Automatic Virtualization of Accelerators

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

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

Amogh Akshintala
The University of North Carolina at Chapel Hill
aakshintala@cs.unc.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) repurposes 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:
Hangchen Yu, Arthur M. Peters, Amogh Akshintala, and Christopher J. Rossbach. 2019. Automatic Virtualization of Accelerators. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3317550.3321423

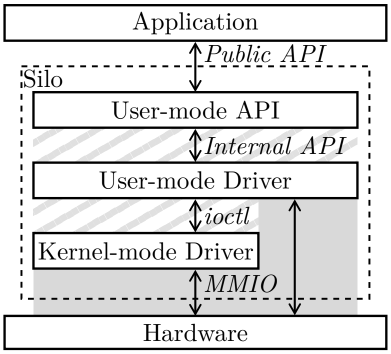

Figure 1. A typical application, accelerator silo, and hardware. The dashed box indicates the tightly coupled silo. The public API and interfaces with stripped backgrounds are virtualizable. All interfaces with backgrounds are *unstable* and cannot be relied on in a new version of the silo.

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

HotOS'19, May 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321423

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 accelator 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

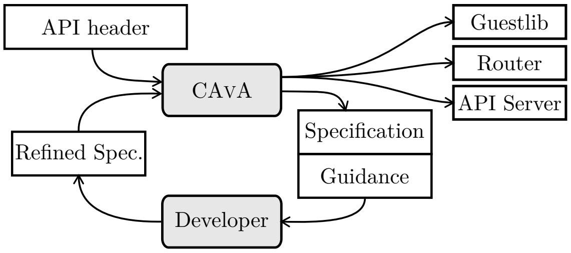

Figure 2. The AvA developer workflow. Rounded boxes represent operations, squared boxes represent input and output data.

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, its 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].

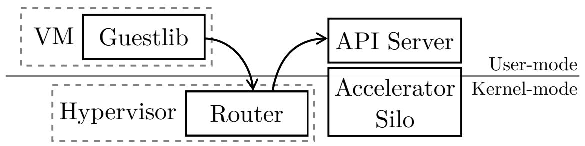

Figure 3. The components of AvA. The accelerator silo is detailed in Figure 1.

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' device contexts.

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 `clEnqueue ReadBuffer`. 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 a inferred to be

```
1  type(cl_int) { success(CL_SUCCESS); }
2  #include <CL/cl.h>
3  cl_int clEnqueueReadBuffer(
4      cl_command_queue command_queue,
5      cl_mem buf, cl_bool blocking_read,
6      size_t offset, size_t size, void *ptr,
7      cl_uint num_events_in_wait_list,
8      const cl_event *event_wait_list, cl_event *event) {
9    if(blocking_read == CL_TRUE) sync; else async;
10   parameter(ptr) { out; buffer(size); }
11   parameter(event_wait_list) {
12     buffer(num_events_in_wait_list); }
13   parameter(event) { out; element { allocates; } }
14 }
```

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].

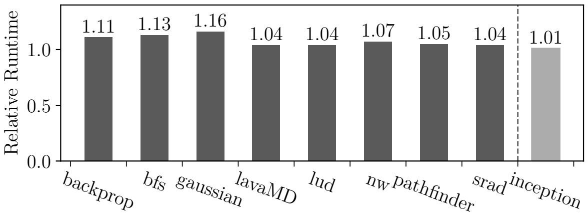

Figure 5. End-to-end relative execution time of benchmarks (normalized to native GPU or Movidius as appropriate). *Inception* is Inception Net v3 ported to Movidius and the others are from Rodinia.

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 interposable 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.

References

[1] [n. d.]. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1. Accessed: 2018-04.

[2] [n. d.]. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types. Accessed: 2018-04.

[3] [n. d.]. AMD multiuser GPU. http://www.amd.com/Documents/Multiuser-GPU-White-Paper.pdf. Accessed: 2018-07.

[4] [n. d.]. Bitfusion: The Elastic AI Infrastructure for Multi-Cloud. https://bitfusion.io/. April. 2019.

[5] [n. d.]. BrainChip Accelerator. https://www.brainchipinc.com/products/brainchip-accelerator. Accessed: 2019-04.

[6] [n. d.]. Cerebras Systems. https://www.cerebras.net/. Accessed: 2019-04.

[7] [n. d.]. Five Reasons Machine Learning Is Moving to the Cloud. https://www.entrepreneur.com/article/300713. [Published Nov 3, 2017].

[8] [n. d.]. Genomics in the Cloud. https://aws.amazon.com/health/genomics. Accessed: 2018-08.

[9] [n. d.]. Google Cloud GPU. https://cloud.google.com/gpu. Accessed: 2018-04.

[10] [n. d.]. Google Cloud Machine Learning Engine. https://cloud.google.com/ml-engine. Accessed: 2018-04.

[11] [n. d.]. Google Cloud TPU. https://cloud.google.com/tpu. Accessed: 2019-01.

[12] [n. d.]. Graphcore Inc. https://www.graphcore.ai. Accessed: 2018-04.

[13] [n. d.]. Habana Labs. https://habana.ai/. Accessed: 2019-04.

[14] [n. d.]. Intel Movidius Myriad 2 VPU. https://www.movidius.com/solutions/vision-processing-unit. Accessed: 2018-04.

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

[16] [n. d.]. Nervana Neural Network Processor. https://ai.intel.com/nervana-nnp. Accessed: 2019-01.

[17] [n. d.]. NVIDIA GPU Cloud. https://www.nvidia.com/en-us/gpu-cloud. Accessed: 2018-04.

[18] [n. d.]. Olympus Cloud Services. https://olympustech.com.au/services/cloud-services. Accessed: 2018-04.

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

[20] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

[21] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. 2018. Mosaic: Enabling Application-Transparent Support for Multiple Page Sizes in Throughput Processors. *ACM SIGOPS Operating Systems Review* 51, 1 (2018), 27–44.

[22] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J Rossbach, and Onur Mutlu. 2018. Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 503–518.

[23] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. 2010. 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*. 1–7. https://doi.org/10.1109/CLUSTERWKSP.2010.5613086

[24] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 44–54.

[25] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38, 2 (2018), 8–20.

[26] Micah Dowty and Jeremy Sugerman. 2009. GPU Virtualization on VMware's Hosted I/O Architecture. *SIGOPS Oper. Syst. Rev.* 43, 3 (July 2009), 73–82. https://doi.org/10.1145/1618525.1618534

[27] Jose Duato, Antonio J. Pena, Federico Silla, Juan C. Fernandez, Rafael Mayo, and Enrique S. Quintana-Orti. 2011. Enabling CUDA acceleration within virtual machines using rCUDA. In *Proceedings of the 2011 18th International Conference on High Performance Computing (HIPC '11)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/HiPC.2011.6152718

[28] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. 2010. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. *Euro-Par 2010-Parallel Processing* (2010), 379–391.

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

[30] Alex Herrera. 2014. NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation. *Nvidia Corp* (2014).

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

[32] Feng Ji, Heshan Lin, and Xiaosong Ma. 2013. RSVM: a region-based software virtual memory for GPU. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*. IEEE, 269–278.

[33] Jens Kehne, Jonathan Metter, and Frank Bellosa. 2015. GPUswap: Enabling oversubscription of GPU memory through transparent swapping. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 65–77.

[34] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. 2012. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 341–352.

[35] Patrick Kutch. 2011. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. *Intel application note* (2011), 321211–002.

[36] Tyng-Yeu Liang and Yu-Wei Chang. 2011. GridCuda: A Grid-Enabled CUDA Programming Toolkit. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*. 141–146. https://doi.org/10.1109/WAINA.2011.82

[37] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 308–317.

[38] Pengyu Nie, Junyi Jessy Li, Sarfraz Khurshid, Raymond Mooney, and Milos Gligoric. 2018. Natural Language Processing and Program Analysis for Supporting Todo Comments as Software Evolves. In *Proceedings of the AAAI Workshop of Statistical Modeling of Natural Software Corpora*.

[39] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1935–1950.

[40] Sébastien Pinneterre, Spyros Chiotakis, Michele Paolino, and Daniel Raho. 2018. vFPGAmanager: A virtualization framework for orchestrated FPGA accelerator sharing in 5G cloud environments. In *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*. IEEE, 1–5.

[41] C. Reano, A. J. Pena, F. Silla, J. Duato, R. Mayo, and E. S. Quintana-Orti. 2012. CU2rCU: Towards the complete rCUDA remote GPU virtualization and sharing solution. *20th Annual International Conference on High Performance Computing* 0 (2012), 1–10. https://doi.org/10.1109/HiPC.2012.6507485

[42] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed {OS} for Hardware Resource Disaggregation. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 69–87.

[43] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. 2012. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Trans. Comput.* 61, 6 (June 2012), 804–816. https://doi.org/10.1109/TC.2011.112

[44] Jike Song, Zhiyuan Lv, and Kevin Tian. 2014. KVMGT: A full GPU virtualization solution. In *KVM Forum*, Vol. 2014.

[45] Baohua Sun, Daniel Liu, Leo Yu, Jay Li, Helen Liu, Wenhan Zhang, and Terry Torng. 2018. MRAM Co-designed Processing-in-Memory CNN Accelerator for Mobile and IoT Applications. *arXiv preprint arXiv:1811.12179* (2018).

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

[47] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. 2003. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 207–222. https://doi.org/10.1145/945445.945466

[48] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments?*/. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 145–158.

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

[50] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 11–20.

[51] Kun Tian, Yaozu Dong, and David Cowperthwaite. 2014. A Full GPU Virtualization Solution with Mediated Pass-through. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 121–132. http://dl.acm.org/citation.cfm?id=2643634.2643647

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

[53] Duy Viet Vu, Oliver Sander, Timo Sandmann, Steffen Baehr, Jan Heidelberger, and Juergen Becker. 2014. 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*. IEEE, 1–6.

[54] Lan Vu, Hari Sivaraman, and Rishi Bidarkar. 2014. GPU Virtualization for High Performance General Purpose Computing on the ESX Hypervisor. In *Proceedings of the High Performance Computing Symposium (HPC '14)*. Society for Computer Simulation International, San Diego, CA, USA, Article 2, 8 pages. http://dl.acm.org/citation.cfm?id=2663510.2663512

[55] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. 2014. GDM: device memory management for GPGPU computing. *ACM SIGMETRICS Performance Evaluation Review* 42, 1 (2014), 533–545.

[56] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 358–369.

[57] Hangchen Yu and Christopher J Rossbach. 2017. Full Virtualization for GPUs Reconsidered. In *Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking*.

[58] Jose Fernando Zazo, Sergio Lopez-Buedo, Yury 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.

[59] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 20.