Generalizing the Utility of GPUs in Large-Scale Heterogeneous Computing Systems

Shucai Xiao (6th year Ph.D. student) and Wu-chun Feng

Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, Virginia 24061
Email: {shucai, wfeng}@vt.edu

***Abstract*—Graphics processing units (GPUs) have been widely used as accelerators in large-scale heterogeneous computing systems. However, current programming models can only support the utilization of local GPUs. When using non-local GPUs, programmers need to explicitly call API functions for data communication across computing nodes. As such, programming GPUs in large-scale computing systems is more challenging than local GPUs since local and remote GPUs have to be dealt with separately. In this work, we propose a virtual OpenCL (*VOCL*) framework to support the *transparent* virtualization of GPUs. This framework, based on the OpenCL programming model, exposes physical GPUs as decoupled virtual resources that can be transparently managed independent of the application execution. To reduce the virtualization overhead, we optimize the GPU memory accesses and kernel launches. We also extend the VOCL framework to support live task migration across physical GPUs to achieve load balance and/or quick system maintenance. Our experiment results indicate that VOCL can greatly simplify the task of programming cluster-based GPUs at a reasonable virtualization cost.**

***Keywords*-graphics processing unit (GPU), virtual OpenCL, task migration**

I. INTRODUCTION

Today, the growth rate of computational power required for some scientific computations has been outstripping that of the computational capabilities of traditional processors. On the other hand, gains in computational horsepower by increasing the clock speed have hit a wall in the mid-2000s due to the fabrication process and power/leakage constraints. As a consequence, parallelism must be relied upon in order to improve computational performance. At the hardware level, performance improvements have been driven by increasing the number of cores per processor, which is true for both traditional x86 multicore processors and many-core GPU processors. These multi- and many-core processors in turn have been aggregated into compute nodes, which are then aggregated into cluster supercomputers.

Originally designed solely for graphics processing, GPUs have evolved into programmable processors for general-purpose computation. With parallel programming models such as CUDA [10] and OpenCL [7]), programming GPUs has become easier than ever. Many applications have now been parallelized on GPUs, and significant performance improvements have been reported [11].

However, current GPU programming models only support the utilization of GPUs installed locally. When a program needs to use GPUs that are not locally installed, API functions such as TCP sockets and Message Passing Interface (MPI) [8] should be called explicitly to handle data communication across machines. Because of this limitation, application parallelization in large-scale heterogeneous systems is much more challenging than that on a single node. Moreover, GPUs continue to make in-roads as an accelerator in supercomputers. In a recent Top500 list [1], three of the top five fastest supercomputers in the world used GPUs as accelerators. In light of this trend, we expect the need for *transparent GPU virtualization*, i.e., using all GPUs in the same way as if they were installed locally, to become increasingly important.

In this dissertation, we investigate generalizing the utility of GPUs in large-scale heterogeneous systems. We propose a virtual OpenCL (VOCL) environment for the transparent virtualization of GPUs. VOCL provides the OpenCL-1.1 API but with the primary difference that it allows an application to view all GPUs available in the system (including remote GPUs) as local virtual GPUs. VOCL internally uses the MPI for data management associated with remote GPUs and utilizes several techniques, including kernel argument coalescing and data transfer pipelining, to improve performance. In addition, we extend VOCL to support live virtual GPU migration for quick system maintenance and load rebalance across GPUs. Overall, VOCL provides the usability and reliability of GPUs and maximizes system flexibility and utility to a wide range of users.

II. RELATED WORK

There have been many studies available for the GPU virtualization and task migration. Athalye et al. implemented a preliminary version of GPU-Aware MPI (GAMPI) for GPU-based clusters [2]. It is a C library with an interface similar to MPI, allowing application developers to visualize an MPI-style consistent view of all GPUs within a system. Duato et al. [5] presented a GPU virtualization middleware that makes remote GPUs available to all compute nodes in a cluster. However, both solutions require using GPUs in a different way and thus are *nontransparent*. Barak et al. [4] proposed a framework that implements the same functionality as our VOCL framework. However, it has limitations in two aspects: 1) an API proxy is needed on each local node for resource management. 2) They did not consider data transfer between host memory and device memory. Both aspects can cause large overhead to program executio. In contrast, VOCL has no such an API proxy and optimizes the data transfer to achieve high data transfer bandwidth.

For task migration, Takizawa et al. [6] showed the feasibility of migrating a GPU program from one node to another. This work is similar to ours. However, in their work, an API

proxy is first added to store the image file, which makes OpenCL function calls become a two-phase procedure even for local GPUs. Second, when migration is triggered, the process needs to be terminated and restarted; thus, it is not live migration. Third, the process image is stored on disk, which can put a heavy burden on the storage subsystem. In contrast, VOCL provides an approach for transparent, live task migration. In summary, our proposed VOCL framework provides a unique and interesting enhancement to the state-of-art in GPU virtualization.

III. Virtual OpenCL Framework

VOCL is a re-implementation of the OpenCL programming model. It enables programmers to utilize both local and remote GPUs through GPU virtualization. For local GPUs, VOCL internally calls native OpenCL functions. When the target GPU is non-local, OpenCL function calls are forwarded to the machine where the physical GPU is located.

The VOCL framework consists of the VOCL library and a proxy created on each remote node as shown in Figure 1. The VOCL library exposes the OpenCL API to applications and is responsible for sending information about OpenCL calls to the VOCL proxy using MPI, and returning the proxy responses to the application. The proxy is responsible for handling messages from the VOCL library, executing the actual functionality on physical GPUs, and sending results back to the VOCL library.

To differentiate OpenCL handles on different machines, VOCL provides another level of abstraction. Specifically, VOCL translates each OpenCL handle created on a proxy to a *VOCL handle* with a unique value, even the OpenCL handles share the same value. Also, VOCL stores the data communication information within the VOCL object. In this way, VOCL is able to differentiate OpenCL handles with the same value and send them to the correct proxy. More details about the VOCL framework are described in our previous work [14].

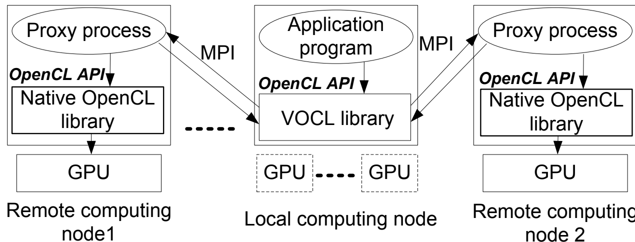

Fig. 1. VOCL enables an application to utilize multiple, distributed GPUs.

IV. VOCL Optimization

Since VOCL internally calls native OpenCL functions for local GPUs, overhead caused by VOCL is mainly related to the utilization of remote GPUs. In this section, we explain our optimizations on reducing the overhead of using remote GPUs.

In a typical OpenCL program, functions for OpenCL object creation and release occupy a very small percentage of the total execution time. Overhead of these functions in the usage of remote GPUs is very small, too. As a result, VOCL optimizations are mainly for kernel execution related functions (e.g., GPU memory read/write, kernel argument setting), which can cause large overhead to program execution in some applications.

We propose the approach *kernel argument coalescing* to reduce the overhead of setting kernel arguments. The basic idea is to combine the message transfers for multiple `clSetKernelArg()` calls. Instead of sending one message for each call of `clSetKernelArg()`, we send kernel arguments to the remote node only once per kernel launch, irrespective of how many arguments the kernel has. Specifically, when `clSetKernelArg()` is called, VOCL library just caches the arguments locally at the VOCL library. Then in the kernel launch, the VOCL library sends the arguments to the proxy, which performs two steps on being notified of a kernel launch: (1) it receives the argument message and sets the individual kernel arguments, and (2) it launches the kernel.

When remote GPUs are used, data transfer between host memory and device memory includes two stages — between local host memory and remote host memory and between remote host memory and remote GPU memory. In a naïve implementation, these two stages are serialized, which can seriously restrict the data transfer bandwidth. In order to improve the bandwidth, we design a data pipelining mechanism. Specifically, we overlap the first stage transfer of one data chunk with the second stage of another. Also, we segment large data chunks into multiple data blocks, whose transfer can also be pipelined across each other. Figure 2 shows such a scenario for the GPU memory write. As we can see, data transfer of the two stages can be overlapped if multiple data blocks are transferred consecutively.

GPU memory read can be optimized in the same way.

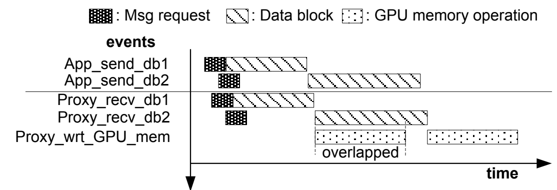

Fig. 2. Nonblocking write to the GPU memory

V. VOCL Extension

In this section, we present the extension of VOCL to support transparent, live task migration across physical GPUs [13]. Migration in VOCL is based on the *virtual GPU*, which represents the GPU resources utilized by an application on each physical GPU. Virtual GPUs exist in both the proxy and the VOCL library with a one-to-one mapping relationship. That is, for each virtual GPU in a proxy, there is a corresponding virtual GPU in the VOCL library. A virtual GPU in the proxy contains OpenCL resources and is referred to as *OpenCL VGPU*. Similarly, a virtual GPU in the VOCL library contains VOCL resources, which is referred to as *VOCL VGPU*;

Migration is achieved by transparently moving the OpenCL VGPU state between physical GPUs and remapping the VOCL-to-OpenCL VGPUs. When a migration is initiated, as

shown in Figure 3, the OpenCL VGPU is migrated from the source proxy to the destination proxy. In the VOCL library, the corresponding VOCL VGPU must also be mapped from the source OpenCL VGPU to the destination OpenCL VGPU. As a result, GPU computation on the source physical GPU will be performed on the destination physical GPU after migration.

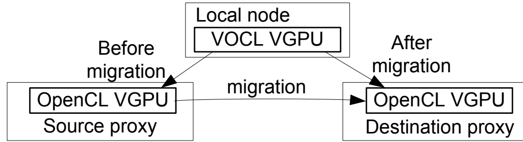

Fig. 3. Migration scenario

VI. Preliminary Results

In this section, we evaluate the efficiency of the proposed VOCL framework via a few micro-benchmarks and application kernels. The evaluation includes four aspects: First, performance improvements corresponding to the optimization strategies; Second, overhead of program execution in the utilization of remote GPUs; Third, speedup achieved by using multiple virtual GPUs; Fourth, performance improvements brought by the task migration when load imbalance exists in the system.

We use computing nodes connected with QDR InfiniBand for our experiments. On each node, there are two Magny-cours AMD CPUs and two NVIDIA Tesla M2070 GPU cards. The nodes are installed with the Centos Linux operating system and the CUDA 3.2 toolkit. We use the MVAPICH2 [9] MPI implementation that can support the InfiniBand. Each of our experiments was conducted three times and the average is reported.

A. Performance Improvements from VOCL Optimizations

In this section, we show the performance improvements brought by our optimization strategies in the VOCL.

Table I presents the kernel execution times of Smith-Waterman [12] for aligning a pair of 6K-letter sequences in three scenarios — native OpenCL on a local GPU and VOCL on a remote GPU with and without the kernel argument coalescing optimization. As we can see, without the optimization, `clSetKernelArg()` has very large overhead of 416.12 ms. The reason is that the function is called more than 86,000 times in the program execution (the kernel is called 12,228 times and 7 arguments are set per call.). With the optimization, time of `clSetKernelArg()` is reduced to 4.03 ms. Though a slightly higher overhead for the kernel execution (increase from 1344.01 ms to 1316.92 ms) is observed, which is due to the additional kernel argument message passed to the proxy within this call, the total kernel execution time decreases from 1737.37 ms to 1348.04 ms, or by 22.41%.

As to the data transfer pipelining, Figure 4 shows the data transfer bandwidth in different scenarios. As we can see, 1) the pipelining mechanism almost doubles the data transfer bandwidth compared to that without pipelining. 2) As the message size increases, the bandwidth increases for the native OpenCL as well as VOCL. With the pipelining optimization, VOCL-remote saturates at a bandwidth of around 20-25% lesser than that of native OpenCL. Overall, VOCL achieves about 80% of the bandwidth for GPU memory write in a native nonvirtualized environment. Similar results are observed for the GPU memory read.

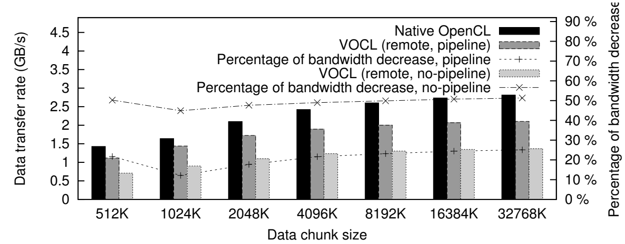

Fig. 4. Data transfer bandwidth of GPU memory write

B. Evaluation with Application Kernels

In this section, we evaluate the efficiency of the VOCL framework using four application kernels — matrix multiplication (MM), n-body (NB), matrix transpose (MT) and Smith-Waterman (SW). Figure 5 shows the performance and the overhead of the application kernels. As we can see, overhead is caused in the utilization of remote GPUs and depends on the computation and data movement properties of each application. For compute-intensive algorithms, the overhead of VOCL is very small; 2.65% for matrix multiplication and less than 1% for n-body. This is because their execution time is dominated by the kernel execution. For algorithms requiring more data movement between host memory and device memory, the overhead of VOCL is higher. For matrix transpose, it is about 21.48%, which is expected because it spends a large fraction of its execution time in data movement. For Smith-Waterman, the overhead is much higher and close to 160%. There are two reasons for that. First, the MPI used in the VOCL proxy has to be initialized to support multiple threads. It is well known in the MPI literature that multi-threaded MPI implementations can cause significant overhead for the transfer of small messages [3]. Second, as explained in Section VI-A, Smith-Waterman relies on a large number of kernel launches, in which large amounts of small messages are transferred. As a result, its performance is seriously impacted.

C. Speedup by Using Multiple Virtual GPUs

In this section, we show performance improvements by using multiple virtual GPUs based on VOCL. Figure 6 shows the total speedup achieved with 1, 2, 4, 8, 16, and 32 virtual GPUs utilized. With one and two GPUs, only local GPUs are used. In other cases, two of the GPUs are local, and the remaining are remote.

As shown in the figure, for compute-intensive algorithms, the speedup can be significant; for instance, with 32 GPUs, the overall speedup of n-body is about 31-fold compared to the single GPU case. For matrix multiplication, the speedup is 11.5-fold (some scalability is lost because of the serialization of the data transfer through a single network link). As to algorithms requiring more data movement, there is almost no performance improvement. In fact, the performance degrades in some cases. For the matrix transpose, the reason is that most of the program execution time is for data transfer between

TABLE I
PERFORMANCE IMPROVEMENTS BROUGHT BY KERNEL ARGUMENT COALESCING

	Local/Native	Remote, w/o optimization		Remote,w/ optimization	
Function name	Runtime	Runtime	Overhead	Runtime	Overhead
clSetKernelArg	4.33	420.45	416.12	4.03	-0.30
clEnqueueNDRangeKernel	1210.85	1316.92	106.07	1344.01	133.17
Total kernel time	1215.18	1737.37	522.19	1348.04	132.71

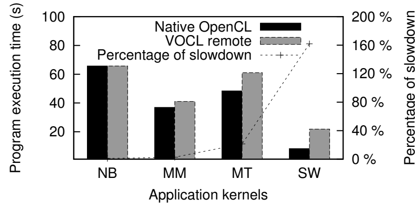

Fig. 5. Overhead of program execution time

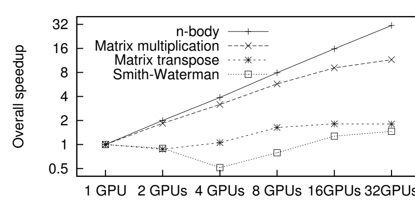

Fig. 6. Performance improvements with multiple VGPU utilized

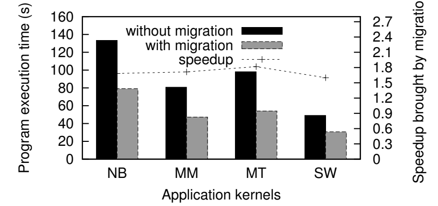

Fig. 7. Performance improvements via load balancing

the host memory and the device memory. As data transfer is serialized to different GPUs, program execution still takes approximately the same amount of time as the single GPU case. For Smith-Waterman, as shown in the previous section, utilization of remote GPUs causes very large overhead. When part of the instances are computed on remote GPUs, it is possible that the overall performance is worse than that using a single local GPU.

D. Performance Impact of Load Balancing

In this experiment, we run two processes for each of the same four algorithms and map both VGPUs to the same physical GPU. In one scenario, no migration is performed and all the computation is on a single GPU. In the other scenario, one of the VGPUs is migrated to an idle GPU. As such two GPUs are used for the computation.

Figure 7 shows the performance improvements by the load rebalancing. As can be seen, with task migration enabled in the framework, performance is improved for all four algorithms. Specifically, speedup of the matrix multiplication is 1.7 times; n-body is 1.9 times faster; matrix transpose is 1.7 times faster; and Smith-Waterman is 1.4 times faster. From these results, we can see that migration is very useful when load imbalance exists across physical GPUs.

VII. SUMMARY AND REMAINING WORK

In this work, we propose the VOCL framework to support the transparent virtualization of GPUs, which in turn can generalize the utilization of GPUs in large-scale heterogeneous computing systems. We implement the framework and the various optimization techniques to reduce the overhead caused by the framework. We also extend the VOCL framework to support live migration of virtual GPUs across physical GPUs.

However, VOCL is still in its early stage and just provides the basic functionalities for virtualization. There are still several problems to be followed. 1) Load metrics on physical GPUs. To achieve load balance, we need a metric to effectively measure the load on each physical GPU. The challenge is that different kernels have different computation and memory access properties and the computation load is also affected by problem size. 2) Based on the metric, we will propose some resource management strategies, which can allocate GPU resources to tasks to achieve the best performance and/or energy consumption.

ACKNOWLEDGEMENTS

Many thanks to Pavan Balaji for his vision on VOCL and for mentoring Shucai Xiao as a summer intern at ANL.

This work is supported in part by the U.S. Department of Energy under Contract DE-AC02-06CH11357 and the NSF CNS-CSR grant 0916719.

REFERENCES

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

[2] A. Athalye, N. Baliga, P. Bhandarkar, and V. Venkataraman. GAMPI is GPU Aware MPI - A CUDA Based Approach, May 2010. http://www.cs.utexas.edu/~pranavb/html/index.html.

[3] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Toward Efficient Support for Multithreaded MPI Communication. In *Proc. of the 15th EuroPVM/MPI*, September 2008.

[4] A. Barak and A. Shiloh. The MOSIX Virtual OpenCL (VCL) Cluster Platform. In *Proc. Intel European Research and Innovation Conference*, October 2011.

[5] J. Duato, F. D. Igual, R. Mayo, A. J. Pena, E. S. Quintana-Orti, and F. Silla. An Efficient Implementation of GPU Virtualization in High Performance Clusters. In *Lecture Notes in Computer Science*, volume 6043, pages 385–394, 2010.

[6] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications. In *Proc. of IPDPS*, May 2011.

[7] Khronos OpenCL Working Group. The OpenCL Specification, June 2010. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[8] Message Passing Interface Forum. The Message Passing Interface (MPI) Standard.

[9] Network-Based Computing Laboratory. MVAPICH2. http://mvapich.cse.ohio-state.edu/overview/mvapich2.

[10] NVIDIA. NVIDIA CUDA Programming Guide-3.2, November 2010. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[11] NVIDIA. High Performance Computing - Supercomputing with Tesla GPUs. 2011. http://www.nvidia.com/object/tesla_computing_solutions.html.

[12] S. Xiao, A. Aji, and W. Feng. On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit. In *Proc. of the 15th ICPADS*, December 2009.

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

[14] 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 the 1st Innovative Parallel Computing (InPar)*, May 2012.