dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems

Philipp Kegel, Michel Steuwer, and Sergei Gorlatch

Department of Mathematics and Computer Science
University of Münster, Münster, Germany
Email: {philipp.kegel,michel.steuwer,gorlatch}@uni-muenster.de

***Abstract*—Modern computer systems are becoming increasingly heterogeneous by comprising multi-core CPUs, GPUs, and other accelerators. Current programming approaches for such systems usually require the application developer to use a combination of several programming models (e. g., MPI with OpenCL or CUDA) in order to exploit the full compute capability of a system.**

In this paper, we present dOpenCL (Distributed OpenCL) – a uniform approach to programming distributed heterogeneous systems with accelerators. dOpenCL extends the OpenCL standard, such that arbitrary computing devices installed on any node of a distributed system can be used together within a single application. dOpenCL allows moving data and program code to these devices in a transparent, portable manner. Since dOpenCL is designed as a fully-fledged implementation of the OpenCL API, it allows running existing OpenCL applications in a heterogeneous distributed environment without any modifications. We describe in detail the mechanisms that are required to implement OpenCL for distributed systems, including a device management mechanism for running multiple applications concurrently. Using three application studies, we compare the performance of dOpenCL with MPI+OpenCL and a standard OpenCL implementation.

***Keywords*-OpenCL, Heterogeneous Systems, Distributed Systems, GPU Computing, dOpenCL**

I. Introduction

Modern distributed computer systems increasingly comprise heterogeneous processing units, e. g., multi-core CPU, GPU, FPGA, and other accelerators. A common approach to exploit the full compute capability of such systems is to employ several programming models simultaneously. For example, consider programming a cluster whose nodes are equipped with a multi-core CPU and a GPU: The programmer has to distribute the data to all compute nodes, e. g., using MPI [1] or even explicit, low-level network programming. To exploit the GPU on each node, another appropriate programming model, like CUDA [2], is needed, which requires the programmer to explicitly transfer data between the node's main memory and GPU. Moreover, in order to use all cores of the CPU, a thread programming model, e. g. Pthreads [3], is usually employed.

The downside of combining several programming models is the complicated task of switching between the models in different phases of an application. Not only the programmer has to master the employed programming models, but he also has to take into account their possible interference when used within a single program. In particular, these programming models do not have a common memory model, such that the programmer has to take care about memory consistency when switching from one model to another, e. g., finish data receipt using MPI before passing the data to a GPU. The programmer needs to have a comprehensive knowledge of all programming models to recognize such problems; moreover, he is often forced to use low-level synchronization mechanisms for ensuring the completion of data transfer in order to allow interaction of different programming models, because higher-level constructs (e. g., collective operations in MPI) are only applicable within a single programming model. While some alleviations have been suggested in the literature, e. g., creating one MPI process per core to avoid multi-threaded programming, programming a heterogeneous distributed system still remains a cumbersome task: the programmer has to deal with multiple programming models that mostly interact at low levels of abstraction and possibly in unexpected ways when used simultaneously.

We propose a uniform approach for programming heterogeneous, distributed computer systems. It is based on OpenCL [4], which recently has emerged as an open, widely accepted standard for heterogeneous systems. We thus rely on OpenCL's uniform programming model that provides access to multiple, possibly heterogeneous processing units, referred to as *devices*, for example multi-core CPU, GPU, or the Cell BE. Unfortunately, OpenCL is limited to stand-alone systems and has to be mixed with other programming models to create applications for distributed systems. Our approach extends the OpenCL programming model to distributed systems, such that the programmer is freed from using other programming models and implementing explicit network communication in a distributed system. While our proposed extension can be used in a transparent manner using standard OpenCL, we also provide an extension of the OpenCL API to explicitly use the additional features of our approach. Moreover, we provide a mechanism to efficiently execute

 DOI: 10.1109/IPDPSW.2012.16

multiple OpenCL-based applications in a distributed system concurrently.

The paper is organized as follows. We start with a discussion of related work in Section II. Section III introduces dOpenCL – an implementation of the OpenCL API based on a distributed runtime system. In Section IV, an extension of dOpenCL is presented which allows for multiple OpenCL applications to run concurrently in a distributed system. In Section V, we present three application studies to compare the performance dOpenCL with MPI and a default OpenCL implementation. We conclude our work in Section VI.

II. Related Work

Several distributed implementations of OpenCL have been proposed recently.

SnuCL [5] implements the OpenCL API based on MPI and provides a number of additional API functions which resemble collective operations in MPI. Unlike our approach, SnuCL only supports a limited set of devices, as it does not use existing OpenCL implementations.

Hybrid OpenCL [6] is based on a modified version of the FOXC OpenCL runtime [7], 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.

The *MOSIX Many GPUs Package* (MGP) [8] 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 these approaches provides a central mechanism for assigning devices to multiple applications that are executed on a distributed system concurrently. The authors also do not explain whether and how they implement the OpenCL mechanisms for memory consistency or synchronization in a distributed setting.

CLuMPI (OpenCL under MPI) is an OpenCL implementation based on MPI [9]. CLuMPI completely hides the distributed systems from the programmer, and, unlike other approaches, presents the system as a single OpenCL device. However, CLuMPI is limited to CPUs and does not give access to GPU or other accelerators.

Several other projects, e. g. DistributedCL [10] (formerly named OpenCLGrid), *CLara* [11], or *SocketCL* [12], have been started to create distributed OpenCL implementations. Most of these projects are in a rather preliminary state and often lack proper documentation.

Besides OpenCL-based approaches for programming distributed systems, solutions based on CUDA have been proposed. *CUDASA* [13] is an extension of CUDA to multi-GPU systems and clusters thereof. POSIX threads and MPI are used to manage the necessary communication between GPUs. *rCUDA* [14] is a distributed implementation of the CUDA API. With rCUDA, multiple CUDA-based applications can access and share GPUs in a network. The CUDA-based approaches are limited to devices that support CUDA, namely NVIDIA GPUs, such that they cannot be used to program multi-core CPUs or GPUs of other vendors.

III. Distributed OpenCL (dOpenCL)

The objective of *distributed OpenCL* (*dOpenCL*[1]) is twofold: 1) to provide access to arbitrary processing units in distributed systems, and 2) to free the application programmer from the complexity of mixing multiple models for programming such systems. dOpenCL is a fully-fledged implementation of the OpenCL programming model which has been extended to distributed systems. With dOpenCL, all devices of a distributed system – (multi-core) CPU, GPU, or other accelerators – are presented to the programmer as if they where installed in a single stand-alone system. dOpenCL completely hides the underlying distributed system from the programmer, such that he can use its devices uniformly by means of the standard OpenCL API. All data transfers between the nodes of the distributed system are performed implicitly. In particular, the programmer can rely on the uniform memory model and synchronization mechanisms which dOpenCL inherits from OpenCL. Thus, when using dOpenCL, the programmer does not have to worry about the possible interference of multiple programming models.

The key idea of the dOpenCL implementation is to forward the OpenCL API calls of an application to the OpenCL implementations that are installed on the nodes of the distributed system. Thus, it can be viewed as a *meta-implementation* of OpenCL, as it uses other OpenCL implementations to implement the OpenCL API. Whenever data has to be transferred between OpenCL implementations on different nodes, the data is transferred between these nodes over the system's network automatically.

To address the major problems that arise when enhancing the OpenCL programming model to work on distributed systems, we developed the following mechanisms in dOpenCL:

- A **runtime system** on each node of a distributed system supports the necessary communication between the nodes, e. g., forwarding OpenCL API calls.
- A mechanism to **automatically connect nodes** of a distributed system, as the OpenCL API does not provide any means for that.
- A mechanism to preserve **consistency of OpenCL management objects** (e. g., contexts, events, and memory objects, see Section III-A) on all nodes of a distributed system, to provide a virtual shared memory as required by the OpenCL programming model.

[1] dOpenCL is not related to *DOpenCL* which is an OpenCL binding for the D programming language

- A ***platform*** – an OpenCL management object – to enable existing OpenCL applications to use the dOpenCL implementation.

A. Programming model

As dOpenCL implements the OpenCL API, it inherits the main principles from the OpenCL programming model. A program is executed on a *host* system and offloads computations to OpenCL *devices*. In dOpenCL, the host system is represented by a single node, while the devices refer to processing units on other nodes of a distributed system. Special functions, called *kernels*, are executed on devices in a parallel manner. The host program specifies how many instances of a kernel – so-called *work-items* – are executed in parallel. All work-items execute the same code in a SPMD (Single Program, Multiple Data) fashion, i. e. the execution can diverge due to branching based on a global identifier, assigned to every work-item.

To address a wide range of different hardware platforms, kernels are compiled during the runtime of the host program. Similar to the standard OpenCL, the kernel's source code is passed to the dOpenCL implementation as a plain string to create an executable binary for a certain device. The binary can then be executed on that device.

In dOpenCL, the host system and the devices have different memory regions. Explicit data transfers are required to exchange data between these regions. API functions allow for allocating memory on the devices and for copying data from the host system to the devices (*upload*) or vice versa (*download*). Devices can share data, such that data is transparently moved between them. dOpenCL inherits OpenCL's so-called *relaxed consistency* memory model: sequential consistency is only guaranteed for read/write operations of a single work-item; to ensure memory consistency between different work-items or devices that share data, explicit synchronization measures are required from the programmer. Note, that dOpenCL does not define the memory consistency model on the host system.

All the functions that the dOpenCL API provides to the programmer, operate on the following set of *management objects* which represent the system and the application state: platforms, servers, devices, contexts, memory objects, programs, kernels, command queues, and events. The dOpenCL implementation itself is represented by a single dOpenCL platform object from which a list of available servers and devices can be queried. A context comprises a number of devices and management objects that are associated with any of these devices. Memory objects, i. e. buffers and images, contain data that is transferred between host and device. Programs represent OpenCL source code from which kernel objects are created. To execute kernels on devices, they are enqueued into a command queue that is associated with the device. As commands are usually executed asynchronously by a command queue, events are provided to synchronize command execution or to query its status if required. Most management objects are *shared* by all devices of a context: these objects and changes to them become visible to all devices at certain synchronization points.

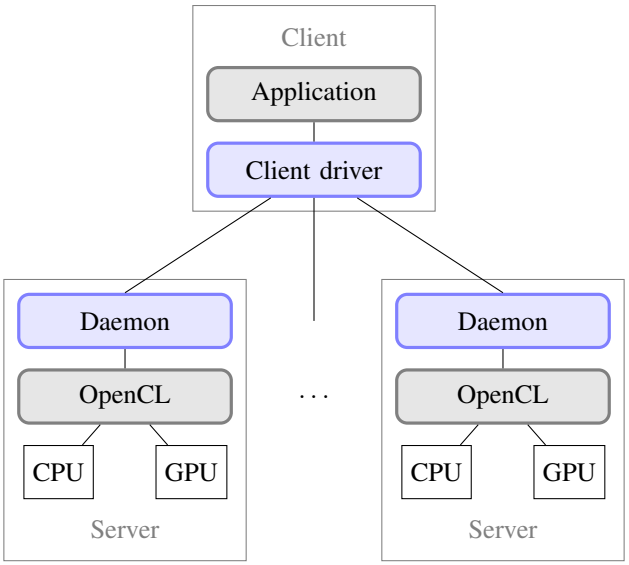

Figure 1. Architecture overview of dOpenCL.

B. Runtime system

The dOpenCL implementation is designed as a distributed application itself, as shown in Figure 1: it comprises one dOpenCL *client driver* and multiple dOpenCL *daemons* which are installed on the nodes of a distributed system. A node with the client driver is called *client*, while nodes that run a daemon are referred to as *servers*. The client (the host system) runs an OpenCL application, while the servers provide access to their devices over a network.

The client driver is a library which provides a fully-fledged implementation of the OpenCL API to an application. It is used as a drop-in replacement for an existing OpenCL implementation, such that an original OpenCL application does not have to be modified in any way (i. e., compiled or linked anew) in order to use dOpenCL. Note that the client driver does not allow an application to access OpenCL devices on the client. However, dOpenCL is compatible with the *ICD loader* (ICD – Installable Client Driver), which has been included in the OpenCL standard in order to use multiple OpenCL implementations simultaneously. Thus, an OpenCL application can use dOpenCL in combination with other OpenCL implementations which give access to the client's devices. Besides, a daemon can also be installed on a client for the same purpose. The main task of the client driver is to intercept calls to OpenCL API functions and redirect them to daemons that own the management objects which the functions refer to. The daemons continuously accept incoming function calls from the client driver and

forward them to their server's OpenCL implementation.

The client driver and the daemons are linked by a communication library which allows the client to control the devices on the servers. Two communication patterns are employed in dOpenCL: *message-based* communication and *stream-based* communication.

Message-based communication is used to execute OpenCL functions on a server and to exchange asynchronous notifications (e. g., object status updates) between client and servers. For executing an OpenCL function on a server, a specific request message is sent to the server. The server indicates the completion of the function call by sending a response message to the client.

Stream-based communication is used to exchange bulk data between client and servers. Unlike message-based communication which introduces a certain overhead due to the communication protocol, in stream-based communication only raw data is transferred. Bulk data transfers (up to several gigabytes in some applications) are used between nodes of a distributed system; they are performed asynchronously, i. e. the client does not wait for a data transfer to complete, as it does for most message-based operations. Moreover, the implementation of some OpenCL functions, e. g., for uploading a program to a device (`clCreateProgramWithSource`), includes bulk data transfers. These functions start by a request-response message exchange to initialize the data transfer before data is actually sent from the client to a server or back synchronously.

In dOpenCL, messages and application data have to be transferred over a network, before an OpenCL implementation on a server actually transfers that data to a device using the server's system bus. Network connections are usually considerably slower than a system bus, such that data transfers in dOpenCL can be up to an order of magnitude slower as compared to other OpenCL implementations which are designed for stand-alone systems. For example, Gigabit Ethernet, a widely used network type, provides a data transfer rate of up to 125 MB/s, whereas a PCIe bus provides data rates of 250 MB/s up to 16 GB/s. Only high-performance network types like Infiniband provide data rates which are similar to that of a system bus (from 250 MB/s up to 12 GB/s). Therefore, using a high-performance network and communication library is crucial for dOpenCL. In our implementation of dOpenCL, we use the Generic Communication Framework (GCF) to implement network communication. GCF is a part of the Real-Time Framework which was originally developed for high-performance communication in distributed real-time applications like massively multiplayer online computer games [15], [16]. In GCF, client and servers are represented by process objects; processes exchange messages which we use to implement message-based communication in dOpenCL. Additionally, we implemented bidirectional data streams for GCF in order to exchange large quantities of binary data as required by dOpenCL for stream-based communication. All communication is performed by GCF asynchronously, i. e. the client never waits for a communication operation to complete before it proceeds with other communications and computations in its program.

Some API functions do not require communication with servers and thus are directly implemented by the client driver. For example, platform information can be queried without any network communication. Also, most information on other OpenCL management objects is immutable and provided to the client driver during object creation, such that this information can be queried from the client driver at later phases of the execution without network communication. All other API functions depend on client-server communication.

The current implementation of dOpenCL is limited to a set of frequently used OpenCL API functions, which cover many OpenCL applications like the ones we present in Section V. API functions, for mapping memory objects (an alternative method for up- and downloading data), for images, samplers, or profiling have not been implemented yet. Moreover, OpenCL API extensions for sharing with OpenGL or Direct3D cannot be implemented for distributed systems, as these render graphical output on the local GPU (i.e. a server's GPU) for output rather than the client's GPU.

C. Connecting client and servers

OpenCL natively does not provide any means to connect or disconnect remote systems in order to access their devices. Rather, an OpenCL application can query a fixed set of devices which are installed in the host system. In dOpenCL, we introduce new functions to the OpenCL API (see Listing 1) which allow the programmer to increase or decrease the number of available devices during the application's runtime.

```
/* Connect to a server, adding its devices to the
   application's device list */
cl_server_WWU clConnectServerWWU(
    const char *url, cl_int *errcode);
/* Disconnect a server; its devices' states become
   'unavailable' */
cl_int clDisconnectServerWWU(
    cl_server_WWU server);
/* Query information about a server */
cl_int clGetServerInfoWWU(cl_server_WWU server,
    cl_server_info param_name, size_t param_size,
    void *param_value, size_t *param_size_ret);
```

Listing 1. OpenCL API extension for dynamic server allocation

The additional functions introduce a new management object called *server* to the OpenCL API. Servers can be connected or disconnected in dOpenCL at runtime, such that their devices become available or unavailable to the application, respectively. Moreover, the programmer can obtain a list of devices which are provided by each server.

The described API extension is optional, and it is not used by existing OpenCL applications. To enable these applications to use dOpenCL, we have implemented an automatic connection mechanism. The user can specify a list of available servers by a configuration file, like the one shown in Listing 2. The configuration file is placed into the

```
# connect to server 'gpuserver.example.com'
gpuserver.example.com
# connect to server in local network
128.129.1.1:7079
```

Listing 2. Example server configuration file. Servers can be specified using either their IP address or host name and an optional port number.

application's execution directory. During the application's initialization phase, when it obtains the list of available devices from the client driver, the client driver automatically connects to the servers 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. Therefore, existing OpenCL applications augmented with a simple configuration file can be executed on a heterogeneous distributed system.

D. Management and consistency of distributed objects

When using OpenCL, applications operate on management objects, e. g., contexts, command queues, or memory objects, provided by an OpenCL implementation. The daemons of dOpenCL use these objects of their server's standard OpenCL implementation to execute the commands from the client driver. For the client these objects are *remote* objects. Since remote objects are tied to the server's OpenCL implementation, two problems arise when using these objects in a distributed system: 1) objects cannot be transferred between client and servers (not even if the same OpenCL implementation is used on all servers), and 2) objects cannot be shared by devices on different servers.

To solve the first problem, we implement object *stubs* which the client driver provides to the OpenCL application as a replacement for the remote objects. Stubs are created by the client driver and assigned a unique ID which corresponds to a remote object. To interact with its remote object, a stub uses message-based communication. When an application calls an API function that refers to a stub, the client driver sends a specific request message with the function's input parameters to the server which holds the stub's remote object. Input parameters that refer to management objects, i. e. other stubs, are replaced by the stubs' IDs. On the server, the daemon replaces these IDs by the associated remote objects and calls the corresponding function of its standard OpenCL implementation. A response message with the function's return code and output parameters is returned to the client driver to confirm the call completion or to report an error. Thus, stubs enable an OpenCL application to control remote objects such that these do not have to be transferred to the client. For the programmer, this solution is transparent, as stubs can be used like management objects from any other OpenCL implementation.

Stubs are sufficient to replace remote devices and command queues, as these remote objects are only used by a single server. However, all other management objects (except for the `dOpenCL` platform object explained in Section III-E) are shared objects and require an enhanced solution. In dOpenCL, these objects have to be shared between servers, because devices that share such an object can reside on different servers. More precisely, a context in dOpenCL as well as every shared object associated with that context is associated with the set of servers that host the devices of that context. Stubs cannot replace shared objects, as they are only associated with a single remote object. Therefore, we introduce *compound* stubs. Unlike the previously described *simple* stubs, compound stubs are associated with multiple remote objects. Compound stubs assert consistency of their associated remote objects on multiple servers such that their state is shared by these servers. With compound stubs, the programmer still uses a single object rather than a set of simple stubs to control multiple remote objects. For example, when the programmer creates a context in dOpenCL using the function `clCreateContext`, appropriate request messages are sent to each server that hosts a device of the distributed context in order to create remote context objects on these servers. On the client, a compound stub along with an ID is created and associated with the remote contexts. Note that the contexts on a particular server are only associated with the devices that are hosted by that server, while the context represented by the compound stub is associated with all devices that have been specified by the programmer.

For samplers, programs, and kernels, compound stubs assert consistency by replicating API functions calls that refer to the compound stub to all associated remote objects. However, other shared objects, namely memory objects and events, require special measures for consistency, because they are not only modified by an application (using the compound stub) but their state can also be changed by a server's OpenCL implementation during kernel execution.

Consistency of memory objects: In order to assert consistency of remote memory objects on multiple servers in dOpenCL, we use a directory-based implementation of the MSI (Modified, Shared, Invalid) coherence protocol [17]. The remote memory objects are viewed as cached versions (copies) of the client's memory object stub and are assigned a status (initially "invalid"). For each memory object stub, the client maintains a status (initially "shared") and a list of servers (the directory) which own a valid remote memory object for this stub. When a remote memory object has been modified on a server, the object is marked "modified" on that server, while all other copies including the client's memory object stub become "invalid". When a server is

about to execute a command, it requires a valid copy of each memory object that will be read during execution of the command. Hence, it verifies the status of its local copies of these memory objects. For each copy whose status is "invalid", the server requests the client to upload a valid copy of this object, such that its status becomes "shared" (as memory objects cannot be copied between servers, here "copying" means to upload data for updating the invalid memory object). If the client also has no valid copy of this memory object (the stub's status is "invalid"), it downloads a valid copy from one of the servers in the memory object's shared list before uploading the object to the server that initially requested the update. The status of the client's memory object thus also becomes "shared" again.

Consistency of events: Unlike other management objects which are created explicitly by an application, events are created implicitly when a command is submitted to a device (using a command queue). Therefore, a remote event can only be created on the server owning that device; copies of this event cannot be created on other servers. In dOpenCL, we overcome this limitation by employing OpenCL *user events* and the following consistency protocol. When a command is submitted to a device in dOpenCL, the command is forwarded to the server that hosts the device, such that a remote event (the *original* event) is created on that server. The event is configured to send a notification to the client when its status changes to "completed" using the function `clSetEventCallback`. On all other servers, a user event is created as a replacement for the original event. Note that while user events can be created without submitting a command to a device, they can otherwise be used like normal events. When the command is completed, the status of the original event is changed to "completed" by the server's OpenCL implementation and a corresponding notification is sent to the client. The client then sets the status of the corresponding user event on all other servers to "completed". Thus, the status of the original event is available on all servers in a consistent manner.

E. The `dOpenCL` platform

Like other management objects, platforms cannot be transferred between client and servers in original OpenCL. This particularly means that each server provides its own platform object, even if the same OpenCL implementation is used on all servers. This has two disadvantages: 1) devices from different servers cannot be associated with the same context in order to share management objects, and 2) the number of platforms increases with each server, which increases the complexity of using devices from multiple servers.

To compensate for these disadvantages, the client driver introduces a platform called `dOpenCL`. This uniform platform is associated with all devices from all servers, such that they can be mixed in one context. Unlike a compound stub (see Section III-D), the uniform platform is not associated with any platform on a server, but is rather a self-contained entity. For example, all platform information is provided by the client driver and does not require communication with a server.

F. Server-to-server communication

In the current implementation of dOpenCL, all message exchange and data transfer is managed by the client, i. e. servers do not communicate directly, but rather use the client as a communication agent. This may considerably reduce communication performance when multiple servers are used, as communication links between servers are not used. To use the available communication bandwidth more efficiently, dOpenCL functions that employ multiple servers will be implemented using server-to-server communication. For example, memory objects on different servers can be synchronized by exchanging their data directly. This would also allow to use a more efficient coherency protocol like MOSI (Modified, Owner, Shared, Invalid) which takes advantage of sharing data between servers. Likewise, event status can be broadcasted directly by the server that owns the original event (see Section III-D). Server-to-server communication is particularly useful for applications which require many servers and high network throughput.

IV. Running multiple applications concurrently

Running multiple applications in a distributed system simultaneously is desirable for two reasons: 1) the system can be shared by multiple users, and 2) the system can be used to its full capacity even if a single application is only able to use a fraction of the system's capacity. However, without managing the allocation of devices to applications, the system may possibly be used inefficiently. For example, consider 4 applications each requiring 1 GPU that are executed on a distributed system comprising 4 servers with 1 GPU each. Each application can freely choose its required device from any of the 4 servers. In particular, all applications might decide to use the GPU of the first server, while the three other servers would stay idle. Specifying different servers 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 servers are already used.

To overcome this problem, we extended the dOpenCL runtime system presented in Section III by a central network-accessible *device manager*. The device manager is either installed on one of the servers or on a dedicated node of the distributed system, such that it can be used by multiple clients simultaneously. The device manager employs sophisticated scheduling strategies to share devices among multiple applications that use dOpenCL. In particular, it ensures that each device is only used by one application at a time when multiple applications are executed concurrently. The key

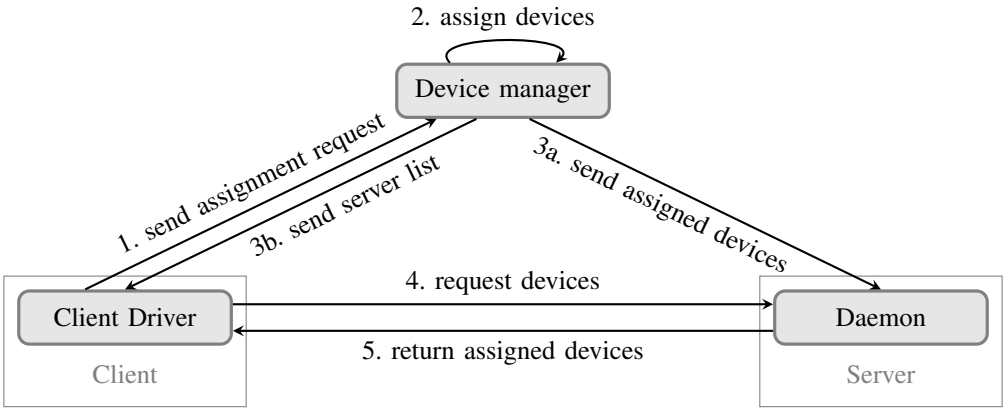

Figure 2. Requesting devices from the device manager.

concept of our device manager is to limit the number of devices a client has access to. Internally, the device manager maintains two sets of devices: devices that are not assigned to a client (free) and assigned devices. The device manager does not replace the connection mechanism described in Section III-C, but rather complements it. Thus, a client can obtain devices from a given list of servers and additionally request devices from a device manager. The device manager is fully transparent to 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 **connect to the servers**, such that it can create a list of all available devices.
- An application (i. e. the client driver) must be able to **request devices** from the device manager.
- A server must only give a client access to **devices** that the device manager has **assigned** to the application running on that client.

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

A. Connecting device manager and servers

To combine a daemon with the device manager, the daemon is started in the so-called *managed* mode. In this mode, the daemon automatically connects to the device manager. The address of the device manager is specified by a command line parameter provided by the user. The device manager receives the list of devices from the daemon's server, and adds it to its list of free devices. Thus, the device manager obtains information about all available servers and devices. Moreover, when in managed mode, the daemon only returns those devices to a client that the device manager has assigned to that client.

B. Requesting devices

To request devices from the device manager, an application specifies the number and properties of the devices it requires. This so-called *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 all properties which can be requested using the OpenCL API function `clGetDeviceInfo` (e. g., minimal memory size, number of processing elements). 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 servers to the client driver (3a) and a list of assigned devices to the daemons (3b). The client driver then connects to the servers (4) from the received list to obtain the assigned devices from the daemons (5).

As the standard OpenCL API does not provide any means for the aforementioned requesting mechanism, we implement a new automatic device request mechanism. The user provides an XML-based configuration file which contains the address of a device manager and a list of properties for each type of device that should be requested from the device manager. An example configuration file is shown in Listing 3. During the application's initialization phase, the client driver reads the configuration file and sends a corresponding assignment request to the device manager specified in the file.

```
<devmngr>devmngr.example.com</devmngr>
<devices>
  <device count="2">
    <attribute name="TYPE">CPU</attribute>
    <attribute name="VENDOR">Intel</attribute>
    <attribute name="MAX_COMPUTE_UNITS">2
      </attribute>
  </device>
  <device>
    <attribute name="TYPE">GPU</attribute>
  </device>
</devices>
```

Listing 3. Example configuration file: two Intel dual-core CPUs and a GPU are requested from the device manager available at `devmngr.example.com`.

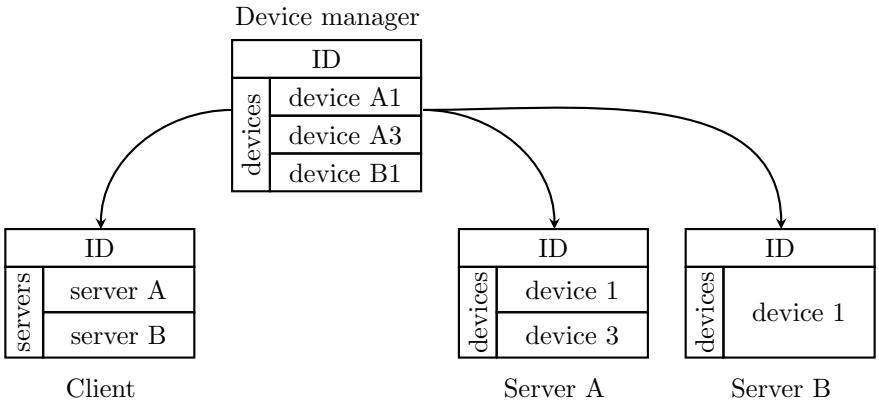

Figure 3. Assignment of devices: a lease is split into a server list for the client and a device list for each of that servers.

C. Assigning devices to clients

In order to assign devices to a client, the device manager has to limit the number of servers and devices the client can access. We implement this restriction mechanism using so-called *leases*. A lease comprises a unique authentication ID, a set of devices, and a set of servers which own these devices. When the device manager receives an assignment request from a client, 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, such that they cannot be associated with other leases. The lease's server set is computed from the device set, such that it comprises all servers that own at least one of the devices from the device set.

The device manager sends the lease's authentication ID and a subset of the lease's devices set to all servers from its server set (see Figure 3). Each server is sent a different subset, as this subset is the intersection of the server's device list and the lease's device set, such that a subset contains the devices which the server should associate with the authentication ID. The client is also sent the authentication ID by the device manager along with the lease's server set. An error code is sent to the client if the device manager was not able to find an appropriate device for some of the devices specified in the assignment request.

To finally obtain the devices it has been assigned, the client requests these devices from the servers in the lease's server set. When connecting to the servers, the client has to provide a valid authentication ID, otherwise the connection is rejected by the server. As the servers are running in managed mode (see Section IV-A), they only give the client access to those devices that are associated with that ID. Thus, a client can only access devices that have been assigned to it by the device manager.

In order to re-assign devices to other clients, idle devices are returned to the device manager's set of free devices. Usually, a client returns a lease when its application is finished, by sending a corresponding *release* message containing the authentication ID to the device manager. The device manager forwards this message to the servers in the lease's server set, such that these servers discard the authentication ID. Besides, the device manager returns all devices from the associated lease's device set to its list of free devices before deleting the lease.

When an application terminates abnormally or if the client is disconnected from the network, it cannot send the release message. Therefore, we implemented an additional mechanism to release assigned devices. When a client disconnects from a server, its authentication ID becomes invalid, i. e. a client cannot access the devices that are associated with that ID any longer. The daemon (rather than the client) sends a release message to the device manager to report the invalidated authentication ID, such that the device manager will return the associated devices to its set of free devices.

V. Experiments

In this section, we present three series of experiments to evaluate the performance of dOpenCL in various application scenarios:

a) a scalability benchmark (Mandelbrot set),
b) a real-world, time-intensive application from the field of computer tomography, and
c) a network bandwidth testing application.

While the first experiment evaluates dOpenCL as an approach for programming clusters using a single programming model, the other experiments show that dOpenCL can be used to share high-performance devices via a network.

A. Application example: Mandelbrot Set

The computation of a fractal of the Mandelbrot set is a popular benchmark application, which we use to study the programming effort and performance of dOpenCL. A

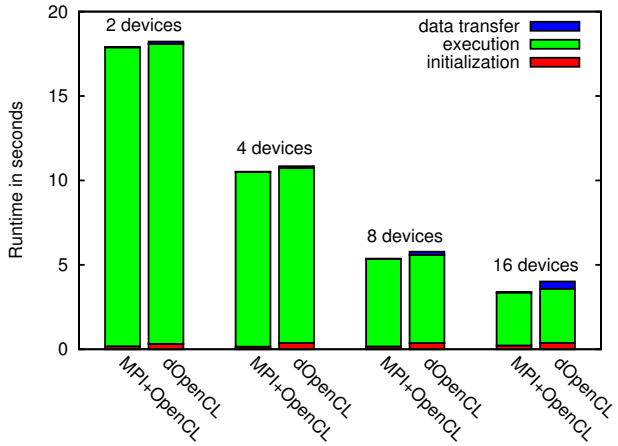

Figure 4. Runtime of the Mandelbrot application using up to 16 devices.

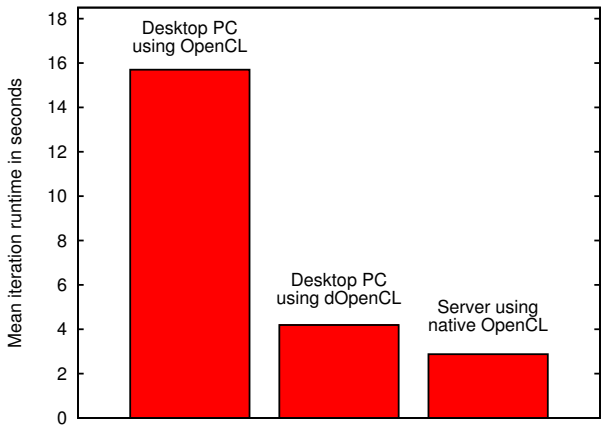

Figure 5. Mean runtime of an iteration of the list-mode OSEM algorithm.

Mandelbrot fractal is a section of the complex numbers plane where each pixel corresponds to a complex number. The pixel's color indicates if the complex number is a member of the Mandelbrot set (black) or not (any other color). An iterative algorithm is used to determine whether a complex number is part of the Mandelbrot set or not. A user-defined threshold limits the number of iterations for each pixel. The higher this threshold is, the higher is the program's algorithmic density.

We compare dOpenCL and a mix of MPI and OpenCL, by adapting an existing OpenCL application for computing a Mandelbrot fractal to both programming approaches. With dOpenCL, we only have to provide a configuration file with a list of servers, while the application is not changed in any way. When using MPI+OpenCL, even this embarrassingly parallel application required the following modifications:

- 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 computing a Mandelbrot fractal.
- The tiles are merged into a result image using the `MPI_Gather` command.
- Initialization and finalization commands for the MPI runtime are added.

In both application versions, each line of the fractal is computed by another device in a round-robin fashion, such that all devices are assigned an equal amount of work.

To evaluate the scalability and runtime of dOpenCL as compared to the MPI+OpenCL implementation, we executed both versions of the application on a cluster, whose compute nodes are connected via Infiniband. Each compute node is equipped with 2 hexa-core CPUs (Intel Westmere X5650, running at 2.6 GHz), which are presented as a single device by the installed OpenCL implementation (AMD Accelerated Parallel Processing SDK [18]). We measured the runtime of both application versions for computing a 4800×3200 fractal image with up to 20000 iterations per pixel on 2, 4, 8, and 16 nodes. The results shown in Figure 4 demonstrate that both, the MPI+OpenCL (left bar) and dOpenCL (right bar) versions scale well. As compared to the MPI+OpenCL program, the dOpenCL program introduces only a moderate and fixed overhead. The stacked view of the runtime for program initialization (bottom), computation (center), and data transfer (top) reveals, that this overhead is only introduced by program initialization and data transfer. Unlike MPI which requires the program binaries to be present on all nodes before execution, dOpenCL transfers function calls and OpenCL program code over the network during the program's runtime. Therefore, we attribute the overhead to message-based communication.

B. Application example: List-mode OSEM

In addition to the benchmark Mandelbrot application, we studied a real-world application from the field of computer tomography. Positron Emission Tomography (PET) is a non-invasive 3D imaging technology based on radioactive *tracers* that are applied to a patient. When scanning the patient, huge amounts of data are recorded and then processed by an imaging algorithm to create 3D images of the patient. We study the *list-mode OSEM* [19], [20] – an iterative, computation-intensive imaging algorithm. In our previous work [21], [22], we developed an OpenCL-based implementation of this algorithm for stand-alone systems with high-end GPUs, which we use for our measurements here. The list-mode OSEM application shows a limited scalability when running on a large number of devices. Hence, using dOpenCL to execute the list-mode OSEM on a cluster, as we did with the Mandelbrot application, is of limited use. But with dOpenCL, the application can be run on a desktop PC, while the computation is performed on a multi-GPU

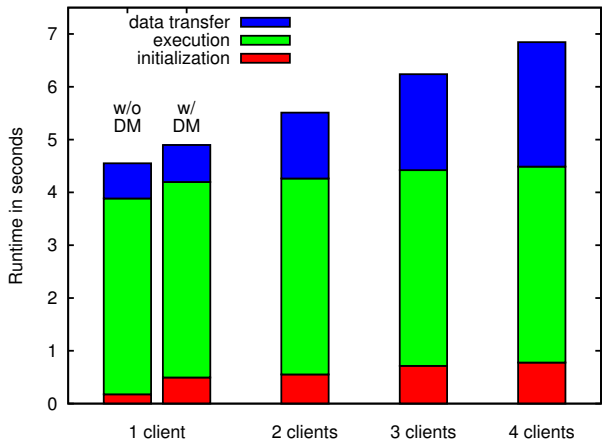

Figure 6. Average runtime of the Mandelbrot application when multiple application instances are executed concurrently on a single server.

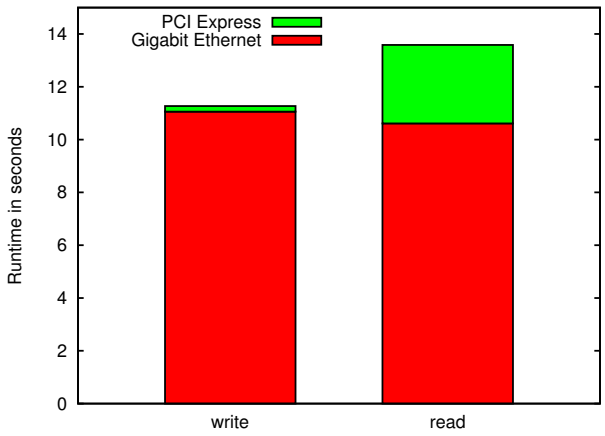

Figure 7. Time to transfer 1024 MB of data over Gigabit Ethernet vs. PCI Express to a device (write) and from a device (read).

server that is connected to the PC via a network. Without dOpenCL, only the local low-end GPU of the desktop PC could be used.

We executed the list-mode OSEM algorithm on a desktop PC with a low-end GPU (NVIDIA NVS 3100M) using dOpenCL. A GPU server 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) is connected to the PC via a Gigabit Ethernet network. The runtimes are shown in Figure 5. We observe that the execution using OpenCL on the low-end GPU is 3.75 times slower than when using dOpenCL (15.7 sec vs. 4.2 sec). Since in real-world applications the number of iterations reaches up to several hundred, this improvement results in reducing the overall runtime by hours. The trade-off for this performance gain is given by the additional costs for data transfer to or from the server, which is not required if the application had been executed on the server directly using its native OpenCL implementation.

C. Device Manager

By means of the dOpenCL device manager, multiple applications can efficiently share the devices of a distributed system. In particular, devices of a single node can be used by these applications concurrently. In order to demonstrate the efficiency of the device manager, we used it to share the devices of our GPU server (see Section V-B) between up to four applications. We connected four desktop PCs to the GPU server and executed the Mandelbrot application from our first experiment on 1 to 4 of these PCs simultaneously. The application had been configured to request a single GPU device from the device manager, rather than connecting to the server directly, while the GPU server ran in managed mode and connected to the device manager.

We measured the average runtime of a single application instance, while 1 to 4 applications are concurrently executed on the server. The results are shown in Figure 6. Note that the results are not comparable to the ones from our experiments in Section V-A as we use GPU devices instead of CPUs and a Gigabit Ethernet Network instead of Infiniband. The results reveal two important features of the device manager: First, the application runtime for computation does not increase when the number of concurrent application instances increases, because the device manager schedules the applications to different devices on the server. Second, the device manager introduces only a small and constant initialization overhead as compared to using the server directly, as shown for the execution of a single application instance. When the number of application instances increases, initialization time of each application increases because the server has to create more management objects. Also, the time for data transfer increases, as the servers network bandwidth has to be shared by all applications. When we performed the same experiment without the device manager all applications still worked correctly, but all dOpenCL commands were scheduled to the same device on the server. Therefore, the commands were arbitrarily interleaved and executed sequentially, such that an application instance ran up to 4 times longer. The runtimes of the application instances differ considerably in each execution, since they depend on the execution order of dOpenCL commands on the server. This order is determined by the order in which the server receives the commands from the client and by the execution strategy of the server's native OpenCL implementation (provided by the NVIDIA GPU driver in our case).

D. Data transfer

In order to evaluate the overhead due to the network communication in dOpenCL, we created a simple OpenCL application that transfers an arbitrary amount of data from the host to a device and vice versa. We measured the transfer time of data chunks of 1 to 1024 MB to and from the

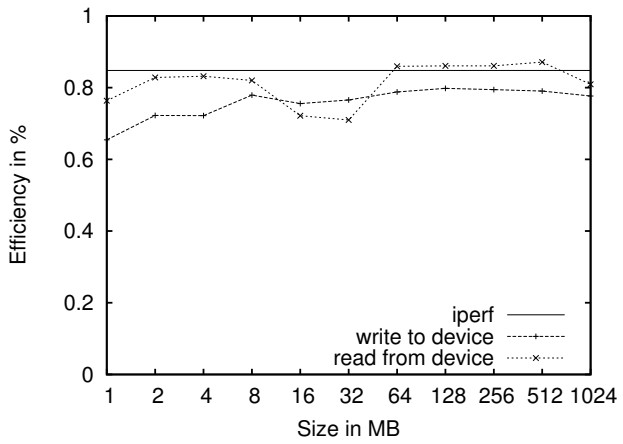

Figure 8. Efficiency of data transfer in dOpenCL using Gigabit Ethernet.

first device of the GPU server we used in our previous experiments. The application has been executed on the server directly using the NVIDIA OpenCL implementation to measure the effective PCI Express bandwidth to and from the device. Measurements reveal that data is written to the device with a bandwidth of about 38.8 GB/s. We observed a significantly slower data transfer when reading data from the device: it took up to 15 times longer than writing the same data to the same device.

When the same application is executed on a remote client that is connect to the server using dOpenCL, the data is transferred over Gigabit Ethernet to the server, then transferred to the device and afterwards sent back to the client. Not surprisingly, the transfers over the Gigabit Ethernet took more time than the transfers to the device: as shown in Figure 7, it is about 4.5 times slower as compared to the PCI Express when reading data from the device and up to 50 times slower when writing data.

To put these results into perspective, we compared the measured transfer times with the achievable bandwidth of Gigabit Ethernet. While the theoretical bandwidth performance of this network is about 125 MB/s, we measured an effective bandwidth of about 106 MB/s by means of the widely used network measurement tool *iperf* [23]. Figure 8 shows the efficiency of the data transfer as the percentage of the theoretically available bandwidth. The solid line at 86% shows the average effective bandwidth measured using the network measurement tool `iperf`. The chart shows that dOpenCL uses the available bandwidth of Gigabit Ethernet quite well, i. e. the overhead introduced by dOpenCL itself is quite small. Therefore, we expect that data transfer times for the dOpenCL will become significantly lower when using high-performance networks, like InfiniBand, since it provides a bandwidth similar or even better than PCI Express.

VI. Conclusion

We presented dOpenCL, an extended OpenCL API for using OpenCL in distributed, heterogeneous computing systems. Using dOpenCL, applications can transparently access OpenCL devices (CPU and GPU) located at remote systems. Our approach, on the one hand, considerably extends the scope of OpenCL; on the other hand, it facilitates a seamless integration with existing OpenCL applications.

Unlike mixed-mode programming approaches such as MPI+OpenCL, dOpenCL does not require existing OpenCL programs to be modified for being executed on a distributed system. For compute-intensive applications, like the presented Mandelbrot example, the overall overhead introduced by dOpenCL is negligible. We demonstrated that our real-world tomography application gains huge performance benefits using dOpenCL by transparently offloading calculations on a high-performance server, rather than using a low-end desktop computer. Our experiments show that the dOpenCL implementation makes efficient use of the available network bandwidth, even though an expectable overhead is introduced to every data transfer as compared to execution on a stand-alone system. Even with a comparatively slow Gigabit Internet, we achieved remarkable performance increase when using devices in a distributed system.

Acknowledgment

We would like to thank NVIDIA for their hardware donation, as well as Thomas Kösters and Klaus Schäfers (European Institute for Molecular Imaging, WWU Münster) for providing the reconstruction software EMrecon [20] and the quadHIDAC PET data used in our application case study. We also thank our students Arne-Knut Hörig, Matthias Kemper, Sebastian Mißbach, Alexander Sajzew, Simon Schroer, and Jan-Gerd Tenberge for their contribution to the implementation of the dOpenCL device manager.

References

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

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

[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] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "OpenCL as a programming model for GPU clusters," in *LCPC'11: Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing*, September 2011.

[6] 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. [Online]. Available: http://dx.doi.org/10.1109/CIT.2010.457

[7] "FOXC, an OpenCL compiler and runtime." [Online]. Available: http://www.fixstars.com/en/opencl/foxc/

[8] 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.

[9] A. Woodland, "CLuMPI (OpenCL under MPI)." [Online]. Available: http://sourceforge.net/projects/clumpi/

[10] A. Tupinambá, "DistributedCL." [Online]. Available: http://sourceforge.net/projects/distributedcl/

[11] B. König, "CLara - OpenCL across the net." [Online]. Available: http://sourceforge.net/projects/clara/

[12] Casten, "SocketCL." [Online]. Available: http://sourceforge.net/projects/socketcl/

[13] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl, "CUDASA: Compute Unified Device and Systems Architecture," in *Eurographics Symposium on Parallel Graphics and Visualization EGPGV08*. Eurographics Association, 2008, pp. 49–56.

[14] J. Duato, J. Peña, A. F. Silla, R. Mayo, and S. Quintana-Orti, E. "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, Caen, France, July 2010, pp. 224–231. [Online]. Available: http://www.rcuda.net

[15] F. Glinka, A. Ploß, J. Müller-lden, 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. [Online]. Available: http://doi.acm.org/10.1145/1326257.1326272

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

[17] J. Handy, *The cache memory book*. San Diego, CA, USA: Academic Press Professional, Inc., 1993.

[18] "AMD Accelerated Parallel Processing SDK." [Online]. Available: http://developer.amd.com/sdks/amdappsdk

[19] A. J. Reader, K. Erlandsson, M. A. Flower, and R. J. Ott, "Fast accurate iterative reconstruction for low-statistics positron volume imaging," *Physics in Medicine and Biology*, vol. 43, no. 4, pp. 823–834, April 1998.

[20] T. Kösters, K. Schäfers, and F. Wübbeling, "EMRECON: An expectation maximization based image reconstruction framework for emission tomography data," in *NSS/MIC Conference Record, IEEE*, 2011. [Online]. Available: http://emrecon.uni-muenster.de

[21] M. Steuwer, P. Kegel, and S. Gorlatch, "SkelCL – A portable skeleton library for high-level GPU programming," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, 2011, pp. 1176–1182.

[22] M. Steuwer, P. Kegel, and S. Gorlatch, "Towards high-level programming of multi-GPU systems using the SkelCL library," in *2012 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, 2012.

[23] "iperf – TCP and UDP bandwidth performance measurement tool." [Online]. Available: http://code.google.com/p/iperf/

BIOGRAPHIES

Philipp Kegel

Philipp Kegel has been a PhD student at the University of Muenster (Germany) since 2008. His main topic of research are programming models for novel parallel system architectures, in particular multi-/many-core CPUs and GPUs. He is the main developer of the dOpenCL middleware. Besides, he has been and is involved in a research project from the field of medical imaging funded by a German national body. Philipp Kegel currently has 9 peer-reviewed publications in renowned international journals and conferences. He holds a Diploma degree in computer science from the University of Muenster.

Michel Steuwer

Michel Steuwer received his Diploma degree in computer science from the University of Muenster (Germany) in 2010. He then started his PhD studies under the supervision of Sergei Gorlatch. His main research interest include high-level programming models for modern parallel architectures, especially GPUs. He is the main developer of the SkelCL library, which allows for simplified application development targeting modern parallel systems, comprising of multiple GPUs and mulit-core CPUs. Michel Steuwer currently has 4 peer-reviewed publications at well-respected international conferences.

Sergei Gorlatch

Sergei Gorlatch has been Full Professor of Computer Science at the University of Muenster (Germany) since 2003. Earlier he was Associate Professor at the Technical University of Berlin, Assistant Professor at the Univeisity of Passau, and Humboldt research Fellow at the Technical University of Munich. Prof. Gorlatch has more than 150 peer reviewed publications in renowned international journals and conferences. He is often delivering invited talks at international conferences and serves at their programme

committees. He was principal investigator in several international research and development projects in the field of parallel, distributed, Grid and Cloud computing, funded by the European Commission, as well as by German national bodies. Among his recent achievements in the area of Internet Technology is the Real-Time Framework (http://www.real-time-framework.com) developed in his group as a novel platform for high-level development of real-time, highly interactive applications like multi-player online games, advanced e-Learning, crowd simulations, etc. Sergei Gorlatch holds MSc degree from the State University of Kiev, PhD degree from the Institute of Cybernetics of Ukraine, and the Habilitation degree from the University of Passau (Germany).