Automatic Virtualization of Accelerators

Hangchen Yu
The University of Texas at Austin
hyu@cs.utexas.edu

Amogh Akshintala
The University of North Carolina at Chapel Hill
aakshintala@cs.unc.edu

Arthur M. Peters
The University of Texas at Austin
amp@cs.utexas.edu

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

Abstract
Applications are migrating en masse to the cloud, while accelerators such as GPUs, TPUs, and FPGAs proliferate in the wake of Moore’s Law. These technological trends are incompatible. Cloud applications run on virtual platforms, but traditional I/O virtualization techniques have not provided production-ready solutions for accelerators. As a result, cloud providers expose accelerators by using pass-through techniques which dedicate physical devices to individual guests. The multi-tenancy that drives their business is lost as a consequence.

This paper proposes automatic generation of virtual accelerator stacks to address the fundamental tradeoffs between virtualization properties and techniques for accelerators. AVA (Automatic Virtualization of Accelerators) re-purposes a para-virtual I/O stack design based on API remoting to present virtual accelerator APIs to guest VMs. Conventional wisdom is that API remoting sacrifices interposition and compatibility. AVA forwards invocations over hypervisor-managed transport to recover interposition. AVA compensates for lost compatibility by automatically generating guest libraries, drivers, hypervisor-level schedulers, and API servers. AVA supports pluggable transport layers, allowing VMs to use disaggregated accelerators. With AVA, a single developer could virtualize a core subset of OpenCL at near-native performance in just a few days.

ACM Reference Format:

Figure 1. A typical application, accelerator silo, and hardware. The dashed box indicates the tightly coupled silo. The public API and interfaces with stripped backgrounds are virtualizable. All interfaces with backgrounds are unstable and cannot be relied on in a new version of the silo.

1 Introduction
Many emerging workloads such as machine learning [7] and genomics [8] rely on cloud computing to achieve economies of scale and lower operating costs. At the same time, the end of Dennard scaling has led to the use of increasingly specialized accelerators such as GPUs, TPUs [20], IPUs [12, 16], and FPGAs [25] to improve the performance of such workloads. Cloud providers have responded by offering VM instances with GPUs [2, 9, 10, 17], TPUs [11], and FPGAs [1, 18], but the lack of practical accelerator virtualization forces them to use pass-through techniques that dedicate physical hardware to VMs. As a result, they sacrifice the consolidation benefits of virtualization that are fundamental to their business.

Virtualizing accelerators is difficult in part because of the associated software frameworks. Accelerator hardware is typically controlled by proprietary programming frameworks that consist of complexly intertwined layers (shown in Figure 1) that lack standardization and stability, and are opaque to system software. Rapid time-to-market and high performance provide very strong incentives to vendors to fuse layers of their software stack with proprietary protocols and kernel bypass techniques, effectively forming what we call silos. Silos are composed of tightly coupled vertical layers that communicate either through proprietary interfaces, and/or using low-level mechanisms such as memory-mapped command queues and MMIO, both of which are very
difficult to efficiently interpose and virtualize. Silos provide only one public interface: the user-mode API. Virtualization based on interposing at layers in the middle of the stack is at best device-specific, and at worst altogether impossible. Silos make traditional virtualization techniques impractical (§2).

While silos are a major obstacle for accelerator virtualization, we hypothesize that silos will remain an enduring feature in future platforms. Vendors are incentivized to build them by market competition and cross-generation compatibility concerns. This suggests a need for tools and techniques that anticipate and integrate accelerator silos into virtualization layers.

We make a case that API remoting is the only virtualization technique that is practical in the presence of silos, because it interposes the only standardized and stable interface (e.g. CUDA, DirectX, or TensorFlow). API remoting [23, 27, 29, 34, 36, 41, 43] forwards user-level API calls to an API server running on the host or in an appliance VM. Compatibility is lost because API remoting typically involves modifying guest libraries. Further, these modified libraries must be manually updated for every new API version for each supported guest OS. API calls typically bypass virtualization layers [27] which renders interposition and hypervisor-enforced isolation impossible. Therefore, we focus our attention on techniques for compensating these lost properties through automation and interposable API remoting transport.

AvA automatically virtualizes APIs rather than para-virtualizing specific devices. It forwards APIs using para-virtual communication infrastructure gaining back hypervisor interposition and resource management. AvA compensates for lost compatibility by automating the construction of accelerator stacks. AvA takes an annotated API header as input and generates a complete virtual API stack. AvA provides near-native performance and minimizes developer effort. A single developer can support a new hypervisor managed API in a matter of days.

### 2 Motivation

This section summarizes limitations of existing virtualization techniques when applied to accelerator silos. We conclude that all conventional techniques have significant drawbacks, which have collectively prevented the emergence of production accelerator virtualization software.

**Full virtualization** [44, 46, 51] virtualizes the hardware interface. For accelerators, this requires trap-based interposition of communication through MMIO and memory BARs. Trapping on every guest access to MMIO and memory BARs results in devastating orders-of-magnitude performance losses [57].

**Para-virtualization** exports a virtual device abstraction to guest software [26], which provides an efficiently interposable interface by construction. However, virtual accelerators require custom drivers and framework libraries in the guest. A further compatibility concern arises from the need to support a single abstraction that encapsulates hardware diversity across vendors and models. For example, GPUs support multiple programming frameworks (e.g. OpenCL, OpenGL, DirectX) common in multiple OSes, and supporting all possible combinations with a virtual device abstraction is a staggering engineering challenge.

**Pass-through and Mediated pass-through (MPT)** techniques expose a physical device directly to guest software, effectively “passing-through” guest communication over the PCIe bus. The technique provides native performance and allows guests to use native drivers and libraries, but bypassing virtualization sacrifices its benefits. It is widely used in production settings currently because there is no alternative for supporting GPGPU compute, which remains unsupported by production hypervisors [26]. MPT [51] changes the balance of costs by interposing only sensitive interfaces and using pass-through for others, yielding a hybrid of pass-through and full virtualization.

**SR-IOV** [35] enables the hardware to directly expose multiple virtual devices (VFs) to system software from a single physical PCIe-attached device. SR-IOV provides an interface and protocol for managing VFs, but does not specify or implement the cross-VF sharing support, which is left to the hardware. Consequently, SR-IOV is an enabling technology for pass-through techniques with strong virtualization properties, but such a vision entails significant support from the hardware to re-implement resource management traditionally implemented by the hypervisor. At present, evidence is scant in the marketplace that such support will become the norm for accelerators. Very few GPUs support SR-IOV [3, 30], and those that do use static resource partitioning to manage sharing, FPGAs commonly support SR-IOV though vendor-provided PCIe IP blocks, and several research systems leverage it virtualize FPGA-accelerators [40, 53, 58]. However, implementing resource management across virtual functions exposed by SR-IOV remains a task for the FPGA programmer. We are aware of no production TPUs that support SR-IOV.

**API remoting** interposes at top-level framework APIs making it the only technique that interposes at a standardized layer for accelerator silos. However, API remoting compromises key virtualization properties as well:

- **Compatibility** is a major challenge due to the rapid pace of accelerator evolution. Frequent updates to accelerator APIs and runtimes requires changes to or reimplementation of virtual devices, guest drivers, and libraries. For example, VMware’s SVGA [26] virtual GPU device supports graphics...
frameworks using an API-remoting subsystem that translates all guest operations (e.g. OpenGL calls) into DirectX commands, but the monumental engineering effort required to maintain guest drivers and framework libraries has caused SVGA to lag behind the latest DirectX 12 by multiple versions. Another API-forwarding framework, GvirtuS [28], took 25,000 LoC and many person-years to build.

- **Performance** for virtualized accelerators is often determined by both the frequency and mode of communication between the guest application and the host API server. Systems such as vCUDA [43] and gvirtuS [28] show 10–40% performance degradation on average. Related systems such as rCUDA [27] and vmCUDA [54] optimize communication and data movement, delivering near-native performance, but these optimizations are specific to CUDA. Evidence that API remoting’s performance provides the most compelling compromise in the design space is emerging as startups and major vendors have started to build user-space production solutions (e.g. BitFusion [4] and Dell XaaS [31]).

- **Interposition** is fundamental to virtualization: hypervisors require interposition to provide indirection between logical and physical resources. Most API remoting solutions bypass the hypervisor by forwarding API calls over simple RPC [27], which gives up interposition and hypervisor-provided benefits as a result. The vCUDA [43] API server spawns threads for all guest applications in a single process, preventing even the most basic resource and fault isolation among applications and VMs.

However, lost interposition and lost isolation are not fundamental to API remoting. VMware’s SVGA device forwards DirectX APIs using hypervisor-managed FIFO queues, which provides an interposition point the hypervisor can leverage to isolate guests and perform resource management.

We observe that when applied to accelerators, all techniques compromise virtualization properties. However, API remoting has a key property not shared with the others: it interposes a stable API that does not require separation of accelerator silo layers. Additionally, its poor interposition can be recovered by ensuring API remoting transport is interposable, as suggested by the SVGA design [26]. Finally, its compatibility losses manifest primarily as increased engineering burden, which can be reduced by automation.

### 3 Vision

We anticipate and accommodate the proliferation of accelerator silos with tools for automatically constructing virtualization layers based on API remoting. Our system, AvA (Automatic Virtualization of Accelerators), automatically virtualizes user-mode accelerator APIs rather than accelerator hardware; embracing silos, instead of attempting to break them.

To reduce the cost of construction and maintenance, AvA automatically generates a custom API remoting stack from an API specification. To manage runtime overheads, AvA interposes the relatively coarse-grained public API, where most calls are infrequent and perform a significant amount of work. AvA uses hypervisor-mediated transport allowing the hypervisor to monitor and control all device accesses and collaborate with the CPU scheduler to improve scheduling decisions. AvA incentivizes vendor use of hypervisor-interposable infrastructure by offering push-button virtualization support at zero engineering cost.

The input API specification includes both the C header and documentation for the API. The documentation is required because C function declarations provide no semantic information about functions or arguments. The AvA prototype uses argument types to infer semantic information, and requires the programmer to verify its results. For example, in the API shown in Figure 4, the const qualifier implies a read-only buffer. The specification can also include a resource usage policy and a scheduling configuration for the accelerators. We envision AvA will be able to leverage natural language processing to extract semantic information from the documentation and comments for API definitions [38, 48–50]: e.g. how to compute the size of a buffer argument from the rest of the arguments of a function.

From the specification, AvA generates API-specific components of the API remoting and interposition stack: a shared library and kernel module for use in the guest, an API command routing module for the hypervisor, and an API server to execute the invoked APIs in the host. For the kernel and hypervisor modules, AvA could also generate assertions and theorems which can be automatically checked to verify that the generated C code is free from specific classes of bugs. The compiled and verified code is then packaged for installation in the guest and host.

In the near term, automatically inferring a complete and correct specification is infeasible, because complex API semantics can not be expressed in the programming languages commonly used to build these APIs. E.g., clEnqueueReadBuffer is synchronous in some cases and asynchronous in others, however, this is never specified formally and only described in the documentation. Therefore, AvA implements a rich declarative API specification language. AvA generates a preliminary specification from the C header file, which the developer refines. Generating assertions and theorems for verification is a key question for future research.

The key research question for AvA is what level of automation can it provide? We believe that, given only a header file and documentation to analyze, AvA can generate an API stack for most functions where the buffer sizes are computable directly from the arguments. Documentation analysis can be replaced with developer provided annotations describing the conventions used in that header (e.g. “the size parameter for every pointer argument has the same name with _size appended”). This simple usage will provide virtualization, but will not enforce any scheduling or resource...


utilization constraints beyond command rate-limiting. However, a developer can further refine the specification to enable the more advanced capabilities of AvA.

AvA relies on the process isolation provided by the runtime libraries, drivers, or hardware to protect one guest from another. Because AvA interposes a remoting transport layer, the knowledge of the underlying device is intentionally limited. Our hypothesis is that this interposition point is sufficient to enable practical best-effort isolation guarantees.

4 Design

AvA consists of an API stack generator, called CAVA, and an API-agnostic runtime used by the generated API stack. CAVA accepts an API specification as input and generates code to para-virtualize that API (see section 4.2). The API specification directly references the original API (e.g., cl.h) and only provides additional information required by AvA. The API agnostic runtime integrates the CAVA generated components into a complete API stack.

Figure 2 shows the development workflow to support a new API with AvA. First, CAVA creates a preliminary API specification from the unmodified header file. Then, the programmer refines the specification with guidance from CAVA, only providing information that CAVA cannot infer. Once the developer is satisfied with the API specification, she invokes CAVA to generate code for the API-specific components of the API para-virtualization stack. She compiles the generated code using standard tools and auto-generated build scripts. Finally, the developer uses auto-generated scripts to integrate the generated components with the API-independent components and deploy them.

4.1 Components

AvA comprises a handful of components many of which implement internal functionality of AvA, for example AvA’s invocation router contains a rate-limiter to enforce sharing policies across VMs. These internal components are separate and can be deployed in different execution contexts (e.g., the hypervisor or an appliance VM), which enables AvA to support a range of configurations and policies with different communication transports and system architectures. The flexibility and modularity enable AvA’s components to be distributed so as to support hardware resource disaggregation, for example, AvA could be integrated with disaggregated systems such as LegoOS [42].

Figure 3 shows the high-level design of AvA, which consists of three components. The guest library intercepts the API calls made by applications running in the guest VM and marshals the arguments. The router verifies the forwarded API calls for security and schedules them according to the resource usage policies (see §4.3). The API server is a non-privileged host process which executes the forwarded API calls on behalf of the guest application. Process-level isolation is required to isolate the applications’ context types.

4.2 Tools

CAVA generates an AvA API implementation from API declarations and AvA-related annotations. The specification provides references to the API’s native implementation, API metadata, annotations on types, and annotations on specific API functions. Simple functions do not need any function-specific annotations.

Most function definitions can be directly translated to the specification. Figure 4 shows an example from the OpenCL specification. Line 1 specifies that the return value from asynchronous calls returning the type cl_int is CL_SUCCESS. Line 2 imports the unmodified OpenCL header. The rest of Figure 4 provides function-specific annotations for clEnqueueReadBuffer. Line 9 specifies that it is synchronous when blocking_read is true. Lines 10–13, provide annotations for the parameters: ptr is an output buffer to be filled by clEnqueueReadBuffer. event_wait_list is an inferred to be

```c
#include "CL/cl.h"

cl_int clEnqueueReadBuffer(
    cl_command_queue command_queue,
    cl_mem buf, cl_bool blocking_read,
    size_t offset, size_t size, void *ptr,
    cl_int num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event) {

    if(blocking_read == CL_TRUE) sync; else async;

    parameter(ptr) { out; buffer(size); }
    parameter(event_wait_list) {
        buffer(num_events_in_wait_list);
    }
    parameter(event) { out; element { allocates; } }

    type(cl_int) { success(CL_SUCCESS); }
```

Figure 4. An example of the CAVA specification format.
an input buffer read by the function because it is a const pointer. Finally, event is a single-element output buffer where the element is freshly allocated. Opaque handles, like cl_mem, are automatically detected by CAVA in many cases, but can be explicitly specified when needed.

The AvA specification language supports structures, nested arrays, callbacks, and resource utilization annotations. All follow the same basic pattern of declarative annotations in the body of function specification.

API functions may be forwarded asynchronously regardless of whether the actual API function is asynchronous when it is annotated to be asynchronous explicitly. For example, clEnqueueReadBuffer has a non-blocking option which is exposed to AvA by an explicit annotation. In addition, clSetKernelArg can be forwarded asynchronously (even though it is synchronous in OpenCL) to reduce the overhead of these calls. This allows AvA to return to the application immediately after such a call is enqueued. Optimizations such as lazy RPC (by vCUDA [43]) and API batching (by rCUDA [27]) will be applied to these API functions with a certain degree of fidelity loss—transparently asynchronous calls are only semantically correct when the call has no output of any kind, meaning asynchronous calls cannot report errors faithfully. In most cases, the error can be delivered from a later API call, but this will not be faithful to a local execution if originally synchronous calls were forwarded asynchronously.

4.3 Resource Management

The router enforces various policies, e.g. rate limiting, at the transport layer. As in previous API remoting systems, the router schedules execution at function call granularity; unlike previous systems, the router runs in the hypervisor and can be federated with schedulers for other resources to improve locality and performance. AvA scheduling relies on resource usage approximations specified for API functions. For example, the code may estimate the bus bandwidth used by a data copy using the number of bytes copied. Some approximations may be much less accurate: for instance, estimating the device time of an OpenCL kernel execution based on the work group sizes and elapsed wall-clock time. However, we conjecture that these approximations will still provide a useful level of performance isolation, and the hypervisor can use the profiling interface of the API for more precise measurements.

AvA supports VM migration by recording and replaying a subset of API function calls and synthesizing operations to copy device memory content. Functions such as global configuration (e.g., cuInit), object allocation or deallocation (e.g., clCreateImage), and object modification (e.g., clCompileProgram) are annotated, so that AvA can select APIs to be recorded (similar to Nooks object tracking [47]) during the normal execution and perform relevant actions during the replay. The API server or the guest library suspends all invocations, synthesizes copies from all extant device buffers to host memory, and frees all in-use device resources. At this point, any migration technique can be used to migrate the VM. Upon arrival, AvA replays the recorded calls to reinitialize the device and reallocate all the device objects, restores the device buffers, and resumes the application’s normal execution. AvA avoids exposing out-of-memory conditions to contending guest VMs by supporting memory swapping at buffer object granularity, which reduces overhead and driver modification relative to page- or chunk-based management [32, 33, 55].

Our key proposition is that the API specification not only annotates how the APIs are transferred, but also annotates how the accelerator resources are managed and the information of the objects’ dependencies, life time, and metadata. AvA can then provide an administration interface to control how much of each specified API resource (e.g., device time or memory) each VM is allotted and how they should be scheduled.

5 Preliminary Results

We have implemented a prototype and optimizations to para-virtualize 39 commonly used OpenCL functions. In addition, we para-virtualized the NCSDK MVNC APIs provided by the Intel Movidius Neural Compute Stick (NCS) [14]. We found that AvA reduced the difficulty in building a para-virtualization system significantly: para-virtualizing the OpenCL and NCSDK APIs from scratch took us mere developer-days.

We ran the Rodinia OpenCL benchmarks [24] and Inception Net v3 on the NVIDIA GTX 1080 GPU and Intel NCS, respectively. Figure 5 shows that AvA’s para-virtualization introduces at most 16% overhead (8% on average) to end-to-end performance for the OpenCL benchmarks. The overhead is about 1% for Inception Net v3 running on the Intel NCS.

We plan to use AvA to auto-virtualize other accelerator APIs, including Intel QuickAssist [15] and BrainChip [5]. We also plan to extend AvA to support dynamic languages, e.g. Python, allowing us to auto-virtualize TensorFlow [20] running on the Google TPU. We are exploring other optimization opportunities as well. For example, the specification allows certain API functions to execute asynchronously.
This optimization improves efficiency by overlapping the API execution with application execution, achieving an 8.6% speedup compared to an unoptimized specification and a 5% overhead compared to native in recent experiments.

6 Discussion

Do users really want virtual accelerators? Major cloud computing providers support TPUs, GPUs, and FPGAs. Even if a single-tenant-per-accelerator model is tolerable in the near term, consolidation is the heart of cloud providers’ business model. Under-utilization is already a problem for GPUs [19, 37, 39, 56]. As the ecosystem matures, the incentive to increase profits will drive providers inevitably toward virtualization and multi-tenancy for accelerators, whether users want them or not.

Is API-remoting just a stop-gap until devices support SR-IOV? We believe API-remoting will remain relevant even as hardware vendors implement virtualization features. There is a plausible future in which all server accelerators support SR-IOV, enabling them to be exposed to guests using PCIe pass-through. However, this vision relies heavily on the hardware to implement all the resource management, policy, and sharing features currently present in commodity hypervisors. While not impossible, we believe this scenario is unrealistic, as it puts a significant burden on accelerator vendors who are not incentivized to implement such features. More importantly, the vision runs counter to conventional wisdom by baking complex resource management functionality into hardware.

Is hand-built API-remoting really so time-consuming that automation is required? The authors’ experience implementing OpenCL support in VMware’s SVGA2 virtual GPU suggests that the answer is a resounding “yes”. Also SVGA2 has only recently announced support for the DirectX 10.1 standard which came out in 2008. This demonstrates that implementing an API remoting system for an accelerator is not simply a matter of marshaling and sending function calls to an API server. It requires implementing guest framework libraries to replace vendor libraries that are “bug-for-bug” compatible [52]. The server component must also manage implicit accelerator state and arbitrate access. In addition, an implementation which preserves interposition in the hypervisor (e.g., SVGA2) requires guest device drivers and API-aware communication and resource management in the hypervisor. Even without hypervisor interposition, accelerator API forwarding is challenging: Bitfusion, a startup focused on API remoting for GPUs, has invested over a year of effort in building remoting support for CUDA alone.

How does automatic generation of API remoting stacks compensate lost compatibility? Along with cross-framework compatibility, API remoting systems typically sacrifice binary compatibility, as is true with all para-virtualization systems: the guest OS must be modified to include binaries provided by the virtualization framework (libraries, drivers, etc.). Automation does not help with this loss form of compatibility. Instead, automation helps address the challenge that is introduced by admitting binary modifications to guest OSes: evolving modified guest libraries and drivers with the changes to the API for each supported guest OS. By separating para-virtual transport from API-specific components, AvA enables automatic construction of these API-specific components, thereby compensating for, rather than recovering lost compatibility, ensuring the system can keep up with rapid accelerator evolution.

How important is it to keep pace with APIs/standards? While ecosystems for accelerators like CUDA GPUs are mature enough that keeping parity with the most recent version may not be critical, we argue that for emerging accelerators (e.g., TPUs) keeping pace with the rapid changes in APIs and frameworks is critical for usability. In the past few years alone, the number of startups focused on hardware acceleration for AI alone is dizzying: GraphCore [12], Movidius [14], GyrFalcon [45], Cambricon [59], Habana [13] Cerebras [6], Intel Nervana [16], multiple TPU variants from Google [11], among others. If the hardware specialization continues, we believe agility will become more and more important.

Do accelerators provide enough isolation between users? Our experience with the AvA prototype is that the vast majority of accelerators support sufficient process-level isolation. The only exception is the emerging Movidius [14] platform. AvA relies on the accelerators memory isolation features for accelerators with private memory. Since this allows guest state to remain in memory while other guests use the accelerator. For accelerators that have minimal onboard memory, AvA can time-share the entire device efficiently, because little or no guest state need be evicted. Our experience is that while performance isolation in accelerators is imperfect, it is sufficient to provide best-effort guarantees similar to what many cloud VMs provide for more traditional hardware like CPUs and memory. Minimizing interference caused by sharing is an active research area [21, 22].

7 Conclusion

Accelerator stacks are silos, making virtualization techniques that rely on clean separation between software layers untenable. We explore an alternative approach, AvA, which virtualizes arbitrary device-centric APIs by automatically constructing interoperable remoting infrastructure. AvA eases the difficulty of building a para-virtualization system and shortens the development cycle for virtualizing each new accelerator.

Acknowledgments

We would like to thank the anonymous reviewers and SCEA group members for their helpful feedback. This research was supported the NSF grant 1618563.
Automatic Virtualization of Accelerators

HotOS’19, May 2019, Bertinoro, Italy

References


[49] Lin Tan, Ding Yuan, and Yuanyuan Zhou. 2007. Hotcomments: how to make program comments more useful?. In HotOS


[58] Jose Fernando Zazo, Sergio Lopez-Buedo, Yuri Audzevich, and Andrew W Moore. 2015. 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. IEEE, 1–6.