Crane: Fast and Migratable GPU Passthrough for OpenCL Applications

James Gleeson, Daniel Kats, Charlie Mei, Eyal de Lara
University of Toronto
Toronto, Canada
{jgleeson,dbkats,cmei,delara}@cs.toronto.edu

ABSTRACT

General purpose GPU (GPGPU) computing in virtualized environments leverages PCI passthrough to achieve GPU performance comparable to bare-metal execution. However, GPU passthrough prevents service administrators from performing virtual machine migration between physical hosts.

Crane is a new technique for virtualizing OpenCL-based GPGPU computing that achieves within 5.25% of passthrough GPU performance while supporting VM migration. Crane interposes a virtualization-aware OpenCL library that makes it possible to reclaim and subsequently reassign physical GPUs to a VM without terminating the guest or its applications. Crane also enables continued GPU operation while the VM is undergoing live migration by transparently switching between GPU passthrough operation and API remoting.

CCS Concepts

• **Computing methodologies** → **Graphics processors;** • **Computer systems organization** → **Heterogeneous (hybrid) systems;** • **Software and its engineering** → **Virtual machines;**

Keywords

virtualization, live migration, passthrough, GPU, GPGPU, OpenCL

1. INTRODUCTION

General purpose GPU (GPGPU) is being used to accelerate parallel workloads as varied as training deep neural networks for machine learning [2], encoding high-definition video for streaming [42], and molecular dynamics [20]. GPGPU programming is primarily done according to one of two APIs: CUDA for NVIDIA's GPUs, and OpenCL, which is implemented on many different GPUs from NVIDIA and other vendors. Major cloud "infrastructure as a service" (IaaS) providers now have virtualization offerings with dedicated GPUs that customers can leverage in their guest virtual machines (VMs). For example, Amazon Web Services allows VMs to take advantage of NVIDIA's high-end GPUs in EC2 [6]. Microsoft and Google recently introduced similar offerings on Azure and GCP platforms respectively [34, 21].

Virtualized GPGPU computing leverages modern IOMMU features to give the VM direct use of the GPU through DMA and interrupt remapping – a technique which is usually called PCI passthrough [17, 50]. By running the GPU driver in the context of the guest operating system, PCI passthrough bypasses the hypervisor and achieves performance that is comparable to bare-metal OS execution [47].

Unfortunately, GPU passthrough is incompatible with VM migration for two key reasons: (1) the hypervisor cannot *safely interrupt guest GPU access* from the guest GPU drivers and GPGPU applications, and (2) the hypervisor cannot *extract and reinject GPU state*. Without the ability of the hypervisor to interrupt the guest OS's access to the GPU, it is impossible for the hypervisor to reclaim the GPU from the guest OS without the guest OS crashing; we confirm this behaviour in the Xen hypervisor (Section 2.1). Without the ability of the hypervisor to extract GPU hardware state prior to migration, any data stored by guest GPGPU applications in GPU device memory are lost once the guest moves to a new machine.

We present Crane, a new approach for achieving passthrough GPU performance without sacrificing VM migration for virtualized OpenCL GPGPU applications. Crane allows both stop-and-copy and live migration. Crane does not require modification to applications, the hypervisor, and importantly, guest OS drivers which are often proprietary. Crane implements virtualization using only the portable OpenCL API, allowing Crane to achieve *vendor independence:* the ability to migrate any current and future GPU model that supports the OpenCL API without vendor support.

Crane operationally consists of three main components: a transparent interposition library, a guest migration daemon, and a host migration daemon. When the guest OS is not migrating, the OpenCL applications run at passthrough speeds with OpenCL calls sent directly to the GPU. While running, the interposition library interposes on OpenCL calls by tracking state needed to recreate OpenCL objects. When an administrator intends to migrate, Crane signals their intent to the guest migration daemon, which coordi-

SYSTOR '17 Haifa, Israel

 ISBN 978-1-4503-5035-8/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3078468.3078478

nates the pre-migration stage. During pre-migration, the guest migration daemon coordinates both *extracting GPU state* to DRAM and *safely interrupting guest GPU access*. Crane implements a universal GPU device memory extraction mechanism built using portable OpenCL APIs for saving GPU state to DRAM. Guest GPU access is safely paused by releasing OpenCL objects, unloading OpenCL shared library handles, and removing guest GPU drivers. With the GPU unused, the host migration daemon safely reclaims the GPU and commences VM migration. Crane enables live migration [13] by temporarily switching from passthrough GPU execution to API remoting, which forwards OpenCL commands over RPC to a stationary proxy domain on the source host. The proxy domain is pre-provisioned with a dedicated passthrough GPU to minimize the time to transition to API forwarding. Once the guest migrates, Crane leverages support in modern OSes and hypervisors for hotplugging PCI devices to re-attach a passthrough GPU [17, 50], and executes the post-migration stage. During post-migration, Crane coordinates resuming execution using the passthrough GPU by *reinjecting GPU device memory* using its universal mechanism, and recreating OpenCL objects from tracked API state.

This paper makes the following contributions:

- An approach to GPU virtualization that simultaneously achieves — **Passthrough performance:** within 5.25% of bare-metal passthrough GPU performance, **Vendor independence:** Crane uses vendor standardized OpenCL APIs to implement virtualization, allowing it to work for current and future GPU models, **VM migration:** Crane supports both stop-and-copy and live migration modes.
- A method for *extraction/reinjection of guest GPU state* and *safe interruption of guest GPU access* without terminating the guest or its OpenCL applications. The approach makes use of standard OpenCL APIs to maintain transparency thereby avoiding application, driver, and hypervisor modifications.
- A method for transparently extracting and injecting OpenCL state, enabling OpenCL applications to continue to execute (albeit at lower performance) while the VM is undergoing live migration via API remoting.

The rest of the paper is structured as follows. Section 2 provides a quick overview of OpenCL and introduces GPU passthrough and how it prevents VM migration. Sections 3 and 4 describe the design and implementation of Crane. Section 5 discusses our solution for dynamically offloading close-source GPU drivers. Section 6 presents our experimental evaluation. Section 7 discusses related work. Section 8 discusses future work such as extending Crane to CUDA. Finally, Section 9 concludes the paper.

2. BACKGROUND

We begin by explaining the key reasons why GPU passthrough is incompatible with VM migration. Next, we explain how OpenCL is used as a GPGPU programming language. This leads us to the design of Crane, which addresses the barriers to using VM migration with OpenCL applications that use GPU passthrough.

2.1 GPU passthrough

Passthrough is a technique that allows guest OS GPU drivers direct use of GPUs as if the guest OS was installed on the system as a bare-metal OS. Passthrough takes advantage of the input-output memory management unit (IOMMU) found on modern chipsets, which allows guests unmediated access to GPU device memory, thereby bypassing any hypervisor involvement. As a result, the guest VM attains GPU performance comparable to a bare-metal OS [54].

Unfortunately, using passthrough sacrifices the ability to migrate VMs. To prove that the hypervisor cannot *safely interrupt GPU access*, we conducted a small experiment with the Xen hypervisor. Modern hypervisors are able to dynamically detach and reattach PCI devices in passthrough without requiring a restart of the guest VM [49, 30]. Our experiment consists of running an OpenCL application in the guest OS, then issuing a detach of its GPU card while the application is running. After the card detaches, the entire guest OS crashes (for experimental setup see Section 6.1). Hence, detaching the card in the hypervisor is equivalent to ripping the physical GPU card out of the PCI-express slot of a bare-metal OS installation.

Even if passthough GPU access could be interrupted by the hypervisor, the hypervisor has no method of *extracting* state from the GPU and *reinjecting* it into the GPU on the new host. Proprietary GPU drivers provide no standard API for GPU state extraction/reinjection.

A simple approach is to terminate the OpenCL applications before migrating then restart them on the new host; however, sheer economic costs [5] in lost computational resources precludes such an approach in industry for batch processing jobs, which can takes many hours to execute [35].

2.2 OpenCL programming

The OpenCL API describes a cross-platform method of writing and executing massively data-parallel programs known as *kernels* on a *device* such as a GPU. The general flow of an OpenCL application is as follows:

1. **Device configuration:** Query and select an available GPU to use throughout the program.
2. **Resource allocation:** Kernels are compiled for the target device, and OpenCL memory buffers are used to write GPU memory as input.
3. **Program execution:** Kernels are enqueued for execution on the GPU, and results are read from GPU device memory back to DRAM.
4. **Cleanup:** Command queues are flushed, and any allocated OpenCL resource handles are freed.

The standardized OpenCL API empowers developers with application portability across all GPU vendors and models, including popular proprietary vendors like NVIDIA and AMD. As a result, the OpenCL shared library provides a universal API over all GPU models, and even other types of data-parallel hardware (e.g. FPGAs [4]).

3. DESIGN

Crane allows OpenCL applications to achieve GPU passthough performance without sacrificing VM migration. Crane enables VM migration by tracking OpenCL API state

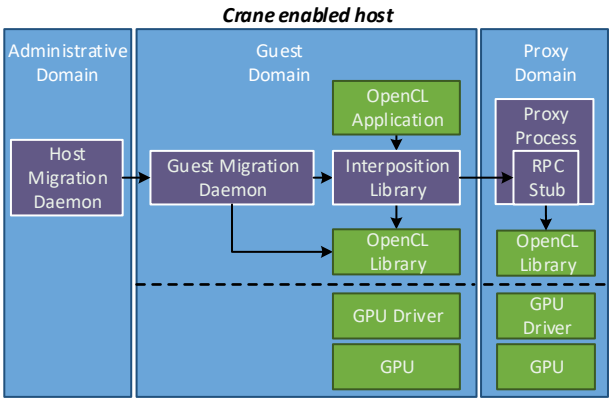


Figure 1: The Crane architecture consists of host/guest migration daemons and a transparent OpenCL interposition library; Crane makes no modification to OpenCL applications, guest OS drivers, or the hypervisor. The dotted line represents the user-kernel boundary.

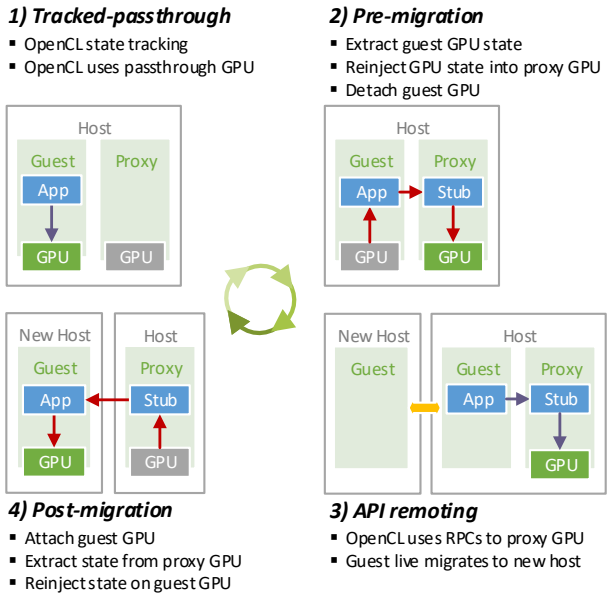


Figure 2: Crane uses a passthrough GPU in *tracked-passthrough* mode, and transitions to *API remoting* during live migration.

as the application runs, and providing a universal state extraction/reinjection API built using only portable OpenCL API calls. When the hypervisor signals its intention to migrate the VM, prior to detaching the card and migrating, Crane coordinates extracting GPU device memory into DRAM and unloading the guest GPU driver. After migration completes and a new GPU is attached, Crane uses its universal API to reinject GPU device memory and reconstruct OpenCL API state.

Crane's approach to VM migration has three important benefits:

1. **Vendor independence.** Crane enables migration of GPU guests for all GPU vendors and all GPU models that support the OpenCL API. Crane will support future GPU models as long as the OpenCL API remains the same.

2. **Passthrough performance.** OpenCL applications achieve passthrough GPU performance when a guest is not migrating.

3. **No application, hypervisor, or OS driver modifications.** Crane is a highly maintainable approach that does not require OpenCL application modification or recompilation.

The rest of this section describes how Crane handles stop-and-copy migration. We then discuss how this functionality can be extended to support *live migration*.

3.1 Stop-and-copy migration

The simplest form of migration is *stop-and-copy* migration [38]. In *stop-and-copy* migration, the guest VM is sent a suspend signal. The suspend signal triggers the guest OS to read in device state from attached PCI-devices, save the state to DRAM, and switch the DRAM into refresh mode. After device state has been preserved in DRAM, devices can be safely detached from the VM, and DRAM can be copied as-is to the new host. When the guest is resumed, the devices on the new host are powered back on, and the device state is loaded back into the new devices.

Unfortunately, there is no standard programming interface for (1) suspending GPU device state to DRAM, (2) safely detaching a passthrough device from a running guest, and (3) resuming GPU device state. Instead, during suspend, the GPU remains powered on, with its device memory being preserved on the device itself. Since there is no kernel API to force GPU device state to be preserved in DRAM during suspend, GPU PCI passthrough is incompatible with guest migration.

3.2 Enabling stop-and-copy migration

Crane supports stop-and-copy migration from a source host to a destination host by solving several challenges.

Extract GPU memory state: Before migration can begin, GPU device memory must be preserved in DRAM. The kernel provides no standard API for extracting GPU device memory to DRAM, and proprietary drivers are a barrier to creating one. However, the OpenCL API is already standardized across all platforms. Crane leverages OpenCL to create a universal GPU state extraction/reinjection API that works across GPU vendors and models. Crane coordinates GPU state extraction across all guest OpenCL applications prior to migration.

Track additional OpenCL API state: Unlike GPU device memory, not all OpenCL state can be extracted using existing OpenCL API calls. Crane must track function arguments used to create OpenCL objects, while retaining GPU passthrough performance.

Source host reclaims the GPU from the guest: To ensure the guest and its OpenCL applications do not crash when the hypervisor reclaims the GPU, Crane must first coordinate unloading the OpenCL shared libraries and the guest GPU driver.

Perform migration: With the GPU state preserved in DRAM, existing DRAM-based hypervisor mechanisms for stop-and-copy migration can be used.

Destination host hot plugs the new GPU into the guest: After migrating to the new host, the new GPU can be "hot plugged" into the guest, and kernel GPU drivers reloaded.

Reinject GPU state: Crane coordinates recreating OpenCL API state and reinjecting GPU device memory into the new GPU.

3.3 Live migration

The primary disadvantage of *stop-and-copy* migration is that while sending guest memory over the network, the entire guest OS and its applications are suspended; in our setup, a guest OS with 2 GB of DRAM is forced to wait 40 seconds before resuming. It is essential that OpenCL applications that have a user-facing component (e.g. a video encoder/decoder) continue to run with minimal stoppage time during live migration, albeit at reduced performance.

To overcome VM stoppage times during migration, modern hypervisors perform *live migration* where the guest OS is stopped for only a brief time on the order of milliseconds [13]. Live migration starts with several *pre-copy* phases, where the guest OS continues to run uninterrupted while its pages are asynchronously copied to the *destination* host. Pages modified during the pre-copy phase are marked as dirty by the hypervisor. A much shorter *stop-and-copy* phase delivers only the remaining dirty pages.

Unfortunately, the shadow page table mechanism used for tracking dirtied guest OS pages will not work for guest OS pages dirtied by DMA operations made by the GPU, since GPU writes do not raise CPU exceptions needed to track DMA writes. PCI express Address Translation Services (ATS) [1, 7, 26] extensions needed to perform hypervisor mediated DMA page faults and unmediated dirty page tracking exist, but none of the high-performance GPU models investigated in this paper (Section 6.1) support this extension. Hence, we require an alternative mechanism for enabling live migration.

3.4 Enabling live migration

In order to support live migration, Crane must allow applications to make OpenCL API calls during the pre-copy phase. However, a GPU cannot be assigned to a guest OS during live migration, since the stop-and-copy phase will fail to capture GPU memory state. To support live migration, Crane allocates a permanent passthrough GPU to a separate "proxy domain" running on the source host. The application's OpenCL calls are forwarded to an RPC stub in a *proxy process* running in the proxy domain (Figure 1).

In the *pre-migration* phase, Crane transitions from *tracked-passthrough* to *API remoting* mode. Crane achieves this by extracting guest GPU state to DRAM, and sending a copy over to the proxy process (Figure 2.2). The program continues executing OpenCL calls over RPC, albeit with degraded performance. With the guest GPU detached, the guest continues to execute as it is migrated over to a new host.

In the *post-migration* phase (Figure 2.4), Crane transitions back from *API remoting* to *tracked-passthrough*. Crane achieves this by extracting modified OpenCL state from the proxy and restoring it on the new passthrough GPU.

3.5 Limitations

Crane currently only supports migrating VMs between hosts with identical GPUs. The significance of this limitation in cloud data centers is debatable given their homogeneous nature [33]. We discuss extending Crane to support migration between heterogeneous GPU models in our future work (Section 8).

4. IMPLEMENTATION

In this section, we go into more detail about the architecture of Crane, which is shown in Figure 1. The interposition library runs in the OpenCL application and takes care of managing OpenCL state and API remoting. The guest migration daemon (running in the guest OS) and the host migration daemon (running in the host) coordinate with each other to extract OpenCL state into DRAM prior to migration, and ensure that the GPU card is removed safely after it is no longer in use by the guest OS.

4.1 Interposition library

The interposition library is a shim that wraps the OpenCL API, and implements the core functionality of Crane using only standard OpenCL APIs. Since the interposition library is a dynamically linked shared library, Crane avoids application modification and recompilation. The interposition library has 3 key roles: (1) tracking and extracting OpenCL state, (2) API remoting during live migration, and (3) reinjecting state into the new GPU after migration finishes.

4.1.1 Tracking and extracting OpenCL state

In order to ensure that all OpenCL and GPU device state can be recreated after migration to the new host, Crane must: (1) track all the arguments to OpenCL API calls that create new objects, and (2) read dirty GPU device memory from OpenCL memory buffer objects into DRAM before migration begins so it is not lost. To allow migration at any time, Crane must do this throughout the app's lifetime (i.e. before, after, and even during live migration).

For each OpenCL object type that a user can create (e.g. `cl_kernel`, `cl_context`) Crane maintains a shadow-object that records the original function arguments passed to the `clCreate*` function call. `clSetKernelArg` is wrapped to record its function arguments in the kernel shadow-object. Other OpenCL functions that manipulate OpenCL object state are simply propagated to the shadow-object. Since the OpenCL library state is encapsulated by APIs that modify OpenCL objects, there is *no need* to record other OpenCL functions calls and replay them later.

To read dirty GPU memory buffers into DRAM, Crane uses the standard OpenCL APIs `clEnqueueReadBuffer`. As an optimization, Crane tracks app writes to GPU device memory (i.e. `clEnqueueWriteBuffer`) and keeps a cached copy of them in DRAM; this reduces app stoppage time prior to migration by avoiding unnecessary GPU memory reads. While state-tracking can track simple buffer writes, it cannot track modifications to OpenCL buffers used as output arguments to an OpenCL kernel. Hence, any OpenCL buffer argument to a kernel that is writable is assumed by Crane to be "dirty", and must either be read from GPU device memory during pre-migration, or sent from the proxy to the OpenCL app during post-migration.

4.1.2 OpenCL object introspection API

OpenCL provides functions for introspecting its object handles. In particular, given an OpenCL object, a user can

query other OpenCL objects it depends on. The introspection API poses a challenge, since it can expose raw OpenCL object handles that Crane must wrap with its own opaque handle. Crane implements this behavior using a hash table mapping raw object handles to opaque handles. To ensure OpenCL applications that do not use introspection functions do not pay the overhead of hash table inserts, Crane inserts hash table entries lazily.

The astute reader may notice that state-tracking (Section 4.1.1) would be unnecessary if we could use the introspection API to query the function arguments used to create an OpenCL object. However, the OpenCL introspection functions are limited in usage, and cannot be used to obtain all function arguments, so state-tracking is required by Crane.

4.1.3 API remoting

The hypervisor prohibits a passthrough device from being attached to a guest during live migration. To ensure user-facing OpenCL apps continue running uninterrupted, the interposition library forwards OpenCL API calls over RPC to the stationary proxy domain residing on the same machine (Figure 1). To optimize the RPC implementation, Crane reduces RPC roundtrips by batching RPC calls. The only RPC calls that the interposition library cannot batch are OpenCL functions that introspect API state and require a return value, and functions that read/write to GPU device memory using an OpenCL app data pointer (e.g. `clEnqueueWriteBuffer`). The Interposition library is able to batch all other OpenCL API calls, including `clCreate*` calls that create new OpenCL objects. When Crane encounters a command that cannot be batched, it must flush any accumulated commands that precede it; Crane makes no attempt to infer whether non-batchable commands can be reordered earlier to further delay a flush.

4.1.4 Reinjecting state

Recreating OpenCL objects simply requires calling the corresponding `clCreate*` OpenCL API function with the original function arguments tracked during state-tracking (Section 4.1). To reinject GPU device memory, Crane uses `clEnqueueWriteBuffer` on each recreated memory buffer handle.

Some OpenCL object creation functions take only data arguments; hence, these objects do *not depend* on other OpenCL objects. However, some OpenCL object creation functions take other OpenCL objects as arguments; these objects *depend* on other OpenCL objects being recreated before them. Crane ensures OpenCL object interdependencies are respected by recreating OpenCL objects in order of their type. For example, `cl_context` objects are created first since they have no dependencies, followed by `cl_program` objects which depend on recreated `cl_context` objects.

4.2 Proxy domain

The proxy domain is a pre-provisioned VM with a permanently assigned passthrough GPU. By having the proxy pre-provisioned, we ensure minimal app stoppage times during pre-migration (Section 3.4). To avoid DRAM and GPU resource waste, multiple guest OSes share the same proxy domain. This provides equivalent security guarantees to previous approaches to GPU virtualization [15] that redirect driver calls to a shared "Driver VM" [19].

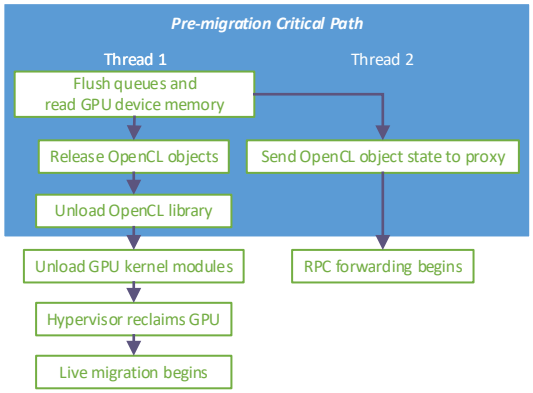


Figure 3: The pre-migration critical path is the transition from *tracked-passthrough* to *API remoting* mode. OpenCL API calls are blocked during this phase.

The proxy domain can support multiple VM migrations simultaneously. However, there are two caveats. First, the proxy domain's ability to multiplex the GPU depends entirely on the vendor-specific GPU driver implementation. Second, if the aggregate GPU device memory usage of multiple VMs and their OpenCL programs exceed the total capacity of the proxy GPU, then either the start of some VM migrations must be delayed, or all VMs can migrate simultaneously with some VMs migrating using stop-and-resume migration to avoid using proxy GPU device memory.

Before the proxy process can handle OpenCL RPC calls, it must first receive all OpenCL object state from the original OpenCL app. The communication medium for state transfer between the proxy and the guest is a TCP connection. Since the proxy process runs in a separate address space from the OpenCL app, the proxy process must translate the app's object handle addresses to its own object handles using a hash table.

4.3 Migration daemons

The host migration daemon and the guest migration daemon transparently intercept administrator migration commands and coordinate in phases referred to *pre-migration* and *post-migration*.

In pre-migration, before existing page-based migration mechanisms can begin, GPU device memory must be extracted into DRAM to ensure it is preserved during migration and available on the new host.

In post-migration, the host migration daemon and the guest migration daemon coordinate to reinject GPU memory state preserved in DRAM into the new GPU, and transition from forwarded OpenCL API calls back to guest local OpenCL calls.

Both pre-migration and the post-migration incur stoppage times where OpenCL API calls are blocked. Stoppage times before and after live migration are determined by the pre-migration critical path and the post-migration critical path respectively. To minimize interruption of service in user-facing workloads, minimizing stoppage times during these paths is essential.

4.3.1 Coordinating pre-migration

The guest migration daemon coordinates extracting GPU memory state to DRAM for each OpenCL app.

The guest migration daemon sends all OpenCL apps a `PREPARE_TO_MIGRATE` message and waits for all their respones, after which the guest migration daemon is assured that all GPU memory state has been moved to DRAM, and existing DRAM-based migration mechanisms can safely begin.

The host migration daemon coordinates safe removal of the GPU. The host migration daemon waits to hear back from the guest migration daemon, and once the guest migration daemon has responded, the hypervisor is assured that the GPU driver in the guest OS has been unloaded, and the GPU can be removed without the guest OS crashing.

The pre-migration critical path is shown in Figure 3, and the steps are as follows:

- **Flush queues and read GPU device memory:** Wait for any outstanding OpenCL kernels to finish by issuing `clFinish`, then read GPU device memory for any dirtied OpenCL memory buffers (Section 4.1.1).
- **Send OpenCL object state to proxy:** For each OpenCL object type, the function arguments and GPU memory state needed to rebuild those objects must be sent to and rebuilt on the proxy.
- **Release OpenCL objects:** Before the OpenCL library unloads, OpenCL objects must be released to ensure that any memory allocated by the library is not leaked.
- **Unload OpenCL library:** The vendor OpenCL library is unloaded, which decrements the reference count of the GPU driver (described in Section 5).

After the pre-migration phase critical path has finished, the interposition library unblocks any OpenCL API calls and forwards them over RPC to the proxy. To minimize stoppage time, Crane uses two threads to overlap sending OpenCL object state to the proxy with releasing objects and unloading the OpenCL library on the guest. The remaining steps in Figure 3 are performed off the pre-migration phase critical path in the background and *do not* contribute to OpenCL app stoppage time:

- **Unload GPU kernel modules:** Unloading the GPU driver prior to the hypervisor reclaiming the GPU card ensures the guest OS does not crash (Section 2.1).
- **Hypervisor reclaims GPU:** The card is safely detached from the guest, allowing it to be assigned elsewhere.

4.3.2 *Coordinating post-migration*

Once live migration completes, the host migration daemon attaches a GPU card to the guest and sends `PREPARE_TO_RESUME` over a socket to the guest migration daemon. With the card now attached, the guest migration daemon loads the GPU drivers, then forwards the message onto each OpenCL app. Once the OpenCL app receives the message, is is assured that the GPU card is inserted and the GPU driver is ready to use. Hence, the OpenCL app reloads the OpenCL library and recreates its OpenCL objects using the state it preserved in DRAM during pre-migration. GPU device memory is reinjected by issuing `clEnqueueWriteBuffer` on each recreated memory buffer object. After OpenCL object recreation, OpenCL apps immediately switch back over to native OpenCL calls without any coordination.

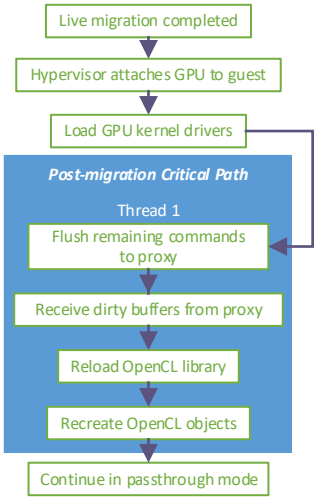


Figure 4: The post-migration critical path is the transition from *API remoting* to *tracked-passthrough* mode. OpenCL API calls are blocked during this phase.

The post-migration phase critical path is shown in Figure 4, and the steps are as follows:

- **Flush remaining commands to proxy:** Since OpenCL commands are batched, any commands that are still queued must be flushed to the proxy and completed.
- **Receive dirty buffers from proxy:** The proxy flushes its command queues and reads GPU device memory, then sends dirty memory buffers of completed kernels back to the application.
- **Reload OpenCL library:** The OpenCL library is reloaded, and function pointer hooks are reset.
- **Recreate OpenCL objects:** OpenCL objects are rebuilt by executing the OpenCL `clCreate*` calls for the corresponding object type (Section 4.1.4).

5. OPENCL LIBRARY UNLOADING

Crane requires idempotent loading and unloading of the OpenCL shared library (`libOpenCL.so`). The naive approach is to use `dlopen` and `dlclose`.

However, reloading only `libOpenCL.so` proved unsuccessful for our NVIDIA-based setup with the OpenCL library returning an error code immediately when attempting to use the OpenCL API. Upon closer inspection, we observed that calling `dlclose` fails to recursively unload shared library dependencies initially loaded by `libOpenCL.so`. From this, we suspected that an incomplete library reload resulted in the error code.

As a workaround to ensure `libOpenCL.so` and its dependencies are all reloaded, we explicitly record recursive shared library dependencies loaded by `libOpenCL.so`, so they can be unloaded manually. Crane creates a wrapper for `dlopen` that: (1) records the library handle of the loaded dependency, and (2) rewrites the Global Offset Table to call the

wrapper instead of `dlopen` so that recursive library dependencies are recorded.

Lastly, to ensure the GPU driver can be unloaded, we must close GPU device driver file descriptors left open by explicitly unloaded libraries. We achieve this on Linux by scanning for open GPU driver file descriptors in `/proc/$PID/fd`.

After performing our explicit unload procedure, we were able to perform an idempotent reload of the OpenCL library without any subsequent errors.

6. EVALUATION

There are three metrics of performance when migrating OpenCL applications: (1) *overhead from state-tracking* during tracked-passthrough mode that deviate from a pristine GPU passthrough environment, (2) *stoppage times* during pre-migration and post-migration, (3) *live migration performance* during API remoting for continued but reduced service.

Importantly, there are wildly different constraints on these three metrics for the two different types of OpenCL workloads.

For **batch processing workloads**, total job completion time is most important, and is impacted most by *overhead from state-tracking* when the VM is stationary. Small but continuous overhead for jobs that go on for hours [35] translate to hours in lost computing resources. However, subsecond stoppage during pre-migration/post-migration or reduced service during a 53 second live migration period are unimportant for jobs that could go on for hours.

For **user-facing workloads**, *stoppage times* are critical since they represent a delay in user experience. *Continued service* during live migration is essential, albeit at reduced performance that is within the real-time constraint of the user (e.g. frames encoded per-second for a video encoder). Overhead from state-tracking is negligible and will not violate the user's real-time constraints.

We structure the evaluation in order of the three metrics, and emphasize the importance of the metric based on whether it is a batch processing or user-facing workload.

6.1 Platform

Our benchmarking setup for Crane consisted of two identical physical hosts connected via a 1 Gbps Ethernet link. Each host contained two Intel Xeon E5-2609 2.4 GHz quad-core processors and 32 GB of RAM. Both hosts were outfitted with an NVIDIA GRID K1 card. For software, Xen 4.7 was used as the hypervisor, CentOS 6 with Linux kernel 3.16.6 for the dom0, and Ubuntu 16.04 LTS with Linux kernel 4.4.0 for the domU. The domU was configured with 2 GB of memory and ran NVIDIA driver version 361 supporting OpenCL 1.2.

To demonstrate Crane's ability to support different GPU models, we also ran experiments using a separate NVIDIA GTX480 card. However, we omit these results from our evaluation for clarity since they are identical.

6.2 OpenCL programs

6.2.1 Data-parallel microbenchmarks

We obtained the "Level One" programs from the Scalable HeterOgeneous Computing (SHOC) benchmark suite [16], which are designed to target common data-parallel tasks found in existing GPGPU applications, including both batch processing and user-facing workloads. We chose SHOC over other benchmarking suites [12] since it provides benchmark implementations for OpenCL, in addition to CUDA.

6.2.2 User-facing OpenCL application

x264: a video encoder. x264 [46] is a free software library used in the popular open-source video player VLC. x264 encodes video files from a lossless format into a lossy compressed format suitable for streaming video playback to a user. x264 allows users to make use of GPU acceleration by using an OpenCL video encoder. We configured the x264 program to use the OpenCL video encoder and to output to MKV format. The input video file is a 4.7GB, 360p video in the uncompressed YUV4MPEG format.

6.2.3 Batch processing OpenCL applications

FAHBench: scientific computing. FAHBench [41] is a benchmarking toolkit for GPU-based molecular dynamics calculations. FAHBench performs the same calculations as those used in Folding@Home [10], a crowd-sourced distributed computing platform for predicting protein folding, which may assist in researching diseases involving protein misfolding such as Alzheimer's and Parkinson's. We ran FAHBench using its default configuration, without any parameters. By default, FAHBench runs a standard simulation of the DHFR protein in an explicit solvent.

DeepCL: training a neural network. Batch processing workloads such as training convolutional neural networks can take hours to complete [35]. It is essential that these workloads are not terminated prematurely, even in the presence of routine maintenance tasks that require migrating a VM so a machine can be powered down. For an image recognition task, the depth of the trained network can lead to better prediction accuracy [40]. However, with increased depth comes increased memory usage and training times [29]. As a result, a variety of machine learning frameworks have begun to appear that make use of GPUs to accelerate the training of large scale neural networks [2, 27, 14]. We consider DeepCL [25] which is one such framework written on top of OpenCL. DeepCL runs a benchmark referred to in the codebase as `maddison-full`. The neural network contains 1 input layer, 12 convolutional layers, 1 fully connected layer, and 1 softmax layer.

6.3 Overhead from state-tracking

Figure 5 shows that in the common case, if an administrator is running a stationary guest OS with Crane enabled, they can expect to see performance comparable to a GPU passthrough environment.

For *data-parallel microbenchmarks*, the performance overhead of state-tracking is low with a mean of 5.25% relative to passthrough performance.

An excellent result is that both batch processing jobs *DeepCL* and *FAHBench* are within measurement error of a GPU passthrough environment, with a mean of -0.1%.

x264 does incur a noticeable state-tracking overhead of 12.5%. However, since x264 is a user-facing workload whose real-time constraint is frames encoded per second, this overhead is acceptable as can be seen in Figure 6 when comparing Passthrough and Crane rates of encoding when the application is not migrating. Upon further investigation, the root cause of this overhead is that `clSetKernelArg` takes

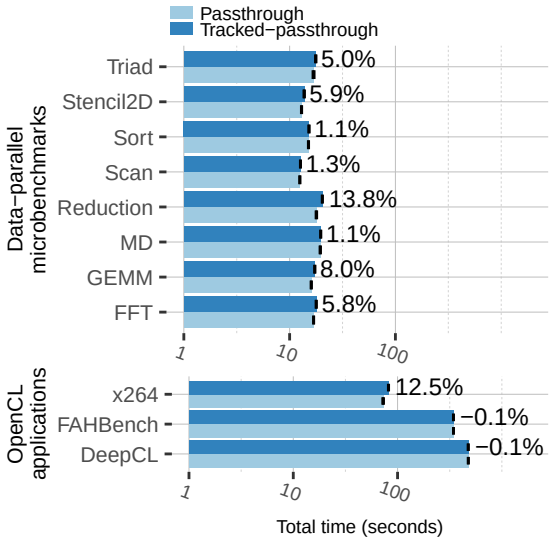


Figure 5: Overhead from state-tracking during GPU passthrough operation

both OpenCL object handles and non-OpenCL object handles as arguments, but does not distinguish type information in the call. This forces the state-tracking layer to perform hash table lookups to determine if the argument is a memory buffer.

6.4 Stoppage times

Stoppage times are incurred during pre-migration before live migration begins, and during post-migration after live migration ends. We first look at pre-migration and post-migration stoppage times and their individual bottlenecks for each application, then we summarize the importance of total stoppage times to the application. Unlike the other applications, stoppage times for DeepCL are bottlenecked by state transferred during pre-migration and post-migration. We provide a breakdown of total state transferred during pre-migration and post-migration for all applications.

6.4.1 *Pre-migration stoppage time*

Figure 7a shows stoppage time during the pre-migration phase when transitioning to API remoting on the proxy, which corresponds to the steps in Figure 3. During pre-migration, we first need to wait for outstanding kernels to finish executing, after which we can extract GPU device memory (Thread 1, a). Once this data is extracted to DRAM, it can be sent to the proxy (Thread 1, b). While objects are sent to the proxy, OpenCL objects on the guest can be released concurrently in a separate thread (Thread 2, a), after which the OpenCL library can be unloaded (Thread 2, b).

Pre-migration stoppage time in FAHBench and x264 is bottlenecked by unloading the OpenCL library (Figure 7a, (Thread 2, b)), and only moderately affected by waiting for outstanding kernels to finish (Thread 1, a). We investigated the kernel execution times of the three applications, and found that the median kernel execution times for x264 and FAHBench are less than 5 ms, and for DeepCL is 5 seconds. DeepCL has the longest kernel execution times with a 85-th percentile kernel execution time of 10 seconds, whereas the 99-th percentile of x264 and FAHBench are less than 10 ms. The longest kernel execution time for DeepCL is 62 seconds, so in the worst case Crane could incur a 62 second pre-migration stoppage for DeepCL. Pre-migration stoppage time in DeepCL is bottlenecked by large state transfers from the guest to the proxy.

6.4.2 *Post-migration stoppage time*

Figure 7b shows stoppage time during the post-migration phase when transitioning back to tracked-passthrough mode on the guest, which corresponds to the steps in Figure 4. During post-migration, the guest sends any remaining RPC calls to the proxy (Thread 1, a), after which it receives any dirty memory buffers from the proxy (Thread 1, b). Then, the guest reloads the OpenCL library using `dlopen` (Thread 1, c) and recreates the OpenCL objects on the guest (Thread 1, d).

Post-migration stoppage time in FAHBench and x264 is bottlenecked by time spent recreating OpenCL objects (Figure 7b, (Thread 1, d)). Again, DeepCL post-migration stoppage times are bottlenecked by large state transfer sizes (Thread 1, b).

6.4.3 *Total stoppage time*

Total stoppage time for pre-migration and post-migration corresponds to the height of the bar graphs in Figures 7a and Figure 7b respectively. The total pre-migration stoppage time corresponds to time spent waiting for the slowest of the two pre-migration threads to finish. The total post-migration stoppage time corresponds to the sum of all the post-migration steps, since they occur synchronously without any overlap.

In total, *x264* incurs 0.1 seconds of stoppage time during pre-migration. Similarly, x264 incurs 0.83 seconds of stoppage time during post-migration. This is an excellent result, since the user-facing workload will incur sub-second stoppage times when servicing encoded video frames.

For the batch processing workloads, stoppage times are less important, since they add a negligible fraction to the total job completion time. However, we include them to provide the reader with insight into the variability of stoppage times between different OpenCL applications.

In total, *FAHBench* incurs small stoppage times during pre-migration (0.073 seconds) and post-migration (0.52) seconds. These stoppage times are acceptable since they are a small fraction of total job time (342.94 seconds).

In total, *DeepCL* has the largest stoppage times, since it sends a large amount of state during pre-migration and post-migration. However, the stoppage times during pre-migration (1.57 seconds) and post-migration (4.90 seconds) are acceptable since they are a small fraction of total job time (475.90 seconds).

We were not able to measure stoppage times for *data-parallel algorithm* microbenchmarks since their runtime is less then the 53 second migration time itself.

6.4.4 *State transfer*

OpenCL object state is sent from guest to proxy during pre-migration, and from proxy to guest during post-migration. For DeepCL, state transfer time bottlenecks both pre-migration and post-migration stoppage times. In all three applications, state transfer is bottlenecked by GPU device memory (`cl_mem`).

Figure 8c shows the worst case maximum state transfer size that could be sent during either pre-migration or post-migration. This corresponds to the maximum GPU device memory allocated by the app over its lifetime, since the amount of GPU device memory transferred is a function of when the VM happens to be migrated by the administrator.

Figure 8a shows OpenCL state sent during pre-migration from the guest to the proxy for recreating OpenCL objects in the proxy process. State sent during pre-migration spans all different OpenCL object types, as recorded during state-tracking (Section 4.1.1). Data being sent includes function arguments needed to recreate the OpenCL objects (e.g. OpenCL source code strings for `cl_program`). The `cl_mem` memory buffer type includes GPU device memory.

Figure 8b shows OpenCL state sent during post-migration for any memory buffers dirtied by kernels that ran on the proxy process. The proxy only needs to send back dirty memory buffers, since all other OpenCL object state is already present on the guest since from performing state-tracking.

DeepCL stoppage times are bottlenecked by sending GPU device memory, with 256MB of data transferred during both pre-migration (Total, Figure 8a) and post-migration (`cl_mem`, Figure 8b). The upper bound on these state transfers is 1GB (`cl_mem`, Figure 8c).

x264 sends 16 MB during both pre-migration and post-migration. The 16 MB sent during pre-migration is not the bottleneck for x264; unloading the OpenCL library is. The 16 MB sent during post-migration is a small contribution to stoppage and is bottlenecked by recreating OpenCL objects.

FAHBench only sends 1 MB during pre-migration resulting in a small contribution to pre-migration stoppage time. For post-migration, FAHBench is the only workload that does not dirty any of its memory buffers during live migration resulting in zero bytes transferred during post-migration; it allocates 16MB of GPU memory later on in its execution after post-migration.

6.5 Live migration performance

Since live migration only lasts 53 seconds, it has little impact on long batch processing workloads. Hence, we limit our analysis to the *x264* user-facing workload.

Figure 6 shows the rate of encoding over the lifetime of an OpenCL application running with Crane. The administrator issues a live migration request in the hypervisor 8 seconds into the application running. After pre-migration stoppage, the OpenCL application transitions to API remoting mode. Crane makes use of batching RPC to minimize TCP roundtrips and achieves an average of 20.1 FPS (without batching it is 4.5 FPS; for clarity this line is omitted). While 20.1 FPS is less than the ideal 60 FPS used in today's video streaming, poor performance during live migration is not an inherent limitation in Crane's API remoting approach and can be optimized by using an inter-VM communication mechanism; we discuss two different approaches in Section 8.1.

Besides the sub-second pre-migration and post-migration stoppage times highlighted in Figure 6, there are other brief periods of reduced frame encoding rates. Following pre-migration stoppage, detaching the GPU PCI device from the hypervisor results in a 1.1 second pause in the VM. However, Crane allows unblocked OpenCL calls over API remoting during this PCI detach-induced pause. We consider any pause from PCI detach to be an artifact of the device that could change between models, hence it is not included in Crane's pre-migration critical path. At roughly 45 seconds into the experiment, the post-copy phase of live migration pauses the VM for 2.5 seconds. The post-copy phase of live migration is always present both with and without Crane, so it does not belong on Crane's post-migration critical path. GPU PCI device reassignment automatically triggers reloading GPU drivers in the VM, and Linux provides no reliable way of being notified of this completing. As a result, we artificially add 10 seconds of additional API remoting after the post-copy phase to avoid a race condition when switching from API remoting back to tracked-passthrough mode. If the guest OS provides a reliable way to block and wait for kernel drivers to load, this additional 10 seconds of API remoting can be removed.

7. RELATED WORK

Dowty & Sugerman [19] provide a taxonomy covering different approaches to GPU virtualization. *Front-end virtualization* runs the driver in the hypervisor and uses API remoting or device emulation for handling guest VM GPU access. *Back-end virtualization* runs the driver in the guest VM using passthrough for maximum performance but sacrifices migration. *Hybrid virtualization* dedicates a "Driver VM" and uses front-end techniques to provide access to the GPU. Crane fills a unique position in this taxonomy that uses back-end virtualization when the VM is stationary for maximum performance, and falls back to hybrid virtualization during live migration to minimize pauses in VM execution.

Front-end GPU virtualization: Many methods focus solely on virtualizing the GPU at the API level, such as forwarding API calls [23, 31, 44, 51, 52, 55]. Unlike Crane, they do not take advantage of new GPU passthrough features for performance. Like Crane, LoGV [22] acknowledges that GPU drivers cannot support migration and proposes saving GPU state to DRAM before migration. However, since LoGV modifies GPU drivers, it only works for 1 GPU model (NVIDIA GTX480), whereas Crane benefits from *vendor independence* (Section 3) by implementing all state extraction and GPU virtualization using vendor standardized OpenCL APIs that work across all GPU models supporting OpenCL.

Back-end GPU virtualization: AMD and NVIDIA provide GPUs tailored to virtualized environments that have a limited form of GPU sharing that configures the GPU to appear either as a single high-performance virtual GPU, or multiple lower performance virtual GPUs assigned via passthrough [8, 36]. Unlike Crane, this does not enable guest VM migration. GPUvm [43] virtualizes the GPU at the hypervisor level by using reverse engineered NVIDIA GPU drivers [32]. However, GPUvm [43] is limited only to GPU models that have been reverse engineered whereas Crane is vendor independent (Section 3). Intel provides an open-source driver (gVirt/Intel GVT) for providing GPU virtualization of its integrated graphics offerings [56, 45]. Since integrated graphics have a shared memory space, GVT is able to use standard shadow page-table mechanisms [9, 3] to trap-and-emulate page table writes needed to share the GPU between VMs, leading to performance degradation that authors refer to as the "Massive Update Issue" [18]. Crane does not suffer from the "Massive Update Issue" by design, since Crane tracks dirty GPU memory at the granularity

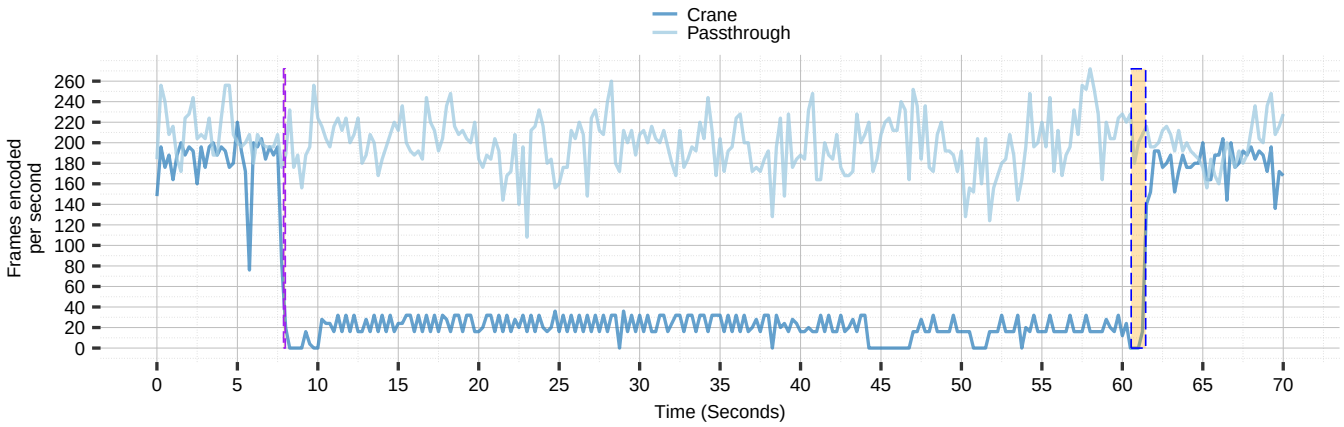
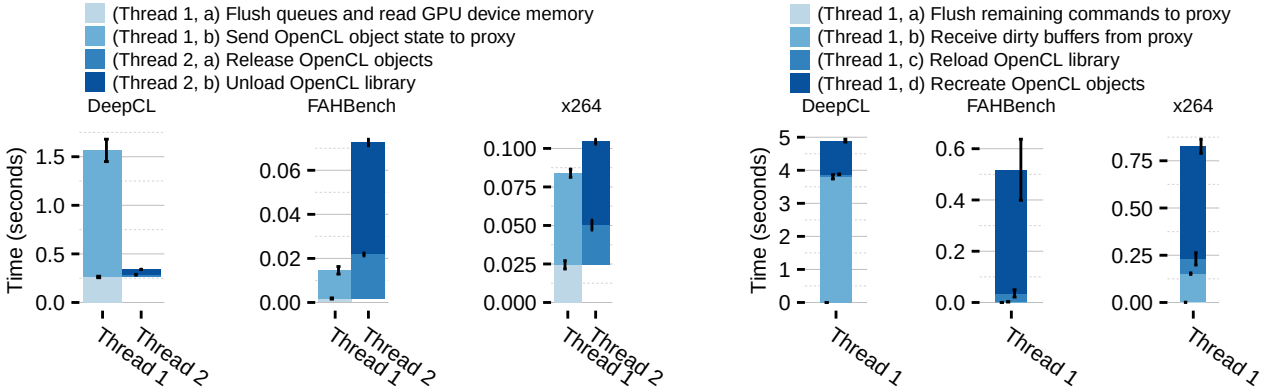


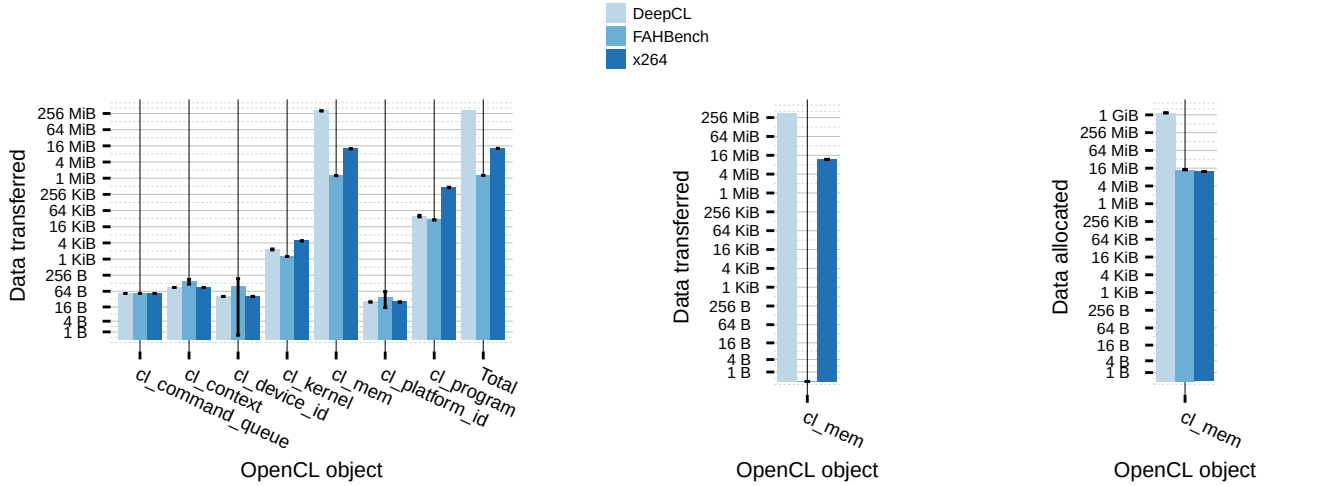
Figure 6: Encoding video frames using the GPU. The administrator issues a live migration command 8 seconds into running the application. Live migration occurs between the boxes and lasts for 53 seconds. The first box shows pre-migration stoppage time; the second box shows post-migration stoppage time.



(a) Pre-migration

(b) Post-migration

Figure 7: Application stoppage time. Pre-migration stoppage makes use of multiple threads to reduce stoppage time, but post-migration stoppage cannot (Sections 4.3.1 and 4.3.2).



(a) Pre-migration: OpenCL API state sent from guest to proxy

(b) Post-migration: dirty GPU device memory sent from proxy to guest

(c) Maximum GPU device memory allocated.

Figure 8: OpenCL API state breakdown. State transfer is bottlenecked by GPU device memory (`cl_mem`); the graph on the right provides an upper bound on GPU device memory that could be transferred during either pre-migration or post-migration.

of OpenCL memory buffers (Section 4.1.1). GVT does not yet support VM migration, whereas Crane is vendor independent and will work on any platform supporting OpenCL, including Intel's.

Address Translation Services: The ATS specification [1, 7, 26] includes a standard wire protocol for performing (1) dirty page tracking through page table entry dirty bit flipping without host processor coordination, and (2) page faults detected by the IOMMU and handled by the host processor. ATS has been shown to work for integrated graphics offerings that share a unified graphics and system memory [24]. However, neither of the high-performance GPU models used in this paper (Section 6.1) support ATS. Crane supports all GPU models both now and in the future, regardless of their support for ATS extensions.

Virtualizing passthrough NIC devices: Numerous studies have investigated enabling VM migration for passthrough NICs [57, 28, 37]. However, all these approaches virtualize at the driver-level, requiring modification of each driver that is virtualized [37], thereby sacrificing *vendor independence* (Section 3). To avoid guest driver modifications, SRVM [53] modifies the hypervisor to track dirty DMA pages during migration using VM introspection of the guest driver. However, tracking dirty pages with introspection requires polling guest memory using a dedicated CPU in order to keep up with the incoming packet rate. Crane avoids needing to track DMA buffers entirely by instead tracking OpenCL memory buffers and their dirty status (Section 4.1.1).

8. FUTURE WORK

8.1 Improving live migration performance

During live migration of the x264 video encoder in Section 6.5, we experience poor OpenCL performance in API remoting mode since we rely on a socket as a communication medium between the proxy OS and the guest OS. While we optimized the performance slightly by batching RPC calls, we can also examine the benefits of using shared memory as a communication medium between the proxy OS and the guest OS instead, similar to other approaches that make use of front-end virtualization techniques like API remoting [23, 39].

Several studies have investigated optimized forms of inter-VM communication that make use of shared memory. XenLoop is a paravirtualized approach that provides high performance by avoiding guest-to-host switches that cause TLB and cache flushes by forgoing page table remappings used in other approaches (e.g. netfront-netback) [48]. XenLoop works by intercepting packets from the source guest destined for guests on the same machine, and copies the packets into a preallocated shared memory FIFO buffer. Fido [11] aims to further reduce inter-VM communication costs by eliminating data copies, which it achieves by pre-mapping a read-only view of every other VM's entire physical memory at fixed virtual offsets. Pre-mapping eliminates page-mapping and data copying costs, but sacrifices isolation guarantees of the data sender by exposing its entire physical memory to the receiver. If all the software in proxy guest OS is trusted, mechanisms like Fido can be used in Crane; otherwise, it is safer to limit the scope of potential guest data leaks to GPU communication by using XenLoop.

8.2 Extending Crane to CUDA

The techniques used in Crane needed to support migration with fast passthrough GPU performance can be extended to support the CUDA framework. Both the CUDA and the OpenCL API treat library-specific objects as opaque handles, and do not allow manipulation of the underlying data except via the library APIs. An API's use of opaque handles ensures we can achieve API remoting and state extraction/reinjection without requiring application modifications, since we can interpose on the opaque handles when referencing recreated or remote API objects. vCUDA [39] takes advantage of opaque handles in CUDA to achieve API remoting without application modifications.

8.3 Migration between heterogeneous GPU models

In this paper, Crane built a universal mechanism for *extracting* GPU state, and *reinjecting* GPU state once the VM has migrated to a new host with a *new GPU*. Since, state extraction/reinjection is not tied to a specific GPU driver, and is performed entirely in user space in vendor independent portable OpenCL, the VM could instead reinject GPU state into a *new heterogeneous GPU*. However, heterogeneous GPU migration will not work with Crane as is. GPUs have different hardware limits such as GPU device memory and levels of available parallelism. Migrating to a weaker capacity GPU will cause the application to fail when it attempts to use resources beyond what are available. In future work, we will explore both application transparent and non-transparent mechanisms for dealing with GPU heterogeneity.

9. CONCLUSIONS

We have described Crane, a solution to GPU virtualization that provides both stop-and-copy and live VM migration while simultaneously achieving passthrough GPU performance. Since Crane implements GPU virtualization using only vendor standardized OpenCL APIs, it achieves two important benefits. First, Crane avoids application, hypervisor, and OS modifications. Second, Crane is vendor independent, thereby enabling virtualization of current and future GPU models that support the OpenCL API without any vendor support.

10. REFERENCES

[1] PCI express - address translation services revision 1.1. 2009.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[3] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGOPS Operating Systems Review*, 40(5):2–13, 2006.

[4] Altera. Implementing FPGA design with the OpenCL standard. *Whitepaper*, 2013.

[5] Amazon Web Services. EC2 Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[6] Amazon Web Services. Linux Accelerated Computing Instances. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using_cluster_computing.html.

[7] AMD. AMD I/O virtualization technology (IOMMU) specification. 2011.

[8] AMD Corporation. AMD FirePro S Series. https://www.amd.com/Documents/FirePro-S-Series-Datasheet.pdf.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[10] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[11] A. Burtsev, K. Srinivasan, P. Radhakrishnan, K. Voruganti, and G. R. Goodson. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *USENIX Annual technical conference*. San Diego, CA, 2009.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[13] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[14] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A MATLAB-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[15] C. I. Dalton, D. Plaquin, W. Weidner, D. Kuhlmann, B. Balacheff, and R. Brown. Trusted virtual platforms: a key enabler for converged client devices. *ACM SIGOPS Operating Systems Review*, 43(1):36–43, 2009.

[16] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[17] Debian Wiki. KVM VGA Passthrough. https://wiki.debian.org/VGAPassthrough.

[18] Y. Dong, M. Xue, X. Zheng, J. Wang, Z. Qi, and H. Guan. Boosting GPU virtualization performance with hybrid shadow page tables. In *USENIX Annual Technical Conference*, pages 517–528, 2015.

[19] M. Dowty and J. Sugerman. GPU virtualization on VMware's hosted I/O architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[20] P. Eastman, M. S. Friedrichs, J. D. Chodera, R. J. Radmer, C. M. Bruns, J. P. Ku, K. A. Beauchamp, T. J. Lane, L.-P. Wang, D. Shukla, T. Tye, M. Houston, T. Stich, C. Klein, M. R. Shirts, and V. S. Pande. OpenMM 4: A reusable, extensible, hardware independent library for high performance molecular simulation. *Journal of Chemical Theory and Computation*, 9(1):461–469, 2013. PMID: 23316124.

[21] Google Cloud Platform. Graphics Processing Units (GPU) | Google Cloud Platform. https://cloud.google.com/gpu/.

[22] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa. LoGV: Low-overhead GPGPU virtualization. In *High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 1721–1726, Nov 2013.

[23] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, pages 17–24, New York, NY, USA, 2009. ACM.

[24] Y.-J. Huang, H.-H. Wu, Y.-C. Chung, and W.-C. Hsu. Building a KVM-based hypervisor for a heterogeneous system architecture compliant system. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 3–15. ACM, 2016.

[25] Hugh Perkins. DeepCL: deep convolutional networks in OpenCL. http://deepcl.hughperkins.com/.

[26] Intel. Intel virtualization technology for directed I/O, revision 2.4. 2016.

[27] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[28] A. Kadav and M. M. Swift. Live migration of direct-access devices. *SIGOPS Oper. Syst. Rev.*, 43(3):95–104, July 2009.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[30] KVM. Hotadd PCI Devices. http://www.linux-kvm.org/page/Hotadd_pci_devices.

[31] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. VMM-independent Graphics Acceleration. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 33–43, New York, NY, USA, 2007. ACM.

[32] Linux open-source community. Nouveau Open-Source GPU Device Driver. http://nouveau.freedesktop.org/.

[33] Matt Kapko. How (and Why) Facebook Excels at Data Center Efficiency. http://www.cio.com/article/2854720/data-center/how-and-why-facebook-excels-at-data-center-efficiency.html.

[34] Microsoft Azure. N-Series GPU enabled Virtual Machines. https://azure.microsoft.com/en-us/pricing/

details/virtual-machines/series/#n-series.

[35] Netflix Inc. Distributed Neural Networks with GPUs in the AWS Cloud. http://techblog.netflix.com/2014/02/distributed-neural-networks-with-gpus.html.

[36] NVIDIA Corporation. Virtual GPU Technology - NVIDIA GRID. http://www.nvidia.ca/object/grid-technology.html.

[37] Z. Pan, Y. Dong, Y. Chen, L. Zhang, and Z. Zhang. Compsc: Live migration with pass-through devices. *ACM SIGPLAN Notices*, 47(7):109–120, 2012.

[38] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, Dec. 2002.

[39] L. Shi, H. Chen, J. Sun, and K. Li. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, 61(6):804–816, June 2012.

[40] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[41] Stanford University. FAHBench. https://fahbench.github.io.

[42] H. Su, M. Wen, N. Wu, J. Ren, and C. Zhang. Efficient parallel video processing techniques on GPU: From framework to implementation. *The Scientific World Journal*, 2014, 2014. Hindawi Publishing Corporation, 19 pages.

[43] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. GPUvm: Why not virtualizing GPUs at the hypervisor? In *USENIX Annual Technical Conference*, pages 109–120, 2014.

[44] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi. CheCL: Transparent checkpointing and process migration of OpenCL applications. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 864–876. IEEE, 2011.

[45] K. Tian, Y. Dong, and D. Cowperthwaite. A Full GPU Virtualization Solution with Mediated Pass-Through. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 121–132, Philadelphia, PA, 2014. USENIX Association.

[46] VideoLAN. x264, the best H.264/AVC encoder. http://www.videolan.org/developers/x264.html.

[47] J. P. Walters, A. J. Younge, D. I. Kang, K. T. Yao, M. Kang, S. P. Crago, and G. C. Fox. GPU Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 636–643, June 2014.

[48] J. Wang, K.-L. Wright, and K. Gopalan. XenLoop: a transparent high performance inter-VM network loopback. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 109–118. ACM, 2008.

[49] Xen Wiki. Xen 4.2: XL and PCI pass-through. https://wiki.xen.org/wiki/Xen_4.2:_xl_and_pci_pass-through.

[50] Xen Wiki. Xen VGA Passthrough. https://wiki.xen.org/wiki/Xen_PCI_Passthrough.

[51] S. Xiao, P. Balaji, J. Dinan, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W. c. Feng. Transparent Accelerator Migration in a Virtualized GPU Environment. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 124–131, May 2012.

[52] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W. c. Feng. VOCL: An optimized environment for transparent virtualization of graphics processing units. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12, May 2012.

[53] X. Xu and B. Davda. SRVM: Hypervisor support for live migration with passthrough SR-IOV network devices. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pages 65–77, New York, NY, USA, 2016. ACM.

[54] C.-T. Yang, J.-C. Liu, H.-Y. Wang, and C.-H. Hsu. Implementation of GPU virtualization using PCI pass-through mechanism. *J. Supercomput.*, 68(1):183–213, Apr. 2014.

[55] Y.-P. You, H.-J. Wu, Y.-N. Tsai, and Y.-T. Chao. VirtCL: A framework for OpenCL device abstraction and management. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 161–172, New York, NY, USA, 2015. ACM.

[56] YouTube Creator Blog. Look ahead: creator features coming to YouTube. https://01.org/igvt-g.

[57] E. Zhai, G. D. Cummings, and Y. Dong. Live migration with pass-through device for Linux VM. In *Ottawa Linux Symposium*, pages 261–268, 2008.