

The Virtual OpenCL (VCL) Cluster Platform

Amnon Barak and Amnon Shiloh

<http://www.MOSIX.org>

Abstract—Heterogeneous computing systems can dramatically increase the performance of parallel applications on clusters. Currently, applications that utilize GPU and APU devices, run their device-specific code only on devices of the same computer where the application runs. This paper presents the Virtual OpenCL (VCL) cluster platform that can run unmodified OpenCL applications transparently on clusters with many devices. VCL provides an OpenCL platform in which all the cluster devices are seen as if they are located in the hosting-node. This platform benefits OpenCL applications that can use many devices concurrently. Applications written for VCL benefit from the reduced programming complexity of a single computer, the availability of shared-memory, multi-threads and lower level parallelism, as in openMP, as well as concurrent access to devices in many nodes, as in MPI. The paper presents the components of VCL and its performance.

Index Terms—APU and GPU computing, heterogeneous clusters, OpenCL, parallel applications, stream computing

I. INTRODUCTION

Heterogeneous computing systems provide an opportunity to dramatically increase the performance of parallel and High-Performance Computing (HPC) applications on clusters, by combining traditional multi-core CPUs, general-purpose GPUs and the emerging Accelerator Processing Units (APUs), such as AMD's Fusion. Currently, applications that utilize GPU devices, run their device-specific (kernel) code only on local devices of the (hosting) computer where they run. Without an adequate run-time environment, it is usually difficult to split an application so that it can run on many computers in parallel.

The main programming paradigms for developing parallel HPC applications are MPI [1] and OpenMP [2]. Development of parallel applications is usually simpler in openMP than in MPI, mainly because OpenMP supports shared-memory, multi-threads and fine granularity. While traditional OpenMP implementations run applications on a single computer, we recently demonstrated [4] that OpenMP can be extended to use a heterogeneous (CPU and GPU) cluster environment, so while the CPU part of the application runs on one node, the GPU kernels run on cluster-wide devices. This extension can provide a simpler programming environment, because unlike MPI, there is no need to split the application into sub-tasks that run on different cluster-nodes.

Regardless of the programming paradigm, HPC programs that are developed to run on heterogeneous clusters necessitate the use of the communication-network, either to distribute tasks and exchange messages as in MPI; or to send kernels, data buffers and collect results as in the extended-OpenMP. In both paradigms, the network latency could become a bottleneck.

This paper presents the Virtual OpenCL (VCL) cluster platform that can run unmodified OpenCL [3] applications transparently on clusters with many devices, including GPU and APU devices of different vendors. VCL benefits OpenCL applications that can use multiple devices concurrently. It is particularly suitable to run extended-OpenMP programs. VCL allows programs to run on a cluster without having to be split, by providing the impression of a single host with many devices. Users can start a parallel application on a hosting computer, then VCL manages and transparently runs the kernels of the application on different nodes.

The VCL cluster platform consists of three components: the VCL library, described in Sec. III-B, is a cluster implementation of the OpenCL standard; the broker, described in Sec. III-C, performs cluster-wide allocation of resources and the back-end daemon, described in Sec. III-D, which runs kernels on behalf of host applications. Combining the above, the VCL components provide an OpenCL platform in which all the cluster devices are seen as if they are located in the hosting-node. The structure of VCL is therefore flexible enough, allowing the incorporation of many algorithms, such as network optimizations, load-balancing and dynamic configurations. Sec. V describes a set of OpenCL extensions that can run at once a sequence of kernels on remote devices.

To show the overhead of VCL, we compare between the time required by an application to run a sequence of kernels using the "native" OpenCL library on a local device and the times to run these kernels using VCL on local and remote devices.

The paper presents the elements of VCL, its performance and the performance of several applications.

II. THE VCL RUN-TIME MODEL

VCL is designed to run applications that combine a CPU process with massive parallel computation on GPUs. The CPU process is responsible for the overall program flow and may also perform some of the computations. It can be multi-threaded in order to utilize all the available cores in the hosting node. While each instance of the application runs on a single hosting node, it can use multiple openCL devices (primarily GPUs) of a cluster. Choosing the location of the openCL devices is transparent to the program – applications need not be aware which cluster nodes and devices are available and where the devices are located. The number of hosting nodes and cluster nodes is configurable and they may overlap.

Currently most openCL applications make no use of multiple devices. Such unmodified applications will run correctly on VCL, but the full benefits of VCL manifest with parallel

applications that can use multiple devices concurrently. Our model simplifies the development of parallel applications and reduces the management complexity of running them on a cluster.

It is well known that the development of parallel applications, which is often done by modifying serial to parallel code, is simpler in openMP [2] than in MPI [1]. However, pure OpenMP is confined to a single computer while MPI can be scaled up to a large number of computers. The VCL runtime model combines the advantages of both programming paradigms. On the one hand, as in openMP, applications written for VCL benefit from the reduced programming complexity of the single computer, the availability of shared-memory, multi-threads and lower level parallelism, while on the other hand, as in MPI, VCL applications have concurrent access to devices in many nodes. To demonstrate the capabilities of the VCL model, we presented in [4] extensions of OpenMP and C++ that make use of VCL on clusters with multiple GPU devices.

For the benefit of existing (unmodified) MPI applications that require intensive GPU computations but do not exceed the capacity of a single multi-core computer, we note that it is still possible to combine MPI with VCL and benefit from shared-memory, by running all the MPI processes on the same computer.

III. THE VCL CLUSTER PLATFORM

A. Overview

The VCL cluster platform is an implementation of the OpenCL standard that allows unmodified applications to transparently utilize many devices in a cluster. Remote nodes perform OpenCL functions on behalf of application(s) on the host(s).

VCL is flexible: applications may choose to create OpenCL contexts that comprise of devices from several nodes; or multiple contexts, each with the devices of a different node; or any combination of the above. Other applications may be split into several independent processes and/or threads, each running on a different set of devices, while using shared-memory between them. Sophisticated applications may be specific about which devices they include in their contexts, but for the benefit of completely unmodified applications, VCL also allows environment-variables to control the device allocation policies. By default, each context that is created includes all the devices of a single node.

VCL consists of three components: the VCL library, the broker and the back-end daemon.

B. The VCL Library

The VCL library implements an openCL platform. When linked with openCL applications it allows transparent access to cluster-wide openCL devices, hiding the actual location of the OpenCL devices from the calling applications.

The VCL library is designed to operate with unmodified OpenCL programs, so platform-specific preferences, such as the policy of choosing devices from the cluster, can be defined

using environment-variables. The VCL library fully supports multi-threading and is thread-safe.

The VCL library incorporates various optimization algorithms. For example, due to network latency, the VCL library attempts to optimize the communication performance by maintaining an independent data-base of OpenCL objects and performing as many OpenCL operations as possible on the host-computer in order to reduce the number of network round-trips to the minimum.

A shell-script is provided for the ease of running programs with the VCL library.

C. The VCL Broker

The VCL broker is a daemon-process that runs on each host-computer where users can run their OpenCL applications. Its responsibilities include:

- 1) Online monitoring of existence and availability of OpenCL devices (e.g. GPUs) in the cluster;
- 2) Reporting about such devices to enquiring applications;
- 3) Intelligent allocation of devices for OpenCL applications when they create contexts, attempting for example to match the number of requested devices with a combination of nodes that have the exact number of such devices.
- 4) Authenticate, route and ensure the integrity of messages between the applications and back-ends.

The broker is connected with the VCL library via a UNIX socket.

D. The Back-end Daemon

The back-end daemon runs on each cluster-node where OpenCL devices are present and supported by an appropriate vendor-specific SDK. The back-end uses whichever vendor-specific OpenCL library(s) that are available on its node to run kernels on behalf of client applications. It emulates all the necessary OpenCL operations as requested by the VCL library. For security protection, so long as GPU devices and SDKs do not allow transparent preemption, OpenCL devices are not shared by the back-end among different client applications - each device is allocated to only one client application at a time.

IV. PERFORMANCE OF VCL

A. The VCL overhead

To find the VCL overheads, we compared between the time taken by an application to run a sequence of identical kernels using the native OpenCL library and the times to run the same kernels using the VCL on local and remote devices.

The tests were performed on a cluster with Intel 4-way Core i7 CPU nodes that were connected by a Quad Data Rate (QDR) Infiniband. Each node included an AMD-ATI 6970 and an NVIDIA GeForce GTX 480 (Fermi) GPUs.

Specifically, we measured the net time to run 1000 pseudo kernels on a designated local and remote devices. The time to compile the program and to copy buffers to/from the device was deliberately excluded. Each test was conducted 5 times and the median result is shown.

The results, for buffer sizes ranging from 4KB - 256MB are shown in Table I: Column 1 lists the size of the buffer that is passed to the OpenCL kernel. Column 2 shows the **native** OpenCL library times. Columns 3 shows the net VCL overheads (on top of the native OpenCL library times) for a local device and Column 4 shows the corresponding overheads for a remote device.

TABLE I
NATIVE OPENCL TIME VS. VCL OVERHEAD

Buffer Size	Native OpenCL Time (ms)	Net VCL Overhead (ms)	
		Local	Remote
4KB	96	35	113
16KB	100	35	111
64KB	105	35	106
256KB	113	36	105
1MB	111	34	114
4MB	171	36	114
16MB	400	36	113
64MB	1354	33	112
256MB	4993	37	111

From Table I it can be seen that the difference between the local/remote times and the corresponding native times are small and practically independent of the buffer size. The overhead in the table shows the net time to start 1000 kernels, therefore starting a single kernel by VCL on a local device takes on average $\sim 35\mu s$ longer than by the native OpenCL library; and $\sim 111\mu s$ longer on a remote device. This is a reasonable overhead for most parallel HPC applications.

B. Performance of the SHOC benchmark

We ran the applications from the Scalable Heterogeneous Computing (SHOC) 1.01 benchmark suite [6], [7], using the default parameters. The tests were performed on the above cluster, with an NVIDIA GeForce GTX 480 (Fermi) GPU in each node. We measured the runtime of each application, first with the native OpenCL library, then with VCL on a local device and again with VCL on a remote device. Each test was conducted 5 times and the median result is shown.

The results are presented in Table II. For each application, Column 2 shows the times to run the application with the native OpenCL library; Columns 3 and 4 show the corresponding times to run the application with VCL on local and remote devices.

Below we quote the brief **SHOC description of each application** followed by an explanation of its VCL performance.

- **BusSpeedDownload and BusSpeedReadback:** measures the bandwidth of transferring data across the PCI bus to/from a device. In both cases, as the VCL library detected that the applications did not run any kernels, the data was never even transferred, so VCL created no overheads.
- **DeviceMemory:** measures the bandwidth of memory accesses to various types of device-memory. Since this test does not involve any computation, the only factor is the overhead of data transfers created by VCL.

TABLE II
SHOC BENCHMARK PERFORMANCE

Application	Native Time (Sec.)	VCL Times (Sec.)	
		Local	Remote
BusSpeedDownload	0.89	0.88	0.88
BusSpeedReadback	0.91	0.89	0.89
DeviceMemory	31.44	56.78	243.81
KernelCompile	5.91	5.93	5.94
MaxFlops	186.98	156.74	211.20
QueueDelay	0.88	0.93	1.22
FFT	7.29	7.15	7.33
MD	14.08	13.66	13.80
Reduction	1.60	1.58	2.88
SGEMM	2.11	2.13	2.43
Scan	2.53	2.54	6.57
Sort	0.98	1.04	1.53
Spmv	3.25	3.30	5.91
Stencil2D	11.65	12.48	18.94
Triad	6.01	11.83	53.37
S3D	32.39	32.68	33.17

Specifically, the local run adds 2 memory copies (via UNIX sockets) and the remote run adds a memory copy plus a TCP/IP network transfer.

- **KernelCompile:** measures the compile time for several OpenCL kernels. The VCL roll is merely to transfer the small source code and instruct the compiler in the appropriate computer to compile it. The VCL overhead is therefore negligent.
- **MaxFlops:** measures the maximum achievable floating point performance. Unfortunately, the VCL performance results in this specific case are misleading because the test is self-calibrating, so that the actual amount of work is different.
- **QueueDelay:** measures the overhead of using the OpenCL command queue. This is an artificial test.
- **FFT: forward and reverse 1D FFT.** Relatively small data, long computation - ideal for VCL.
- **MD: computation of the Lennard-Jones potential from molecular dynamics.** Relatively small data, long computation - ideal for VCL.
- **Reduction: operation on an array of single or double precision floating point values.** Moderate amount of computation resulted in moderate performance.
- **SGEMM: matrix multiplication.** Much data and computation - reasonable VCL overhead.
- **Scan (parallel prefix sum): on an array of single or double precision floating point values.** Much data, little computation - poor results on remote devices.
- **Sort: an array of key-value pairs using a radix sort algorithm.** More computation than Scan, therefore better results.
- **Spmv: sparse matrix vector multiplication.** Huge data, significant computation - moderate results.
- **Stencil2D: a 9-point stencil operation applied to a 2D data set.** Huge data - bandwidth for remote devices is the limiting factor.
- **Triad: a version of the STREAM Triad benchmark**

that includes PCIe transfer time. Huge data, minimal computations - very poor results.

- **S3D: a computationally-intensive kernel from the S3D turbulent combustion simulation program.** Relatively small data, long computation - ideal for VCL.

C. Performance of the ATI-stream SDK Test Suite

We ran selected applications from the AMD ATI-stream SDK version 2.3 test suite [8], using an ATI 6970 GPU in each node. The results are shown in Table III and exhibited a similar pattern as the SHOC benchmark.

TABLE III
ATI-STREAM SDK TEST SUITE PERFORMANCE

Application and Parameters: Iterations (-i) or Array Size	Native Time (Sec.)	VCL Times (Sec.)	
		Local	Remote
AESDecrypt -i 2,000	16.75	17.31	24.50
BinarySearch -s 10K -i 10,000	2.36	3.28	16.66
BinomialOption -i 10,000	4.83	5.04	16.34
BitonicSort 2,000K	2.99	2.83	3.02
BlackScholes -i 2,000	3.02	6.14	43.36
ConstantBandwidth	123.12	122.89	123.51
DwtHaar1D -i 10,000	3.73	4.33	23.97
FFT -i 20,000	7.44	9.15	56.35
FastWalshTransform 1,000K	2.60	2.45	2.52
Histogram 100K X 100K	22.28	21.77	21.75
LDSBandwidth -i 10,000	19.03	19.94	25.05
MatrixMultiply 10K X 10K X 10K	12.46	14.87	27.21
MemoryOptimizations	5.60	6.61	12.55
MonteCarloAsian -c 1K	31.93	32.86	101.35
PCIEBandwidth	5.39	4.76	4.75
PrefixSum 100K	0.98	0.86	0.88
RadixSort -i 1,000	8.98	10.19	27.64
RecursiveGaussian -i 1,000	3.60	5.57	24.93
Reduction 1,000K	0.98	0.86	0.90
ScanLargeArrays 1,000K	2.63	2.50	2.60
SimpleConvolution 10K X 10K	5.43	15.69	34.32
SimpleImage -i 1,000	1.96	3.61	22.54
SobelFilter -i 1,000	1.51	2.29	11.67
URNGi -i 1,000	1.49	2.24	11.63

D. Performance outcome

The Tables show a very wide range of results for taking the GPU computation to other nodes. On the one hand, much more computation power is available throughout the cluster. On the other hand, network bandwidth and especially network latency take their toll on performance. Those tests with relatively long kernels and infrequent buffer-I/O operations are doing well, but those with many short kernels or with frequent or large I/O operations fall behind.

To amend these obvious problems that emerge from running on a cluster, we designed "SuperCL".

V. SUPERCL

When running OpenCL on remote devices, network latency is the main limiting factor. Minimizing the number of network round-trips for standard OpenCL library-calls was the first step, but is not enough.

As the next step, we designed an extension called "SuperCL", whereby a programmable sequence of kernels and/or

memory operations can be sent to device(s) of a cluster-node, usually with just a single network round-trip. When necessary, communication with the host is still possible, but in an asynchronous manner, to avoid the round-trip waiting time.

Bandwidth can also be a limiting factor when huge inputs/outputs are involved, so this is also addressed by SuperCL by allowing buffers to be initialized from back-end files and for results to be stored on back-end files.

Below are some examples of what can be done with SuperCL, beginning with simple cases and continuing onto more advanced options.

- Run a sequence of 3 kernels on a back-end device. As each kernel depends on the output of the former kernel, a temporary buffer is used on the back-end node, without the need to create it on the host, or to transfer its contents over the network.
- As above, but while the first and the third kernels are parallel in nature and therefore run best on a GPU, the middle kernel is sequential in nature and therefore runs best on a CPU.

In this case, SuperCL uses a back-end that supports both its CPU and its GPU(s) as OpenCL devices, where it runs the first and third kernel on the GPU and the middle kernel on the CPU - there is no need to transfer the intermediate results to the host.

- Run 2 alternate kernels N consecutive times (A-B-A-B-A-B...).
- Run 2 alternate iterative kernels: the first kernel computes something and the second determines the accuracy of the result. Repeat running both until an accuracy below a given threshold is achieved.
- As above, but the currently-achieved level of accuracy is also sent to the host after each iteration using asynchronous messages. The host may also stop (or otherwise control, for example by adjusting the threshold) the iterations at any time by sending asynchronous instructions which can then be inspected between kernels.
- Repeat a complex computation (with one or more kernels) over a large matrix, either N times or until a condition is satisfied. Instead of waiting until the computation is complete and only then sending the whole matrix back to the host, whenever parts of the matrix are known to have final values, those parts (and only them) can be sent back to the host, concurrently with the remaining computation.
- As above, but signals are also sent to the host to let it know that some data has been copied.
- In all the above cases, it is possible to obtain the input from common NFS-mounted files and/or from data left over on the back-end nodes by earlier programs. Final results can also be left on the back-end nodes instead of being sent to the host.

SuperCL operations can be queued in the normal way as standard OpenCL operations.

VI. CONCLUSIONS AND FUTURE WORK

Advancements in heterogeneous systems offer new opportunities to increase the performance of parallel HPC applications on clusters. Currently, users are provided with software development and programming environments that can ease the use of GPU and APU devices on a single node, but were not designed to run applications on clusters.

We presented an OpenCL cluster platform that allows unmodified OpenCL applications to transparently and concurrently run on multiple devices in a cluster. This platform allows OpenMP and each task of MPI application to utilize cluster-wide devices.

Performance of parallel applications with VCL shows that running parallel kernels efficiently on remote devices in a cluster is quite feasible. VCL should be able to support large-scale high-end parallel computing applications.

Based on our experience, an ideal cluster for running parallel HPC applications with our platform would be a collection of low-cost servers, each with several GPUs and/or APUs, that are connected by a low-latency, high-bandwidth network to high-end hosting nodes with many cores and large memories.

The work described in this paper could be extended by the development of MOSIX-like algorithms for dynamic resource management, load-balancing among different devices and within an APU, task priorities, fair-share and for choosing the “best” device [5]. Followup projects include porting of VCL to DirectCompute [9] and upgrades to the latest OpenCL specifications.

VCL is currently implemented for Linux platforms. The latest distribution supports OpenCL 1.1. It can be obtained from [10].

ACKNOWLEDGMENTS

This research was supported in part by grants from Dr. and Mrs. Silverston, Cambridge, UK.

REFERENCES

- [1] MPI. *The Message Passing Interface (MPI) standard*, <http://www.mcs.anl.gov/research/projects/MPI/>.
- [2] OpenMP. *The OpenMP API specification for parallel programming*, <http://www.OpenMP.org>.
- [3] OpenCL, *The OpenCL Specification*, A. Munshi (Ed). Khronos Group, 2010, <http://www.khronos.org/opencvl>
- [4] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices, in *Proc. Workshop on Parallel Programming and Applications on Accelerator Clusters (PPAAC), IEEE Cluster 2010*, Sept. 2010.
- [5] A. Barak and A. Shiloh, *The MOSIX management system for Linux cluster, multi-clusters, GPU clusters and Clouds*, http://www.mosix.org/pub/MOSIX_wp.pdf.
- [6] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, “The Scalable Heterogeneous Computing (SHOC) benchmark suite,” in *Proc. 3-rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*, March 2010, pp. 63–74.
- [7] SHOC, ORNL future tech wiki - Scalable Heterogeneous Computing (SHOC) benchmark suite, <http://ft.ornl.gov/doku/SHOC/start>.
- [8] AMD, *ATI SDK 2.3 test suite*, <http://AMD.developer.com/GPU/AMDappsdk/downloads/pages/default.aspx>.”
- [9] DirectCompute, <http://msdn.microsoft.com/Directx>.
- [10] VCL, http://www.mosix.org/txt_vcl.html.