VOCL: An Optimized Environment for Transparent Virtualization of Graphics Processing Units

Shucai Xiao[1] Pavan Balaji[2] Qian Zhu[3] Rajeev Thakur[2] Susan Coghlan[4]
Heshan Lin[1] Gaojin Wen[5] Jue Hong[5] Wu-chun Feng[1]
[1]Dept. of Computer Science, Virginia Tech. {shucai, hlin2, wfeng}@vt.edu
[2]Math. and Comp. Sci. Div., Argonne National Lab. {balaji, thakur}@mcs.anl.gov
[3]Accenture Technology Labs, qian.zhu@accenture.com
[4]Leadership Comp. Facility, Argonne National Lab. smc@alcf.anl.gov
[5]Shenzhen Inst. of Adv. Tech., Chinese Academy of Sciences. {gj.wen,jue.hong}@siat.ac.cn

ABSTRACT

Graphics processing units (GPUs) have been widely used for general-purpose computation acceleration. However, current programming models such as CUDA and OpenCL can support GPUs only on the local computing node, where the application execution is tightly coupled to the physical GPU hardware. In this work, we propose a virtual OpenCL (*VOCL*) framework to support the *transparent* utilization of local or remote 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. The proposed framework requires no source code modifications. We also propose various strategies for reducing the overhead caused by data communication and kernel launching and demonstrate about 85% of the data write bandwidth and 90% of the data read bandwidth compared to data write and read, respectively, in a native nonvirtualized environment. We evaluate the performance of VOCL using four real-world applications with various computation and memory access intensities and demonstrate that compute-intensive applications can execute with negligible overhead in the VOCL environment.

Keywords

Graphics Processing Unit (GPU), Transparent Virtualization, OpenCL

1. INTRODUCTION

General-purpose graphics processing units (GPGPUs or GPUs) are becoming increasingly popular as accelerator devices for core computational kernels in scientific and enterprise computing applications. The advent of programming models such as NVIDIA's CUDA [21], AMD/ATI's Brook+ [1], and Open Computing Language (OpenCL) [15] has further accelerated the adoption of GPUs by allowing many applications and high-level libraries to be ported to them [17, 19, 23, 26]. While GPUs have heavily proliferated into high-end computing systems, current programming models require each computational node to be equipped with one or more local GPUs, and application executions are tightly coupled to the physical GPU hardware. Thus, any changes to the hardware (e.g., if it needs to be taken down for maintenance) require the application to stall.

Recent developments in virtualization techniques, on the other hand, have advocated decoupling the application view of "local hardware resources" (such as processors and storage) from the physical hardware itself. That is, each application (or user) gets a "virtual independent view" of a potentially shared set of physical resources. Such decoupling has many advantages, including ease of management, ability to hot-swap the available physical resources on demand, improved resource utilization, and fault tolerance.

For GPUs, virtualization technologies offer several benefits. GPU virtualization can enable computers without physical GPUs to enjoy *virtualized* GPU acceleration ability provided by other computers in the same system. Even in a system where all computers are configured with GPUs, virtualization allows allocating more GPU resources to applications that can be better accelerated.

However, with the current GPU programming model implementations, such virtualization is not possible. To address this situation, we have investigated the role of accelerators such as GPUs in heterogeneous computing environments. Specifically, our goal is to understand the feasibility and efficacy of virtualizing GPUs in such environments, allowing for compute nodes to ***transparently and efficiently*** view remote GPUs as ***local virtual GPUs***. To this end, we propose a new implementation of the OpenCL programming model, called Virtual OpenCL, or ***VOCL***. The VOCL framework provides the OpenCL-1.1 API and it enables application programs to utilize all GPUs in the same way as local GPUs. VOCL internally calls native OpenCL functions for local GPUs and it calls the Message Passing Interface (MPI) [18] for data transfer across different machines. VOCL also utilizes several techniques, including argument caching and data transfer pipelining, to improve performance.

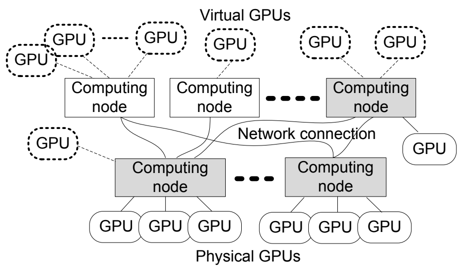


Figure 1: Transparent GPU virtualization

We note that this work does not deal with using GPUs on virtual machines, which essentially provide operating system-level or even lower-level virtualization techniques (that is, full or paravirtualization). Instead, it deals with user-level virtualization of the GPU devices themselves. Unlike full or para-virtualization using virtual machines, VOCL does not handle security and operating system-level access isolation. However, it does provide similar usage and management benefits, and the added benefit of being able to transparently utilize remote GPUs. As illustrated in Figure 1, VOCL allows a user to construct a virtual system that has, for example, 32 virtual GPUs, even though no physical machine in the entire system might have 32 collocated physical GPUs.

We describe here the VOCL framework as well as the optimizations to improve its performance, which is of great importance to virtual GPUs since performance improvements brought by the GPUs would otherwise be killed by overhead involved in the virtualization. We also present a detailed evaluation of the framework. This includes microbenchmark evaluation measuring data transfer overheads to and from GPUs associated with such virtualization. Also provided is a detailed profiling of overheads for various OpenCL functions. We evaluate our framework with several real application kernels, including SGEMM/DGEMM, N-body computations, matrix transpose kernels, and the computational biology Smith-Waterman application. We observe that for compute-intensive kernels (high ratio of computation required to data movement between host and GPU), VOCL's performance differs from native OpenCL's performance by only a small percentage. However, for kernels that are not compute-intensive (low ratio of computation required to data movement between host and GPU) and where the PCI-Express bus connecting the host processor to the GPU is already a bottleneck, such virtualization does have some impact on performance, as expected.

The rest of the paper is organized as follows. Section 2 provides an overview of the OpenCL programming model and the MPI standard. Sections 3 and 4 describe the VOCL framework, its implementation challenges, and the various performance optimization techniques we used. Section 5 presents the performance evaluation. Section 6 describes related work, and Section 7 presents our conclusions.

2. BACKGROUND

In this section, we provide a brief overview of the OpenCL programming model.

OpenCL [15] is a framework for programming heterogeneous computing systems. It provides functions to define and control the programming context for different platforms. It also includes a C-based programming language for writing *kernels* to be executed on different platforms such as the GPU, CPU, and Cell Broadband Engine (Cell/BE) [7]. A kernel is a special function called on the host and executed on the aforementioned device. Usually, the data-parallel and compute-intensive parts of applications are implemented as kernels to take advantage of the computational power of the various accelerators. A kernel consists of a few work groups, with each work group consisting of a few work items. In OpenCL, kernels use a different memory space from the host program and OpenCL provides API functions for data copy between host memory and device memory for kernel execution on the GPU.

In general, execution of a typical OpenCL program includes the following steps. The program first allocates OpenCL objects such as contexts, programs, command queues, device memories, etc, on the device. Then it copies kernel inputs to device memory and performs computation on the device. After kernel execution is completed, the program copies the results back to the host memory for program output. Finally, the program releases the OpenCL objects. Compared to programs executed on the traditional CPU processors, data copy is needed between host memory and device memory in OpenCL programs. As to the data-copy time, in compute-intensive programs, it can be negligible; while in programs requiring more data movement, such data copy can occupy a significant portion of the total program execution time.

The current implementations of the OpenCL programming model only provide capabilities to utilize accelerators installed locally on a compute node.

3. VIRTUAL OPENCL ENVIRONMENT

VOCL consists of the VOCL library on the local node and a VOCL proxy process on each remote node, as shown in Figure 2. The VOCL library exposes the OpenCL API to applications and is responsible for sending information about the OpenCL calls made by the application to the VOCL proxy using MPI, and returning the proxy responses to the application. The VOCL proxy is essentially a service provider for applications, allowing them to utilize GPUs remotely. The proxies are expected to be set up initially (for example, by the system administrator) on all nodes that would be providing virtual GPUs to potential applications. 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. When an application wants to use a virtual GPU, its corresponding VOCL library would connect to the appropriate proxy, utilize the physical GPUs associated with the proxy, and disconnect when it is done.

We chose OpenCL as the programming model for two reasons. First, OpenCL provides general support for multiple accelerators (including AMD/ATI GPUs, NVIDIA GPUs, Intel accelerators, and the Cell/BE), as well as for general-purpose multicore processors. By supporting OpenCL, our VOCL framework can support transparent utilization of varieties of remote accelerators and multicore processors. Second, OpenCL is primarily based on a library-based interface, rather than a compiler-supported user interface such as CUDA. Thus, a runtime library can easily implement the OpenCL interface without requiring to design a new compiler.

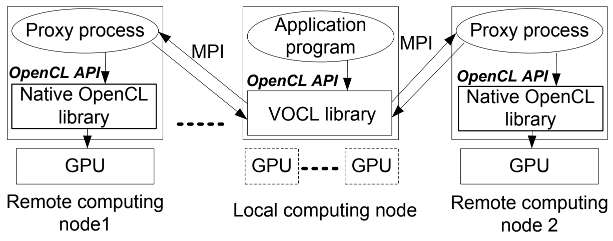


Figure 2: Virtual OpenCL framework

3.1 VOCL Library

VOCL is compatible with the native OpenCL implementation available on the system with respect to its abstract programming interface (API) as well as its abstract binary interface (ABI). Specifically, since the VOCL library presents the OpenCL API to the user, all OpenCL applications can use it without any source code modification. At the same time, VOCL is built on top of the native OpenCL library available on the system and utilizes the same OpenCL headers on the system. Thus, applications that have been compiled with the native OpenCL infrastructure need only to be relinked with VOCL and do not have to be recompiled. Furthermore, if the native OpenCL library is a shared library and the application has opted to do dynamic linking of the library (which is the common usage mode for most libraries and default linker mode for most compilers), such linking can be performed at runtime just by preloading the VOCL library through the environment variable `LD_PRELOAD`.

The VOCL library is responsible for managing all virtual GPUs exposed to the application. Thus, if the system has multiple nodes, each equipped with GPUs, the VOCL library is responsible for coordinating with the VOCL proxy processes on all these nodes. Moreover, the library should be aware of the locally installed physical GPUs and call the native OpenCL functions on them if they are available.

3.1.1 VOCL Function Operations

When an OpenCL function is called, VOCL performs the following operations.

- Check whether the physical GPU to which a virtual GPU is mapped is local or remote.
- If the virtual GPU is mapped to a local physical GPU, call the native OpenCL function and return.
- If the virtual GPU is mapped to a remote physical GPU, check whether the communication channels between applications and proxy processes have been connected. If not, call the `MPI_Comm_connect()` function to establish the communication channel.
- Pack the input parameters of the OpenCL functions into a structure and call `MPI_Isend()` to send the message (referred to as *control message*) to the VOCL proxy. Here, a different MPI message tag is used for each OpenCL function to differentiate them.
- Call `MPI_Irecv()` to receive output and error information from the proxy process, if necessary.
- Call `MPI_Wait()` when the application requires completion of pending OpenCL operations (e.g., in blocking OpenCL calls or flush calls).

3.1.2 VOCL Abstraction

In OpenCL, kernel execution is performed within a host-defined context, which includes several objects such as devices, program objects, memory objects, command queues, and kernels. A node can contain multiple devices; therefore, objects such as command queues need to be mapped onto a specific device before computation can be performed. As such, in environments where a node is equipped with multiple physical GPUs, to do this mapping, the OpenCL library includes additional information in each object that lets it identify which physical GPU the object belongs to. For example, when OpenCL returns a command queue (i.e., `cl_command_queue`), this object internally has enough information to distinguish which physical GPU the command queue resides on.

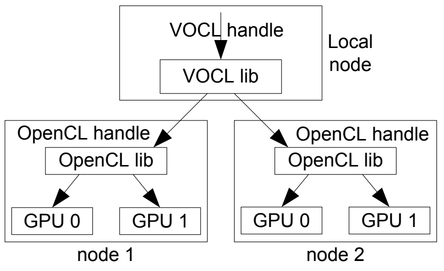


Figure 3: VOCL abstraction

With VOCL, since the physical GPUs might be located on multiple physical nodes, the VOCL library needs to coordinate with the native OpenCL library on multiple nodes (through the VOCL proxy). To this end, we propose another level of abstraction for the OpenCL object management. Specifically, we define an equivalent *VOCL object* for each OpenCL object as shown in Figure 3. Then for each OpenCL object, its handle is translated to a *VOCL handle* with a unique value even OpenCL handles share the same value. Together with the native OpenCL handle, the VOCL object contains additional information to identify which physical node (and thus, which native OpenCL library instance) an OpenCL object corresponds to.

When a VOCL handle is used, VOCL will first translate it to the OpenCL handle. Then it sends the OpenCL handle to the corresponding computing node based on the MPI communication information contained in the VOCL object. However, care must be taken when a memory handle is used as a kernel input. As we know, a kernel argument is set by calling the function
`clSetKernelArg(cl_kernel kernel, cl_uint arg_index, size_t arg_size, const void *arg_value)` and the argument `arg_value` should be a pointer to an OpenCL memory handle. But with the VOCL abstraction, `arg_value` is a pointer to a VOCL handle, which is invalid for kernel execution. Moreover, from the arguments in the function `clSetKernelArg()`, it is impossible to figure out whether a kernel argument is a memory handle or not. To address this problem, we wrote a parser to parse the kernel source code and figure out the device memory arguments in the kernel[1]. As such, when `clSetKernelArg()` is called, VOCL can first translate a VOCL memory handle to an OpenCL memory handle based on the parser output. Then VOCL uses the OpenCL memory handle as input to the native OpenCL function `clSetKernelArg()`. In this way, kernel arguments can be set correctly.

3.2 VOCL Proxy

The VOCL proxy is responsible for (1) receiving connection requests from the VOCL library to establish communication channels with each application process, (2) receiving inputs from the VOCL library and executing them on its local GPUs, (3) sending output and error codes to the VOCL

[1]This approach needs the kernel source code to be available.

library, and (4) destroying the communication channels after the program execution has completed.

3.2.1 Managing Communication Channels

Communication channels between the VOCL library and VOCL proxy are established and destroyed dynamically. Each proxy calls `MPI_Comm_accept()` to wait for connection requests from the VOCL library. When such a request is received, a channel is established between them, which is referred to as the *control message channel*. Once the application has completed utilizing the virtual GPU, the VOCL library sends a termination message to the proxy. Then `MPI_Comm_disconnect()` is called by both the VOCL library and the VOCL proxy to terminate the communication channel.

In the VOCL framework, each application can utilize GPUs on multiple remote nodes. Similarly, GPUs on a remote node can be used by multiple applications simultaneously. In addition, applications may start their execution at different times. Thus, the proxy should be able to accept connection request from application processes at any arbitrary time. To achieve this, we use an additional thread at the proxy that continuously waits for new incoming connection requests. When a connection request is received, this thread updates the communication channels such that messages sent by the VOCL library can be handled by the main proxy process, and the thread waits for the next connection request.

3.2.2 Handling Native OpenCL Function Calls

Once a control message channel is established between the VOCL proxy and the VOCL library, the proxy preposts buffers to receive control messages from the VOCL library (using nonblocking MPI receive operations). Each VOCL control message is only a few bytes large, so the buffers are preposted with a fixed maximum buffer size that is large enough for any control message. When a control message is received, it contains information on what OpenCL function needs to be executed as well as information about any additional input for the OpenCL function call. If any data needs to be transferred for the function call, the proxy posts additional receive buffers to receive this data from the VOCL library. It is worth noting that the actual data communication happens on a separate communicator to avoid conflicts with control messages; this communicator will be referred to as the *data channel*.

Specifically, for each control message, the proxy process performs the followed steps.

- When a control message is received, the corresponding OpenCL function is determined based on the message tag. Then the proxy process decodes the received message according to the OpenCL function. Depending on the specific OpenCL function, other messages may be received in the data channel as inputs for the function execution.
- Once all of the input data is available, the native OpenCL function is executed.
- Once the native OpenCL function completes, the proxy packs the output and sends it back to the VOCL library.
- If dependencies exist across different functions or if the current function is a blocking operation, the proxy waits for the current operation to finish and the result sent back to the VOCL library before the next OpenCL function is processed. On the other hand, if the OpenCL function is nonblocking, the proxy will send out the return code and continue processing other functions.
- Another nonblocking receive will be issued to replace the processed control message to receive additional control messages.

Since the number of messages received is not known beforehand, the proxy process uses an infinite loop waiting to receive messages. It is terminated by a message with a specific tag. Once the termination message is received, the loop is ended.

4. VOCL OPTIMIZATIONS

The VOCL framework enables applications to utilize all GPUs in the same way to accelerate their execution. However, we should reduce the virtualization overhead to as little as possible, since otherwise, performance improvements brought by using virtual GPUs would be killed by such overhead. Since VOCL internally calls native OpenCL functions for local GPUs, overhead of using local GPUs is expected to be very little as shown later. Thus in the following, our optimization focuses on reducing the overhead of using remote GPUs.

Compared to local GPUs, function calls on remote GPUs need one more phase of data transfer between the local node and the remote node. Specifically, if a function is executed without reading or writing device memory, data transfer is performed only between local node and remote node for remote GPUs. On the other hand, if device memory reads or writes are performed in a function call, we should consider data transfer between local host memory and remote device memory, which includes two phases – between local host memory and remote host memory and between remote host memory and remote GPU memory. In general, overhead of data transfer depends on the amount of data to be transferred, network bandwidth, as well as the times of data transfer.

In a typical OpenCL program, API functions for allocating and releasing OpenCL objects are called only a few times. Inputs and outputs of these functions are of tens of bytes. As a result, overhead of these functions is negligible in practice. But functions related to kernel execution (GPU memory read/write, kernel argument setting, and kernel launch) can cause significant overhead for program execution in some scenarios. According to the work of Gregg et al. [14], even when local GPUs are used, data transfer between host memory and device memory can be the bottleneck, which can make kernel execution take 2 to 50x longer than the GPU processing alone. With one more phase of data transfer for the remote GPU utilization, it is expected that data transfer will cause more overhead to program execution. Thus, optimizing such data transfer is of great importance in the remote GPU utilization.

To reduce these overheads, we have implemented three optimizations: (1) kernel argument caching to reduce the times of data transfer; (2) data transfer pipelining to improve the bandwidth between local host memory and GPU memory; and (3) modifications to error handling.

4.1 Kernel Argument Caching

When remote GPUs are used, execution of functions without accessing GPU memory needs the data transfer only between local node and remote node and the amount of data is of tens of bytes in general. If these functions are called only a few times (e.g., OpenCL object allocation and release), the data transfer overhead involved in remote GPU utilization can be ignored. But if a function is called thousands of times, the overhead can be very large. One such function is the kernel argument set function `clSetKernelArg()`, which can be called thousands of times in some applications.

Table 1: Overhead (in ms) of kernel execution for utilization of remote GPUs

Function Name	Runtime Local	Runtime Remote	Overhead
clSetKernelArg	4.33	420.45	416.12
clEnqueueND-RangeKernel	1210.85	1316.92	106.07
Total time	1215.18	1737.37	522.19

In Table 1, we compare the kernel execution overhead for VOCL (using a remote GPU) vs. the native OpenCL library for aligning 6K base-pair sequences using the Smith-Waterman application [25, 27] on an NVIDIA Tesla M2070 GPU with the QDR InfiniBand as the network connection between different nodes. In this example, `clSetKernelArg()` has an overhead of 416.12 ms, which is 4x more than the kernel execution overhead (106.07ms). The reason is that this function is called more than 86,000 times (the kernel is called 12,288 times, and 7 parameters have to be set for each call). Though overhead of each function call is small, it causes significant overhead in total.

The basic idea of kernel argument caching is to combine the arguments to be transferred for multiple `clSetKernelArg()` calls. Rather than sending the arguments in each call of `clSetKernelArg()` to the proxy, we send kernel arguments to the remote node only once per kernel launch, no matter how many arguments are set in the kernel launch. Since all arguments should be set before the kernel is launched, we just cache all the arguments locally at the VOCL library. When the kernel launch function is called, the arguments are sent to the proxy. The proxy performs two steps on being notified of the kernel launch: (1) it receives the argument message and sets the individual kernel arguments, and (2) it launches the kernel.

Table 2 shows the execution time of Smith-Waterman for aligning the same 6K base-pair sequences using our kernel argument caching approach. As we can see in the table, the execution time of `clSetKernelArg()` decreases from 420.45 ms (Table 1) to 4.03 ms (Table 2). We notice a slight speedup compared with native OpenCL; the reason is that, in VOCL, arguments are cached in host memory and are not passed to the GPU immediately. We also notice a little more time spent on the kernel execution (increase from 1316.92 ms to 1344.01 ms). The reason is that kernel argument data need to be passed to the proxy and they have to be set for the kernel execution within this call. On the whole, the total kernel execution time decreases from 1737.37 ms to 1348.04 ms, or by $(1737.37 - 1348.04)/1737.37 = 22.41\%$.

Table 2: Overhead (in ms) of kernel execution with kernel argument caching optimization

Function Name	Native OpenCL	VOCL remote	Overhead
clSetKernelArg	4.33	4.03	-0.30
clEnqueueND-RangeKernel	1210.85	1344.01	133.17
Total time	1215.18	1348.04	132.71

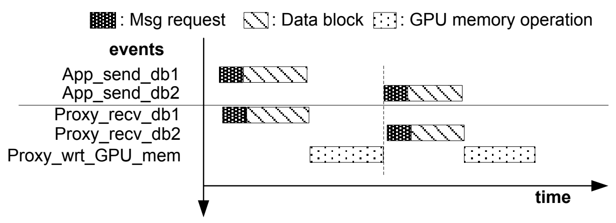
4.2 Data Transfer Pipelining

Two types of data need to be transferred between the VOCL library and the VOCL proxy for remote GPU utilization. The first type is the input arguments to OpenCL functions without GPU memory accesses involved; This type of data is transferred from the local host memory to the remote host memory. The size of such input arguments is generally of a few hundred bytes and the transfer cannot be started in one function until execution of its previous functions is completed. Their data transfers cause negligible overhead and pipelining them brings no useful benefits for program execution.

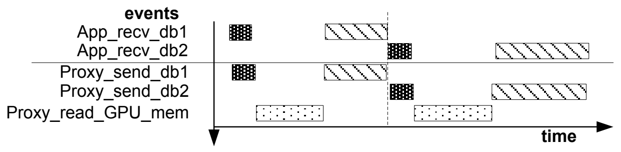
The second type is the GPU memory accesses, in which data are transferred from the local host memory to the remote GPU memory. This type of data transfer has two stages: (1) between the VOCL library and the VOCL proxy and (2) between the VOCL proxy and the GPU. In a naive implementation of VOCL, these two stages would be serialized and buffers to store the data are dynamically allocated and released in the proxy. Such an implementation, though simple, has two primary problems. First, there is no pipelining of the data transfer between the two stages. In another word, for each data chunk, the second stage can be started only after the first stage is finished. Moreover, transfer of a data chunk cannot be started until transfers of its previous data chunks are completed as shown in Figure 4. Second, since the temporary buffer used for storing data in the proxy is dynamically allocated and freed, this buffer is not statically registered with the local GPU device and has to be registered for each data transfer;[2] this causes additional loss of performance.

In order to optimize the data transfer overhead within VOCL, we designed a data pipelining mechanism with statically registered buffer pool for data storage in the proxy. Specifically, with pipeline, the first stage transfer of one data chunk can be done concurrently with the second stage transfer of another as shown in Figure 5. As to the buffer pool, each buffer is of size B bytes and is statically allocated and maintained in the proxy. As such, it does not encounter the buffer allocation or buffer registration overheads that we face in the nonpipelined approach. When the VOCL library needs to write some user data to the GPU, it segments this data into blocks of size at most B bytes, and transfers them to the VOCL proxy as a series of nonblocking sends. The VOCL proxy, on receiving each block, initiates the transfer of that data block to the GPU. The read operation is similar, but in the opposite direction. Figure 6 illustrates this

[2]All hardware devices require host memory to be registered, which includes pinning virtual address pages from swapping out, as well as caching virtual-to-physical address translations on the device.

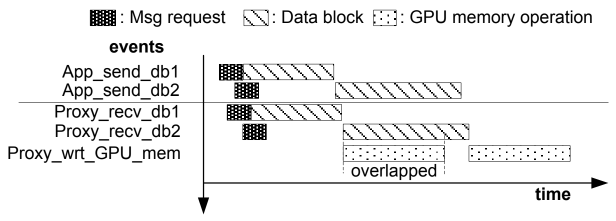


(a) Blocking Write to the GPU Memory

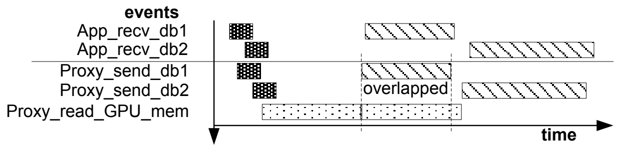


(b) Blocking Read from the GPU Memory

Figure 4: Blocking data transmission scenarios



(a) Nonblocking Write to the GPU Memory



(b) Nonblocking Read from the GPU Memory

Figure 5: Nonblocking data transmission scenarios

buffer pool model utilized in VOCL. In the example shown, data segments 1 and 2 are smaller than the maximum size of each buffer in the buffer pool. Thus, they are transmitted as contiguous blocks. Data segment 3, however, is larger than the maximum size, and hence is segmented into smaller blocks. Since the number of buffers in the pool is limited, we reuse buffers in a circular fashion. Note that before we reuse a buffer, we have to ensure that the previous data transfers (both from the network transfer as well as the GPU transfer) have completed. The number of buffers available in the pool dictates how often we need to wait for such completion, and thus has to be carefully configured.

We also note that, at the VOCL proxy, the tasks of sending/receiving data from the VOCL library, and writing/reading data from the GPU, are performed by two different threads. This allows each thread to perform data movement in a dedicated manner without having to switch between the network communication and GPU communication. This approach allowed us to further improve the data transfer performance by a few percent.

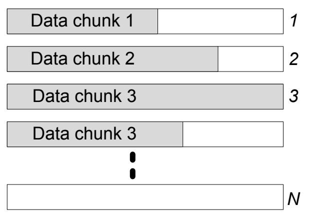


Figure 6: Buffer pool on proxy processes

4.3 Error Return Handling in Nonblocking Operations

Most OpenCL functions provide a return code to the user: either `CL_SUCCESS` or an appropriate error code. Such return values, however, are tricky for VOCL to handle, especially for nonblocking operations. The OpenCL specification does not define how error codes are handled for nonblocking operations. That is, if the GPU device is not functional, is a nonblocking operation that tries to move data to the GPU expected to return an error?

While the OpenCL specification does not describe the return value in such cases, current OpenCL implementations do return an error. For VOCL, however, since every OpenCL operation translates into a network operation, significant overhead can occur for nonblocking operations if the VOCL library has to wait until the OpenCL request is transferred over the network, a local GPU operation is initiated by the VOCL proxy, and the return code sent back.

We believe this is an oversight in the OpenCL specification, since all other specifications or user documents that we are aware of (including MPI, CUDA, and InfiniBand) do not require nonblocking operations to return such errors—the corresponding *wait-for-completion* operation can return these errors at a later time. In our implementation, therefore, we assume this behavior and return such errors during the corresponding *wait* operation.

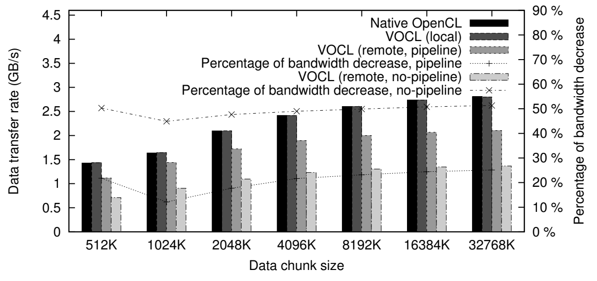
5. EXPERIMENTAL EVALUATION

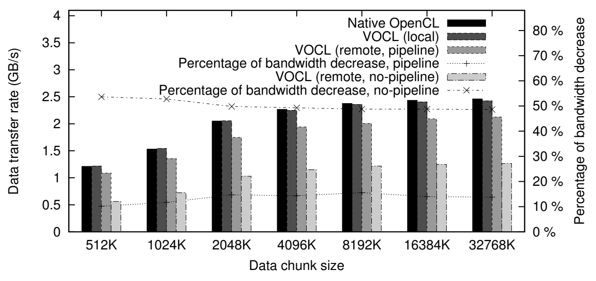
In this section, we evaluate the efficiency of the proposed VOCL framework. First, we analyze the overhead of individual OpenCL operations with VOCL and the bandwidth increase brought by the data transfer pipelining. Then, we quantitatively evaluate the VOCL framework with several application kernels: SGEMM/DGEMM, matrix transpose, n-body [22], and Smith-Waterman [25, 27].

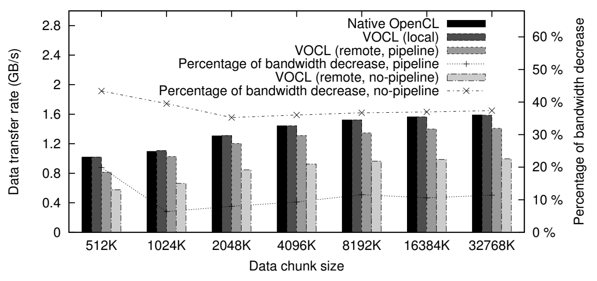
The compute nodes used for our experiments are connected with QDR InfiniBand. Each node is installed with two Magny-cours AMD CPUs with host memory of 64 GB and two NVIDIA Tesla M2070 GPUs each with 6 GB global memory. The two GPU cards are installed on two different PCIe slots, one of which shares the PCIe bandwidth with the InfiniBand adapter as shown in Figure 7. The computing nodes are installed with the Centos Linux operating system and the CUDA 3.2 toolkit. We use the MVAPICH2 [20] MPI implementation. Each of our experiments was conducted three times and the average is reported.

5.1 Microbenchmark Evaluation

In this section, we study the overhead of various individual OpenCL operations using the SHOC benchmark suite [8] and a benchmark suite developed within our group.


(a) Bandwidth from Host Memory to Device Memory (Local transfer is from **CPU1** to **GPU1** and remote transfer is from **CPU3** to **GPU1**.)


(b) Bandwidth from Device Memory to Host Memory (Local transfer is from **GPU1** to **CPU1** and remote is from **GPU1** to **CPU3**.)


(c) Bandwidth from Host Memory to Device Memory (Local transfer from **CPU1** to **GPU0** and remote transfer is from **CPU3** to **GPU0**.)

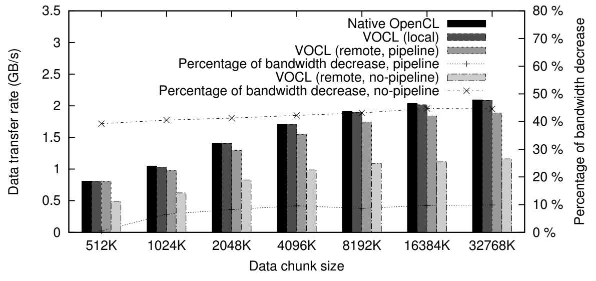

(d) Bandwidth from Device Memory to Host Memory (Local transfer is from **GPU0** to **CPU1** and remote transfer is from **GPU0** to **CPU3**.)

Figure 8: Bandwidth between host memory and device memory for nonblocking data transfer

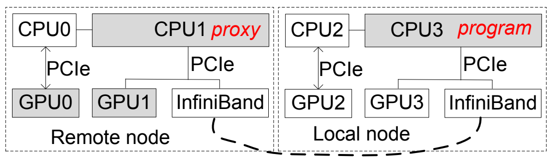


Figure 7: GPU configuration and the scenario for the bandwidth test

5.1.1 Initialization/Finalization Overheads

In this section, we study the performance of initializing and finalizing OpenCL objects within the VOCL framework. Overhead of these functions are mainly caused by the transfer of function parameters as described in Section 4.2. These functions and their overhead are listed in Table 3. As we can notice in the table, for most functions, the overhead caused by VOCL is minimal. The one exception to this is the `clGetPlatformIDs()` function which has the overhead of 402.68 ms. The reason for this overhead is that `clGetPlatformIDs()` is typically the first OpenCL function executed by the application in order to query the platform. Therefore, the VOCL framework performs most of its initialization during this function, including setting up the MPI communication channels as described in Section 3.

The total overhead caused by all the initialization and finalization functions together is a few hundred milliseconds. However, this overhead is a one-time overhead unrelated to the total program execution time. Thus, in practice, for any program that executes for a reasonably long time (e.g., a few tens of seconds), these overheads play little role in the noticeable performance of VOCL.

Table 3: Overhead of OpenCL API Functions for Resource Initialization/Release (Unit: ms)

Function Name	Native OpenCL	VOCL (remote)	Overhead
clGetPlatformIDs	50.84	453.52	402.68
clGetDeviceIDs	0.002	0.173	0.171
clCreateContext	253.28	254.11	0.83
clCreateCommandQueue	0.018	0.044	0.026
clCreateProgramiWithSource	0.009	0.042	0.033
clBuildProgram	488.82	480.90	-7.92
clCreateBuffer	0.025	0.051	0.026
clCreateKernel	0.019	0.030	0.011
clReleaseKernel	0.003	0.012	0.009
clReleaseMemObj	0.004	0.011	0.007
clReleaseProgram	0.375	0.291	-0.084
clReleaseCmdQueue	0.051	0.059	0.008
clReleaseContext	177.47	178.43	0.96

5.1.2 Performance of Kernel Execution on the GPU

Kernel execution on the GPU would be the same no matter which host processor launches the kernel. Thus, utilizing remote GPUs via VOCL should not affect the kernel execution on the GPU card. By evaluating VOCL with the SHOC microbenchmark [8], we verified that the maximum flops, on-chip memory bandwidth, and off-chip memory bandwidth are the same as native OpenCL. These results are not provided here because they show no useful difference in performance.

5.1.3 Data Transfer between Local Host Memory and GPU Memory

In this section, we measure the data transfer bandwidth increase brought by our pipelining mechanism and the bandwidth achieved for GPU write and read operations using VOCL. The experiment is performed with different message sizes. For each message size, a window of 32 messages is issued in a nonblocking manner, followed by a flush operation to wait for their completion. The bandwidth is calculated as the amount of data transferred per second. A few initial "warm up" iterations are skipped from the timing loop.

Figure 8 shows the performance of native OpenCL, VOCL when using a local GPU (legend "VOCL (local)"), VOCL with the pipelining mechanism when using a remote GPU (legend "VOCL (remote, pipeline)"), and VOCL without pipelining for remote GPUs (legend "VOCL (remote, no-pipeline)"). Native OpenCL uses only the local GPU. Two scenarios are shown—bandwidth between CPU3 and GPU0 (Figures 8(c) and 8(d)) and between CPU3 and GPU1 (Figures 8(a) and 8(b)); see Figure 7. In our experiments, the VOCL proxy is bound to CPU1. For native OpenCL, the application process is bound to CPU1.

As shown in the figures, when remote GPUs are used, the pipelining mechanism significantly increases the data transfer bandwidth for all message sizes. Compared to the bandwidth without pipelining, data transfer bandwidth is almost doubled. This is important for GPU computing since data transfer between host memory and device memory can cause large overhead in some applications [14].

Compare the bandwidth in the above various scenarios, VOCL-local has no degradation in performance as compared to native OpenCL, as expected. VOCL-remote, however, has some degradation in performance because of the additional overhead of transmitting data over the network. As the message size increases, the bandwidth increases for native OpenCL as well as VOCL (both local and remote). But VOCL-remote saturates at a bandwidth of around 10-25% lesser than that of native OpenCL if the pipelining mechanism is used. Comparing the bandwidth between GPU0 and GPU1, we notice that the absolute bandwidth of native OpenCL as well as VOCL (local and remote) is lesser when using GPU0 as compared to GPU1. The reason for this behavior is that data transfer between CPU1 and GPU0 requires additional hops compared to transfer between CPU1 and GPU1, causing some drop in performance. This lower absolute performance also results in lesser difference between VOCL-remote (with data transfer pipelining) and native OpenCL (10% performance difference, as compared to the 25% difference when transmitting from CPU1 to GPU1). The results for reading data from the GPU are similar.

We also note that the shared PCIe between the network adapter and GPU1 does not degrade performance because for most communication the direction of data transfer to/from the network and to/from the GPU does not conflict. Specifically, when the application is writing data to the GPU, the proxy needs to read the data from the network and write it to the GPU. Similarly, when the application is reading data from the GPU, the proxy needs to read the data from the GPU and write it to the network. Since PCIe is a bidirectional interconnect, data transfers in opposite directions do not share the bandwidth. This allows transfers to/from GPU1 to achieve a higher bandwidth as compared with GPU0. Consequently, the performance difference for VOCL is higher for GPU1 than for GPU0.

For the remaining results, we use GPU1 with data transfer pipelined because of the higher absolute performance it can achieve.

5.2 Evaluation with Application Kernels

In this section, we evaluate the efficiency of the VOCL framework using four application kernels: SGEMM/DGEMM, n-body, matrix transpose and Smith-Waterman. Table 4 shows the computation and memory access complexities for these four kernels. The first two kernels, SGEMM/DGEMM and n-body, can be classified as compute-intensive based on their computational requirements, while the other two require more data movement.

Table 4: Computation and the amount of memory transferred between host memory and device memory of the four applications. 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 amount of data transferred between host and device memories for kernel execution.

Application Kernels	Computation	Memory Size
SGEMM/DGEMM	$O\left(n^3\right)$	$O\left(n^2\right)$
N-body	$O\left(n^2\right)$	$O\left(n\right)$
Matrix transpose	$O\left(n^2\right)$	$O\left(n^2\right)$
Smith-Waterman	$O\left(n^2\right)$	$O\left(n^2\right)$

The difference in computational intensity of these four kernels is further illustrated in Figure 9, where the percentage of time spent on computation for each of these kernels is shown. As we can see in the figure, the n-body kernel spends almost 100% of its time computing. SGEMM/DGEMM spend a large fraction of time computing, and this fraction increases with increasing problem size. Matrix transpose spends a very small fraction of time computing. While Smith-waterman spends a reasonable amount of time computing (70-80%), most of the computational kernels it launches are very small kernels which, as we will discuss later, are harder to optimize because of the large number of small message transfers they trigger.

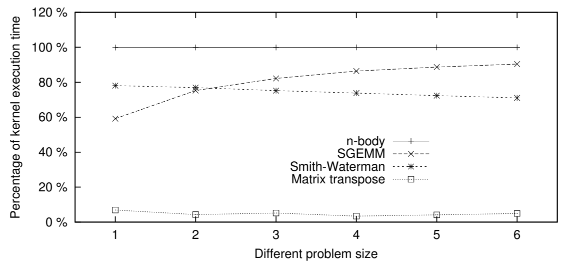
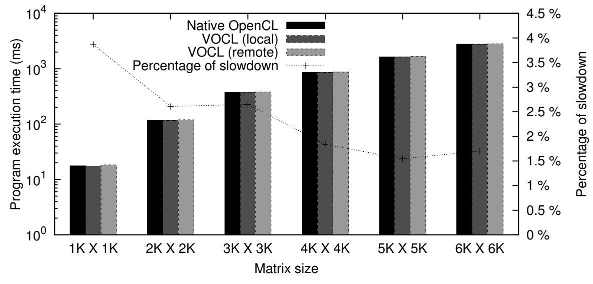
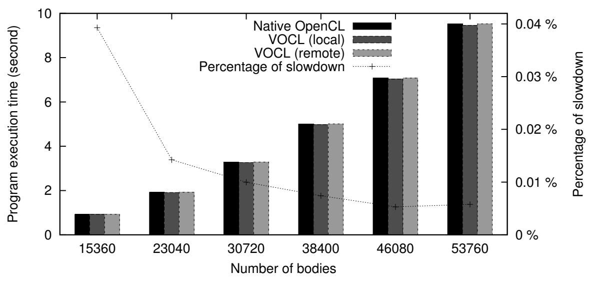


Figure 9: Percentage of the kernel execution time in the single precision case

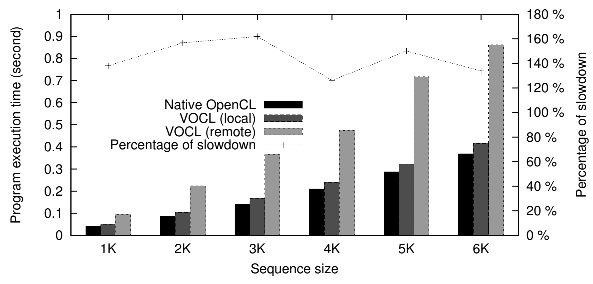
Next we evaluate the overhead of program execution time with different problem sizes. Recall that the program execution time in this experiment includes the data transfer time, kernel argument setting, and kernel execution. We ran both



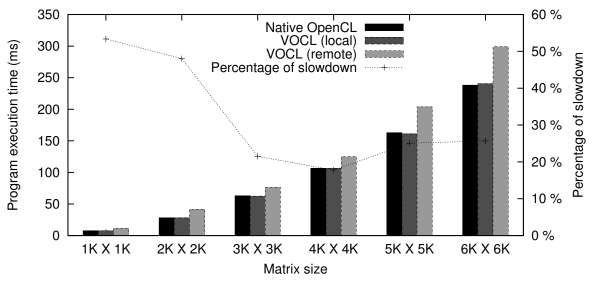
(a) SGEMM



(b) N-body



(c) Smith-Waterman



(d) Matrix Transpose

Figure 10: Overhead in total execution time for single-precision computations

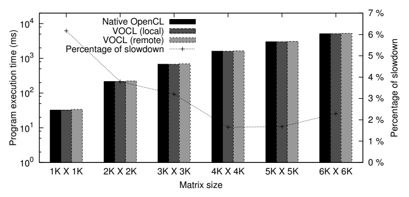
the single-precision and double-precision implementations of all application kernels, except Smith-Waterman since sequence alignment scores are usually stored as integers or single-precision floats in practice. We ran multiple problem instances in a nonblocking manner to pipeline data transfer and kernel execution. After we issue all nonblocking function calls, the OpenCL function `clFinish()` is called to ensure that all computation and data transfer has completed before measuring the overall execution time.

Figure 10 shows the performance and the overhead of the application kernels for single precision computations. We notice that the performance of native OpenCL is almost identical to that of VOCL-local; this is expected as VOCL does very little additional processing (e.g., translation between OpenCL and VOCL handles) in this case. For VOCL-remote, however, the performance depends on the application. For compute-intensive algorithms, the overhead of VOCL is very small; 1-4% for SGEMM and nearly zero for n-body. This is because for these applications the total execution time is dominated by the kernel execution. For SGEMM, we further notice that the overhead decreases with increasing problem size. This is because the computation time for SGEMM increases as $O(N^3)$ while the amount of data that needs to be transferred only increases as $O(N^2)$; thus, the computation time accounts for a larger percentage of the overall execution time for larger problem sizes as shown in Figure 9.

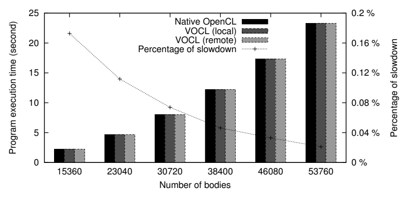
For algorithms that need more data movement, the overhead of VOCL is higher. For matrix transpose, for example, this is between 20-55%, which is expected because it spends a large fraction of its execution time in data transfer (based on Figure 9, matrix transpose spends only 7% of its time computing). With VOCL-remote, this data has to be transmitted over the network causing significant overhead. For Smith-Waterman, the overhead is much higher and closer to 150%. This is because of two reasons. First, since the VOCL proxy is a multi-threaded process, the MPI implementation has to be initialized to support multiple threads. It is well known in the MPI literature that multi-threaded MPI implementations can add significant overhead in performance, especially for small messages [3, 4, 9, 13]. Second, Smith-Waterman relies on a large number of kernel launches for a given amount of data [27]. For a 1K sequence alignment, for example, more than 2000 kernels are launched causing a large number of small messages to be issued, which, as mentioned above, cause a lot of performance overhead. We verified this by artificially initializing the MPI library in single-threaded mode and noticed that the overhead with VOCL comes down to around 35% by doing so.[3]

Figure 11 shows the performance and the overhead of the application kernels for double precision computations. The observed trends for double precision are nearly identical to the single-precision cases. This is because the amount of data transferred for double precision computations is double that of the single precision computations; and on the NVIDIA Tesla M2070 GPUs, the double precision computations are about twice as slow as the single precision computations. Thus, both the computation time and the data transfer time double and result in no relative difference. On other architectures, such as the older NVIDIA adapters where the double precision computations were much slower than the single precision computations, we expect this balance to change and the relative overhead of VOCL to reduce since time percentage of kernel execution will be higher than that on the Tesla M2070.

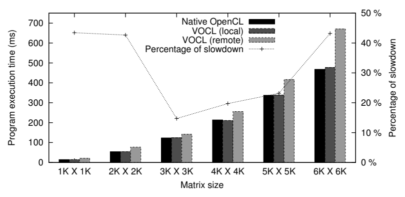
[3]Note that, in this case, the VOCL proxy can accept only one connection request each time it is started. After an application finishes its execution and disconnects the communication channel, we would need to restart the proxy process for the next run; a process that is unusable in practice. We only tried this approach to understand the overhead of using a multi-threaded vs. single-threaded MPI implementations.



(a) DGEMM



(b) N-body



(c) Matrix Transpose

Figure 11: Overhead in total execution time for double-precision computations

5.3 Multiple Virtual GPUs

OpenCL allows applications to query for the available GPUs and distribute their problem instances on them. Thus, with native OpenCL, an application can query for all the local GPUs and utilize them. With VOCL, on the other hand, the application would have access to all the GPUs in the entire system; thus, when the application executes the resource query function, it looks like it has a very large number of GPUs.

In this section, we perform experiments with a setup that has 16 compute nodes, each with 2 local GPUs; thus, with VOCL, it would appear to the applications that they have 32 local (virtual) GPUs. In this environment, the application can distribute its work on 32 GPUs instead of the 2 GPUs that it would use with the native OpenCL. Figure 12 shows the total speedup achieved with 1, 2, 4, 8, 16, and 32 virtual GPUs utilized. Same as the overhead evaluation in Section 5.2, we assign multiple problem instances to each GPU in the nonblocking way, then call the function `clFinish()` to wait for their completion. With one and two GPUs, only local GPUs are used. In other cases, two of the GPUs are local, and the remaining GPUs are remote.

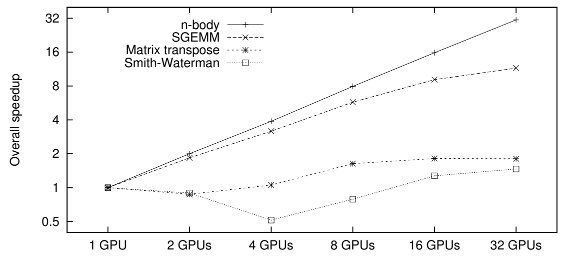


Figure 12: Performance improvement with multiple virtual GPUs utilized (single precision)

As shown in the figure, for compute-intensive applications such as SGEMM and n-body, significant speedup is achieved. For instance, with 32 GPUs, the n-body achieves a speedup of about 31-fold. For SGEMM, it is about 11.5-fold, which is due to the serialization of the data transfer over a single network link. For data-intensive applications such as matrix transpose and Smith-Waterman, on the other hand, almost no performance improvement is observed; in fact, the performance becomes worse in some cases. For the matrix transpose, the reason for this behavior is that it spends most of the execution time on data transfer between the host memory and the device memory. Since all data transfers share the same network link, program execution still needs about the same time as with that using a single GPU. As to the Smith-Waterman, as shown in the previous section, data transfer associated with remote GPUs causes large overhead to its execution. With large portion of the computation performed on remote GPUs, the overall performance can be worse than the single GPU case.

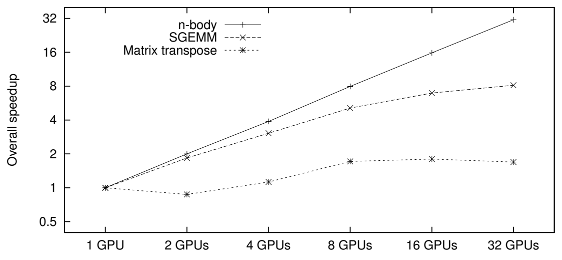


Figure 13: Performance improvement with multiple virtual GPUs utilized (double precision)

Figure 13 illustrates a similar experiment, but for double-precision computations. Again, we notice similar trends as single precision computations because there is no relative difference in data transfer time and computation time between the single-precision and double-precision applications, as explained above.

6. RELATED WORK

Several researchers have studied GPU virtualization and data transfer between computing nodes and GPUs.

Lawlor et al. proposed the cudaMPI library, which provides an MPI-like message-passing interface for data communication across different GPUs [16]. They compared the performance of different approaches for inter-GPU data communication and suggested various data transfer strategies. Their work is complementary to VOCL and can be adopted in our framework for efficient data transfer.

Athalye et al. designed and implemented a preliminary version of GPU-Aware MPI (GAMPI) for CUDA-enabled device-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. With such API functions, all GPUs, local or remote, in a cluster can be used in the same way. This work provides a general approach for using both local and remote GPUs. However, their solution requires invoking GPUs in a way different from the CUDA or OpenCL programming models, and thus is *nontransparent*. In comparison, our framework supports both local and remote GPUs without any source code modification.

Duato et al. [10, 11, 12] presented a GPU virtualization middleware that makes remote CUDA-compatible GPUs available to all compute nodes in a cluster. They implemented the software on top of TCP sockets to ensure portability over commodity networks. Similar to the work by Athalye et al., API functions have to be called explicitly for data transfer between the local node and the remote node. Thus, additional efforts are needed to modify the GPU programs to use their virtualization middleware. Further, this work requires all CUDA kernels to be separated into a different file, so this file can be shipped to the remote node and executed. This is a fundamental limitation of trying to utilize remote GPUs with the CUDA programming model, because of its dependency on a compiler. With OpenCL, on the other hand, the compilation of the computational kernel is embedded into the runtime library thus allowing such virtualization to be done transparently.

Shi et al. [24] proposed a framework that allows high performance computing applications in virtual machines to benefit from GPU acceleration, where a prototype system was developed on top of KVM and CUDA. This work considers OS-level virtualization of GPUs installed locally, and the overhead comes from the usage of virtual machines. Therefore, GPUs that can be used in vCUDA are restricted by local GPUs. In contrast, our VOCL framework provides the transparent utilization of both local and remote GPUs. Overhead in VOCL applies only to remote GPUs, which comes from data transfer between the local and remote nodes.

Barak et al. [6] provided a framework to transparently use cluster-based GPUs based on the MOSIX [5]. This framework implements the same functionality as VOCL. However, VOCL is distinct in the following aspects: 1) In MOSIX, an API proxy is used by each application on the local node for resource management to use all GPUs. As such, significant overhead is caused, even for local GPUs. In contrast, VOCL does not need such an API proxy and it calls the native OpenCL functions directly for local GPUs. As a result, the same performance as the native OpenCL library can be achieved on local GPUs. 2) The MOSIX framework does not consider the overhead of data transfer between host memory and device memory. However, data transfer can slow down the program execution on local GPUs by 2 to 50x in some applications [14]. For remote GPUs, it is expected that this overhead becomes even more and application performance can be further impacted. If the data transfer is not optimized, it is possible performance improvements brought by the GPU will be killed. To this end, our VOCL framework optimizes the data transfer between host memory and device memory and it can achieve 80% - 90% of the data transfer bandwidth of the native OpenCL. 3) The VOCL framework is designed to be a true virtualization framework, rather than simple wrappers to execute regular OpenCL functions. One aspect of these functionalities is its ability to perform live migration of "virtual GPUs" between different physical GPUs in a cluster.

In summary, the proposed VOCL framework provides a unique and interesting enhancement to the state-of-art in GPU virtualization.

7. CONCLUDING REMARKS

GPUs have been widely adopted to accelerate general-purpose applications. However, the current programming models, such as CUDA and OpenCL, can support usage of GPUs only on local computing nodes. In this work, we proposed an optimized environment to support the transparent virtualization of GPUs, which in turn allows applications to use local and remote GPUs as if they were installed locally. In this environment, we proposed several mechanisms to reduce the overhead for virtualization. Also, we studied the overhead of the VOCL framework using various microbenchmarks as well as four application kernels with various compute and memory access intensities.

8. ACKNOWLEDGEMENTS

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

9. REFERENCES

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

[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] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming. *International Journal of High Performance Computing Applications (IJHPCA)*, 24(1):49–57, 2009.

[5] A. Barak and A. Shiloh. The MOSIX Cluster Operating System for High-Performance Computing on Linux Clusters, Multi-Clusters, GPU Clusters and Clouds, 2011. A white paper.

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

[7] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and Its First Implementation – A Performance View. *IBM developerWorks*, Nov 2005.

[8] A. Danaliszy, G. Mariny, C. McCurdyy, J. S. Meredithy, P. C. Rothy, K. Spaffordy, V. Tipparajuy, and J. S. Vetter. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite, March 2010. `http://ft.ornl.gov/doku/shoc/start`.

[9] G. Dozsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur. Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems. In *Proc. of the 15th EuroMPI*, September 2010.

[10] 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.

[11] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Orti. rCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters. In *Proc. of International Conference on High Performance Computing and Simulation (HPCS)*, pages 224–231, June 2010.

[12] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Orti. Performance of CUDA Virtualized Remote GPUs in High Performance Clusters. In *Proc. of International Conference on Parallel Processing (ICPP)*, September 2011.

[13] D. Goodell, P. Balaji, D. Buntinas, G. Dozsa, W. Gropp, S. Kumar, B. R. de Supinski, and R. Thakur. Minimizing MPI Resource Contention in Multithreaded Multicore Environments. In *Proc. of the IEEE International Conference on Cluster Computing*, September 2010.

[14] C. Gregg and K. Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2011.

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

[16] O. S. Lawlor. Message Passing for GPGPU Clusters: cudaMPI. In *IEEE Cluster PPAC Workshop*, August 2009.

[17] S. A. Manavski and G. Valle. CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment. *BMC Bioinformatics*, March 2008.

[18] Message Passing Interface Forum. The Message Passing Interface (MPI) Standard. `http://www.mcs.anl.gov/research/projects/mpi`.

[19] Y. Munekawa, F. Ino, and K. Hagihara. Design and Implementation of the Smith-Waterman Algorithm on the CUDA-Compatible GPU. In *Proc. of the 8th IEEE International Conference on BioInformatics and BioEngineering*, pages 1–6, October 2008.

[20] 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`.

[21] 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`.

[22] L. Nyland, M. Harris, and J. Prins. Fast N-Body Simulation with CUDA. *GPU Gems*, 3:677–695, 2007.

[23] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W. W. Hwu. GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications. In *Proc. of the Conference on Computing Frontiers*, pages 273–282, May 2008.

[24] L. Shi, H. Chen, and J. Sun. vCUDA: GPU Accelerated High Performance Computing in Virtual Machines. *IEEE Transactions on Computers*, 99, 2011.

[25] T. Smith and M. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[26] G. M. Striemer and A. Akoglu. Sequence Alignment with GPU: Performance and Design Challenges. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2009.

[27] S. Xiao, A. Aji, and W. Feng. On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit. In *Proc. of the International Conference of Parallel and Distributed System*, December 2009.