Transparent Accelerator Migration in a Virtualized GPU Environment

Shucai Xiao,* Pavan Balaji,† James Dinan,† Qian Zhu,‡ Rajeev Thakur,† Susan Coghlan,§
Heshan Lin,* Gaojin Wen,¶ Jue Hong,¶ Wu-Chun Feng*

*Dept. of Computer Science, Virginia Tech. {shucai, hlin2, wfeng}@vt.edu
†Math. and Comp. Sci. Div., Argonne National Lab. {balaji, dinan, thakur}@mcs.anl.gov
‡Accenture Technologies. qian.zhu@accenture.com
§Leadership Comp. Facility, Argonne National Lab. smc@alcf.anl.gov
¶Shenzhen Inst. of Adv. Tech., Chinese Academy of Sciences. {gj.wen, jue.hong}@siat.ac.cn

***Abstract*—This paper presents a framework to support transparent, live migration of virtual GPU accelerators in a virtualized execution environment. Migration is a critical capability in such environments because it provides support for fault tolerance, on-demand system maintenance, resource management, and load balancing in the mapping of virtual to physical GPUs. Techniques to increase responsiveness and reduce migration overhead are explored. The system is evaluated by using four application kernels and is demonstrated to provide low migration overheads. Through transparent load balancing, our system provides a speedup of 1.7 to 1.9 for three of the four application kernels.**

***Keywords*-GPU, Virtualization, OpenCL, Migration, VOCL**

I. Introduction

Over the past several years, processor designers have encountered significant power constraints that have derailed CPU clock frequency scaling trends. Transistor scaling trends, however, continue to provide increasing numbers of devices within a chip. As a result, current microprocessors provide multiple cores per chip that must utilize application-level parallelism to achieve performance gains. Graphics processing units (GPUs) are many-core processors that have recently gained widespread use as computation accelerators for high-performance computing applications, and three of the top five machines on the November 2011 Top500 list [1] utilize GPUs.

Accelerator programming models, such as AMD's Brook+ [2], NVIDIA's CUDA [3], and the OpenCL [4] standard, have greatly eased the complexity in programming these devices and have quickened the pace of adoption. However, these models are currently limited to interacting with devices that are physically installed on the nodes where an application is executed. In cluster computing environments where GPUs are not available on every node or are underutilized on some nodes, programmers must construct their own system for coordinating the use of these resources.

In our prior work, we created Virtual OpenCL (VOCL) [5], an implementation of the OpenCL programming model that provides the ability to utilize and share nonlocal computation accelerators through device virtualization. VOCL establishes device proxies that manage the mapping of virtual to physical OpenCL contexts and forward OpenCL commands from the application to the physical device. VOCL can be used immediately by any OpenCL application and provides a runtime system that automatically manages the mapping of virtual to physical GPUs.

The ability to migrate virtual devices is a key capability in any virtualized environment. VOCL provides the ability to migrate virtual GPUs when a failure is detected; perform on-demand maintenance of compute resources; and dynamically adjust the mapping of virtual to physical devices to manage resource allocation and balance the workload.

In this work, we extend VOCL to support transparent, live migration of virtual OpenCL GPUs. Migration is achieved by transparently moving the virtual GPU state between VOCL proxies and physical devices and remapping the virtual-to-physical translation. Asynchronous, one-sided communication is used to decouple and coordinate migration. In addition, a command queueing strategy is introduced that allows the proxy greater control over the migratability of the virtual GPU and increases its responsiveness to migration events. We evaluate our migration framework on four application kernels from three application domains: dense linear algebra, n-body calculations, and data-intensive bioinformatics. Results indicate that, with our queueing technique, VOCL incurs low migration overheads while maintaining fast response time to migration events. In addition, through migration-enabled load balancing, applications achieve speedups of 1.7 to 1.9.

The remainder of this paper is organized as follows. In Section II, we provide an overview of the OpenCL programming model, the VOCL framework, and the Message Passing Interface (MPI) used internally by VOCL. In Sections III and IV, we present our virtual GPU migration framework and evaluate the overhead of migration and benefits from migration. In Section V, we discuss related work. In Section VI, we summarize our conclusions.

II. Background

In this section, we provide an overview of the OpenCL accelerator programming model, our prior work on the VOCL framework; and MPI, which VOCL uses internally for coordination and communication.

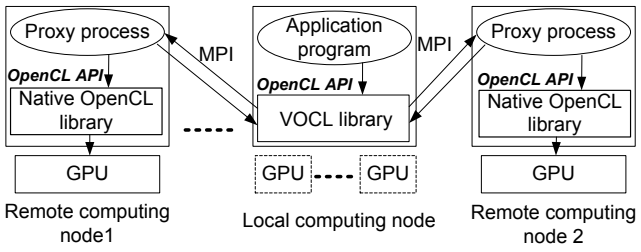


Fig. 1. VOCL framework enabling an application to utilize multiple, distributed GPUs.

A. OpenCL Programming Model

OpenCL (the Open Computing Language) [4] is a framework for programming various accelerators in heterogeneous computing environments. OpenCL provides an API that can be used to manage accelerator devices and execution contexts on various platforms. In addition, it defines a C-based programming language for writing *kernels*, which are special functions called on the host processor and executed on an accelerator device, such as a GPU, Cell Broadband Engine [6], or conventional CPU. An instance of a kernel is referred to as an OpenCL *work-item*, which executes on a single execution resource in the target device. A grid of work-items can be launched, where each grid coordinate corresponds to an element in the domain of the computation. Work-items can be grouped together for synchronization as an OpenCL *work-group*.

Current implementations of OpenCL provide the ability to utilize only accelerators that are local (e.g., connected by a high-speed PCIe link) to the host CPU. If a user wishes to utilize accelerators that are not local, mechanisms such as TCP/IP sockets or MPI are needed for data communication and coordination between the different machines.

B. The Virtual OpenCL Framework

In our prior work, we presented the Virtual OpenCL framework [5]. VOCL is a reimplementation of the OpenCL programming model that enables the programmer to utilize both local and remote accelerators through device virtualization. For local devices, VOCL directly calls the native OpenCL functions. For remote devices, VOCL forwards function calls to the location of the physical device.

The VOCL framework consists of the VOCL library and a proxy that is created on each remote node. The VOCL library shares the same interface as the native OpenCL library for their API functions. It forwards OpenCL function calls to the corresponding physical GPUs and receives GPU computing results for program output. The VOCL proxy is located on each remote node. It is responsible for receiving inputs from the application, calling the native OpenCL functions, and sending results back to the application once computation is completed. With the VOCL framework, all GPUs can be used as if they were installed locally, as shown in Figure 1.

OpenCL can handle multiple devices on a machine. Similarly, VOCL needs to handle devices on multiple computing nodes. To this end, VOCL provides another level of abstraction for resource management. Specifically, VOCL translates each OpenCL handle to a *VOCL handle* with a unique value; even two OpenCL handles share the same handle value. When a VOCL handle is used, the library first translates it to the corresponding OpenCL handle and then generates the corresponding information for data communication (if there is any). Next it sends the OpenCL handle to the device for the native OpenCL function to be called. Since OpenCL handles sharing, the same handle values are translated to different VOCL handles; VOCL is able to differentiate them and send function calls to the correct device. More details about the VOCL framework and the VOCL abstraction are described in our prior work [5].

C. The Message Passing Interface

The Message Passing Interface [7] is the industry standard for parallel programming and is available on essentially every parallel computing platform. MPI provides a rich communication interface, including basic point-to-point sends and receives; collective operations; and asynchronous, one-sided data transfer operations. MPI also provides the ability to dynamically establish and destroy new communication channels between different MPI processes. For these reasons, we have selected MPI as the communication runtime system on which to construct VOCL. While the VOCL runtime utilizes MPI internally, the application need only utilize OpenCL to take advantage of VOCL.

III. Transparent Virtual GPU Migration

In this work, we extend the VOCL framework with the ability to migrate virtual GPU images across different physical GPUs. We first describe the virtual GPU abstraction, followed by the virtual GPU migration algorithm. We then introduce a queueing mechanism that can be used to improve the performance of migration.

A. Virtual GPU Abstraction

A *virtual GPU* (or *VGPU*) represents the resources used by an application process on a physical GPU. These resources include OpenCL contexts, command queues, memory buffers, programs, and kernels. An application process can use multiple physical GPUs, with each represented by a virtual GPU. Similarly, one physical GPU can be shared by multiple applications; in such a case, an individual virtual GPU is created for each application. We illustrate the two cases in Figure 2.

In the VOCL framework, virtual GPUs exist both in the VOCL library and in the proxy with a one-to-one mapping relationship. That is, there is a virtual GPU in the VOCL library corresponding to each virtual GPU in the proxy. A virtual GPU in the VOCL library contains VOCL resources, referred to as *VOCL VGPU*; a virtual GPU in the proxy contains OpenCL resources and is referred to as *OpenCL VGPU*. Throughout the paper, we use the *source GPU* to indicate the GPU from which migration is originated and *destination GPU* for the migration destination. A *source proxy* is the proxy that contains the source GPU. Similarly, the destination GPU belongs to the *destination proxy*.

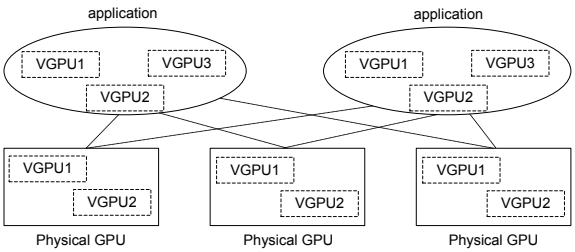


Fig. 2. Mapping of VOCL VGPUs to OpenCL VGPUs for two applications and three physical GPUs.

An OpenCL VGPU in the proxy is identified by the OpenCL device ID and the index of the application that is using the device. Once an application selects a physical GPU, a VGPU is created, and all OpenCL resources created by the application on the physical GPU will be saved in its VGPU. Besides OpenCL handles, information used to create the handles is also stored. For instance, when an OpenCL program is created, besides the program handle, we need to store its source code and build options. The reason is that OpenCL handles created on one physical GPU may be invalid on another. Therefore, when a VGPU is migrated, we need to recreate all the OpenCL resources on the destination GPU that requires all the information.

VOCL VGPUs reside in the VOCL library and have a one-to-one correspondence with OpenCL VGPUs. Each VOCL VGPU is identified by the VOCL device ID and the index of the proxy where the device is located. VOCL VGPUs store information such as VOCL contexts, VOCL command queues, and VOCL programs. In contrast to OpenCL VGPUs, which are created in the destination proxy and released in the source proxy in a migration, the update of a VOCL VGPU propagates from the source OpenCL VGPU to the destination OpenCL VGPU. Specifically, for each VOCL handle in the VOCL VGPU, its corresponding OpenCL handle and MPI data communication information will be replaced by its counterpart in the destination OpenCL VGPU. As a result, all OpenCL function calls will be directed to the destination proxy and GPU computation will be performed on the destination GPU. Since we keep the same VOCL handle in the VOCL VGPU, migration is transparent to the application.

B. Migrating Virtual GPUs

Migrating a VGPU across physical GPUs requires manipulation of the OpenCL and VOCL VGPUs as well as management of the GPU device and transfer of the VGPU image. When a migration is triggered, the OpenCL VGPU image is copied from the source proxy to the destination proxy. In the VOCL library, the corresponding VOCL VGPU needs to be mapped from the source OpenCL VGPU to the destination OpenCL VGPU accordingly. As a result, as shown in Figure 3, GPU computation performed on the source physical GPU is migrated to the destination physical GPU. However, careful synchronization must be employed to preserve data consistency and provide migration that is transparent to the user.

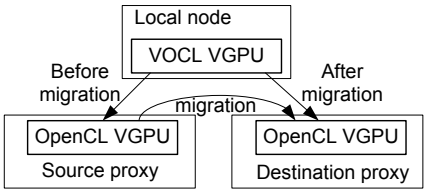


Fig. 3. Virtual GPU migration scenario.

We argue that migration should start as quickly as possible with minimum overhead. To achieve this, we extend the VOCL framework in two ways: (1) an internal command queue in the proxy to reduce the time for waiting the issued kernels to be completed, in order to support a quick start of migration; and (2) atomic enqueueing of OpenCL function calls to reduce the migration overhead. In the following, we first describe the conditions in which migration should be triggered; we then explain the steps involved in the migration. Finally, the above extensions to the VOCL framework to improve the migration performance are presented.

Currently, we consider two scenarios in which migration can be triggered: to free the GPU resources on a node (e.g. to perform maintenance) and to rebalance the VOCL to OpenCL mapping of VGPUs. To free the GPU resources on a given node, we provide a tool, *voclForcedMigration*, to send messages to the proxy. When a proxy receives the forced migration message, it will move all its tasks to other nodes. The second scenario is for load balance. To enable this, we provide a function called `voclRebalance()` to check the loads on the physical GPUs. If the load difference across physical GPUs is larger than a threshold value, migration will happen to rebalance the loads on different GPUs. Currently, we use a threshold of half of the internal queue depth N described in Section III-C, which means if the difference of the number of function calls that are issued to the native OpenCL library but not completed on two physical GPUs is larger than $N/2$, migration will be triggered. Many other strategies are possible and VOCL provides an interface that allows users to define new load-balancing modules.

Migration of a virtual GPU image across physical GPUs is performed by using the following algorithm:

1) **Lock the VGPU:** The VGPU is locked to prevent commands from being issued during migration.
2) **Drain command queue:** Before starting the migration procedure, the source proxy must wait for completion of all issued OpenCL function calls by invoking the OpenCL function `clFinish()`.
3) **Select physical GPU:** Select the physical GPU to which the virtual GPU will be created. Many criteria are possible; we select the physical GPU with the least computational load. In this step, the source proxy queries the load on each available physical GPU.
4) **Transfer OpenCL VGPU:** The source proxy marshalls the source OpenCL VGPU and transmits it to the destination proxy. In the destination proxy, an OpenCL VGPU is created using this information.
5) **Update VOCL VGPU:** The VOCL VGPU is updated

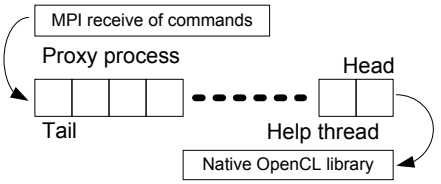


Fig. 4. Queueing of OpenCL operations in the VOCL proxy.

and mapped to the newly created OpenCL VGPU.

6) **Transfer contents of device memory:** Data in the device memory of the source VGPU is sent to the destination VGPU. Here, the data transfer is pipelined to reduce the data transfer overhead.
7) **Release source GPU:** After data transfer is completed, the source VGPU is released.
8) **Unlock the VGPU:** The migration lock on the VGPU is released, allowing the client to resume issuing OpenCL commands to the destination VGPU.

C. Queueing Virtual GPU Operations

One of the first steps in migration is to wait for completion of all issued OpenCL commands. In this step, if a large number of function calls are issued and migration is necessary, we may need to wait a long time before migration can start. This situation affects the delay until a fault can be migrated around, maintenance can be performed, or the load can be rebalanced.

To reduce the waiting time, instead of issuing all received OpenCL function calls to the GPU, the proxy creates an internal command queue to queue up the received functions, as shown in Figure 4. When an OpenCL function call is received, the proxy enqueues it into the command queue. Also, the proxy creates a help thread to issue function calls to the GPU. Each time the help thread issues a fixed number, N, of OpenCL function calls to the GPU and calls `clFinish()` to wait for their completion. After that, the help thread issues another N functions and calls the `clFinish()`, and so forth. In this way, when migration is necessary, the proxy process needs to wait for the completion of at most N function calls. Thus, the wait for completion time can be significantly reduced compared with issuing all OpenCL function calls to the GPU. However, this approach will add overhead to the program execution because kernel execution becomes synchronous in some degree by calling `clFinish()`. For example, if $N = 1$, `clFinish()` is called after every kernel launch. Consequently, a kernel cannot be launched until its previous kernel finishes the computation. This approach can cause significant overhead because the kernel launch is overlapped with the previous kernel's execution in general. Note that by tuning the *queue depth*, or N value, we can control the overhead of this approach in a low level, as we will show in our experimental results.

After the OpenCL VGPU is migrated to the destination proxy, the source proxy will send the unissued function calls to the destination proxy. After receiving the function calls, the destination proxy will first update the OpenCL handles to those in the destination OpenCL VGPU and will then enqueue them to its internal queue.

D. Atomic Enqueueing Commands in the Presence of Migration

When migration starts, the source proxy stops issuing OpenCL function calls to its GPU. Instead, it waits for completion of the issued function calls. At this time, any OpenCL function calls received from the VOCL library are stored in its internal command queue. Unissued function calls will be sent to the destination proxy, thereby incurring additional overhead. On the other hand, if the VOCL library stops issuing function calls to the source proxy, the number of function calls to be sent from the source proxy to the destination proxy can be reduced, and migration overhead is reduced as a result. To this end, we use a *migration lock* to prevent the VOCL library from issuing function calls to the source proxy when migration is in progress.

We utilize the MPI one-sided mutex algorithm of Ross et al. [8] to establish a migration lock between the application and the VOCL proxy. When migration starts, the source proxy acquires the mutex and holds it until migration is complete. In the VOCL library, the application must acquire the mutex before it can issue OpenCL function calls to the device. If there is no migration, the application can acquire the mutex immediately. On the other hand, if migration is in progress in the proxy, the application must wait for completion of the migration, when the mutex will be released before it can issue a function call to the proxy. In this way, function calls are restricted in the application when migration is in progress.

Since the application is expected to issue OpenCL function calls much more frequently than migration will occur, the mutex structure is located in the VOCL library on the application's node to reduce the overhead of locking.

IV. Experimental Evaluation

We used four application kernels, shown in Table I, to evaluate our system. Matrix multiplication and n-body are compute intensive, whereas matrix transpose and Smith-Waterman need more data movement between host memory and device memory. Using these kernels, we measured the cost of migration; demonstrated the performance impact of rebalancing the mapping of VGPUs to physical GPUs; and explored the tradeoff between a shallow queue depth, which decreases the time-to-migration, and a deeper queue depth, which can improve the efficiency of kernel execution.

Experiments were conducted on four QDR InfiniBand-connected compute nodes. Each node contains two AMD Magny-cours CPUs, 64 GB of memory, and two NVIDIA Tesla M2070 GPUs, each with 6 GB of global memory. The two GPUs are connected to different PCI express links, and one GPU shares its PCI express link with the InfiniBand NIC. In our experiment, we use two of the nodes as the remote GPU nodes and the other two as the local nodes on which only CPU is used. Each node runs the Centos Linux operating system, and the CUDA 3.2 toolkit is installed to provide OpenCL support. In addition, we use the MVAPICH2 [9]

TABLE I
COMPUTATION AND DATA ACCESS COMPLEXITIES FOR FOUR KERNELS. THE VALUE n CORRESPONDS TO MATRIX DIMENSIONS IN MATRIX MULTIPLICATION AND TRANSPOSE, THE NUMBER OF BODIES IN N-BODY, AND THE LENGTH OF THE INPUT SEQUENCE IN SMITH-WATERMAN. "MEMORY SIZE" ALSO CORRESPONDS TO THE VOLUME OF DATA TRANSFERRED BETWEEN HOST AND DEVICE MEMORIES FOR KERNEL EXECUTION.

Application Kernels	Computation	Memory size
Matrix multiplication	$O(n^3)$	$O(n^2)$
N-body	$O(n^2)$	$O(n)$
Matrix transpose	$O(n^2)$	$O(n^2)$
Smith-Waterman	$O(n^2)$	$O(n^2)$

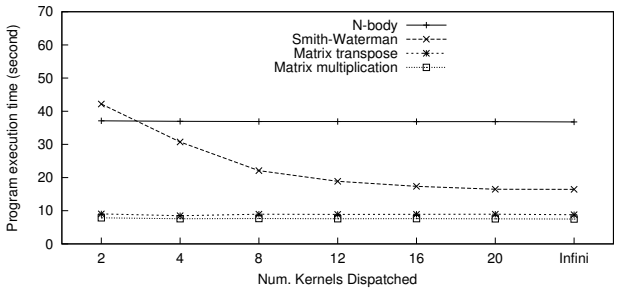


Fig. 5. Overhead caused by internal queue in proxy.

MPI implementation, which supports the QDR InfiniBand interconnect.

A. Impact of Command Queue Depth

As described in Section III-C, the proxy issues batches of N kernels to the OpenCL library and then blocks on their completion by calling `clFinish()`. The value of N determines the depth of the OpenCL command queue before waiting for completion. High values of N can improve kernel execution efficiency, whereas low values of N reduce the delay between issuing a migration event and when migration can be performed.

Figure 5 shows the total execution time with no migrations over a range of queue depths. `Infini` indicates that the queue has an infinite depth and that the proxy does not periodically invoke `clFinish()`. As can be seen, for matrix multiplication, matrix transpose, and n-body, program execution time with different queue depths shows little variation. Specifically, with $N = 2$, which means `clFinish()` is called after every two kernel launches, program execution time increases by 4.6%, 2.7%, and 0.8% respectively. The reason is that these three applications utilize long-running kernels that mitigate performance degradation from synchronous kernel execution. Smith-Waterman, on the other hand, launches a large number of short kernels, resulting in a slowdown of 256% when $N = 2$. Increasing of the N value reduces overhead for all four applications.

While increasing queue depth improves device utilization, it also impacts the waiting time before migration can be initiated. As Figure 6 shows, with the increase of the N value from 2 to 20, the wait for completion time increase for all four applications. Note that the wait for completion time

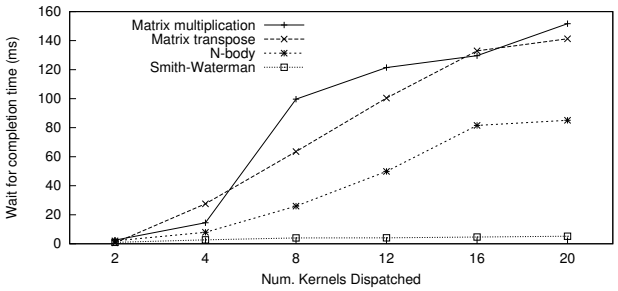


Fig. 6. Wait for completion time with different N values.

affects only the time interval between a migration event and when migration can be performed; it does not correspond to overhead.

Using the data from these two experiments, we chose a value of $N = 20$ as the default queue depth in VOCL (N is an user-adjustable parameter). When $N = 20$, the overhead caused by queueing is less than 2%, and the wait for completion time is also very low, only a few hundred milliseconds for our application kernels.

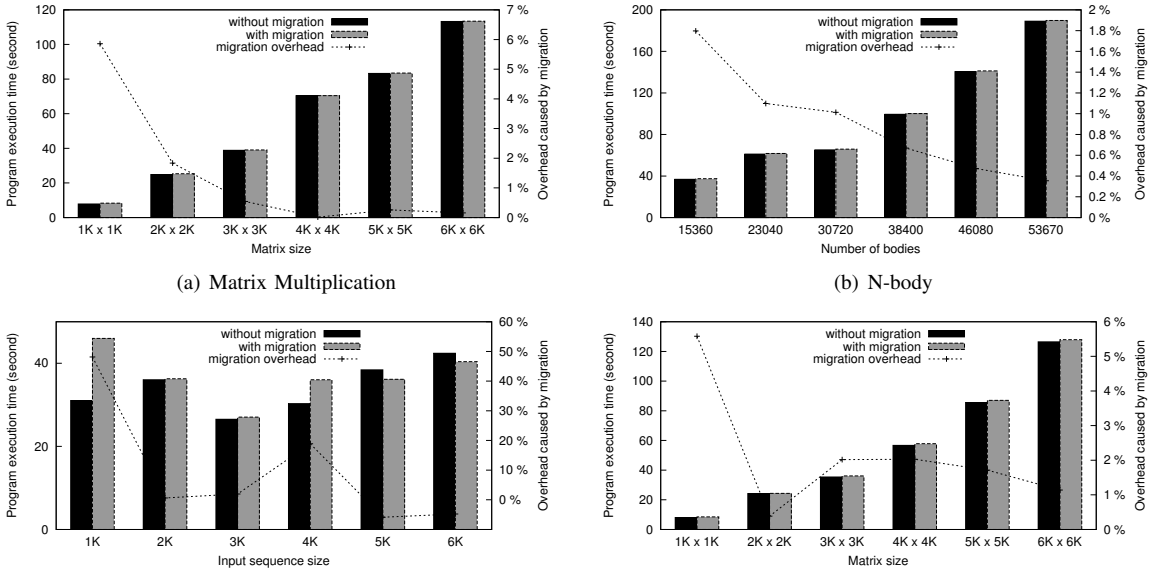
B. Analysis of Migration Overhead

In Figure 7, we show the total execution time for each kernel with no migration and with a single migration. From the figure we see that, overall, as the problem size increases, the relative overhead decreases. The reason is that the execution time increases faster than the migration overhead with regard to the problem size. Thus, less relative overhead is caused in programs running a larger problem size. In addition, migration overhead is a few hundred milliseconds. For programs that run long enough, other factors such as network congestion and system noise affect the total execution time more than migration does. From these results, we conclude that performance degradation caused by migration can be negligible for programs running a reasonably long time (e.g., a few tens of seconds).

Figure 8 shows a detailed breakdown of the migration overhead for the four benchmarks across all input sizes. The time for virtual GPU creation includes the latency of draining the device's command queue and the program build time for the destination VGPU, which dominate the overhead in all four applications. For Smith-Waterman, we see that the large number of queued function calls generated by the application also increases the cost of migrating the queue of unissued functions. For n-body, which contains two kernel programs each of which needs to be built separately, the copy VGPU time is about twice that of the other three.

C. Impact of VGPU Oversubscription

Virtualization introduces the opportunity to oversubscribe a physical GPU by binding multiple VGPUs to the same device. This can have a positive effect on computational efficiency by providing the device with ample computation to hide latencies and maximize occupancy. In Figure 9, we show the total execution time for each benchmark when multiple instances of the application are running on the smallest input problem



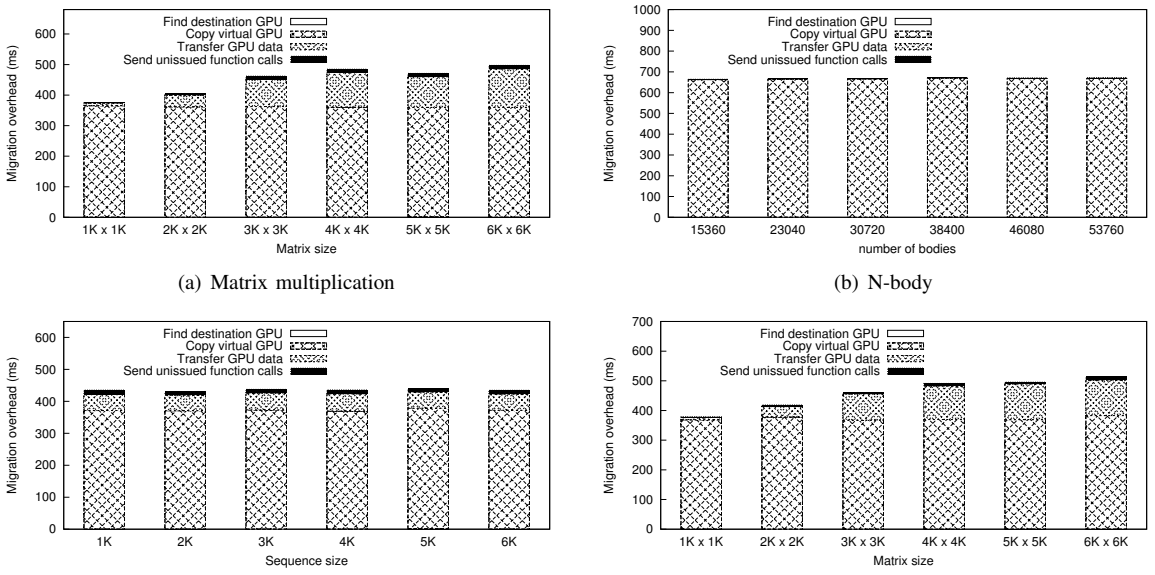
(a) Matrix Multiplication

(b) N-body

(c) Smith-Waterman

(d) Matrix Transpose

Fig. 7. Total execution time for each kernel over a range of input sizes with and without migration.



(a) Matrix multiplication

(b) N-body

(c) Smith-Waterman

(d) Matrix transpose

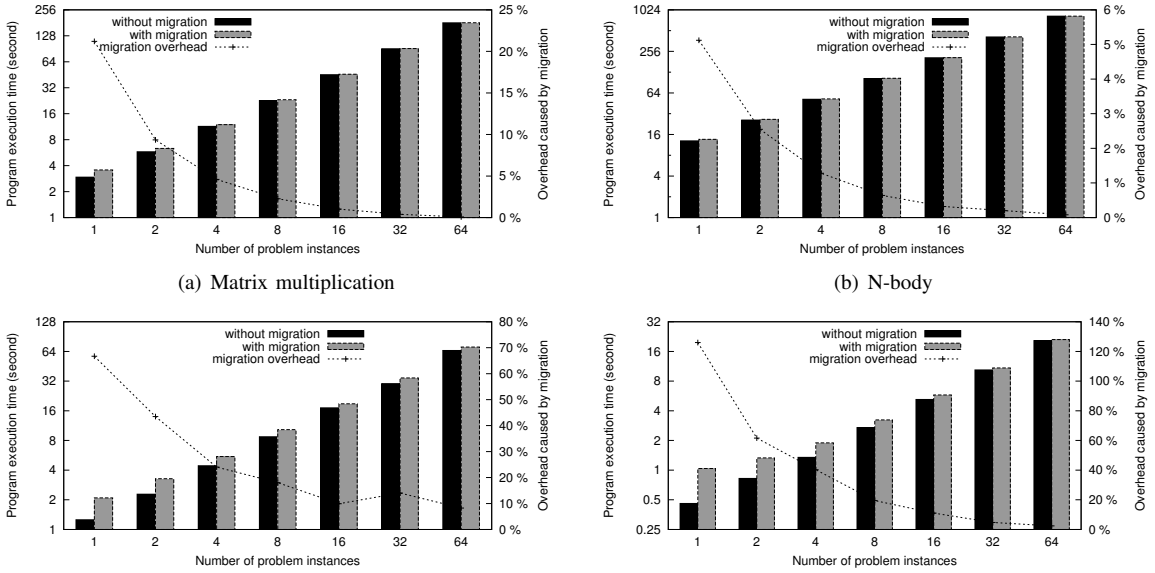
Fig. 8. Breakdown of migration overheads for each benchmark across all input sizes.

and all share the same VGPU. For this graph, we show the execution time with no migration, the execution time when a single migration is performed, and the percent difference in execution time due to migration. We see that increasing the degree of oversubscription significantly decreases the cost of migration by an order of magnitude or more across all application kernels.

D. Performance Impact of Load Balancing

Migration also adds the capability to balance the VGPU workload across physical GPUs. In Figure 10 we show the performance improvement from VOCL's load balancing. For this experiment, we ran two instances of the same application and mapped both VGPUs to the same physical device. This represents a scenario where one device is initially occupied when the new VGPUs are created. In the baseline case, both instances share the physical GPU for their full execution. In the migration case, one of the applications triggers the VOCL load balancer, which performs migration. This corresponds to a scenario in which resources become available while the application is running. After migration, each VGPU is mapped to a separate physical GPU.

From the data we see that, with task migration enabled in the framework, application performance is improved for all



(a) Matrix multiplication

(b) N-body

(c) Smith-Waterman

(d) Matrix Transpose

Fig. 9. Total execution time for each kernel over a varying degree of oversubscription with and without migration.

four applications. Specifically, the total time to complete the matrix multiplication reduces by a factor of 1.7; n-body is 1.9 times faster; matrix transpose is 1.7 times faster; and Smith-Waterman is 1.4 times faster. Among the four applications, the speedup of n-body is the highest and Smith-Waterman the lowest. These results are consistent with the varying degree of overhead incurred by migration in all four applications, as shown in Figure 7; the least amount of data is transferred in the migration of n-body, whereas Smith-Waterman requires the largest amount of data transfer. In addition, total execution time of n-body is much larger than that of Smith-Waterman. Hence, migration overhead has far less impact on its performance than on that of Smith-Waterman.

V. Related Work

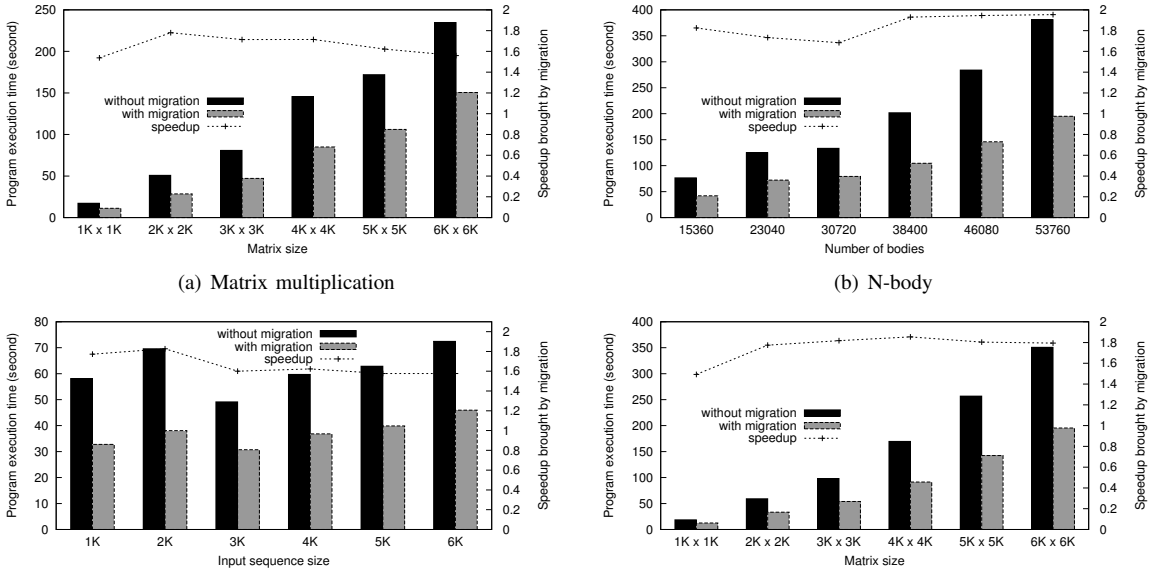
Our work is related to task migration across different GPUs and different nodes. Many studies have been conducted on migration in large-scale computing systems.

Process migration can be achieved by the checkpoint approach. One study is the Berkeley Lab Checkpoint/Restart (BLCR) [10]. It writes the process image to a file and then restarts the process from the saved process image file. This approach can be used for migrating a process from one node to another, but it considers only the process image of CPU processes.

Based on BLCR, Ouyang [11] used a proactive job migration scheme to enhance the fault tolerance of the MVAPICH2 [9]. This work implements the checkpoint/restart procedure by transferring the process image to a healthy spare node for the purpose of resuming the process. Wang et al. [12] proposed a process-level live migration mechanism to support continued execution of MPI processes. This work is integrated into an MPI execution environment to transparently sustain health-inflated node failures, which eradicates the need to restart and requeue MPI jobs. These studies are related to the task migration in our VOCL framework in that we support live migration of VGPUs from one physical GPU to another and migration is transparent to the program execution.

Takizawa et al. [13], [14] demonstrated the feasibility of migrating a GPU program from one node to another. This work is similar to ours. However, our work is distinct in the following ways. First, in their work, an API proxy is added to store the image file, which makes OpenCL function calls become a two-phase procedure. Thus, when large amounts of data are transferred between host memory and device memory, large overhead can be caused to the program execution even in the local GPU usage. In contrast, in our VOCL framework, there is no such API proxy on the local node, and no additional overhead is caused to the usage of local GPUs. Second, when migration is triggered, in Takizawa et al.'s work, execution of the process needs to be terminated and restarted on the target machine. In contrast, migration in the VOCL framework is transparent to the application program and happens during its execution, where process termination and restart are not necessary. Third, process image is stored in the hard disk in Takizawa et al.'s migration approach, which puts a heavy burden on the storage subsystem and can cause significant overhead for restarting the process. In contrast, we do not use the hard disk; all data are transferred over the network.

Another strategy for task migration is based on the virtual machine such as Xen [15], which enables migration of virtual operating system (OS) instances across different compute nodes. One example is vCUDA [16]. In this approach, all API function calls on the target OS need to be redirected to the host OS when migration happens. As a result, it causes significant migration overhead on both the host and the target nodes.



(a) Matrix multiplication

(b) N-body

(c) Smith-Waterman

(d) Matrix Transpose

Fig. 10. Total execution time for each benchmark over a range of input sizes without and with VOCL load balancing.

VI. Conclusion

In this paper, we extend our VOCL framework to support transparent, live virtual GPU migration. VGPU migration enables VOCL to provide fault tolerance, perform resource management and load balancing, and provide facilities for quick system maintenance. When a migration is triggered, the VOCL framework collects the OpenCL and application states from the source GPU and transmits it to the new destination GPU. VOCL reconstructs the VGPU at the destination and transparently updates the application's handle to the VGPU to establish a connection with the new VGPU. To reduce migration latency, we extend VOCL with an OpenCL dispatch queue that limits the command queue depth, thereby reducing the time to migration.

We evaluate the VOCL framework using four application kernels. Results indicate that schemes to improve time to migration and migration overhead can effectively reduce performance penalties. In addition, VOCL's VGPU load balancing is used to demonstrate performance improvement when additional resources become available to the application. VOCL load balancing is shown to provide a speedup of 1.7 and 1.9 for two compute-intensive benchmarks and 1.4 and 1.7 for applications needing more data movement between host memory and device memory.

Acknowledgments

This work is supported in part by the U.S. Department of Energy under Contract DE-AC02-06CH11357, the NSF grant CSR 0916719, the National Natural Science Foundation of China (Grant No. Y046021001, 60903116), and the Shenzhen Science and Technology Foundation (Grant No. ZYC201006130310A, JC200903170443A).

References

[1] "Top500 Supercomputing Sites," http://www.top500.org/.

[2] AMD/ATI, "Stream Computing User Guide," April 2009, http://developer.amd.com/gpu_assets/Stream_Computing_User_Guide.pdf.

[3] NVIDIA, "NVIDIA CUDA Programming Guide-4.0," May 2011, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[4] Khronos OpenCL Working Group, "The OpenCL Specification," June 2011, http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[5] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W. Feng, "VOCL: An Optimized Environment for Transparent Virtualization of Graphics Processing Units," in *Proc. of Innovative Parallel Computing*, May 2012.

[6] T. Chen, R. Raghavan, J. Dale, and E. Iwata, "Cell Broadband Engine Architecture and Its First Implementation – A Performance View," *IBM Journal of Research and Development*, vol. 51, no. 5, p. 559, Nov 2007.

[7] MPI Forum, "MPI-2: A message-passing interface standard, version 2.2," University of Tennessee, Knoxville, Tech. Rep., Sept. 2009.

[8] R. Ross, R. Latham, W. Gropp, R. Thakur, and B. Toonen, "Implementing MPI-IO Atomic Mode without File System Support," in *IEEE International Symposium on Cluster Computing and the Grid*, May 2005.

[9] Network-Based Computing Laboratory, "MVAPICH2 (MPI-2 over OpenFabrics-IB, OpenFabrics-iWARP, PSM, uDAPL and TCP/IP)," http://mvapich.cse.ohio-state.edu/overview/mvapich2.

[10] P. H. Hargrove and J. C. Duell1, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," in *Proc. of SciDAC*, June 2006.

[11] X. Ouyang, S. Marcarelli, R. Rajachandrasekar, and D. K. Panda, "RDMA-Based Job Migration Framework for MPI over InfiniBand," in *Proc. of IEEE International Conference on Cluster Computing*, September 2010.

[12] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive Process-Level Live Migration in HPC Environments," in *Proc. of ACM/IEEE conference on Supercomputing*, June 2006.

[13] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "CheCUDA: A checkpoint/restart tool for cuda applications," in *Proc. of International Conference on Parallel and Distributed Computing, Applications and Technologies*, December 2009.

[14] ——, "CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications," in *Proc. of 25th Intl. Parallel and Distributed Processing Symp.*, May 2011.

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Wareld, "Xen and the Art of Virtualization," in *Proc. of ACM Symposium on Operating Systems Principles*, December 2003.

[16] L. Shi, H. Chen, and J. Sun, "vCUDA: GPU Accelerated High Performance Computing in Virtual Machines," in *IEEE Transactions on Computers*, 2011.