*cl*OpenCL - Supporting Distributed Heterogeneous Computing in HPC Clusters[⋆]

Albano Alves[1], José Rufino[1], António Pina[2], and Luís Paulo Santos[2]

[1] Polytechnic Institute of Bragança, Bragança, Portugal
{albano,rufino}@ipb.pt
[2] University of Minho, Braga, Portugal
{pina,psantos}@di.uminho.pt

Abstract. Clusters that combine heterogeneous compute device architectures, coupled with novel programming models, have created a true alternative to traditional (homogeneous) cluster computing, allowing to leverage the performance of parallel applications. In this paper we introduce *cl*OpenCL, a platform that supports the simple deployment and efficient running of OpenCL-based parallel applications that may span several cluster nodes, expanding the original single-node OpenCL model. *cl*OpenCL is deployed through user level services, thus allowing OpenCL applications from different users to share the same cluster nodes and their compute devices. Data exchanges between distributed *cl*OpenCL components rely on Open-MX, a high-performance communication library. We also present extensive experimental data and key conditions that must be addressed when exploiting *cl*OpenCL with real applications.

Keywords: Heterogeneous/Cluster/GPGPU Computing, OpenCL.

1 Introduction

Clusters of heterogeneous computing nodes provide an opportunity to significantly increase the performance of parallel and High-Performance Computing (HPC) applications, by combining traditional multi-core CPUs coupled with accelerator devices, interconnected by high throughput and low latency networking technologies. However, developing efficient applications to run in clusters that integrate GPUs and other accelerators often requires a great effort, demanding programmers to follow complex development methodologies in order to suit algorithms and applications to the new heterogeneous parallel environment.

Cluster nodes with GPUs are usually exploited by an hybrid approach: MPI is used to distribute the application across multiple CPUs, and OpenCL or CUDA are used to run specific routines (kernels) on GPU(s) at each node [1,2]; more

⋆ This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-010067.

complex approaches have also been experimented [3], where OpenMP exploits CPU parallelism inside each MPI process and CUDA takes advantage of GPUs.

The former examples show that porting single-node-multi-GPU applications to a multi-node-multi-GPU execution environment is not straightforward. Thus, the benefits of a platform that would allow to efficiently run the same and unmodified program, whether in a single-node-multi-accelerator or a multi-node-multi-accelerator scenario are obvious.

Our initial goal was to contribute to the efficient implementation of a parallel path tracer, by improving the BVH operation [4], but we now target a more general one: to boost the performance of existing solutions by exploiting the hardware of an heterogeneous cluster. In this paper we introduce *cl*OpenCL (*cluster* OpenCL), an integrated, transparent and efficient approach to take advantage of compute devices spread across an heterogeneous cluster.

The remaining of the paper is organized as follows: section 2 reviews some related work; section 3 describes the architecture of *cl*OpenCL; section 4 presents and discusses a preliminary evaluation scenario based on matrix multiplication; finally, section 5 concludes and points out directions for future work.

2 Scaling Up OpenCL

2.1 The OpenCL Model

OpenCL is an open standard for programming different kinds of computing devices [5]. An OpenCL application comprises a *host* program and a set of *kernels* intended to run on compute *devices*; the OpenCL specification defines a language for kernel programming, and an API for transferring data between the host and devices (and to execute kernels on the later). Currently there are three major implementations of the OpenCL specification, supporting different compute devices: i) AMD APP SDK (for CPUs and AMD GPUs), ii) Nvidia's implementation (for NVIDIA GPUs only) and iii) Intel OpenCL SDK (for CPUs only).

But OpenCL poses a major problem: applications can only utilize the *local devices* present on a single machine. Thus, the number of OpenCL devices available to an application may be rather limited. For instance, the number of PCIe bus slots in a machine limits the number of usable GPUs. And although it would be possible to increase the number of slots through the use of bus extenders, the only way to achieve a scalable platform is to use a cluster of nodes, each one with its own (limited) set of computing devices.

Since in the original OpenCL model an application runs on a single node, a new/modified model is then required for OpenCL applications to be able to use several nodes. In the new model, the host application must be able to transfer/share data to/with *remote devices.*

2.2 Related Work

Running unmodified OpenCL applications on clusters with GPUs has been a common goal of several projects. Different approaches have been undertaken, but the fact is that none combines the simplicity and the performance required for heterogeneous clusters shared by distinct users (usually operated in batch).

The Many GPUs Package (MGP) [6] can run extended OpenMP, C++ and unmodified OpenCL applications transparently on clusters with many GPU devices. It provides a simple API and the illusion of a single (virtual) host with many GPUs (single-system-image). The whole system uses the MOSIX VCL layer [7] to create the abstraction of a global OpenCL platform combining all GPUs present in a cluster (the CPU part of the application runs in a single node). Communications rely on TCP sockets and only binaries are distributed.

The Hybrid OpenCL [8] project integrates the network layer in the OpenCL runtime, translating in a "bridge program" (service) per cluster node. The system was developed for a particular device independent OpenCL implementation (FOXC OpenCL) which currently supports only x86 CPUs, thus preventing it to exploit high performance GPUs. Data exchanges are RPC based.

dOpenCL (*distributed* OpenCL) [9] has resemblances to *cl*OpenCL: both support transparent multi-node-multi-accelerator OpenCL applications and combine a wrapper client library with remote services. However, dOpenCL is oriented to general distributed environments, uses a TCP/UDP based communication framework, and devices may not be concurrently shared. In turn, *cl*OpenCL targets HPC clusters, uses Open-MX to maximize the utilization of commodity Gigabit Ethernet links, and devices are fully shareable. Both approaches work on top of any OpenCL platform and so are able to exploit many device types.

CUDA applications may also benefit from similar approaches. With rCUDA [11], applications may use CUDA-compatible GPUs installed in remote computers as if they were local. rCUDA follows the client-server model: clients use a wrapper library and a GPU network service listens for TCP requests.

In GVirtuS [10], a different approach is taken, in order to fill the gap between in-house hosted computing clusters (equipped with custom devices) and pay-for-use high performance virtual clusters (deployed via public or private computing clouds). GVirtuS allows a virtual machine to access CUDA powered GPGPUs in a transparent way, with an overhead slightly greater than a real machine setup.

However, in addition to the limited number of compute devices that CUDA can handle (NVIDIA only), none approach takes explicit advantage of the high performance interconnection technologies available in modern clusters.

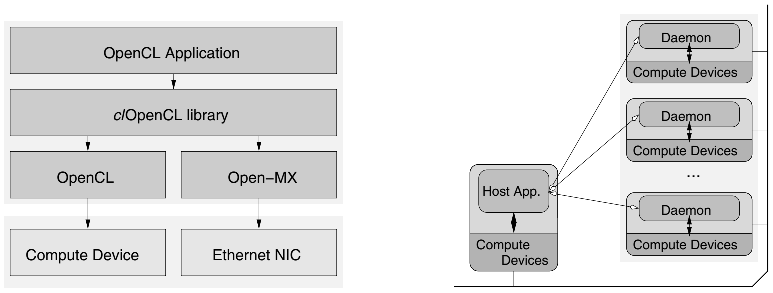
3 Our Approach: *cl*OpenCL

3.1 General Concept

The *cl*OpenCL platform comprises a wrapper library and a set of user-level daemons. Every call to an OpenCL primitive is intercepted by the wrapper library which redirects its execution to a specific daemon at a cluster node or to the local OpenCL runtime. *cl*OpenCL daemons are simple OpenCL programs that handle remote calls and interact with local devices. A typical *cl*OpenCl application starts running at a particular cluster node and will create OpenCl contexts, command queues, buffers, programs and kernels across all cluster nodes.

For the exchange of data between the wrapper library and remote daemons, we adopted Open-MX, an open-source message passing stack over generic Ethernet [12], which provides low-level communication mechanisms at user-level space and allows to achieve low latency communication and low CPU overhead.

Figure 1 presents a) the software/hardware layers of the host component of an *cl*OpenCL application, and b) the *cl*OpenCL operation model upon which the host component interacts with multiple compute devices (local or remote).



(a) layers (b) operation model

Fig. 1. *cl*OpenCL architecture

3.2 Distributed Operation

Running a *cl*OpenCL application requires the prior launching of *cl*OpenCL per-user daemons at specific cluster nodes; each user may then choose the node (sub-)set to be effectively used by the application; different users may exploit distinct configurations, sharing or not particular nodes and compute devices.

When the host program starts, the *cl*OpenCL wrapper library (which also wraps `main`) discovers all daemons by interacting with the Open-MX mapper service. This mapper creates a distributed directory that registers each process with an opened Open-MX end-point (along with the user id of the process owner).

In a traditional OpenCL application, the programmer has only to manipulate objects returned by the OpenCL API, namely: platform and device identifiers, contexts, command queues, buffers, images, programs, kernels, events and samplers. These objects are in fact pointers to complex OpenCL data structures, whose internals are hidden from the programmer. In a distributed/parallel environment, where OpenCL primitives are executed in multiple daemons, these pointers cannot be used to uniquely identify objects, because each daemon has its own address space. Thus, for each object created by OpenCL, the wrapper library returns a “fake pointer” used as a global identifier, and stores the real pointer along with the corresponding daemon location.

Each time the wrapper library redirects an OpenCL primitive, its parameters are packed in an Open-MX frame and sent to the remote daemon that will execute the primitive. Any parameters that reference OpenCL objects are previously mapped into their real pointers and the daemon is determined accordingly.

The library and daemons do not keep any information about calls to OpenCL primitives, *i.e.*, they are stateless. Any data needed for subsequent primitive calls is kept by the OpenCL runtime at each cluster node. As such there's no need to manage complex data structures related to OpenCL operation and state.

3.3 Using *cl*OpenCL

Porting OpenCL programs to *cl*OpenCL only requires linking with both the OpenCL and *cl*OpenCL libraries. This is accomplished by taking advantage of the `-Xlinker --wrap` GCC directives for function wrapping in link-time.

Typically, an OpenCL application starts by querying the runtime for platforms and devices which, by design, are *local* (*i.e.*, provided by the node where the host application component is started). In *cl*OpenCL such discovery returns the *local* platforms and devices (if any, once *cl*OpenCL doesn't strictly require local ones), and also returns the set of *remote* platforms and devices provided by the cluster nodes where the user-specific *cl*OpenCL services are running.

In order to know to which cluster node a certain platform belongs, we have extended the OpenCL primitive `clGetPlatformInfo` with the special attribute `CL_PLATFORM_HOSTNAME`. Having the possibility of selecting specific cluster nodes where to run the OpenCL kernels may be useful, *e.g.*, for load balancing.

Currently, we do not support the mapping of buffer and image objects. However, at its current state, the *cl*OpenCL platform is enough to meet the purpose of testing its general concept, including running basic OpenCL applications.

4 Evaluation

4.1 Testbed Cluster

The testbed is a small Linux commodity cluster of 4 nodes, with an Intel Q9650 CPU (3GHz 4-core with 12 Mb of L2 cache), 8Gb of RAM (DDR3 1333MHz) and two Ethernet 1Gbps NICs (on-board Intel 82566DM-2 and a PCI64 SysKonnect SK-9871), per node. The nodes are also fitted with NVIDIA GTX 460 GPUs (1GB of GDDR5 RAM): one node (`node-0`) has 2 GPUs and three nodes (`node-{1,2,3}`) have 1 GPU each. The OpenCL platforms used were AMD SDK 2.6 (for CPU devices) and CUDA 4.1.28 (for the NVIDIA GPUs). Open-MX 1.5.2 was used with the SysKonnect NICs, interconnected via a dedicated ethernet network using a gigabit switch with jumbo frames (mtu 9000) enabled.

4.2 Test Application

We chose the *matrix product* as the test application because it's a simple and "embarrassingly parallel" case study, enough to verify the correctness and scalability of *cl*OpenCL (our aim is not to offer a reference HPC implementation).

We narrowed our study to square matrices of order $n \in \{8K, 16K, 24K\}$ and single-precision (floats) elements. These 3 different orders were chosen to support a minimal scalability study and, at the same time, to allow all the 3 matrices involved (the operands A and B, and the result $C = AB$) to be fully instantiated in the RAM (8Gb) of the node with the application host component (`node-0`); in the worst case ($n = 24K$) each matrix uses 2.25 Gb, for a total of 6.75 Gb.

The matrix product operation was parallelized using a block-based approach: A (B) was partitioned in horizontal (vertical) blocks $subA$ ($subB$) of order $slice \times n$ ($n \times slice$), so that $subC = subA\ subB$ is a block, of order $slice \times slice$, of C.

The OpenCL kernel used to produce a $subC$ block is shown in Figure 2. It is a “naive” implementation, not optimized to exploit advanced OpenCL features.

```
_kernel void sbmp(const int n, const int slice, __global float *subA,
                          __global float *subB, __global float *subC){
    int i, j, k; float v=0;

    i = get_global_id(0); j = get_global_id(1);
    for(k=0; k<n; k++)
        v += subA[i*n+k] * subB[j*n+k];
    subC[i*slice+j] = v;
}
```

Fig. 2. `sbmp` - a simple kernel for `s`ingle-precision `b`lock-based `m`atrix `p`roduct

Producing a $subC$ requires $slice^2$ kernel executions (or work-items); this is achieved by the parameter `int global_work_size[2]={slice,slice}` of the OpenCL function that triggers the kernel execution (`clEnqueueNDRangeKernel`).

We set $slice = 1K, 2K, 4K$ when multiplying matrices of order $n = 8K, 16K$, $24K$, respectively. The $slice$ values were chosen to allow any device to be able to fully store the triplet $< subA, subB, subC >$; in the worst case, with $n = 24K$ and $slice = 4K$, $subA$ and $subB$ need 384 Mb each and $subC$ needs 64 Mb, for a total of 832 Mb, still less than the 1 Gb of RAM of any GPU. At the same time, the values of $slice$ also ensure enough blocks for a fine-grain load-balancing among the various OpenCL devices; this comes from the observation that, in our cluster, a GPU is approximately twice as fast as a CPU when executing the `sbmp` kernel; thus, with 5 GPUs and 4 CPUS, we need at least $2 \times 5 + 4 = 14$ kernel runs (one per $subC$) to keep all devices busy; by requiring at least two kernel runs per device, we need at least 28 kernel runs overall; finally, $slice$ comes by solving $(n/slice)^2 \geq 28$ where $(n/slice)^2$ is the total number of kernel runs.

The test application is multi-threaded (PThreads) and follows a dynamic model of work (auto-)assignment: a thread is created for each OpenCL device involved in the matrix product; the threads select mutually exclusive pairs $< subA, subB >$, send them to their devices, trigger the kernel run and collect the results $subC$. Data transfers and kernel runs for remote devices are mediated by *cl*OpenCL but, for local devices, they are direct (no daemon is involved).

4.3 Test Configurations

The test application was always started in the cluster node with the most performant set of OpenCL devices. That way, the utilization of that device set is maximized, once communication with its devices is purely local. In our testbed cluster, the node that fits this criterion is `node-0`, with 1 CPU and 2 GPUs.

Overall, under the constraints of our testbed cluster, there are 74 combinations of OpenCL devices, where `node-0` is always used (with at least one device), and zero or more remote nodes are used (with at least one device). All combinations were evaluated, but Table 1 shows only the subset that provides the best performance, for a certain number of CPUs (#C) and GPUs (#G) used; in each combination, the CPUs are denoted by C and the GPUs by G; they are represented in comma separated groups, with one group per each cluster node used to support the combination; in each combination, the 1st group of devices is from `node-0` and the next (eventual) groups are from `node-1` to `node-3` respectively.

Table 1. Performance Optimal Combinations of OpenCL Devices

#C \ #G	0	1	2	3	4	5
0		G	GG	GG,G	GG,G,G	GG,G,G,G
1	C	GC	GGC	GGC,G	GGC,G,G	GGC,G,G,G
2	C,C	GC,C	GGC,C	GGC,G,C	GGC,G,G,C	GGC,GC,G,G
3	C,C,C	GC,C,C	GGC,C,C	GGC,G,C,C	GGC,GC,G,C	GGC,GC,GC,G
4	C,C,C,C	GC,C,C,C	GGC,C,C,C	GGC,GC,C,C	GGC,GC,GC,C	GGC,GC,GC,GC

All combinations of Table 1 obey to the same general rule: they use the maximum possible number of local devices (on `node-0`) and scatter as much as possible the remote devices necessary (through `node-1` to `node-3`). Moreover, our tests showed that this rule is valid, regardless the order n of the matrices.

4.4 Test Results

Figure 3.a) represents the time took by the matrix product for n=24K, for all device combinations of Table 1. Times range from a maximum of ≈4800s, when using a single local CPU (combination C), to a minimum of ≈469s, when using all 5 GPUs of the cluster but only 3 CPUs (in the combination GGC,GC,GC,G)[1]. Figure 3.b) zooms in Figure 3.a), for combinations where #C≥1 and #G≥2.

Figure 4.a) shows the speedups of all combinations, relative to combination GGC (with speedup = 1). Figure 4.b) zooms in figure 4.a) for #C≥1 and #G≥2.

The point of coordinates #C=1 and #G=2, that marks the lower end of the range defined by #C≥1 and #G≥2, translates to the combination GGC, meaning that all local devices of the starting node (`node-0`) are used. From such point onwards, the only way to increase performance is to use remote devices,

[1] Using all 4 CPUs (combination GGC,GC,GC,GC) takes ≈489s.

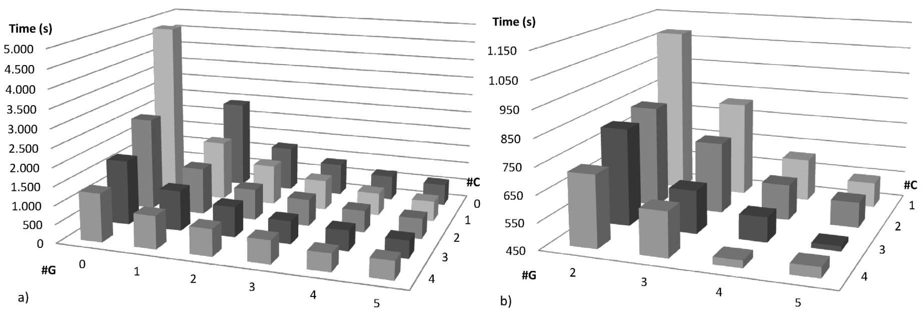


Fig. 3. Execution time for n=24K (a) all combinations; b) #C≥1 and #G≥2)

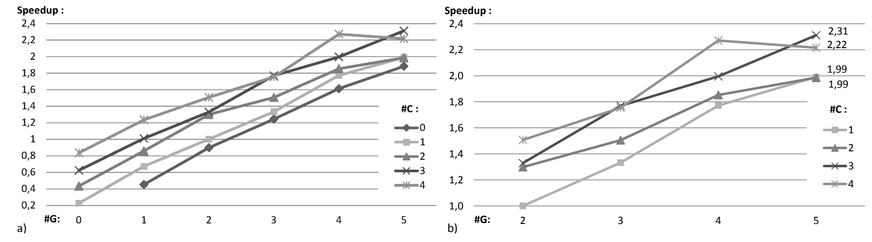


Fig. 4. Speedup for n=24K (a) all combinations; b) #C≥1 and #G≥2)

but the speedups will be modest, as shown in Figure 4.b). Such demonstrates the importance of the combination GGC: if one wants to minimize the number of cluster nodes involved in the application execution, while maximizing performance, then using a node holding the device combination GGC is mandatory, and that node should be the one where the application starts. Moreover, GGC is the best scenario that a traditional OpenCL approach could exploit and so it makes sense to use it as the starting point to deploy *cl*OpenCL applications.

Figures 5 to 6 show similar graphics for matrices of order 16K and 8K. As expected, the matrix product times decrease (ranging from ≈1422s to ≈157s for $n = 16K$, and from ≈179s to ≈22s for $n = 8K$). Moreover, speedup values also decrease (with peek-values of 2.15 and 1.76 for orders 16K and 8K, against 2.31 for order 24K), showing that the scalability improves with the problem size. Differently from order 24K, orders 16K and 8K achieve the best times when using all the 4 CPUs and 5 GPUs of the cluster (combination GGC,GC,GC,GC).

Real Speedup versus Ideal Speedup. An alternative (and perhaps more objective) measure of the merits of our approach results from the comparison of the maximum speedups achieved by *cl*OpenCl (real speedups) with the maximum speedups theoretically achievable under ideal conditions (ideal speedups).

For order 24K, we recall that the worst execution time is T(C)≈4800s, for a single CPU, and the best is T(GGC,GC,GC,G)≈469s, when 5 GPUs and 3 CPUs are involved. The evaluation of the execution times also reveals that a

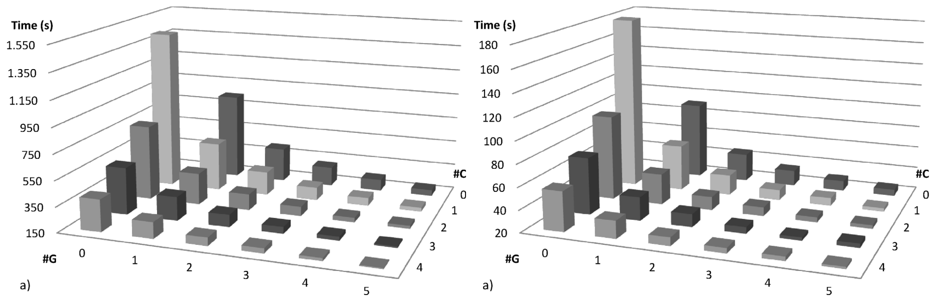


Fig. 5. Execution time (all combinations) for a) n=16K; b) n=8K

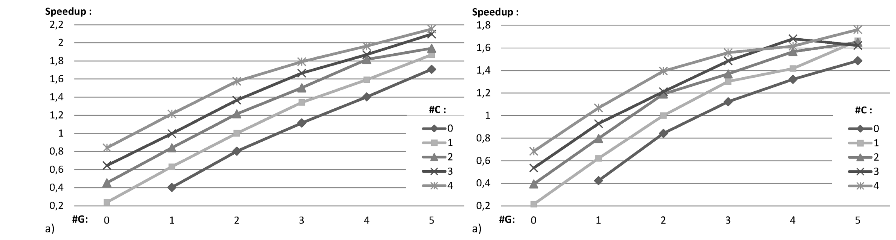


Fig. 6. Speedup (all combinations) for a) n=16K; b) n=8K

single GPU takes T(G)≈2403s, which is approximately half of T(C). Thus, the time that a CPU takes to process a single work slice (T_{slice}(C)), is twice the time that a GPU would take (T_{slice}(G)). Or, conversely, the rate at which a GPU can process work slices (λ_{slice}(G)) is approximately twice the rate of a CPU (λ_{slice}(C)). So, if it were possible to host, in the same cluster node, 5 GPUs and 3 CPUs, then, for a time interval of duration T_{slice}(C), we would have 5 × 2 = 10 work slices being processed by the 5 GPUS, and 3 × 1 = 3 work slices by the 3 CPUs, for a total of 13 work slices. It then follows that the maximum theoretical speedup is S^{ideal}_{max}=13 (this assumes that there are enough work slices to keep all devices simultaneously busy). Now, the maximum effective speedup is S^{real}_{max} = T(C) / T(GGC,GC,GC,G) = 10.23. Finally, S^{real}_{max} / S^{ideal}_{max} = 78.72%.

For n=16K, the worst and best execution times are T(C)≈1422s and T(GGC, GC,GC,GC)≈157s, respectively. Also, T(G)≈840s, which is 59% of T(C), relatively near from the 50% ratio achieved with n=24K. However, with 5 GPUs and 4 CPUs, the maximum theoretical speedup is now S^{ideal}_{max} ≈14 and the maximum effective speedup is S^{real}_{max} = T(C) / T(GGC,GC,GC,GC) = 9.05. Then, S^{real}_{max} / S^{ideal}_{max} = 64.69%, a smaller ratio in comparison to the one with n=24K.

Finally, for n=8K, the worst time is T(C)≈179s and the best is T(GGC,GC, GC,GC)≈22s. Also, T(G)≈91s, which is ≈50% of T(C). Again, with 5 GPUs and 4 CPUs, the maximum theoretical speedup is S^{ideal}_{max} ≈14. The maximum effective speedup is now S^{real}_{max} = T(C) / T(GGC,GC,GC,GC) = 8.21. Thus, S^{real}_{max} / S^{ideal}_{max} = 58.68%, a smaller ratio in comparison to the one with n=16K.

The above S^{real}_{max} / S^{ideal}_{max} ratios shed new light on the (somehow) modest speedups apparently achieved by *cl*OpenCL against the best pure OpenCL scenario (GGC). In particular, they prove that for bigger problem sizes (as illustrated by the matrix product of order $24K$), *cl*OpenCL exhibits good scalability.

5 Conclusions

We have presented the design and implementation details of a new platform that facilitates the execution of OpenCL applications in heterogeneous clusters. Other projects have also focused on the same objective, but *cl*OpenCL has two main advantages: it is able to take full advantage of commodity networking hardware through Open-MX, and programmers/users do not need special privileges neither exclusive access to scarce resources to deploy the desired running environment.

We used an embarrassingly parallel program to evaluate *cl*OpenCL, with different problem sizes and cluster device combinations. Results show that *cl*OpenCL is useful for exploiting multi-node-multi-GPU environments: porting previous OpenCL applications is straightforward and performance gains are attractive.

In the future, we will expand the set of OpenCL primitives supported by *cl*OpenCL. Also, adding BSD sockets as an alternative communication layer to Open-MX will allow for *cl*OpenCL to be used in more distributed scenarios.

References

1. Lawlor, O.: Message Passing for GPGPU Clusters: cudaMPI. In: IEEE Cluster PPAC Workshop, pp. 1–8 (2009)
2. Stefanski, T., Chavannes, N., Kuster, N.: Hybrid OpenCL-MPI parallelization of the FDTD method. In: International Conference on Electromagnetics in Advanced Applications (ICEAA), pp. 1201–1204 (2011)
3. Yang, C.-T., Huang, C.-L., Lin, C.-F.: Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. Computer Physics Communications 182, 266–269 (2011)
4. Goldsmith, J., Salmon, J.: Automatic creation of object hierarchies for ray tracing. IEEE Computer Graphics & Applications 7(5), 14–20 (1987)
5. Munshi, A.: The OpenCL Specification. Khronos OpenCL Working Group (2009)
6. Barak, A., Ben-nun, T., Levy, E., Shiloh, A.: A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices. Science, 1–7 (2010)
7. Barak, A., Shiloh, A.: The Virtual OpenCL (VCL) Cluster Platform. In: Proc. Intel European Research & Innovation Conference, p. 196 (2011)
8. Aoki, R., Oikawa, S., Nakamura, T., Miki, S.: Hybrid OpenCL: Enhancing OpenCL for Distributed Processing. In: IEEE 9th International Symposium on Parallel and Distributed Processing with Applications Workshops, pp. 149–154 (2011)
9. Kegel, P., Steuwer, M., Gorlatch, S.: dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems. In: 26th IEEE Int. Parallel and Distributed Processing Symposium Workshops, pp. 174–186 (2012)

10. Giunta, F., Montella, R., Laccetti, G., Isaila, F., Blas, F.: A GPU Accelerated High Performance Cloud Computing Infrastructure for Grid Computing Based Virtual Environmental Laboratory. In: Advances in Grid Computing (2011)
11. Duato, J., Peña, A., Silla, F., Mayo, R., Quintana-Ortí, E.: Reducing the number of GPU-based accelerators in high performance clusters. In: International Conference on High Performance Computing and Simulation, pp. 224–231 (2010)
12. Goglin, B.: High-Performance Message Passing over generic Ethernet Hardware with Open-MX. Elsevier Journal of Parallel Comp. (PARCO) 37(2), 85–100 (2011)