# dOpenCL: Towards Uniform Programming for Distributed Systems with Multi-Cores and GPUs

Philipp Kegel and Sergei Gorlatch

***Abstract*—Modern computer systems are becoming distributed and heterogeneous by comprising multi-core CPUs, GPUs, and other accelerators. However, to program such systems, the user currently has to use a combination of several programming models (e.g., MPI with OpenCL or CUDA), which is difficult and error-prone. We present *dOpenCL (distributed OpenCL)* – a uniform approach to programming distributed systems with accelerators. Our approach is based on the OpenCL standard and it allows the user to run existing OpenCL applications unmodified in a heterogeneous distributed environment. The dOpenCL system also supports transparent execution of multiple OpenCL applications in one distributed, multi-user environment. We describe dOpenCL as an implementation of the OpenCL programming model on distributed systems, and we experimentally compare the performance of dOpenCL with MPI+OpenCL and standard OpenCL implementations.**

***Index Terms*—OpenCL, heterogeneous systems, distributed systems, GPU computing, dOpenCL, multi-cores.**

## I. Introduction

MODERN systems increasingly comprise heterogeneous processing devices, e.g., multi-core CPUs, GPUs, and other accelerators. The state-of-the-art approaches to program such systems usually employ several programming models in combination. For example, let us consider programming a cluster where each node contains a multi-core CPU and a GPU. First, the programmer has to distribute the data to all compute nodes, e.g., using MPI [1] or explicit, low-level network programming. Second, to exploit the GPU at each node, another appropriate model, e.g., CUDA [2] is needed, which requires the programmer to explicitly transfer data between the node's main memory and GPU. Furthermore, in order to use all cores of the CPU, a thread programming model, e.g., Pthreads [3] is usually employed. The main drawback of combining programming models is that the programmer has to master several different models, and he also has to take into account their possible interference when used together in a single program.

We present *dOpenCL (distributed OpenCL)* – a uniform programming approach for heterogeneous, distributed systems. It is based on OpenCL [4] – an open, widely accepted standard for heterogeneous systems. However, the original OpenCL is limited to stand-alone systems and has to be augmented by other programming models to create applications for distributed systems. The dOpenCL approach transparently implements the OpenCL programming model for distributed systems, such that the programmer no longer has to combine it with other programming models.

Note that our dOpenCL is not related to *DOpenCL* which is an OpenCL binding for the D programming language [5].

Manuscript received December 28, 2012; revised January 23, 2013.

The authors are with the Institute of Computer Science, University of Muenster, 48149 Muenster, Germany.
Email: gorlatch@uni-muenster.de

The paper is organized as follows. Section II introduces the design of dOpenCL as an implementation of the OpenCL API extended for distributed systems. In Sections III, an extension of dOpenCL is presented which allows for multiple OpenCL applications to run concurrently in a distributed system. In Section IV, we present application studies to experimentally evaluate the performance of dOpenCL and to compare it to approaches that mix MPI and OpenCL for programming distributed systems. We proceed with a discussion of related work in Section V and we conclude our work in Section VI.

## II. The Distributed OpenCL (dOpenCL) System

The *dOpenCL* system has two objectives: 1) provide the application programmer with access to various compute devices in a heterogeneous distributed system, and 2) avoid mixing different models for programming such systems.

We develop dOpenCL as a fully-fledged implementation of the OpenCL programming model for distributed systems. In dOpenCL, all devices of the target architecture – (multi-core) CPUs, GPUs, or other accelerators – are presented to the programmer as if they were available in a single stand-alone system. Therefore, dOpenCL hides from the application programmer the underlying distributed system structure: the programmer can use heterogeneous devices of the system uniformly by means of the standard OpenCL API.

The main idea of the dOpenCL implementation is to merge the native OpenCL implementations on the nodes of the target distributed system into a single so-called OpenCL *platform*. A dOpenCL platform describes the target system consisting of a single *host* and a number of *compute nodes* connected to it.

The most important task of dOpenCL which maps the OpenCL programming model to distributed systems is to hide from the programmer the network which connects the host and the compute nodes. This is a challenging task since the implementation has to:

- automatically connect nodes of a distributed system, as the OpenCL API has no means for that;
- distribute contexts and associated objects transparently across multiple compute nodes;
- ensure consistency of distributed objects according to OpenCL's release consistency model;
- synchronize command execution across distributed compute nodes.

In the following, we describe in some detail how the dOpenCL runtime system addresses these challenges.

The dOpenCL runtime system is designed as a distributed application shown in Figure 1: it comprises one *client driver* and multiple *daemons* which are installed on the nodes of the target distributed system. A node with the client driver

ISBN: 978-988-19251-8-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

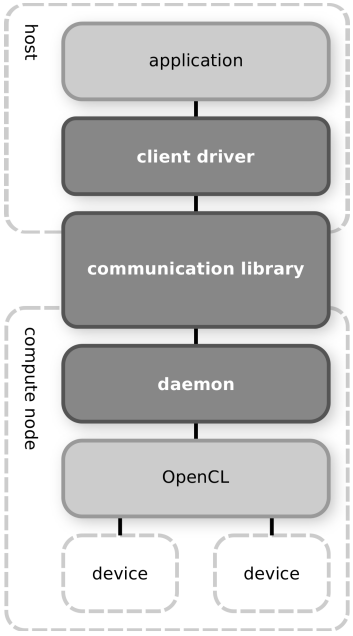


Fig. 1. Overview of the dOpenCL runtime system.

is called the *host*, while each node that runs a daemon is referred to as a *compute node*. The host runs the OpenCL application's host program, and the compute nodes execute application kernels on their local devices.

The client driver provides a full implementation of the OpenCL API which transparently replaces an existing OpenCL implementation on the host: the original OpenCL application does not have to be modified in any way in order to use dOpenCL. The client driver does not interact with local devices but only with compute nodes. On a compute node, its daemon uses a native OpenCL implementation to interact with the compute node's devices on behalf of the client driver. The client driver and the daemons are connected using a communication library which allows the host to interact with the compute nodes and their devices.

In the current implementation of dOpenCL, we use the Generic Communication Framework (GCF) to implement network communication. GCF is a part of the Real-Time Framework [6], [7] which was originally developed for high-performance communication in distributed real-time applications like massively multi-player online computer games. While the GCF is not restricted to a particular communication stack, its current version supports TCP- and UDP-based communication.

In contrast to the standard OpenCL that does not provide any means to connect or disconnect remote systems in order to access their devices, in dOpenCL, an automatic connection mechanism enables OpenCL applications to run on a distributed system.

```
# connect to compute node 'gpuserver.example.com'
gpuserver.example.com
# connect to compute nodes in local network
128.129.1.1
128.129.1.2
```

Listing 1. Example configuration file listing available compute nodes.

The user can specify a list of available compute nodes in the target distributed system by a configuration file, like the one shown in Listing 1. The file is placed into the application's execution directory. Creating this file is the only additional effort required from the user in order to execute existing OpenCL applications on a heterogeneous distributed system. When the application requests a list of available devices from the client driver for the first time, the client driver automatically connects to the compute nodes specified in the configuration file. From each daemon, it obtains the list of available devices and merges them into a single list which is returned to the application.

## III. Using dOpenCL for Multi-User Systems

It is often desirable to run multiple applications simultaneously in a distributed system. Thereby, the system can be shared by multiple users, and the system can be used to its full capacity even if a single application is able to use only a fraction of the system's capacity. In both cases, the dOpenCL implementation presented in the previous section has a disadvantage: because each application can access all devices of the system, a device would be possibly used by multiple applications concurrently. While this is permitted by the dOpenCL implementation, the system may be used inefficiently.

As an illustration example, let us consider four applications, each requiring one GPU, that are executed on a distributed system comprising four nodes with one GPU each. With the dOpenCL implementation presented in the previous section, each application can choose its device from any of the four nodes. In particular, all applications might choose the GPU of the first node, while the three other nodes would remain idle. Specifying different nodes for each application is not an option, because the applications are independent from each other (e.g., started by different users), such that they do not know which nodes are already used or will be used next.

We extended the dOpenCL runtime system presented in Section II by a central, network-accessible *device manager*, in order to overcome this problem. The device manager partitions devices among multiple applications by restricting the host's access to particular devices. It ensures that each device is only used by one application at a time. A host can obtain devices from a given list of compute nodes and additionally request devices from the device manager. Like the connection mechanism, the device manager runs transparently for the application.

In order to integrate the device manager with the client driver and daemon, the following mechanisms are required:

- The device manager must be able to determine the available devices on all compute nodes.
- An application (i.e. its client driver) must be able to request devices from the device manager.
- A compute node must only give a host access to devices that the device manager has assigned to the application running on that host.

In the following, we briefly describe how these mechanisms are implemented in dOpenCL.

The device manager is installed either on one of the compute nodes or on a dedicated node of the distributed system, such that it can be used by multiple hosts simultaneously.

ISBN: 978-988-19251-8-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Internally, the device manager maintains two sets of devices: devices that are not assigned to a host (free) and assigned devices. In order to fill these initially empty sets, the daemons of the system's compute nodes connect to the device manager which obtains lists of devices from these nodes and adds them to its set of free devices. Thus, the device manager obtains information about all available compute nodes and devices. Unlike the client driver, the device manager does not actively connect to a fixed set of compute nodes, but rather waits for incoming connections from compute nodes. Compute nodes can also request to disconnect from the device manager, such that their devices are no longer used by an application.

When using the device manager, a daemon is started in the so-called *managed* mode. In this mode, the daemon automatically connects to the device manager, lets it obtain a device list and passes access control for these devices to the device manager. The address of the device manager is specified by a command line parameter provided by the user.

To request devices from the device manager, an application specifies the number and properties of the devices it requires. This assignment request contains the number and type of devices to be allocated and a set of properties that the requested devices should have. Eligible device properties are a subset of properties which can be requested using the standard OpenCL function `clGetDeviceInfo` (e.g., minimal global memory size and number of processing elements).

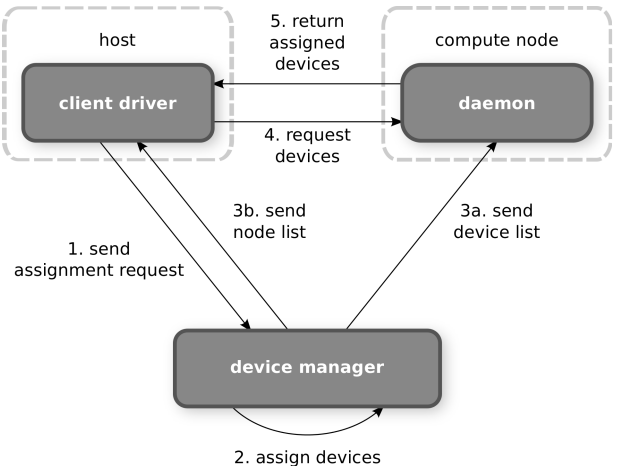


Fig. 2. Requesting devices from the dOpenCL device manager.

In order to request devices (see Figure 2), the client driver sends an assignment request to the device manager (1). The device manager assigns the devices (2) and returns a list of compute nodes to the client driver (3b) and a list of assigned devices to the daemons (3a). The client driver then connects to the compute nodes (4) from the received list to obtain the assigned devices from the daemons (5). This mechanism simplifies the configuration of applications, as the user only has to know the device manager's URL rather than all compute nodes' URLs.

As the standard OpenCL API does not provide any means for the described requesting process, we implement a new automatic device request mechanism. The user provides an XML-based configuration file which contains the address of the device manager and a list of properties for each type of device that should be requested from the device manager.

In order to assign devices to a host, the device manager of dOpenCL restricts other hosts' access to these devices. We implement this restriction mechanism using so-called *leases*. A lease comprises a unique authentication ID, a set of devices, and a set of compute nodes which own these devices. When the device manager receives an assignment request from a host, a new lease with a unique authentication ID is created. To create a device set for the lease, the device manager searches its set of free devices for devices which comply with the devices' properties from the assignment request. Appropriate devices are added to the device set and removed from the device manager's set of free devices.

The host requests the devices assigned to it from the compute nodes in the lease's compute node set. When connecting to the compute nodes, the host provides a valid authentication ID, otherwise the connection is rejected by the compute node. As the compute nodes are running in managed mode, they only give the host access to those devices that are associated with that ID. Thus, a host can only access devices that have been assigned to it by the device manager.

## IV. Application Studies and Experiments

We evaluate the performance of dOpenCL by running it on a number of heterogeneous desktop PCs. A dual-core (Intel Core2 6300 running at 1.86 GHz) computer is used as host, while several quad-core (Intel Core i7 860 running at 2.8 GHz) computers are used as default compute nodes. Besides, we use a GPU system equipped with a quad-core CPU (Intel Xeon E5520, 2.27 GHz) and an NVIDIA Tesla S1070 (4 GPUs with 4 GB of memory each) as a high-performance compute node. On the default compute nodes, the AMD Accelerated Parallel Processing SDK [8] is installed which provides the system's CPU as a single OpenCL device. The high-performance compute node uses the NVIDIA OpenCL driver (version 304.51) to provide four GPU devices. All nodes are connected via a Gigabit Ethernet network.

### A. Scalability: Basic Linear Algebra Subroutines

In order to evaluate the scalability of dOpenCL, we created OpenCL-based implementations of the SAXPY and SGEMM subroutines from the well-known Basic Linear Algebra Subprograms (BLAS) [9] API. BLAS subroutines are popular benchmarks as they are frequently used as building blocks in numerical applications. SAXPY (Single-precision real Alpha X Plus Y) tends to be memory-bound as it usually requires more time for reading and writing data than for computations, whereas SGEMM (Single-precision General Matrix Multiply) is usually compute-bound, i.e. it spends more time for computations than for reading/writing data.

Table I lists the sizes of the vectors and matrices we pass to each subroutine in our experiments along with the amount of data that has to be transferred between the host and compute nodes. We execute each subroutine on the host with and without using dOpenCL for comparison. With dOpenCL, we executed the subroutines on up to 8 compute nodes.

Figure 3 shows, for each subroutine, the measured average time for initialization, kernel execution, and data transfer. With dOpenCL, the kernel execution time for all subroutines

ISBN: 978-988-19251-8-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

| subroutine | vector size | matrix size | upload (MB) | download (MB) |
|---|---|---|---|---|
| SAXPY | 67108864 | – | 512 | 256 |
| SGEMM | – | 2048×2048 | 24 + 16×d | 16 |

TABLE I
SIZE OF VECTOR, MATRICES, AND DATA TRANSFERS FOR BLAS ROUTINES ON $d$ DEVICES.

decreases, as a single compute node has more computational power than the host (leftmost bar "w/o dOpenCL"). But the bandwidth for data transfers also decreases, as in dOpenCL the Gigabit Ethernet, rather than the host's system bus, limits the achievable bandwidth. This multiplies the applications' data transfer time by an order of magnitude. For the memory-bound SAXPY, the decreased computation time does not compensate for the increased data transfer time, such that the overall runtime increases.

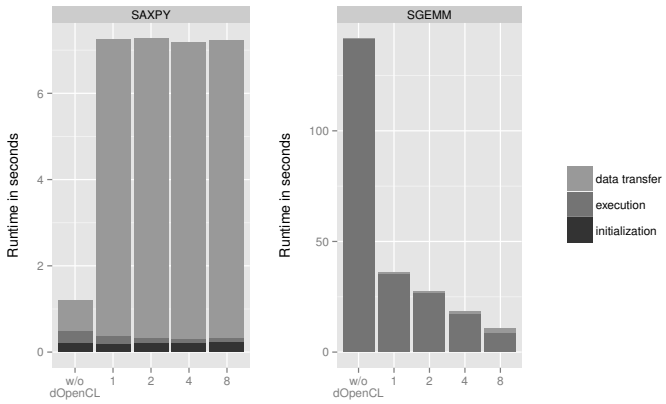


Fig. 3. Runtime of BLAS subroutines on up to 8 compute nodes.

We observe that the compute-intensive SGEMM is up to 12.8 times faster when using dOpenCL, as the increased data transfer time is negligible as compared to the decreased compute time. As the number of compute nodes increases, the kernel execution time of SGEMM reduces proportionally, whereas it stays constant for SAXPY. Unlike SAXPY, SGEMM does not equally split its input data to all compute nodes, but has to send a full copy of one of its input matrices to all compute nodes. Therefore, the data transfer time for SGEMM increases proportionally to the number of compute nodes. The time for initialization remains roughly constant in all subroutines.

### *B. dOpenCL vs. MPI+OpenCL: Mandelbrot Set*

To compare the programming effort and performance of dOpenCL against MPI mixed with OpenCL (MPI+OpenCL), we use the computation of a fractal of the Mandelbrot set which is a very popular benchmark application.

We adapted an existing OpenCL implementation of Mandelbrot computation to both programming approaches. With dOpenCL, we only have to provide a list of available compute nodes (like in Listing 1), while the application is not changed in any way. When using MPI+OpenCL, no node list is needed, but even such an embarrassingly parallel application as Mandelbrot required the following significant modifications of the original OpenCL program:

- Based on the MPI process rank and communicator size, an image tile (specified by its offset and size) is assigned to each node.
- This tile, rather than the complete image, is passed to the program's algorithm for Mandelbrot computation.
- The tiles are merged into a complete image using the `MPI_Gather` command.
- Initialization and finalization commands for the MPI runtime are added.

In order to compare the scalability and runtime of the dOpenCL- and MPI+OpenCL-based implementation, we executed both implementations on a cluster, with compute nodes connected via 4x QDR Infiniband. Each compute node is equipped with 2 hexa-core CPUs (Intel Westmere X5650, running at 2.6 GHz), which are accessible as a single device in OpenCL.

We measured the runtime of both application versions for computing a $4800 \times 3200$ fractal image with up to 10,000 iterations per pixel on 2, 4, 8, and so forth up to 32 nodes. In both versions, each line of the Mandelbrot fractal is computed by another device in a round-robin fashion, such that all devices are assigned an equal amount of work.

The results shown in Figure 4 demonstrate that both, the dOpenCL (left bar) and the MPI+OpenCL (right bar) versions scale well. As compared to the MPI+OpenCL program, the dOpenCL program introduces only a moderate overhead.

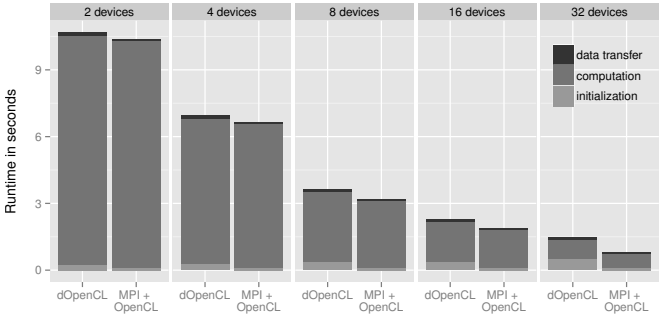


Fig. 4. Runtime of the Mandelbrot application using up to 32 devices.

The stacked view of the runtime with program initialization (bottom), computation (center), and data transfer (top) reveals that this overhead is introduced by program initialization and data transfer. The overhead for data transfer is fixed and does not increase as more nodes are employed by the application. MPI performs better in bulk data transfer than dOpenCL, because it directly uses the cluster's Infiniband fabric, while the current implementation of dOpenCL uses a TCP/IP stack which achieves only about 56% of the performance of the native Infiniband library. The initialization overhead is only relevant in dOpenCL and slightly increases with the number of nodes. Unlike the MPI runtime system which is initialized before the application is started, dOpenCL sets up its runtime system during the application's

ISBN: 978-988-19251-8-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

runtime, i.e. it connects to all compute nodes and obtains a list of available devices. Moreover, OpenCL's initialization process for building the OpenCL program binaries requires dOpenCL to transfer the OpenCL program code to all compute nodes over the network. In MPI, the compiled program binary has to be present on each node before the application is started. While the setup of the runtime system increases the initialization time, this also makes dOpenCL more flexible, as the runtime system is created dynamically before each execution.

## V. Related Work

Several distributed implementations of the OpenCL standard API have been proposed recently. *SnuCL* [10] implements the OpenCL API using MPI and provides a number of additional API functions which resemble collective operations in MPI. *Hybrid OpenCL* [11] is based on a modified version of the FOXC OpenCL runtime [12], such that it not only provides access to the devices of the system it is running on (the host system), but also to the devices on remote systems. An approach named *clOpenCL* [13] is an implementation of the OpenCL API for clusters. It uses Open-MX as communication library which provides a Myrinet communication stack for Ethernet networks. The *MOSIX Many GPUs Package* (MGP) [14] is a library and runtime system which aim at simplifying the programming of clusters with GPUs. MGP provides an API layer called *MOSIX Virtual OpenCL* which enables unmodified OpenCL applications to be executed on clusters.

While the objectives of the aforementioned approaches are similar to ours, none of them provides a central mechanism for assigning devices to multiple applications that are executed on a distributed system concurrently. We specifically address this issue in our approach. Moreover, our approach is not particularly focused on parallel execution on clusters, but also aims at distributed computing in local area networks.

## VI. Conclusion

This paper presents dOpenCL – a novel approach based on OpenCL for uniformly programming distributed heterogeneous systems comprising multi-core processors and multiple GPUs. Using dOpenCL, standard OpenCL applications can transparently access remote devices (CPU and GPU). Our approach, on the one hand, considerably extends the scope of OpenCL: by means of the device manager, devices of a distributed system can be shared efficiently between multiple OpenCL applications, and additional devices can be added transparently from clouds. On the other hand, our approach facilitates a seamless integration with existing OpenCL applications without the necessity to rewrite them.

Unlike mixed-mode programming approaches such as MPI+OpenCL, the dOpenCL programming model does not require existing OpenCL programs to be modified for being executed on a distributed system. For compute-intensive applications, like the presented SGEMM and Mandelbrot benchmarks, the overall runtime overhead introduced by dOpenCL is shown to be negligible. Even with a comparatively slow Gigabit Ethernet network, we achieved remarkable performance increase when using devices in a distributed system. We expect an additional performance imrovement by using an alternative implementation of dOpenCL for Infiniband networks which is currently under active development.

## Acknowledgment

The authors would like to thank NVIDIA Corp. for their generous hardware donation.

## References

[1] *MPI: A Message-Passing Interface Standard*, http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, Message Passing Interface Forum, 2009, version 2.2.

[2] *NVIDIA CUDA API Reference Manual*, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_Toolkit_Reference_Manual.pdf, February 2011, version 4.0.

[3] *POSIX, Part 1: System API, ANSI/IEEE Std 1003.1c, Amendment 2: Threads Extension*, IEEE Standards Press, Technical Committee on Operating Systems and Application Environments of the IEEE., 1996.

[4] A. Munshi, *The OpenCL Specification*, Beaverton, OR, 2010, version 1.1, Document Revision: 33.

[5] "DOpenCL," http://www.ohloh.net/p/DOpenCL.

[6] F. Glinka, A. Ploß, J. Müller-Iden, and S. Gorlatch, "RTF: A Real-Time Framework for Developing Scalable Multiplayer Online Games," in *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games*, ser. NetGames '07. New York, NY, USA: ACM, 2007, pp. 81–86.

[7] F. Glinka, A. Ploß, and S. Gorlatch, "RTF: Real Time Framework," http://www.real-time-framework.com/.

[8] "AMD Accelerated Parallel Processing SDK," http://developer.amd.com/sdks/amdappsdk.

[9] "BLAS (Basic Linear Algebra Subprograms)," http://www.netlib.org/blas/.

[10] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 341–352.

[11] R. Aoki, S. Oikawa, R. Tsuchiyama, and T. Nakamura, "Hybrid OpenCL: Connecting Different OpenCL Implementations over Network," in *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, ser. CIT '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 2729–2735.

[12] "FOXC, an OpenCL Compiler and Runtime," http://www.fixstars.com/en/opencl/foxc/.

[13] A. Alves, J. Rufino, A. Pina, and L. P. Santos, "clOpenCL – Supporting Distributed Heterogeneous Computing in HPC Clusters," in *Euro-Par 2012 Parallel Processing Workshops*, ser. Lecture Notes in Computer Science. Springer, to appear.

[14] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh, "A package for OpenCL based heterogeneous computing on clusters with many GPU devices," in *Workshop on Parallel Programming and Applications on Accelerator Clusters (PPAAC10), IEEE Cluster 2010*, September 2010.