtualization*, 2008.

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

[54] 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.

[55] 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.

[56] H Esmaeilzadeh, 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.

[57] Patrick Paul "Pat" Gelsinger. Private Communication, 1998.

[58] 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.

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

[60] 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.

[61] 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.

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

[63] 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.

[64] 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.

[65] 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.

[66] Alex Herrera. Nvidia grid: Graphics accelerated vdi with the visual performance of a workstation. *Nvidia Corp*, 2014.

[67] Chun-Hsian Huang and Pao-Ann Hsiung. Hardware resource virtualization for dynamically partially reconfigurable systems. *IEEE Embedded Systems Letters*, 1(1):19–23, 2009.

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

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] Khronos Group. *Vulkan 1.0.64 - A Specification*, 2017.

[76] 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.

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

[78] 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.

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

[80] Yossi Kuperman, Eyal Moscovici, and Joel Nider. Paravirtual Remote I/O.

[81] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. Paravirtual remote i/o. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 49–65. ACM, 2016.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] Microsoft. Microsoft azure goes back to rack servers with project olympus, 2017.

[89] Microsoft Inc. *Windows GDI*, 2017.

[90] Sun Microsystems. Rfc1050: Rpc: Remote procedure call protocol specification, 1988.

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

[92] 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.

[93] 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.

[94] 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.

[95] 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.

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

[97] 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.

[98] 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.

[99] 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.

[100] 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.

[101] 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.

[102] 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.

[103] Russel Sandberg. The sun network file system: Design, implementation and experience. In *in Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, 1986.

[104] Mark Segal and Kurt Akeley. The opengl graphics system: A specification. Technical report, Silicon Graphics Inc., December 2006.

[105] 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.

[106] 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.

[107] 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.

[108] Pci Sig. Single Root I/O Virtualization and Sharing Specification Revision 1.1. Technical report, January 2010.

[109] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: Integrating a file system with gpus. 2013.

[110] Prakalp Srivastava, Maria Kotsifakou, and Vikram S. Adve. HPVM: A portable virtual instruction set for heterogeneous parallel systems. *CoRR*, abs/1611.00860, 2016.

[111] 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.

[112] 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.

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

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

[115] 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.

[116] 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.

[117] 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.

[118] 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.

[119] 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.

[120] 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.

[121] 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.

[122] VMware, X.org, Nouveau. *Tungsten Graphics Shader Infrastructure*, 2012.

[123] 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.

[124] 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.

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

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

[127] 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.

[128] 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.

[129] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian 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.

[130] 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.

[131] 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.

[132] 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.

[133] 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.

[134] 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.

[135] Hangchen Yu and Christopher J Rossbach. Full virtualization for gpus reconsidered. In *14th Workshop on Duplicating, Deconstructing, and Debunking (WDDD), ISCA*, 2017.

[136] 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.

[137] 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.

[138] 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.