AvA: Accelerated Virtualization of Accelerators

Hangchen Yu[†], Arthur Michener Peters[†], Amogh Akshintala[§], Christopher J. Rossbach[†¶]
[†]The University of Texas at Austin, [§]The University of North Carolina at Chapel Hill, [¶]VMware Research
{hyu,amp}@cs.utexas.edu,aakshintala@cs.unc.edu,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 trends are in conflict: cloud applications run on virtual platforms, but existing virtualization techniques have not provided production-ready solutions for accelerators. As a result, cloud providers expose accelerators by **dedicating** physical devices to individual guests. Multi-tenancy and consolidation are lost as a consequence.

We present AvA, which addresses limitations of existing virtualization techniques with automated construction of hypervisor-managed virtual accelerator stacks. AvA combines a DSL for describing APIs and sharing policies, device-agnostic runtime components, and a compiler to generate accelerator-specific components such as guest libraries and API servers. AvA uses **Hypervisor Interposed Remote Acceleration** (HIRA), a new technique to enable hypervisor-enforcement of sharing policies from the specification.

We use AvA to virtualize nine accelerators and eleven framework APIs, including six for which no virtualization support has been previously explored. AvA provides near-native performance and can enforce sharing policies that are not possible with current techniques, with orders of magnitude less developer effort than required for hand-built virtualization support.

CCS Concepts • **Software and its engineering** **Virtual machines**; *Operating systems*; *Source code generation*.

Keywords Virtualization, Code generation

ACM Reference Format:
Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. 2020. AvA: Accelerated Virtualization of Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3373376.3378466

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378466

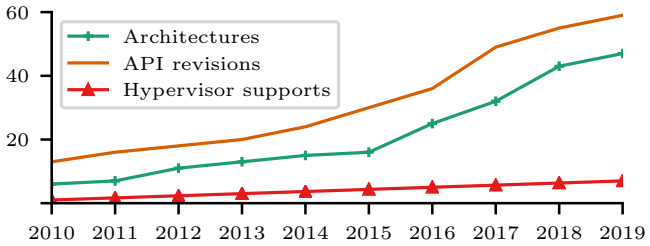

Figure 1. The number of accelerators (discrete GPUs and AI accelerators) and APIs released since 2010 compared to the number of accelerators officially supported by production hypervisors (VMware ESX, Citrix XenServer, and Microsoft Hyper-V). This data was drawn from release notes and specification sheets.

1 Introduction

Applications are migrating to the cloud in search of scalability, resilience, and cost-efficiency. At the same time, silicon scaling has stalled, precipitating a wave of new specialized hardware accelerators such as TPUs [53] and IPUs [14], and on-demand GPU and FPGA support from cloud providers. Accelerators have driven the success of emerging application domains in the cloud [9, 10], but cloud computing and hardware specialization are on a collision course. Cloud applications run on virtual infrastructure, but practical virtualization support for accelerators has yet to arrive. Cloud providers routinely support accelerators [2, 3, 11, 12, 17, 18], but do so using PCIe pass-through techniques that dedicate physical hardware to VMs, despite the wealth of techniques known in the literature [24, 38–43, 63, 76, 84, 87, 90, 94, 95, 97, 98, 105]. This sacrifices the consolidation that drives their business, and leads inevitably to hardware underutilization [19, 70, 72, 100].

The problem is increasingly urgent, as hypervisors have not kept pace with accelerator innovation. Figure 1 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 [2, 3, 11–13, 17, 18]. The challenge is technical: hypervisor-level accelerator virtualization requires substantial engineering effort and the design space features multiple fundamental tradeoffs 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 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 (§2.3).

This paper 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 [5, 40, 51, 79, 97], 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 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 [38] or BitFusion's FlexDirect [5]. 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 at GitHub utcs-scea/ava. We make the following contributions:

- We demonstrate feasibility of automatically constructed virtual accelerator support, using a single technique to support many architectures, APIs, versions, and policies.
- We introduce Hypervisor Interposed Remote Acceleration (HIRA) to enable hypervisor-enforced isolation and sharing policies unachievable with current SR-IOV and API remoting systems (§3).
- We describe a novel DSL, Lapis, for describing API functions, resources, and policies to enable automatic construction of virtual stacks starting from native API header files.
- We evaluate AvA on nine accelerators showing low effort, strong properties, and good performance (§6).

2 Background

Cloud providers currently support compute accelerators using PCIe pass-through, which dedicates the device exclusively to a single guest. Despite evidence of accelerator under-utilization [19, 21, 70, 72, 100], and abundant research [19, 70, 72, 100, 105, 109], no practical alternative yet exists. This section provides background to explain this trend and motivates AvA's design.

2.1 Virtualization Properties

Virtualization is a huge area—see [32] for comprehensive treatment. We focus on properties critical for accelerator virtualization: *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 [98]. 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 [90, 107]; *incomplete* interposition compromises the hypervisor's ability to enforce isolation.

Compatibility captures multiple related dimensions, including robustness to evolution of interposed interfaces and adjacent stack layers, and applicability across multiple platforms or related devices. For example, full virtualization of a accelerators's hardware interface (§2.3) has *poor* compatibility in that it works only with that device. However, it has *good* compatibility with guest software, which will work without modification. 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 [5, 16, 40] can have poor isolation (§2.3) because the hypervisor is bypassed.

2.2 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,

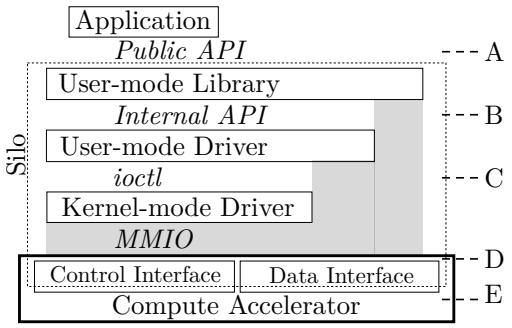

Figure 2. An accelerator silo. The public API and the interfaces with striped backgrounds are interposition candidates. All interfaces with backgrounds are proprietary and subject to change.

interposing opaque, frequently-changing interfaces 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 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 (§2.3). 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.

2.3 Existing Accelerator Virtualization Techniques

PCIe pass-through is the current *de facto* standard technique for exposing an accelerator to guest VMs. It works by dedicating the hardware interface directly to the guest. Consequently, it does not interpose *any* interface. All benefits of virtualization are lost, but native performance is preserved.

Full virtualization (FV) [87, 90, 94] interposes the hardware interface (D in Figure 2). For accelerators, this interface is memory mapped I/O (MMIO), necessitating trap-based interposition (e.g. using memory protection or deprivileging), which devastates performance (e.g., 100× or more [90, 107]). Accelerator hardware interfaces are proprietary and device-specific, so FV has poor compatibility, even across different devices of the same type (e.g. AMD vs NVIDIA GPUs).

Mediated pass-through (MPT) is a hybrid of pass-through and full virtualization. MPT [73, 94, 102] uses pass-through for data plane operations, and provides a privileged control plane interface for sensitive operations (D in Figure 2). MPT 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 and, to our knowledge, Intel integrated GPUs are the only accelerators to support MPT [94].

Para-virtualization (PV) *creates* an efficiently interposable interface in software and adjusts adjacent stack layers to use it, rather than interposing an existing interface in the stack. This necessitates a custom driver and library in every supported guest OS for the virtual device. The technique is powerful because enables encapsulation of diverse hardware behind a single interposable interface. However, compatibility is compromised: guest library and OS modifications are required, and the para-virtual device interface must be maintained as interfaces evolve. For example, VMware's SVGA II [38] encapsulates multiple GPU programming frameworks, but keeping up with the evolution of those frameworks has proved untenable: SVGA remains multiple versions behind current frameworks [22, 23] (e.g., DirectX 12 [8]).

SR-IOV. Accelerators with PCIe SR-IOV [60] present multiple virtual devices (VFs) to system software. SR-IOV provides an interface and protocol for managing VFs, but the device vendor must *implement* any cross-VF sharing support (E in Figure 2) *in silicon*. The technique can provide strong virtualization guarantees [36, 37], but hardware-level resource management is inflexible and unevolvable: current implementations are trivially vulnerable to fragmentation and unfairness pathologies that cannot be changed. Moreover, evidence is scant that broad SR-IOV support will emerge for accelerators: only two current GPUs support it [4, 46], none of the TPUs we evaluate support it; and SR-IOV *interface* IP blocks from FPGA vendors (used by [47, 76, 96, 108]) do not implement resource management. We do not expect this to change any time soon: SR-IOV requires significant engineering effort vendors are not incentivized to invest.

API Remoting [30, 39, 41, 43, 44, 63, 64, 79, 101, 103] interposes the top of the stack, between application and framework library (A in Figure 2). Similar to system call interposition [25, 26, 59, 91], API calls are intercepted and forwarded to a user-level API framework [84] in an appliance VM [97], or remote server [40, 56]. Limited interposition frequency, batching opportunities [40] and high-speed networks [5, 7] reduce overheads, making it appealing to industry. Dell XaaS [51], BitFusion FlexDirect [5], and Google Cloud TPUs [13] use it to support GPUs, FPGAs, and TPUs. However, API 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 [5] show vulnerability to massive unfairness pathologies (up to 88.1%) that AvA's hypervisor-level enforcement avoids (§6.5).

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

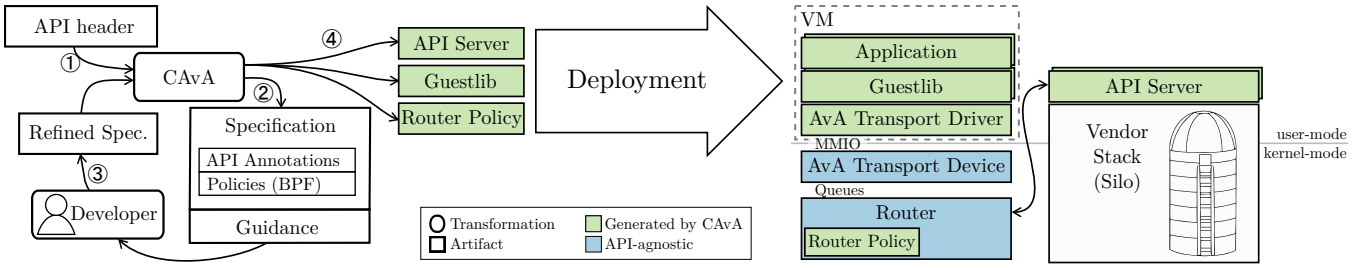

Figure 3. Overview of AvA.

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 (§4), which introduces the need to specify API semantics and policies beyond the ability of existing Interface Description Languages (IDLs) [61, 68].

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 3 shows the work-flow to support a new API with AvA (§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 [78].

3.1 AvA Components

Figure 3 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.

3.2 Developer Work-flow

AvA's API-agnostic components (§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 3 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.

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).

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 (§4.5). This solution admits no concurrency between tenants, but enables protected sharing for devices which cannot otherwise be shared.

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 (§5.2.2) 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.

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.

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 [82]. 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.

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 the stream (`cuSynchronizeStream`). The Lapis source for the four of the five CUDA API functions required is shown in Figure 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

```
1  continuous_rc device_memory;
2  instantaneous_rc kernel_invocations;
3  policy("policy_kern.c");
4  struct fn_arg_info {
5    // CUfunction type information, filled by other LAPIS code
6    int fn_argc;
7    char fn_arg_is_handle[64];
8    size_t fn_arg_size[64];
9  };
10 declare_metadata(struct fn_arg_info);
11 buf_registry async_DtoH;
12 type(CUdeviceptr) { handle; }
13 cuMemAlloc(CUdeviceptr *pp, size_t size) {
14   allocates_rc(device_memory, size);
15   argument(pp) { out; element { obj_record; allocate; } } }
16 cuMemcpyDtoHAsync(void* dst, CUdeviceptr src,
17                   size_t size, CUstream stream) {
18   async; success(CUDA_SUCCESS);
19   argument(dst) { no_copy; buffer(size); lifetime_manual; }
20   register_buf(async_DtoH, stream, dst, size); }
21 cuLaunchKernel(CUfunction f,
22     uint gridDimX, uint gridDimY, uint gridDimZ,
23     uint blockDimX, uint blockDimY, uint blockDimZ,
24     uint sharedMemBytes, CUstream hStream,
25     void **kernelParams, void **extra) {
26   async; consumes_rc(kernel_invocation, 1);
27   argument(kernelParams) {
28     in; buffer(metadata(f)->fn_argc);
29     element {
30       if (metadata(f)->fn_arg_is_handle[index]) {
31         type_cast(CUdeviceptr*); buffer(1);
32       } else {
33         type_cast(void*); buffer(metadata(f)->fn_arg_size[index]);
34       } } }
35   argument(extra) { in; ... } }
36 cuStreamSynchronize(CUstream stream) {
37   contextual_argument void** bufs = get_bufs(async_DtoH, stream);
38   argument(bufs) {
39     in; buffer(get_n_bufs(async_DtoH, stream));
40     element {
41       buffer(get_buf_size(async_DtoH, stream, index));
42       out; deallocate; } } }
```

Figure 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 7.

declarations, can be flexibly scoped (Table 1), and applied to values (Table 2) or functions (Table 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 4 (Line 16). The argument `src` is only used by the application through API calls (AvA does not currently support UVM, see §4.7 for details). This is true of the type `CUdeviceptr` in general, so it is expressed on line 12 with **`type`**`(CUdeviceptr){ `**`handle`**`; }`.

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

Table 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 2. LAPIS descriptors for specifying values and objects.

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 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 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 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 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 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 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.

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

Runtime Library. The LAPIS standard library is illustrated (in excerpted form) in Table 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 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  void api_server_handle_command(call) {
2    switch (call->command_id) {
3    case CALL_CU_STREAM_SYNCHRONIZE: {
4      // For handles, get the real address
5      CUstream stream = handle_deref(call->stream);
6      // Get pointers to shadows of the attached buffers
7      void **bufs = get_buffer(call, call->bufs);
8      size_t nbufs = get_n_bufs(async_DtoH, stream);
9      for (size_t i = 0; i < nbufs; i++)
10       bufs[i] = get_shadow_buffer(call, bufs[i]);
11     // Perform call
12     CUresult stat = cuStreamSynchronize(stream);
13     // Construct and send the return command
14     cu_stream_synchronize_ret *ret_cmd = alloc();
15     ret_cmd->command_id = RET_CU_STREAM_SYNCHRONIZE;
16     ret_cmd->call_id = call->call_id; ret_cmd->ret = stat;
17     // Attach sync buffers to return command
18     void **tmp_bufs = alloc();
19     for (size_t i = 0; i < nbufs; i++)
20       tmp_bufs[i] = attach_buffer(ret_cmd, bufs[i],
21         get_buf_size(async_DtoH, stream, i));
22     ret_cmd->bufs = attach_buffer(ret_cmd, tmp_bufs,
23       nbufs * sizeof(void *));
24     send_command(ret_cmd);
25     break; }
26   case CALL_CU_MEMCPY_DTOH_ASYNC: {
27     ...Extract dst, stream, and size from call...
28     // Get or allocate the shadow buffer
29     void *dst = get_shadow_buffer(call, call->dst);
30     // Execute state-tracking code from spec
31     register_buf(async_DtoH, stream, dst, size);
32     // Perform call
33     CUresult stat = cuMemcpyDtoHAsync(dst, src, size, stream);
34     ...Construct and send the return command...
35     break; } ... } }
```

Figure 6. An outline of the generated API server code for `cuStreamSynchronize` and `cuMemcpyDtoHAsync`.

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 [89]) 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.

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, 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 API server. The hypervisor enforces resource sharing policy at the router based on the resource accounting and policy descriptors. The API server, in Figure 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.

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 (e.g. GPUfs [85]), but will not function correctly with AvA. The limitation is not fundamental, and we plan to address it in future work.

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.

```
1  map_def cmd_cnt, priority;
2  kernel_calls = ava_get_rc_id("kernel_calls");
3  tot_id = 0; // ID of total in cmd_cnt
4  int consume(__sk_buff *skb) {
5    vm_id = load_ava_vm_id(skb)
6    amount = load_ava_rc_amount(skb, kernel_calls);
7    vm_cnt = map_lookup_elem(&cmd_cnt, &vm_id);
8    tot_cnt = map_lookup_elem(&cmd_cnt, &tot_id);
9    fetch_and_add(vm_cnt, amount);
10   fetch_and_add(tot_cnt, amount); }
11 int schedule(__sk_buff *skb) {
12   ...Load vm_id, vm_cnt, vm_pri, tot_cnt, and tot_pri...
13   if((*vm_cnt) *(*tot_pri) <= (*vm_pri) *(*tot_cnt))
14     return HIGH_PRIO;
15   else return LOW_PRIO; }
```

Figure 7. An example eBPF policy program (simplified for clarity). This is referenced from Figure 4 as `policy_kern.c`.

CAvA generates code to compute resource usage information in the hypervisor from call arguments. This code makes the call arguments available, similarly to Figure 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 4, the specification of `cuLaunchKernel` includes resource usage descriptors and a reference to a custom scheduler. Figure 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 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 [1, 65, 80].

4.5 VM Migration

AvA supports VM migration using *record-and-replay* [67], 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.

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 [28, 29, 62]. 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 [52, 54, 99].

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 §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 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 (§5.2). 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.

5.1 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.

5.2 Hypervisor Interposition and Mediation

AvA enables hypervisor mediation by interposing the transport channel. We extended QEMU *virtio-vsock* [45, 81] (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.

5.2.1 Policies in eBPF

AvA supports policies written as eBPF (Extended Berkeley Packet Filter [66]) 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.

5.2.2 Scheduling

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

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

Feedback control where the adaptive delay, $d_v(t)$, is computed by the additive-increase multiplicative-decrease (AIMD) algorithm [1] below ($a = 1$ ms and $b = 1/2$; see §6.5).

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

5.3 Shadow Resources

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

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.

API	Gen	#	LoC	Churn	Benchmark	Hardware
OpenCL 1.2	✗	39	7514	14 318	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 [55]	✓	4	30	40	BitCoin	AWS F1
QuickAssist 1.7	✓	19	444	676	QATzip	Intel QAT 8970
HIP	✓	41	624	990	Galois [75]	AMD Vega 64

Table 5. Development effort for forwarding different APIs, along with the benchmarks [33, 86, 92] 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.

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 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.

6.1 Development Effort

We present our experience building API remoting systems by hand and with CAvA as evidence of the significant reduction

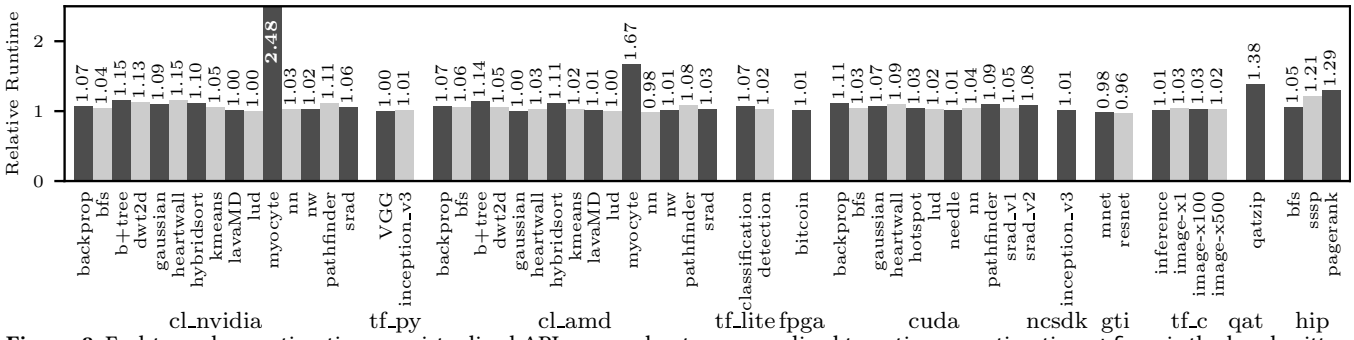

Figure 8. End-to-end execution time on virtualized APIs or accelerators normalized to native execution time. `tf_py` is the handwritten TensorFlow Python API remoting with AvA API-agnostic components.

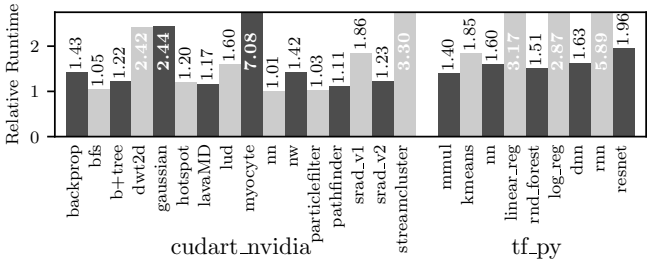

Figure 9. End-to-end execution time on virtualized CUDART and CUDA-accelerated TensorFlow APIs normalized to native execution time.

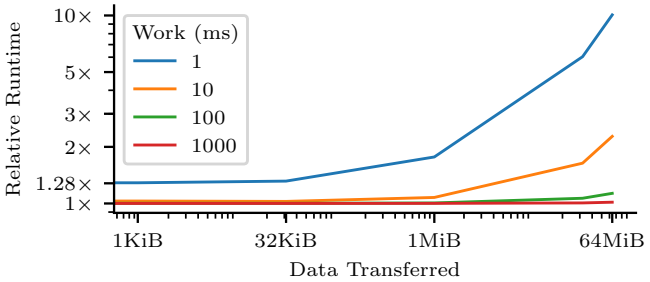

Figure 10. Overhead on a micro-benchmark with varying work per call and data per call. The plot is log-log and the trend is linear.

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), and the number of lines of code modified (**Churn** in Table 5) during development (counted from commits). Building an API remoting system for OpenCL, which supports all the APIs needed to run the Rodinia [33] benchmarks, by hand took more than 3 developer-months, and spanned 7,514 LoC (see row 1 of Table 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).

6.2 End-to-end Performance

Figure 8–9 shows the end-to-end runtime, normalized to native, for all benchmarks, accelerator and API combinations we support (see Table 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 [107]. `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 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 [35] 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.

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 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).

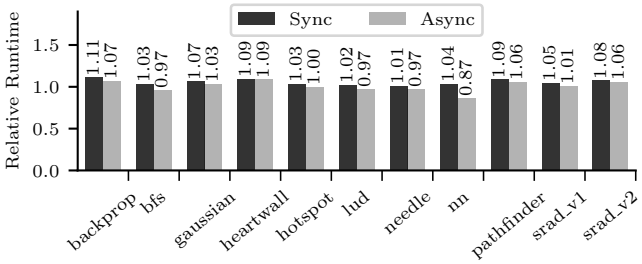

Figure 11. End-to-end runtime of CUDA benchmarks (relative to native) using synchronous and asynchronous specifications.

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) [84] and rCUDA (API batching) [40].

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

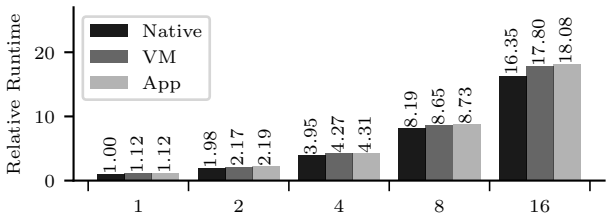

Figure 12. Scalability of multiple VMs running a single application each, and multiple applications in a single VM, with AvA. Runtime is relative to running the same number of applications natively.

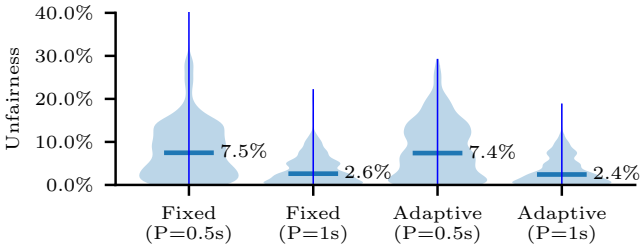

Figure 13. Unfairness of the fixed and adaptive scheduling algorithms with two different measurement periods. The width of the shaded areas show the probability of the bias (unfairness) being a specific value in any given measurement window. The horizontal bar shows the median and the vertical line runs from the minimum to the maximum.

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 12. The GPU kernel execution has an average 5.7% slowdown each time the number of VMs and applications is doubled. This slowdown is small due better utilization of the physical device and other system resources.

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.

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 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.

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, 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

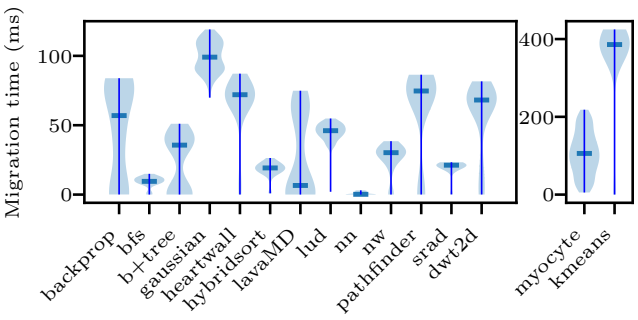

Figure 14. Live migration downtime for single-threaded OpenCL benchmarks on NVIDIA GTX 1080. This downtime is in addition to the ~75 ms of downtime of the VM migration itself. Migration downtime does not include time spent waiting for executing kernels to complete (accounted as latency), as the application is still performing useful computation on the accelerator during that time.

or GPU usage takes 19 s with a 75 ms downtime on average.

Figure 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 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.

7 Related Work

Section 2 provides detailed related work for AvA; this section addresses related work not covered there. Our previous work [106] proposed many key ideas in AvA; this paper fleshes out and evaluates those ideas.

GPU virtualization. Table 6 shows prior GPU virtualization and trade-offs across virtualization properties (see §2).

FPGA virtualization has a long history [31, 34, 48, 57, 58, 69, 74, 77, 88]. Most prior work relies on hardware-specific features, focuses on sharing in a single protection domain [55], or virtualization primitives [83]. AvA can be combined with any of these techniques to virtualize FPGA accelerators.

Nooks [91] 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 [27, 71, 93, 104] 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.

Technique	System	lib unmod	OS unmod	lib compat	HW compat	sharing	isolation
Full-virtual	GPUvm [90]	✓		✓		✓	✓
	gVirt [94]	✓		✓		✓	✓
PCIe Pass-thru	AWS GPU [3]	✓	✓				✓
API remoting	GViM [43]				✓	✓	
	gVirtuS [41]				✓	✓	
	vCUDA [84]		✓		✓	✓	
	vmCUDA [97]		✓		✓	✓	✓
	FlexDirect [5]		✓		✓	✓	✓
Distributed API remoting	rCUDA [7, 40]		✓		✓	✓	✓
	GridCuda [64]		✓		✓	✓	
	SnuCL [56]		✓			✓	
	VCL [30]		✓		✓	✓	
Para-virtual	GPUvm [90]					✓	✓
	HSA-KVM [49]	✓				✓	✓
	LoGV [42]	✓		✓		✓	✓
	SVGA2 [38]	✓				✓	✓
	Paradice [25]	✓		✓		✓	✓
	VGVM [95]				✓	✓	✓
	AvA	*	*	✓	✓	✓	✓

Table 6. Comparison of existing compute GPU virtualization proposals. **lib unmod** and **OS unmod** indicate support for unmodified guest libraries and OS/driver. **lib compat** and **HW compat** indicate support for a virtual interface that is independent of the *framework* or *hardware* virtualized. **sharing** and **isolation** indicate cross-domain sharing and isolation. Grayed-out cells indicate that the metric is meaningless for that design. * indicates that while AvA requires library and OS modifications, it automates these changes.

Program specification languages [50] allow programmers to specify properties of functions and their behavior, and are generally used to check correctness.While such languages allow (nearly) arbitrary predicates on programs, they are not designed to provide semantic information to other tools, and do not support features like state tracking.

Foreign Function Interface tools allow one language to call functions written in another, such as C. Some [20] make use of C headers, but require manual annotations in many common cases. Unlike AvA, language specific DSLs [6, 15], do not support marshalling data structures and encapsulate rather than export the C API.

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.

Acknowledgments

We thank the PC and our shepherd Mark Silberstein for their insightful feedback. This research was supported by NSF grants CNS-1618563 and CNS-1846169.

References

[1] [n.d.]. Additive Increase/Multiplicative Decrease. https://en.wikipedia.org/wiki/Additive_increase/multiplicative_decrease. Accessed: 2019-08.

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

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

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

[5] [n.d.]. Bitfusion: The Elastic AI Infrastructure for Multi-Cloud. https://bitfusion.io. Accessed: 2019-04.

[6] [n.d.]. Cython: C-Extensions for Python. https://cython.org/. Accessed: 2019-08.

[7] [n.d.]. Deploying rCUDA in Cloud Computing Environments. http://www.rcuda.net/pub/white_paper_cloud_v2.pdf. Published: 2016-12.

[8] [n.d.]. DirectX Version History. https://en.wikipedia.org/wiki/DirectX#Version_history. Accessed: 2019-08.

[9] [n.d.]. Five Reasons Machine Learning is Moving to the Cloud. https://www.datanami.com/2015/04/29/5-reasons-machine-learning-is-moving-to-the-cloud. Published: 2015-04.

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

[11] [n.d.]. Google Cloud AI Platform. https://cloud.google.com/ai-platform. Accessed: 2020-01.

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

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

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

[15] [n.d.]. Java Native Access (JNA). https://github.com/java-native-access/jna. Accessed: 2019-08.

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

[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] [n.d.]. SWIG: Simplified Wrapper and Interface Generator. http://www.swig.org. Accessed: 2019-08.

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

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

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

[24] Amogh Akshintala, Hangchen Yu, Arthur Peters, and Christopher J Rossbach. 2019. Trillium: The code is the IR. In *The Second Special Session on Virtualization in High Performance Computing and Simulation (VIRT '19)*. Dublin, Ireland.

[25] Ardalan Amiri Sani, Kevin Boos, Shaopu Qin, and Lin Zhong. 2014. 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)*. ACM, New York, NY, USA, 319–332. https://doi.org/10.1145/2541940.2541943

[26] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. 2014. 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*. ACM, 259–272.

[27] Apache Software Foundation. [n.d.]. Apache Thrift. https://thrift.apache.org. Accessed: 2019-04.

[28] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*. 136–150.

[29] 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 *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*. 503–518.

[30] Amnon Barak, Tal Ben-Nun, Ely Levy, and Amnon Shiloh. 2010. A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices. In *2010 IEEE international conference on cluster computing workshops and posters (CLUSTER WORKSHOPS)*. IEEE, 1–7.

[31] Alexander Brant and Guy GF Lemieux. 2012. ZUMA: An Open FPGA Overlay Architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 93–96.

[32] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. Hardware and Software Support for Virtualization. *Synthesis Lectures on Computer Architecture* 12, 1 (2017), 1–206.

[33] 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.

[34] André DeHon, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, and John Wawrzynek. 2006. Stream Computations Organized for Reconfigurable Execution. *Microprocessors and Microsystems* 30, 6 (2006), 334–354. https://doi.org/10.1016/j.micpro.2006.02.009

[35] Sebastian Deorowicz. 2003. *Universal Lossless Data Compression Algorithms*. Ph.D. Dissertation. Silesian University of Technology. http://sun.aei.polsl.pl/~sdeor/pub/deo03.pdf

[36] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2012. High Performance Network Virtualization with SR-IOV. *J. Parallel and Distrib. Comput.* 72, 11 (2012), 1471–1480.

[37] Yaozu Dong, Zhao Yu, and Greg Rose. 2008. SR-IOV Networking in Xen: Architecture, Design and Implementation.. In *Workshop on I/O Virtualization*, Vol. 2.

[38] Micah Dowty and Jeremy Sugerman. 2009. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review* 43, 3 (2009), 73–82.

[39] José Duato, Francisco D Igual, Rafael Mayo, Antonio J Peña, Enrique S Quintana-Ortí, and Federico Silla. 2009. An Efficient Implementation of GPU Virtualization in High Performance Clusters. In *European Conference on Parallel Processing*. Springer, 385–394.

[40] José 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 *2011 18th International Conference on High Performance Computing*. IEEE, 1–10.

[41] 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.

[42] Mathias Gottschlag, Marius Hillenbrand, Jens Kehne, Jan Stoess, and Frank Bellosa. 2013. LoGV: Low-overhead GPGPU virtualization. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded Ubiquitous Computing (HPCC_EUC)*. IEEE, 1721–1726.

[43] 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.

[44] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. 2011. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *2011 USENIX Annual Technical Conference (USENIX ATC'11)*. 31.

[45] Stefan Hajnoczi. [n.d.]. Virtio-vsock, Zero-configuration Host/Guest Communication. https://vmsplice.net/~stefan/stefanha-kvm-forum-2015.pdf. Accessed: 2019-04.

[46] Alex Herrera. 2014. NVIDIA GRID: Graphics Accelerated VDI with the Visual Performance of a Workstation. *Nvidia Corp* (2014).

[47] Chun-Hsian Huang and Pao-Ann Hsiung. 2009. Hardware Resource Virtualization for Dynamically Partially Reconfigurable Systems. *IEEE Embedded Systems Letters* 1, 1 (2009), 19–23.

[48] Chun-Hsian Huang and Pao-Ann Hsiung. 2009. Hardware Resource Virtualization for Dynamically Partially Reconfigurable Systems. *IEEE Embed. Syst. Lett.* 1, 1 (May 2009), 19–23. https://doi.org/10.1109/LES.2009.2028039

[49] Yu-Ju Huang, Hsuan-Heng Wu, Yeh-Ching Chung, and Wei-Chung Hsu. 2016. 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)*. ACM, New York, NY, USA, 3–15. https://doi.org/10.1145/2892242.2892246

[50] Microsoft Inc. 2016. Using SAL Annotations to Reduce C/C++ Code Defects. https://docs.microsoft.com/en-us/visualstudio/code-quality/using-sal-annotations-to-reduce-c-cpp-code-defects. Accessed: 2019-11.

[51] 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.

[52] 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.

[53] Norm Jouppi. 2016. Google Supercharges Machine Learning Tasks with TPU Custom Chip. *Google Blog, May* 18 (2016).

[54] 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.

[55] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 107–127.

[56] 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.

[57] Robert Kirchgessner, Alan D. George, and Greg Stitt. 2015. Low-Overhead FPGA Middleware for Application Portability and Productivity. *ACM Trans. Reconfigurable Technol. Syst.* 8, 4 (Sept. 2015), 21:1–21:22. https://doi.org/10.1145/2746404

[58] Robert Kirchgessner, Greg Stitt, Alan George, and Herman Lam. 2012. VirtualRC: A Virtual FPGA Platform for Applications and Tools Portability. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 205–208.

[59] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. 2016. Paravirtual Remote I/O. In *ACM SIGARCH Computer Architecture News*, Vol. 44. ACM, 49–65.

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

[61] David Alex Lamb and David Alex. 1987. IDL: Sharing Intermediate Representations. *ACM Transactions on Programming Languages and Systems* 9, 3 (jul 1987), 297–318. https://doi.org/10.1145/24039.24040

[62] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. 49–63.

[63] Teng Li, Vikram K Narayana, Esam El-Araby, and Tarek El-Ghazawi. 2011. GPU Resource Sharing and Virtualization on High Performance Computing Systems. In *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 733–742.

[64] Tyng-Yeu Liang and Yu-Wei Chang. 2011. GridCuda: A Grid-Enabled CUDA Programming Toolkit. In *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*. IEEE, 141–146.

[65] Mikael Lindberg. 2007. *A Survey of Reservation-based Scheduling*. Citeseer.

[66] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture.. In *USENIX winter*, Vol. 46.

[67] Violeta Medina and Juan Manuel García. 2014. A Survey of Migration Mechanisms of Virtual Machines. *ACM Computing Surveys (CSUR)* 46, 3 (2014), 30.

[68] Microsoft. [n.d.]. MIDL Compiler. https://docs.microsoft.com/en-us/windows/win32/com/midl-compiler. Accessed: 2019-08.

[69] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. 2006. Tartan: Evaluating Spatial Computation for Whole Program Execution. *SIGOPS Oper. Syst. Rev.* 40, 5 (Oct. 2006), 163–174. https://doi.org/10.1145/1168917.1168878

[70] 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.

[71] Object Management Group. 2006. CORBA Component Model. https://www.omg.org/spec/CCM/4.0/PDF. Accessed: 2019-04.

[72] 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.

[73] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. 2018. MDev-NVMe: A NVMe Storage Virtualization Solution with Mediated Pass-Through. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*. 665–676.

[74] K. Dang Pham, A. K. Jain, J. Cui, S. A. Fahmy, and D. L. Maskell. 2013. Microkernel Hypervisor for a Hybrid ARM-FPGA Platform. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. 219–226. https://doi.org/10.1109/ASAP.2013.6567578

[75] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. 2011. The TAO of Parallelism in Algorithms. In *ACM Sigplan Notices*, Vol. 46. ACM, 12–25.

[76] Sébastien Pinneterre, Spyros Chiotakis, Michele Paolino, and Daniel Raho. 2018. 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)*. IEEE, 1–5.

[77] Christian Plessl and Marco Platzner. 2005. Zippy-A Coarse-grained Reconfigurable Array with Support for Hardware Virtualization. In *null*. IEEE, 213–218.

[78] Gerald J Popek and Robert P Goldberg. 1974. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM* 17, 7 (1974), 412–421.

[79] Carlos Reaño, Antonio J Peña, Federico Silla, José Duato, Rafael Mayo, and Enrique S Quintana-Ortí. 2012. CU2rCU: Towards the Complete rCUDA Remote GPU Virtualization and Sharing Solution. In *2012 19th International Conference on High Performance Computing*. IEEE, 1–10.

[80] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 233–248.

[81] Rusty Russell. 2008. virtio: Towards a De-facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

[82] R Sandberg, D Golgberg, S Kleiman, D Walsh, and B Lyon. 1988. Design and Implementation of the Sun Network File System. In *Innovations in Internetworking*, C Partridge (Ed.). Artech House, Inc., Norwood, MA, USA, Chapter Design and, 379–390. http://dl.acm.org/citation.cfm?id=59309.59338

[83] Eric Schkufza, Michael Wei, and Christopher J. Rossbach. 2019. Just-In-Time Compilation for Verilog: A New Technique for Improving the FPGA Programming Experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. 271–286. https://doi.org/10.1145/3297858.3304010

[84] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. 2012. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Trans. Comput.* 61, 6 (2012), 804–816.

[85] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 485–498. https://doi.org/10.1145/2451116.2451169

[86] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[87] Jike Song, Zhiyuan Lv, and Kevin Tian. 2014. KVMGT: a Full GPU Virtualization Solution. In *KVM Forum*, Vol. 2014.

[88] Greg Stitt and James Coole. 2011. Intermediate Fabrics: Virtual Architectures for Near-instant FPGA Compilation. *IEEE Embedded Systems Letters* 3, 3 (2011), 81–84.

[89] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in science & engineering* 12, 3 (2010), 66–73.

[90] 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.

[91] Michael M Swift, Brian N Bershad, and Henry M Levy. 2003. Improving the Reliability of Commodity Operating Systems. In *ACM SIGOPS operating systems review*, Vol. 37. ACM, 207–222.

[92] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[93] The gRPC Authors. [n.d.]. gRPC. https://grpc.io. Accessed: 2019-04.

[94] Kun Tian, Yaozu Dong, and David Cowperthwaite. 2014. A Full GPU Virtualization Solution with Mediated Pass-Through. In *2014 USENIX Annual Technical Conference (USENIX ATC'14)*. 121–132.

[95] Dimitrios Vasilas, Stefanos Gerangelos, and Nectarios Koziris. 2016. VGVM: Efficient GPU capabilities in virtual machines. In *International Conference on High Performance Computing & Simulation, HPCS 2016, Innsbruck, Austria, July 18-22, 2016*. 637–644. https://doi.org/10.1109/HPCSim.2016.7568395

[96] 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.

[97] 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*. Society for Computer Simulation International, 2.

[98] Carl Waldspurger and Mendel Rosenblum. 2012. I/O Virtualization. *Commun. ACM* 55, 1 (Jan. 2012), 66–73. https://doi.org/10.1145/2063176.2063194

[99] 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.

[100] 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.

[101] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. 2017. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.

[102] Lei Xia, Jack Lange, Peter Dinda, and Chang Bae. 2009. Investigating Virtual Passthrough I/O on Commodity Devices. *ACM SIGOPS Operating Systems Review* 43, 3 (2009), 83–94.

[103] Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. 2012. Transparent Accelerator Migration in a Virtualized GPU Environment. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 124–131.

[104] Zhonghua Yang and Keith Duddy. 1996. CORBA: A Platform for Distributed Object Computing. *SIGOPS Oper. Syst. Rev.* 30, 2 (April 1996), 4–31. https://doi.org/10.1145/232302.232303

[105] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G Rogers. 2017. Pagoda: Fine-grained GPU Resource Virtualization for Narrow Tasks. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 221–234.

[106] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J Rossbach. 2019. Automatic Virtualization of Accelerators. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, 58–65.

[107] Hangchen Yu and Christopher J Rossbach. 2017. Full Virtualization for GPUs Reconsidered. (2017).

[108] 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.

[109] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. 2018. G-NET: Effective GPU Sharing in NFV Systems. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association.

A Artifact

This appendix describes the experimental workflow, artifacts, and results from this paper. The evaluation focuses on *automatically* generated accelerator virtualization stacks for accessible hardware on a local machine, and provides configurations and execution steps to reproduce the expected productivity, end-to-end performance, and fairness results.

- **Compilation:** GCC 5.5+, LLVM 7.0 (customized), Python 3.6+, Bazel 0.26.1
- **Platform:** Ubuntu 18.04, Linux kernel 4.14 (customized)
- **Virtual machine:** QEMU 3.1 (customized), guest kernel 4.10+
- **Data set:** Table 5 in paper
- **Run-time environment:** `AVA_CHANNEL=SHM` `AVA_WPOOL=TRUE` `AVA_ROOT=/path/to/ava/source` (A.1, A.3) `DATA_DIR=/path/to/rodinia/data` (A.1)
- **Hardware:** Intel Xeon CPU E5-2643, NVIDIA GTX 1080 GPU, Intel Movidius stick v1
- **Execution:** start API server in the host and run benchmarking scripts in the guest VM
- **Output:** all benchmarks output the elapsed execution time
- **Experiments:** parameters are set and tunable in benchmarking scripts
- **Selected frameworks:** OpenCL 1.2, CUDA 10.0 (driver and runtime), NCSDK 2.10, TensorFlow 1.14
- **Selected accelerators:** NVIDIA GTX 1080 GPU, Intel Movidius stick v1
- **How much disk space required (approximately)?:** 40 GB excluding VM images
- **How much time is needed to prepare workflow (approximately)?:** one day
- **How much time is needed to complete experiments (approximately)?:** one day
- **Publicly available?:** Yes
- **Code licenses:** BSD 2-clause
- **Archive:** DOI:10.5281/zenodo.3596935

A.1 Description

The source and benchmarks are host on GitHub utcs-scea/ava and utcs-scea/ava-benchmarks, and archived at Zenodo/3596935.

Hardware dependencies. This AE requires NVIDIA GTX 1080 GPU with CUDA 10.0, and Intel Movidius stick v1 with NCSDK 2.10.

Data sets. Please refer to benchmark directories for data set information. Most data sets are contained in those directories, except Rodinia 3.1 data set which is available at Virginia LAVA lab. To prepare the Rodinia data set, download and unpack `rodinia_3.1.tar.bz2` to `$DATA_DIR`. To prepare the NCSDK input graph, NCSDK needs to be installed in host. We generate the input (as described in README) in host and copy it to guest.

A.2 Install accelerator drivers and runtime

A.2.1 OpenCL 1.2 and CUDA 10.0 for NVIDIA GTX 1080

Install cuda_10.0.130_410.48_linux and verify the installation by `clinfo` and `nvidia-smi`. CUDA SDK also needs to be installed in guest to compile CUDA benchmarks, but NVIDIA driver is not required.

A.2.2 NCSDK 2.10 for Intel Movidius stick

Follow the README of NCSDK benchmarks to install NCSDK v2.

A.2.3 TensorFlow 1.14 and cuDNN 7.5

Download and install cuDNN 7.5 (or 7.6.4) at NVIDIA developer zone. The guest VM requires a customized TensorFlow from (yuhc/tensorflow-cudart-dynam). The build instructions are in the repository's BUILD.md.

A.3 Installation

Clone utcs-scea/ava then follow its README.md to clone associated repositories, install software dependencies, setup environment variables, and build customized kernel, QEMU, and LLVM. Before compiling the source, turn off the debug mode completely by applying `#undef DEBUG` at `$AVA_ROOT/include/debug.h:9`.

Generated stack and API server. Generate and build the virtualization stack by following the instructions in the README. The optimized specifications such as `opencl.async.nw.c` and `cuda.async.nw.c` can be used in the same way.

API server manager. Simply run `make` in `$AVA_ROOT/worker`.

A.4 Experimental workflow

A.4.1 Line count

Example specifications are put in `$AVA_ROOT/cava/samples`. The length of specification (LoS) is growing over the time. LoS and number of virtualized APIs (NoA) are counted by:

```
$ wc -l opencl.nw.c
$ ctags -R -f - cuda.nw.c | grep -w f | wc -l
```

A.4.2 Start API server

Follow AvA's README to start API server. This step needs to be done for testing every new API framework (NCSDK, CUDA, etc.). API servers and manager can be terminated by `Ctrl+C` and restarted at any time. For fairness experiment, start manager by `sudo -E ./manager --policy`. This will install an integrated fair scheduler for device time.

A.4.3 Start guest VM

There is an example script to start VM in `$AVA_ROOT/vm`. In `$AVA_ROOT/vm/scripts/environment`, `IMAGE_FILE` is the path to the VM's image, and `DIR_QEMU` is the path to QEMU. The VM boot script is `$AVA_ROOT/vm/run.sh`, where the QEMU parameter must contain:

```
## In $AVA_ROOT/vm/run.sh
-enable-kvm -machine accel=kvm -cpu host,kvm=on \
-device vhost-vsock-pci,guest-cid=5 \
-device ava-vdev
```

When creating multiple VMs for fairness experiment, each VM must be assigned a different guest CID. The end-to-end performance experiment needs one VM, and the fairness experiment needs two.

A.4.4 Setup guest VM

The `guestdrv` source and compiled `guestlib` need to be copied to VM. Furthermore, the benchmark must load `guestlib` instead of original framework library at runtime. It is necessary to install essential tool chains to compile benchmarks in guest, e.g. CUDA (`nvcc`).

Guest directory structure. Set `AVA_ROOT` in guest. To simplify the following process, clone benchmarks to `$AVA_ROOT/benchmark`, and create `$AVA_ROOT/cava` (or repeat A.1 in guest).

Guestdrv and guestlib. Run `$AVA_ROOT/vm/setup.sh` to setup guestdrv and guestlib in the two VMs.

A.4.5 Execute benchmarks

Make sure `AVA_CHANNEL` (=`SHM`) and `AVA_WPOOL` (=`TRUE`) are consistent between guest and host. Also set `AVA_ROOT` (=`/home/hyu/ava` in the demo VM) and `DATA_DIR` (to Rodinia data set).

End-to-end performance experiments.

OpenCL. Every Rodinia OpenCL benchmark directory contains a `run` script to execute the benchmark. To run the same benchmark in host, comment `util/make.mk:7-8` before compiling the benchmark. To run all benchmarks, use `scripts/test_cl.sh`, and parse the result with `scripts/parse.sh`.

NCSDK. The benchmarking scripts are `run_movidius.sh` and `test_movidius.sh`.

CUDA driver. The benchmarks[1] are under `$AVA_ROOT/benchmark/rodinia/cuda`. The instructions are same as those for OpenCL benchmarks.

CUDA runtime. The benchmarks are under `$AVA_ROOT/benchmark/rodinia/cudart`. Link guestlib using script `link_lib.sh` which creates symbolic links to `libguestlib.so`. To run a benchmark, use `LD_LIBRARY_PATH=../util ./run` in guest and `./run` in host. To run all benchmarks, use `scripts/test_cudart.sh`.

TensorFlow. The benchmarks are under `$AVA_ROOT/benchmark/tensorflow/python`, and its README includes steps to run the benchmarks.

Fairness experiments

Recompile kernel. Define `KVM_MEASURE_POLICY` in the header `$AVA_ROOT/include/devconf.h:21`, Then recompile and reinstall kernel.

Setup environment. Generate `guestlib` and API server with specification `samples/opencl.sched.c`, which is a simplified OpenCL specification but with resource (device time) report to hypervisor.

Prepare two VMs. Spawn two VMs with different guest CID, and setup guest driver and library by `./setup.sh cl_nw`.

Get measurement data. In `/sys/kernel/debug/tracing`, set current tracer to `nop`, enable tracing (`echo 1 > tracing_on`), and read data from `trace_pipe`.

Execute benchmarks in parallel. In `$AVA_ROOT/benchmark/rodinia/micro/rate-limit`,

```
$ ./rate-limit.sh gaussian gaussian
```

[1] All benchmarks were ported from original Rodinia to use CUDA driver API. `Srad2_v2` faults occasionally even when running in host.

The two VMs' host names are specified in in `pssh-hosts` and `run.sh`. The script runs `gaussian` and `gaussian` in the two VMs correspondingly in parallel. The guest `AVA_ROOT` and `DATA_DIR` must be set in `rate-limit.sh` to access the appropriate directories. The two benchmarks can be picked from the set {gaussian, heartwall, hotspot, hotspot3D, lavaMD, nn, nw, pathfinder, streamcluster}.

A.5 Evaluation and expected result

A.5.1 Productivity

The LoS and NoA counted in A.4.1 should be close to the numbers shown in Table 1. {LoS/NoA} indicates the average lines of specification required to virtualize one API.

A.5.2 End-to-end performance

OpenCL, NCSDK, CUDA driver. The virtualization overhead of these three APIs should be close to the result in Figure 8.

CUDA runtime and TensorFlow. The overhead of CUDA runtime API and optimized TensorFlow API virtualizations should be close to the result in Figure 9. For unoptimized version, loading TensorFlow library takes ≈ 50 s. This time is reduced to be less than 3 s with the optimized version.

A.5.3 Fairness

At running benchmarks on two VMs simultaneously, the device time usages of VMs are printed via `trace_pipe`.

```
## Running gaussian and heartwall
# cat trace_pipe
device_time_measure_timer_callback: [intvl=500 ms]
vm#1 consumed device time = 256984 us
device_time_measure_timer_callback: [intvl=500 ms]
vm#2 consumed device time = 212812 us
```

The device time usage is measured based on the reports from API server. The benchmark's kernel execution time that crosses the measurement window will be counted into the next interval. But this bias is suppressed when the window increases, e.g., to 1 s.

The unfairness is calculated by $|t_1 - t_2| \,/\, (t_1 + t_2)$, where t_i is the device time used by VM_i in the time window. The distribution of unfairness values should match Figure 13.

A.6 Notes

We select representative accelerators and functionalities to be evaluated in this AE. Other supported features require specific hardware (e.g., PCIe devices, high-speed NIC, and dual machines), platforms (e.g., AWS F1 and AmorphOS for FPGA, Google Cloud for TPU), and very different (some are exclusive) configurations and environments. We leave those specifications and evaluation instructions in `$AVA_ROOT/cava/samples` and `$AVA_ROOT/benchmark` for further reference.