LoGV: Low-overhead GPGPU Virtualization

Mathias Gottschlag*, Marius Hillenbrand*, Jens Kehne*, Jan Stoess†, Frank Bellosa*

*System Architecture Group, Karlsruhe Institute of Technology
†HStreaming

***Abstract*— Over the last few years, running high performance computing applications in the cloud has become feasible. At the same time, GPGPUs are delivering unprecedented performance for HPC applications. Cloud providers thus face the challenge to integrate GPGPUs into their virtualized platforms, which has proven difficult for current virtualization stacks.**

In this paper, we present LoGV, an approach to virtualize GPGPUs by leveraging protection mechanisms already present in modern hardware. LoGV enables sharing of GPGPUs between VMs as well as VM migration without modifying the host driver or the guest's CUDA runtime. LoGV allocates resources securely in the hypervisor which then grants applications direct access to these resources, relying on GPGPU hardware features to guarantee mutual protection between applications. Experiments with our prototype have shown an overhead of less than 4% compared to native execution.

I. Introduction

Running high performance computing applications in the cloud has become a viable alternative to buying and maintaining dedicated compute clusters. At the same time, GPGPUs have started to deliver unprecedented performance for HPC applications. Cloud providers are thus facing the challenge of integrating GPGPUs into their platforms, which typically make extensive use of virtualization. However, integrating GPGPUs into current virtualization stacks has proven difficult [1].

A straightforward solution employs a pass-through mechanism that gives a VM direct, exclusive access to the GPGPU. While this method can achieve near-native performance, it also diminishes the advantages of the cloud platform: First, exclusive access to a GPGPU makes sharing of a physical GPGPU among multiple VMs impossible, leading to poor resource utilization. Second, the hypervisor has no control over the state of the GPGPUs, which makes VM migration difficult.

There have been multiple attempts to implement true virtualization of GPGPUs [2]–[6]. However, these attempts intercept all GPGPU commands in the guest in order to maintain a consistent GPGPU state. While this method allows for sharing of GPGPUs as well as VM migration, even the most sophisticated previous approach (vCUDA) still adds a runtime overhead of up to 21% [2].

In this paper, we present LoGV, a novel approach to GPGPU virtualization, which leverages the protection mechanisms already present in GPGPU hardware to fully virtualize GPGPUs without adding significant overhead. LoGV only intercepts commands related to resource allocation in the hypervisor, which then maps allocated resources directly into the guest VM, using the hardware's protection mechanisms guarantee protection between VMs. VMs can then access the mapped resources without intervention from the hypervisor. Our initial experiments show that the performance of LoGV is less than 4% below that of native execution for several common GPGPU algorithms.

Furthermore, LoGV also allows for live migration of VMs without interrupting currently executing GPGPU applications. To that end, our virtualization solution temporarily unmaps GPGPU resources from the VM and replaces them with shadow copies. These copies are then synchronized with the physical GPGPU on the destination system. From the application's point of view, this method maintains the illusion of uninterrupted access to the GPGPU.

The rest of this paper is organized as follows: We first describe some of the isolation features present in modern GPGPUs in Section II. Then, we present our proposed design in Section III. Section IV describes our prototype implementation and our initial performance evaluation. Finally, Section V presents related work and Section VI concludes the paper.

II. GPGPU Overview

Modern GPGPUs are typically usable by multiple applications at the same time. To guarantee mutual protection, GPGPU commands execute within virtual address spaces on the GPGPU. The CPU interacts with the GPGPU through ring buffers called *command submission channels*. Each channel is attached to exactly one address space on the GPGPU, and commands submitted to a channel can only access data in that channel's address space. Recent Nvidia GPGPUs support both multiple command submission channels and address spaces. The driver can thus allow multiple applications to share a GPGPU by allocating a separate set of address spaces and command submission channels for each application.

A. Memory Management

In order to ensure protection between applications, the device driver assigns a separate virtual address space to each application. To that end, the GPGPU features its own MMU as depicted in Figure 1. If a GPGPU command from an application accesses any GPGPU memory, the GPGPU's MMU resolves that access using that application's own page tables. Commands from different applications therefore cannot access memory outside their own address space.

Applications running on the CPU can also access GPGPU memory directly. The device driver can enable that access in three different ways:

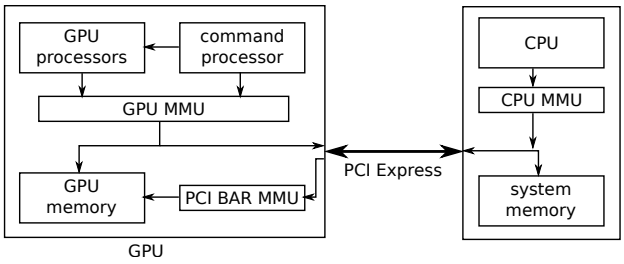

Figure 1. Memory access paths in a system consisting of a CPU and a GPU

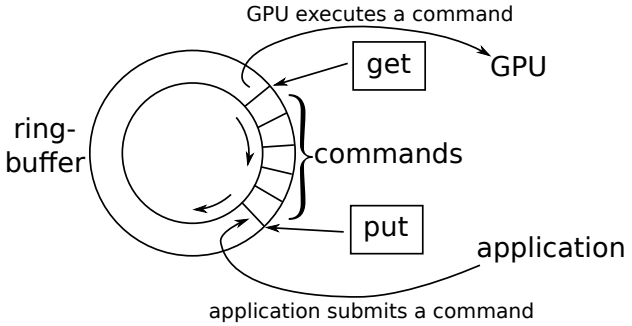

Figure 2. GPGPU command submission

1) Mapping a window of GPGPU memory into a CPU address space. The application can then access that memory in the same way as ordinary RAM.
2) Mapping a window of system RAM into a GPGPU address space. GPGPU applications can then use that window like regular GPGPU memory.
3) Using DMA data transfer between CPU and GPGPU memory. GPGPU applications can implement DMA transfers by copying data from mapped system RAM into GPGPU memory.

B. Command Submission

Modern GPGPUs work asynchronously to the CPU. The GPGPU device driver allocates a ring buffer in the application's GPGPU address space and maps it into that application's CPU address space. Figure 2 depicts this command submission channel. The application enqueues commands to the GPGPU into that ring buffer. The GPGPU in turn detects submitted commands through changes of the put pointer, which resides in a memory-mapped device register. The application can thus submit commands to the GPGPU without explicitly invoking the device driver.

III. Design

LoGV uses the protection mechanisms present in modern GPGPUs to grant virtual machines direct access to GPGPU resources. Modern GPGPUs support virtual address spaces for GPGPU memory similar to those on the CPU. In LoGV, the hypervisor performs resource allocations on behalf of the VMs to ensure that the VMs only have access to their own address spaces. Since each GPGPU command is confined to the address space of the application submitting the command, the applications in the VMs can then be allowed to submit commands to their own address spaces without hypervisor intervention. If the GPGPU supports multiple command submission channels, the hypervisor can grant each application its own set of command submission channels, allowing applications in multiple VMs to share one GPGPU while still guaranteeing mutual protection.

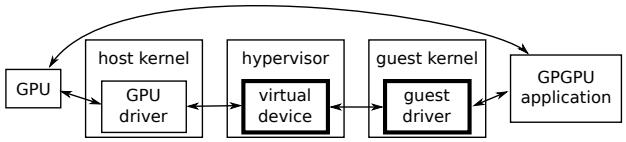

Figure 3. Overview of our virtualization layer. The two components of LoGV are highlighted.

LoGV enables VM migration by allowing the hypervisor to temporarily suspend access from a VM to the GPGPU. While the GPGPU is suspended, no changes to non-mapped GPGPU memory can occur. The hypervisor can then extract the VM's GPGPU memory for migration.

A. Basic architecture

LoGV consists of two parts as shown in Figure 3: An extension to the hypervisor and a guest kernel driver. The hypervisor extension processes all resource allocation and mapping requests to ensure protection, while the guest driver performs any operations directly related to the VM's state.

In order to ensure protection, all requests for the allocation of memory or command submission channels as well as mappings between GPGPU and system address spaces must be processed by the hypervisor. In LoGV, the application's user-mode software stack – typically CUDA or OpenCL – passes all such requests to the guest driver, which forwards the requests to the hypervisor. The hypervisor performs any necessary checks to ensure isolation, and subsequently forwards the request to the host driver, which performs the actual allocation. The hypervisor then returns the result of the allocation to the guest driver, which maps the allocated resource into the requesting application's address space. Since the application then has direct access to the mapped resource, the user-mode software stack can execute all other requests without intervention from the hypervisor. Note that no changes to the host device driver or the application's user-mode software stack are necessary.

B. Memory Allocation and Mapping

In LoGV, the hypervisor manages the allocation of GPGPU memory and maps that memory into system RAM. Handling memory management in the guest would allow the guest to affect other VMs, either by exhausting all available GPGPU memory or by establishing mappings to GPGPU address spaces of other VMs. By checking all allocation and mapping operations in the hypervisor, LoGV can prevent such malicious accesses without having to filter every subsequent memory access.

To ensure that applications stay confined to their GPGPU address spaces, the hypervisor processes all memory allocation

requests. Whenever an application requests GPGPU memory, the guest driver forwards that request to the hypervisor. The hypervisor records all allocation requests to track which GPGPU memory segments are assigned to which VM. At the same time, the hypervisor may also perform other checks, for example to enforce memory quotas. Finally, the hypervisor forwards the request to the device driver in the host, which performs the actual allocation.

To prevent VMs from accessing other VMs' GPGPU address spaces, LoGV must also secure access to GPGPU memory from the CPU. However, since the hypervisor tracks all memory allocations, it can easily implement protection for the three cases described in Section II-A:

1) When mapping GPGPU memory into an application address space, the hypervisor can verify that the mapped GPGPU memory was allocated by the same VM the request originated from.
2) When mapping system RAM into a GPGPU address space, the hypervisor can verify that the mapped RAM belongs to the same VM that created the destination address space on the GPGPU.
3) DMA data transfers are implemented as memory copies on the GPGPU, with either the source or the destination being a mapped window of system RAM. However, the hypervisor already enforces protection when mapping that window into the GPGPU address space.

However, the hypervisor can only implement protection between VMs in this way: Since the hypervisor has no knowledge of individual applications within the VMs, LoGV currently leaves protection between those applications to the guest driver.

C. GPGPU Command Submission

The hypervisor manages command submission channels for different VMs in the same way as the device driver manages channels for different applications. When an application in a VM requests a command submission channel, the hypervisor ensures that the GPGPU address space associated with the request belongs to the requesting VM, forwards the request to the host driver, and passes the allocated channel to the guest driver, which maps the channel into the address space of the requesting application. Since commands submitted to the channel can only access memory in the address space associated with the channel, it is then safe to let the application submit arbitrary commands directly to the channel without further intervention from the hypervisor. However, the hypervisor can only ensure protection between VMs this way, while protection between applications within the same VM is left to the guest driver.

If the GPGPU supports multiple command submission channels, the hypervisor can allow applications from multiple VMs to share a single GPGPU by granting each application its own set of command submission channels. The GPGPU will then execute commands from all channels in a round-robin fashion. However, since current GPGPUs do not take the execution times of individual commands into account, LoGV currently does not guarantee fairness.

D. Migration

LoGV has the ability to migrate virtual machines. During the migration, access to the GPGPU must be suspended to extract a consistent snapshot of GPGPU state. However, live migration of the remaining VM memory is possible nonetheless.

Suspend and Resume: During the migration, the GPGPU's state must be transferred between the two involved physical systems. However, GPGPU commands execute asynchronously to the CPU and can change the contents of GPGPU memory at any time. Furthermore, the CPU cannot monitor the execution of GPGPU commands in real time, which makes it difficult to predict when and where these commands will change GPGPU memory. It is therefore difficult to save a consistent image of that memory while the GPGPU is running. Therefore, LoGV suspends GPGPU command submission from a virtual machine before it migrates the VM.

Some GPGPUs allow the device driver to temporarily disable a command submission channel. A disabled channel will not execute any new commands after the currently running command has completed. Once all channels in an address space have been disabled, the hypervisor can migrate the remaining commands in those channels along with the contents of the address space.

If the GPGPU cannot disable command submission channels, another way to pause a GPGPU is to prevent the application from submitting new commands and wait until all queued commands have completed. In order to prevent the application from submitting new commands, the hypervisor unmaps all command submission channels from the VM and replaces them with shadow copies in RAM. All commands subsequently submitted by the VM will be stored in these shadow copies. The hypervisor can then wait for the real command submission channels to drain before performing the migration.

On the destination system, the hypervisor replays any migrated commands to newly allocated command submission channels. If necessary, it then replaces the shadow copies with the newly allocated channels before resuming the VM. The VM thus has the illusion of uninterrupted access to its command submission channels.

Migrating GPGPU memory: In order to reduce the down-time of the VMs, most migration solutions transfer the memory of running VMs using either pre-copy [7] or post-copy live migration [8]. Since GPGPU commands are not preemptible and potentially run for a long time, using live migration is desirable for GPGPU memory as well. However, we found it difficult to migrate GPGPU memory.

In theory, pre-copy live migration is possible as the CPU has direct access to GPGPU memory even while the GPGPU is executing commands. However, accessing GPGPU memory from the CPU is typically orders of magnitude slower than CPU memory, which makes monitoring an entire GPGPU address space for changes infeasible. Using DMA instead is also not possible: DMA transfers are initiated by writing a command to a command submission channel. However, that command will only execute after all other commands in that channel have

drained, at which point no further modifications to GPGPU memory will occur.

On the other hand, post-copy live migration is driven by page faults: Whenever the guest tries to access memory which has not been transferred yet, it is paused by the page fault handler until the memory has been fetched from the source system. However, most current GPGPUs do not support page faults in the same way as the CPU: Instead of interrupting the faulting command while the fault is handled, the GPGPU simply aborts the faulting command. A pagefault-driven approach is therefore not possible on current GPGPUs.

Since we did not find a feasible approach to live migration of GPGPU memory, our current design reverts to pausing the GPGPU before migrating its memory. We hope to devise a feasible strategy for pre-copy live migration in the future.

Migration Strategy: Even though live migration of GPGPU memory has proven difficult, live migration of the VMs system RAM is still possible. Our current design uses the following strategy to migrate virtual machines:

1) **Restricting access to the GPGPU:** The hypervisor either suspends the command submission channels or unmaps them from the virtual machine and replaces them with shadow copies in RAM. In the latter case, all commands submitted after this point will be written into the shadow copies.
2) **Transferring system memory:** The hypervisor transfers the VM's system memory using pre-copy live migration. The transfer includes any shadow command submission structures as well as any GPGPU memory currently mapped into the VM. This step can be executed in parallel with steps 3 and 4.
3) **Pausing the GPGPU:** The hypervisor waits until all command submission channels of the VM have stopped executing commands. After the last command has been executed, the GPU is idle and no further changes to the VM's GPGPU address spaces will occur.
4) **Transferring GPGPU memory:** The hypervisor transfers the parts of GPGPU memory which are not mapped into the virtual machine. Since the GPGPU is already paused, it cannot modify any data in these memory regions at this time.
5) **Transferring remaining virtual machine and GPGPU state:** The hypervisor halts the virtual machine and transfers all remaining information about GPGPU memory allocation and mappings. This data typically consists of only a few bytes per GPGPU memory allocation.
6) **Restoring GPGPU state:** On the destination system, the hypervisor initializes the GPGPU address spaces and transfers the migrated memory content to the GPGPU. If there are any outstanding commands in suspended command submission channels or shadow copies, the hypervisor replays these commands to newly allocated channels.
7) **Resuming the virtual machine:** At this point, initialization of the virtual machine is complete and the state of the GPU is the same as before the migration.

IV. Initial Evaluation

In order to evaluate the feasibility of our design, we implemented a prototype of our hypervisor extension. In this section, we present that prototype as well as the results of our initial experiments.

A. Prototype Implementation

We integrated our prototype implementation into the Linux Kernel virtual machine (KVM). In the host, we used an unmodified "pscnv" GPU driver [9]. Our guest driver provides the same API as pscnv and forwards any calls to the host after performing the appropriate checks. For the userspace parts of the GPGPU stack, we used an unmodified Gdev CUDA runtime [10]. For the sake of simplicity, the prototype is limited to one specific GPU model (NVidia GeForce GTX480).

Our prototype implements all features described in Section III except resource isolation. Furthermore, our migration code is currently limited to CUDA applications. In principle, however, it is possible to save and restore the state of non-CUDA applications in the same way as that of CUDA applications. We plan to implement support for non-CUDA applications in the future.

B. Functionality

In order to evaluate LoGV, we followed the criteria proposed by Dowty and Sugerman [11]:

- **Fidelity:** In the proposed design, the guest can send arbitrary commands to the GPGPU without hypervisor intervention. Thus, all features of the underlying GPGPU are available to the guest.
- **Multiplexing:** In LoGV, a virtual machine behaves like an application towards the GPGPU. Given the GPGPU supports multiple concurrently running applications, multiple virtual machines can share one GPGPU.
- **Interposition:** While the proposed design does not support advanced interposition for example for live migration of GPGPU memory, the hypervisor can temporarily revoke access to the GPGPU without damaging the GPGPU's application state. Therefore, it is possible to implement migration of VMs using the GPGPU.
- **Performance:** Our initial experiments show a performance overhead of less than 4% compared to native execution for several common GPGPU algorithms. We believe that overhead to be acceptable.

C. Benchmarks

In order to measure the virtualization overhead of our prototype, we selected four benchmarks from the test programs of the Gdev CUDA runtime and from the Rodinia benchmark suite [12]. We selected those benchmarks to represent a wide range of GPGPU applications as well as to test specific parts of the prototype implementation:

- **mmul** implements a simple matrix multiplication by squaring a random matrix with 2048×2048 entries. The benchmark executes only one rather long-running GPGPU kernel. Therefore the benchmark mainly shows that the

Benchmark	Host	Guest	Difference	Overhead
mmul	2922ms	2920ms	-1.72ms	-0.06%
lud	868ms	864ms	-4.88ms	-0.56%
nn	26ms	26ms	-0.08ms	-0.33%
backprop	51ms	53ms	1.82ms	3.55%

Table I
VIRTUALIZATION OVERHEAD BENCHMARK RESULTS

performance of virtualized GPGPU applications does not differ from that of applications running natively.

- **lud** implements LU decomposition, a common method to solve systems of linear equations. Since the problem is split into multiple shorter GPGPU kernel invocations, this benchmark places more emphasis on the overhead during GPGPU command submission.
- **backprop** implements a machine learning algorithm which trains a layered neural network. The computation of this benchmark alternates between CPU and GPGPU. Therefore, this benchmark spends a large part of its time transferring data between the two.
- **nn** computes the nearest neighbors from a set of points. This benchmark launches multiple short-running kernels and has a low total runtime.

All benchmarks were executed on a test system with two Intel Xeon E5-2620 CPUs, 32GB RAM and a NVidia GeForce GTX 480. Both the host system and the VMs were running Ubuntu 12.04 with Linux 3.5.7. In all cases, we used the same host GPU driver and userspace CUDA runtime. We conducted the migration tests on a single physical machine. The migration was performed through a loopback TCP connection, using trickle [13] to simulate a network bandwidth of 100MB/s and an average latency of 1 millisecond.

During all experiments, all but one CPU core were deactivated in both the host and the virtual machine. With more than one core, the benchmarks showed significant variations which were not caused by GPGPU virtualization. We believe that these differences were caused by scheduling effects which are not relevant for the evaluation of our design.

D. Virtualization Overhead

To demonstrate the performance of LoGV, we executed all benchmarks described above both natively and inside a VM. Since we are only interested in the performance of the GPGPU parts of the benchmarks, we measured the time from the first call into the CUDA API until the completion of the last call. We ran each benchmark 21 times and dropped the first result as it was slowed down significantly by file system operations.

Table I shows the results of the virtualization overhead benchmarks. Our results show that the performance of the mmul, lud and nn benchmarks running in a VM is roughly identical to native execution. The backprop benchmark experienced the highest overhead (3.55%) when run inside a VM. This benchmark has a short total runtime; therefore, the memory allocation overhead of LoGV has a larger impact on backprop than on the longer-running benchmarks. Note that the nn benchmark does not suffer from the same problem since it allocates less memory than backprop. However, since in all cases the total slowdown is well below 5%, we consider the overhead to be acceptable.

We also ran the lud benchmark concurrently in two VMs. We chose lud because it launches many short-running commands which the GPGPU can interleave. Running two VMs concurrently approximately doubled the runtime to 1696 ms, which indicates that, as we expected, each VM receives about half of the GPGPU's computational resources.

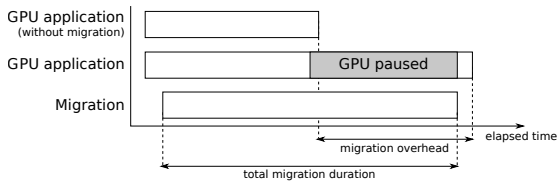

Figure 4. Relation between total migration duration and migration overhead

Benchmark	Downtime	Overhead
mmul	149.2ms	962.9ms
lud	196.0ms	1903.9ms

Table II
MIGRATION OVERHEAD AND DOWNTIME

E. Migration Performance

To evaluate our migration strategy, we measured both the downtime of the VM and the total migration overhead for running GPGPU applications. Here, as shown in Figure 4, the migration overhead is the difference between the runtimes of the application with and without migration. Since the GPGPU application continues to run during large parts of the migration, we expect the migration overhead to be smaller than the total migration duration. The downtime of the VM is the maximum period during which the VM is paused during the migration.

In order to measure the downtime as observed by the VM, we took advantage of the fact that pausing a VM causes a discontinuity in the clock of that VM: We sampled the system time once per millisecond and computed the downtime from the differences between consecutive clock values. Our experiments have shown that this sampling introduces less than 1% overhead to the runtime of the benchmark. As Table II shows, the resulting downtimes for the mmul and lud benchmarks were 149.2 and 196.0 milliseconds, respectively. We believe these downtimes to be manageable for most applications and therefore consider the migration to be live.

Benchmark	Overhead	Data Size	GPU Program Runtime	Migration Duration
mmul	962.9ms	48MiB	2801.8ms	3051.3ms
lud	1903.9ms	39MiB	660.9ms	2293.0ms

Table III
MIGRATION OVERHEAD COMPARED TO ALLOCATED MEMORY SIZE, PROGRAM RUNTIME WITHOUT MIGRATION AND TOTAL MIGRATION TIME

To measure the overall migration overhead, we compared the benchmark runtimes with and without migration. As seen in Table III, we measured an average overhead of 962.9 ms for mmul and 1903.9 ms for lud. Note that the overall migration overhead of the two benchmarks is higher than the virtual machine downtime because it includes times where the virtual machine is still running but the GPGPU is already paused.

The overhead of the lud benchmark was particularly high because that benchmark submits many short-running GPGPU commands. Using the proposed migration strategy, our prototype unmaps the command submission channels from the VM at the same time it starts to transfer the VM's memory. If the remaining commands in the unmapped channel finish quickly, the GPGPU will then idle until the migration is complete. A possible solution to this problem may be to perform some pre-copy iterations before unmapping the command submission channels from the VM.

V. Related Work

Several previous projects have addressed GPGPU sharing and migration. GViM [5] uses Xen's mechanisms [14] to enable efficient communication between guest VM and GPGPU. rCUDA [4], gViruS [3] and VOCL [6] can forward GPGPU commands to remote machines, which allows migration of VMs in the sense that the GPGPU state need not be migrated along with the VM. vCUDA [2] is a more recent approach using an optimized RPC protocol between guest and host. In addition to access to remote GPGPUs, vCUDA can also suspend local GPGPUs and save their state, which is a prerequisite for migration. However, all these approaches maintain state by intercepting all GPGPU commands through modified CUDA or OpenCL libraries in the guest and therefore share the same two drawbacks: First, all of them only support applications using one specific user-mode software stack. Second, intercepting all commands adds processing overhead to every command submitted by the guest. In contrast, our approach allows guest applications to submit raw commands directly into the GPGPU's command submission structures and therefore i) is not tied to a specific application software stack, and ii) does not add any overhead to command submission.

Zhai et al [15] employ PCI pass-through to grant VMs direct access to host devices, and then use ACPI S3 events to make the VM save the device state for migration. While the same approach could be applied to GPGPUs, its migration strategy is neither transparent nor live. Furthermore, the approach does not allow multiple VMs to share the same device.

VI. Conclusion

In this paper, we have presented LoGV, a novel approach to GPGPU virtualization, which leverages the existing protection mechanisms of modern GPGPUs. In LoGV, the hypervisor performs resource allocation requests on the VMs' behalf, and grants the VMs direct access to the allocated resources, relying on GPGPU hardware features to guarantee protection. LoGV also supports live migration of VMs using GPGPUs by temporarily unmapping GPGPU resources from the VM and replacing them with shadow copies. Our initial experiments indicate that LoGV achieves a runtime overhead of less than 4% compared to native execution.

For the future, we plan to implement resource- and performance isolation in LoGV. To achieve resource isolation, we are considering ways to swap GPGPU memory into system RAM in order to guarantee each VM a fair share of GPGPU memory while maintaining high utilization. Similarly, we are currently examining more sophisticated scheduling techniques [16] to guarantee performance isolation while sharing a GPGPU between VMs. We also intend to reduce LoGV's migration overhead by keeping the GPGPU active during the first pre-copy iterations. Finally, we plan to investigate the feasibility of live migration of GPGPU memory.

References

[1] M. Vinaya, N. Vydyanathan, and M. Gajjar, "An evaluation of CUDA-enabled virtualization solutions," in *Proceedings of the 2nd IEEE International Conference on Parallel Distributed and Grid Computing*, ser. PDGC '12, 2012, pp. 621–626.

[2] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 804–816, 2012.

[3] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds," in *Euro-Par 2010 - Parallel Processing*, ser. Lecture Notes in Computer Science, P. D'Ambra, M. Guarracino, and D. Talia, Eds. Springer Berlin Heidelberg, 2010, vol. 6271, pp. 379–391.

[4] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Orti, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *International Conference on High Performance Computing and Simulation (HPCS)*, Caen, France, 2010, pp. 224–231.

[5] 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*, ser. HPCVirt '09. New York, NY, USA: ACM, 2009, pp. 17–24.

[6] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W. chun Feng, "VOCL: An optimized environment for transparent virtualization of graphics processing units," in *Innovative Parallel Computing*, ser. InPar '12, San Jose, CA, 2012, pp. 1–12.

[7] 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 Symposium on Networked Systems Design & Implementation*, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286.

[8] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *ACM SIGOPS operating systems review*, vol. 43, no. 3, pp. 14–26, 2009.

[9] PathScale, "Pscnv," https://github.com/pathscale/pscnv.

[10] S. Kato, "Gdev cuda runtime," https://github.com/shinpei0208/gdev.

[11] M. Dowty and J. Sugerman, "GPU virtualization on VMware's hosted I/O architecture," *ACM Operating Systems Review*, vol. 43, no. 3, pp. 73–82, Jul 2009.

[12] "Rodinia benchmark suite," https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page.

[13] M. A. Eriksen, "Trickle: A userland bandwidth shaper for unix-like systems," in *Proc. of the USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005.

[14] 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 *Proceedings of the 19th Symposium on Operating System Principles*, Bolton Landing, NY, USA, Oct. 2003, pp. 164–177.

[15] E. Zhai, G. D. Cummings, and Y. Dong, "Live migration with pass-through device for Linux VM," in *Proceedings of the 2008 Ottawa Linux Symposium*, ser. OLS '08, Ottawa, Canada, Jul. 2008, pp. 261–268.

[16] M. Bautin, A. Dwarakinath, and T.-c. Chiueh, "Graphic engine resource management," in *Proceedings of the Annual Multimedia and Networking Conference*. International Society for Optics and Photonics, 2008.