libWater: Heterogeneous Distributed Computing Made Easy

Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer
Institute of Computer Science, University of Innsbruck, Austria
{grasso, spellegrini, cosenza, tf}@dps.uibk.ac.at

ABSTRACT

Clusters of heterogeneous nodes composed of multi-core CPUs and GPUs are increasingly being used for High Performance Computing (HPC) due to the benefits in peak performance and energy efficiency. In order to fully harvest the computational capabilities of such architectures, application developers often employ a combination of different parallel programming paradigms (e.g. OpenCL, CUDA, MPI and OpenMP), also known in literature as hybrid programming, which makes application development very challenging. Furthermore, these languages offer limited support to orchestrate data and computations for heterogeneous systems.

In this paper, we present libWater, a uniform approach for programming distributed heterogeneous computing systems. It consists of a simple interface, compliant with the OpenCL programming model, and a runtime system which extends the capabilities of OpenCL beyond single platforms and single compute nodes. libWater enhances the OpenCL event system by enabling inter-context and inter-node device synchronization. Furthermore, libWater's runtime system uses dependency information enforced by event synchronization to dynamically build a DAG of enqueued commands which enables a class of advanced runtime optimizations. The detection and optimization of collective communication patterns is an example which, as shown by experimental results, improves the efficiency of the libWater runtime system for several application codes.

Categories and Subject Descriptors

C.1.3 [**Other Architecture Styles**]: Heterogeneous (hybrid) systems; D.3.2 [**Language Classifications**]: Concurrent, distributed, and parallel languages; D.3.4 [**Processors**]: Runtime Environments

Keywords

OpenCL, MPI, distributed computing, heterogeneous computing, programming model, runtime system

ICS'13, June 10–14, 2013, Eugene, Oregon, USA.

1. INTRODUCTION

Recently, the major chip manufacturers have faced the problem of three walls [7]: the power wall, the instruction level parallelism wall, and the memory wall. This turned the development of hardware design towards multi- and many-core CPUs, next to special purpose hardware and accelerators such as GPUs. As a result, engineers and academics are pervasively embracing the use of heterogeneous architectures to attain the highest performance possible from a single compute node.

However, heterogeneous computing also poses the new challenge of how to handle the diversity of execution environments and programming models. The Open Computing Language [19] introduces an open standard for general-purpose parallel programming of heterogeneous systems. An OpenCL program may target any OpenCL-compliant device and today many vendors such as Adapteva, Altera, AMD, IBM, Intel and NVidia provide an implementation of the OpenCL standard. An OpenCL program comprises a *host* program and a set of *kernels* intended to run on a compute device. It also includes a language for kernel programming, and an API for transferring data between host and device memory and for executing kernels.

Single node hardware design is shifting to a heterogeneous nature. At the same time many of today's largest HPC systems are clusters that combine heterogeneous compute device architectures [4]. Although OpenCL has been designed to work with multiple devices, it only considers local devices available on a single machine. However, the host-device semantics can be potentially applied to remote, distributed devices accessible on different compute nodes.

Porting single-node multi-device applications to clusters that combine heterogeneous compute device architectures is not straightforward and, in addition, it requires the use of a communication layer for the data exchange between nodes (e.g. MPI [25]). Writing programs for such platforms is error prone and tedious. Therefore new abstractions, programming models and tools are required to deal with this problem.

This paper introduces *libWater*, a library-based extension of the OpenCL programming paradigm that simplifies the development of applications for distributed heterogeneous architectures. *libWater* aims to improve both productivity and implementation efficiency when parallelizing an application targeting a heterogeneous platform by achieving two design goals: (*i*) transparent abstraction of the underlying distributed architecture, such that devices belonging to a remote node are accessible like a local device; (*ii*) access to

performance-related details since it supports the OpenCL kernel logic.

The main contributions of this paper are:

- The presentation of the *libWater* programming model, which extends the OpenCL standard by replacing the host code with a simplified interface. The definition of a novel device query language (DQL) for OpenCL device management and discovery.
- A lightweight distributed runtime environment which dispatches the work between remote devices, based on asynchronous execution of both communications and OpenCL commands. *libWater* runtime also collects and arranges dependencies between commands in the form of a powerful representation called *command DAG*.
- A demonstration of how the *command DAG* can be effectively exploited to improve the scalability. For this purpose we introduce a collective communication pattern recognition analysis and optimization that matches multiple single point-to-point data transfers and dynamically replaces them with a more efficient collective operation (e.g. *scatter*, *gather* and *broadcast*) supported by MPI.
- A study of the scalability of *libWater* on a real production cluster using up to 64 homogeneous compute nodes. Results show that an efficiency of around 64% is achieved, on average, for 6 application codes. Finally we demonstrate the suitability of *libWater* for a heterogeneous GPU cluster for two codes.

The rest of the paper is organized as follows. Section 2 and 3 provide an introduction to OpenCL and libWater programming model. Section 4 describes the distributed runtime system and the underlying command DAG representation. The runtime optimizations are treated in Section 5 and the experimental evaluation is presented in Section 6. Section 7 and 8 discuss related work and conclusions.

2. THE OPENCL PROGRAMMING MODEL

OpenCL is an open industry standard for programming heterogeneous systems. The language is designed to support devices with different capabilities such as CPUs, GPUs and accelerators. The platform model comprises a *host* connected to one or more *compute devices*. Each device logically consists of one or more compute units (CUs) which are further divided into processing elements (PEs). Within a program, the computation is expressed through the use of special functions called *kernels* that are, for portability reason, compiled at runtime by an OpenCL driver. Interaction with the devices is possible by means of *command-queues* which are defined within a particular OpenCL *context*. Once enqueued, commands – such as the execution of a kernel or the movement of data between host and device memory – are managed by the OpenCL driver which schedules them on the actual physical device.

Commands can be enqueued in a blocking or non-blocking way. A non-blocking call places a command on a command-queue and returns immediately to the host, while a blocking-mode call does not return to the host until the command has been executed on the device. For synchronization purpose, within a context, *event* objects are generated when kernel and memory commands are submitted to a queue. These objects are used to coordinate execution between commands and enable decoupling between host and devices control flows.

Despite being a well designed language that allows the access to the compute power of heterogeneous devices from a single, multi-platform source code base, OpenCL has some drawbacks and limitations. One of the major drawbacks is that, because being created as a low-level API, a significant amount of boilerplate code is required even for the execution of simple programs. Developers have to be familiar with numerous concepts (i.e. *platform*, *device*, *context*, *queue*, *buffer* and *kernel*) which make the language less attractive to novice programmers. Another important limitation is that, although it was designed to address heterogeneous systems, in case of devices from different vendors, objects belonging to the context of one vendor are not valid for other vendors. This limitation clearly becomes a problem when synchronization of command queues across different contexts is needed.

3. THE LIBWATER PROGRAMMING INTERFACE

libWater is a C/C++ library-based extension of the OpenCL programming paradigm that simplifies the development of distributed heterogeneous applications. It inherits the main principles from the OpenCL programming model trying to overcome its limitations. While maintaining the notion of host and device code, *libWater* exposes a very simple programming interface based on four key concepts: *device*, *buffer*, *kernel* and *event*. A *device* represents a compute device, but differently from the original paradigm this single object is an abstraction of the OpenCL platform, device, queue and context concepts. Such simplification reduces the number of source code lines necessary for the initialization of the devices, and thus avoids the boilerplate configuration code that is usually present in every OpenCL program. Furthermore, the library is not restricted to a single node but, taking internally advantage of the message passing model, it provides access to devices on remote nodes as if they were locally available.

Since *libWater* can grant access to a large number of distinct devices, the selection of a particular one can be cumbersome. In order to simplify this important aspect, *libWater* introduces a novel domain specific language for querying devices. A *device query language* (DQL) query statement follows an SQL-like structure, that is composed of 4 basic clauses with the following syntax:

```
SELECT          [ALL | TOP k | POS i]
    FROM NODE   [n [, ...]]
    WHERE       [restrictions attribute values]
    ORDER BY    [attribute [, ...]]
```

The `SELECT` clause (the only one which is mandatory) respectively allows the selection of all the devices, the first top *k*, or a particular device from the device list generated under the restrictions on the following clauses. With `FROM NODE` a single node or a list of nodes can be specified narrowing the range of selectable devices to those particular nodes. The clauses `WHERE` and `ORDER BY` allow the control of the device restrictions on attribute values and the order in which the devices will be returned. The possible attribute values are currently those exposed by the OpenCL `clGetDeviceInfo` function. A DQL use case is shown and discussed in Section 4.2. DQL

Device Management (**wtr_**)	
`void` **init_devices**`('DQL', ...)`	`device` **get_device**`('DQL', ...)`
`int` **get_num_devices**`()`	`void` **release_devices**`()`
`void` **print_device_infos**`(device)`	

Buffer Management (**wtr_**)
`buffer` **create_buffer**`(device, mem_flag, size, evt)`
`void` **write_buffer**`(buffer, size, source_ptr, wait_evt, evt)`
`void` **read_buffer**`(buffer, size, dest_ptr, wait_evt, evt)`
`void` **release_buffer**`(buffer, wait_evt, evt)`

Kernel Management (**wtr_**)
`kernel` **create_kernel**`(device, name, kernel_name, build_options, flag, evt)`
`void` **run_kernel**`(kernel, work_dim, global_work_size, local_work_size, wait_evt, evt, num_args, ...)`
`void` **release_kernel**`(kernel, wait_evt, evt)`

Event Management (**wtr_**)	
`event` **create_event**`()`	`void` **release_event**`(evt)`
`event` **merge_events**`(num, ...)`	`void` **wait_for_events**`(num, ...)`
`void` **init_event_array**`(num, evt)`	
`void` **release_event_array**`(num, evt)`	

Table 1: The complete *libWater* API.

queries can be used for both device initialization and device selection. The latter must be a subset of the former and since *libWater*'s device concept represents a single device only, the function `wtr_get_device` only accepts queries that make use of the `POS` clause.

Table 1 presents the *complete* API of the *libWater* library. The prefix `wtr_` and the C language pointer syntax has been removed from the table for readability reasons. Initialization and selection of devices is done, respectively, by using the `wtr_init_devices` and the `wtr_get_device` routines. Once a *device* is created, it is possible to allocate data and execute computation on it. In *libWater*, this is done through the use of the *buffer* and the *kernel* concepts. These two objects are similar to their respective OpenCL versions, with the main difference that, during their creation, they are bound to a specific device. For this reason no device must be specified for buffer and kernel related functions. The principal kernel functions are `wtr_create_kernel` and `wtr_run_kernel`. The former receives as parameter a `flag` that specifies whether the `name` input argument contains the kernel code or it is the name of a file containing the OpenCL kernel. The latter is used for executing a kernel in the previously bound device. The parameters `work_dim`, `global_work_size` and `local_work_size` are the same specified in the OpenCL `clEnqueueNDRangeKernel`. The `num_args` parameter states the number of input arguments accepted by the kernel. This parameter is followed by a list of a variable number of pairs. Each pair consists of a size (in bytes) and a pointer to the corresponding kernel argument. The first value of the pair distinguishes between buffers – when is equal to 0 – or a valid address in the host memory. The fourth concept in *libWater* is the *event* object. Most of *kernel* and *buffer* functions have one or two parameters called `wait_evt` and `evt`. The latter is an output argument which is used by the invoked command to generate an event object. If not specified, *libWater* assumes blocking semantics for the routine. The former specifies the event object on which the execution of the command depends. If not present, the command has no dependencies and thus it can be immediately executed. Since there can be a dependency between several commands, the `wtr_merge_events` function can be used to merge multiple event objects into one.

The last major difference between *libWater* and the OpenCL model is the fact that initialization and release of buffers and kernels can be invoked using a non-blocking semantics. The main reason for this is to increase the amount of operations that the runtime system can overlap. In the next section we explain how dependency information enforced by events are then exploited by *libWater*'s runtime system.

4. THE LIBWATER DISTRIBUTED RUNTIME SYSTEM

While the main focus of the programming interface of *libWater* is on simplicity and productivity, the underlying runtime system aims at low resource utilization and high scalability. Calls to *libWater* routines are forwarded to a distributed runtime system which is responsible for dispatching the OpenCL commands to the addressed devices and for transparently and efficiently moving data across the cluster nodes. The *libWater* distributed runtime is written in C++ and internally uses several paradigms, such as pthreads, OpenMP and MPI for parallelization.

4.1 Runtime System Architecture

Figure 1 shows the organization of the *libWater* distributed runtime system. The *host code*, which directly interacts with *libWater*'s routines, runs on the so called *root node*, which by default is the cluster node with rank 0. This thread will be referred to as the *host thread*. In the background, a second thread, i.e. the *scheduler thread*, is allocated to execute an instance of the `WTRScheduler`. On the remaining cluster nodes, a single *scheduler thread* is spawned independently of the number of available devices (only one MPI process is allocated per node). This thread executes an instance of the `WTRScheduler` which represents the backbone of *libWater*'s distributed runtime system.

Each `WTRScheduler` continuously dequeues `wtr_commands` from the local command queue. `wtr_commands` in the system are generated in two ways, either by (*i*) *libWater*'s routines (step 1), or (*ii*) by delegation from the root scheduler (step 3). Calls to the *libWater*'s interface are converted into command descriptors (i.e. command design pattern) and immediately enqueued into the root node local command queue (step 1) of Figure 1. Since all `wtr_commands` are generated by the root node itself, we refer to its queue as the runtime *global* command queue.

`wtr_commands` are either wrappers for OpenCL commands or data transfer jobs (i.e. `send_job` or `recv_job`) which are generated by the library routines whenever the device addressed by a read or write buffer operation is located in a remote (i.e. *rank* $\neq 0$) compute node. The descriptor of a `wtr_command` is *self-contained* since it carries all the information necessary for its execution . To be portable across cluster nodes, OpenCL objects such as kernels, buffers and events are identified, within the `wtr_command` object, by a unique ID. The root scheduler continuously fetches the `wtr_commands` from the global command queue, decodes its content and – depending on the targeted device – dispatches the command to the correct node. When the `wtr_command` addresses

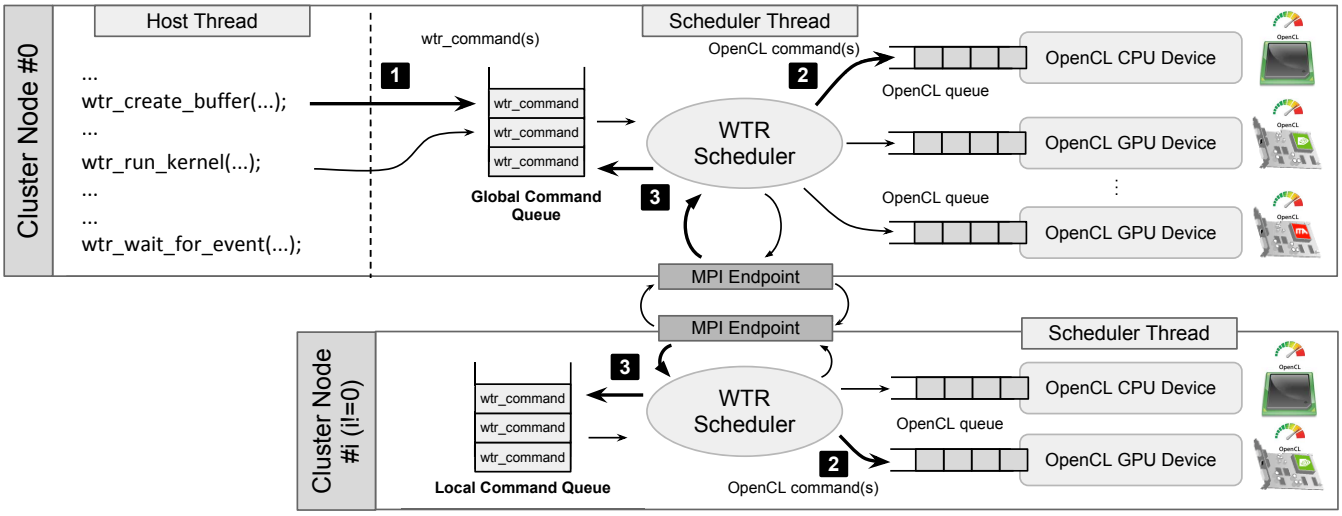

Figure 1: *libWater*'s distributed runtime system architecture.

one of the local OpenCL devices, the corresponding OpenCL command is created and enqueued into the device command queue (step 2). When a remote OpenCL device is addressed, an MPI message is generated – serializing the content of the `wtr_command` descriptor – and dispatched to the cluster node hosting the requested device. The `WTRScheduler` of the target node then de-serializes the `wtr_command` and, instead of immediately executing it, enqueues the `wtr_command` instance into the local command queue (step 3). The same `WTRScheduler` is then responsible to dispatch the corresponding OpenCL command into one of its local device queues (step 2).

The heartbeat of the `WTRScheduler` is an advanced event system which allows the management of an entire compute node – hosting multiple OpenCL devices – using only a single application thread. Indeed, because one instance of the `WTRScheduler` runs on every cluster node, trying to keep the resource usage as low as possible is of paramount importance in order to avoid wasting CPU cycles which can be used to run an OpenCL kernel. Different from related work, e.g. the SnuCL runtime system [20], which exclusively reserves an entire cluster node and a physical CPU core in each compute node only for scheduling purposes, our system does not exclusively reserve any user resources for scheduling. Furthermore, using a single thread, for both executing local `wtr_commands` and for performing scheduling decisions, reduces the amount of synchronization since accesses to event and the command queues do not need to be synchronized.

Relying on a single thread can however easily become a performance bottleneck. An interesting example is the interaction with MPI routines. By default many MPI implementations implement blocking behaviour with a *spin-lock* mechanism in order to minimize latency. This means for example that a blocking receive, waiting for a message from the communication channel, continuously checks for incoming data usually saturating the cycles of a CPU core. In an environment like ours, where CPU cores may be used to run OpenCL kernels, this behaviour must be avoided. Our solution is to avoid in every event handler routine any call to blocking MPI or OpenCL routines and always use the non-blocking semantics. The main idea is the creation of periodic events, handled by the event system using a priority queue based on timestamps, to check for the completion of pending operations. For OpenCL routines, we exploit the OpenCL event system and the associated callback mechanism. In this way, the `WTRScheduler` is able to dispatch several commands on the OpenCL devices, or MPI data transfers, which although being issued sequentially (by the single flow of the execution) are concurrently executed by the available resources (i.e. OpenCL devices and the network controller). The same event-based technique utilized to manage multiple OpenCL devices in a single node is also exploited on the large scale across cluster nodes.

4.2 Event-based Command Scheduling

As already explained in the previous section, *libWater* puts a strong emphasis on events. Following the semantics of OpenCL, dependency information enforced by programmers are used to select `wtr_commands`, which can be safely enqueued into one of the cluster nodes. *libWater* provides an event object, i.e. `wtr_event`. Internally, `wtr_events` are mapped either to an OpenCL `cl_event` object, or to a `wtr_command` identifier which is automatically generated for each `wtr_command` enqueued into the system. These dependencies allow the runtime system to organize enqueued `wtr_commands` into a DAG.

A complete multi-device *libWater*-based host program is shown in Listing 1. This code initializes all the available NVidia GPU devices. It then selects two devices belonging respectively to node rank 0 and 1, with a global memory larger than 1024MB. For each device the code in Listing 1 does the following: create a kernel (i.e. `kern`, in line 10) and a read/write buffer (i.e. `buff`, line 11). Then the contents from the host memory is written into the device buffer by the `wtr_write_buffer` command (line 12) and the `wtr_run_kernel` command is issued providing `buff` as an input argument (lines 14-16). The computed result is then retrieved by the `wtr_read_buffer` command (line 17) which moves data from the device memory back to the host memory. From the runtime system point of view, the execution of the previous code generates a set of dependent commands structured as the DAG depicted in Figure 2. The DAG $G(V, E)$ is composed of vertices, i.e. `wtr_commands` $\in V$, interconnected

```
1  wtr_init_devices(
2    "SELECT ALL WHERE (type = gpu AND vendor = nvidia)");
3  wtr_event* evts[2];
4  for (int i=0; i<2; ++i) {
5    size_t offset=size/2*i;
6    wtr_device* dev = wtr_get_device("SELECT POS 1 FROM
         NODE %d WHERE global_memory > 1024MB",i);
7    assert(dev != NULL && "Device does not exist!");
8    wtr_event* e[8];
9    wtr_init_event_array(7,e);
10   wtr_kernel* kern = wtr_create_kernel(dev,"kernel.cl","fun"
         , "", WTR_SOURCE, e+0);
11   wtr_buffer* buff = wtr_create_buffer(dev,
         WTR_MEM_READ_WRITE, size/2, e+1);
12   wtr_write_buffer(buff, size/2, ptr+offset, e+1, e+2);
13   e[7] = wtr_merge_events(2, e+0, e+2);
14   wtr_run_kernel(kern,1,(size_t[1]){size/2},NULL,e+7,e+3,2,
15                  0, buff,
16                  sizeof(size_t), &offset);
17   wtr_read_buffer(buff, size/2, ptr+offset, e+3, e+4);
18   wtr_release_buffer(buff, e+4, e+5);
19   wtr_release_kernel(kern, e+3, e+6);
20   evts[i] = wtr_merge_events(2, e+5, e+6);
21   wtr_release_event_array(8, e);
22 }
23 /* Blocks until buffers and kernels are released */
24 wtr_wait_for_events(2, evts+0, evts+1);
25 wtr_release_event_array(2, evts);
```

Listing 1: A complete multi-device program example using *libWater*'s routines

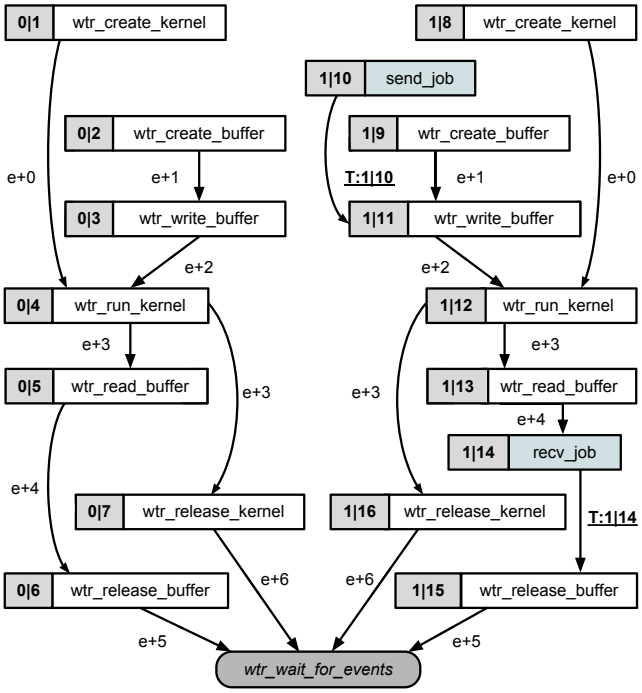

Figure 2: DAG of `wtr_commands` generated during the execution of the code snippet in Listing 1.

through directed edges $(a,b) \in E | a,b \in V$, or *events*, which guarantee that the correct order of execution, and therefore the semantics of the input program, is maintained. The *set* of dependencies associated with a command $c \in V$ is defined as $c.deps = \{v \in V | (v,c) \in E\}$. It is worth mentioning that not all *libWater* library routines generate a corresponding `wtr_command`. For example, creation, merging and release of events are only meaningful in the root node, therefore there is no need for serializing them. In Figure 2, each `wtr_command` carries a descriptor in the form $x|y$ where x represents the node rank, *c.node_id*, on which the targeted device, *c.dev_id*, is hosted and y is the unique command identifier assigned by the runtime system. As already mentioned, for buffer operations on remote devices (i.e. device on node 1) explicit data transfers are automatically inserted by the *libWater* library (e.g. `wtr_commands` 10 and 14).

Events determine when a `wtr_command` can be scheduled for execution. The scheduler uses a *just-in-time* strategy to select the next `wtr_command` from the local command queue. The logic works as follows: enqueued `wtr_commands` are analyzed in a FIFO fashion and, for each ready command, the scheduler checks whether dependencies – explicitly specified by event objects – are satisfied. If a command has no dependencies, it can be executed. Since the host program generates all the commands solely on the root node, scheduling is done at this node. However, a centralized scheduler on a single node is not an effective strategy since it limits command throughput and thus the overall scalability of the system.

In order to solve this problem, we rely on the fact that the OpenCL runtime system already has the capability of scheduling commands and handling dependencies by using events. It is worth noting that in OpenCL this mechanism is limited since events cannot be used to perform command synchronization across different contexts. *libWater* unifies event handling through `WTRScheduler` instances which manage inter-context synchronization and offload intra-context synchronization to the OpenCL driver.

We implemented a *three-level hierarchal scheduling* approach as described in Algorithm 1. At the top level, the root node of the *libWater* runtime system *pro-actively* schedules `wtr_commands` from the global queue to the targeted cluster nodes. *cmd*, fetched from the command queue, is sent to the target node (i.e. *cmd.node_id*) only if each of its dependent commands (i.e. the set *cmd.deps*) are to be executed on the same remote node (lines 6–9). The second level scheduling is local to each node (lines 11–14). The scheduler checks whether *cmd* only depends on `wtr_commands` addressing the same OpenCL device. In such case, the command is enqueued into the corresponding device queue (i.e. *dev.dev_id*) and dependencies are mapped to local OpenCL events. Alternatively, if a `wtr_command` C_1 depends on a second `wtr_command` C_2, scheduled in another context (of the same node), the local `WTRScheduler` ensures that C_1 is not enqueued into the OpenCL device queue before C_2 is completed. The third-level scheduling is implemented by the OpenCL runtime system itself which is responsible of managing single device queues. If *cmd* cannot be scheduled, due to unsatisfied dependencies, then it is pushed back in the command queue.

Command dependencies are automatically updated when a `wtr_command` *c* completes. Locally, a *command completion event* is generated. The associated *callback* function is depicted in Algorithm 2. The function removes, for every command in the local queue, any dependence on *c*. Additionally, nodes notify the root scheduler with a message (lines 5–7) triggering

Algorithm 1 The `WTR_Scheduler`'s algorithm

```
1: cmd_queue                                   ▷ Local FIFO wtr_command queue
2: my_rank                                     ▷ MPI process rank
3: while true do
4:   cmd ← cmd_queue.pop();
5:   if cmd.node_id ≠ my_rank then
6:     if ∀ d ∈ cmd.deps | d.node_id = cmd.node_id then
7:       send(cmd, cmd.node_id, SCHED)         ▷ Delegates cmd to node
8:       continue
9:     end if
10:  else
11:    if ∀ d ∈ cmd.deps | d.dev_id = cmd.dev_id then
12:      issue(cmd.cl_cmd, cmd.deps)           ▷ Delegates to corresp. dev.
13:      continue
14:    end if
15:  end if
16:  cmd_queue.push(cmd)                       ▷ Failed to schedule event due to deps.
17: end while
```

Algorithm 2 Update `wtr_command` dependencies

```
1: function CALLBACK_CMD_COMPLETION(c)
2:   for cmd in cmd_queue do
3:     cmd.deps.remove(c)                      ▷ Removes c from the dependencies
4:   end for
5:   if my_rank ≠ 0 then
6:     send(c, 0, DONE)                        ▷ Notifies the root node of c completion
7:   end if
8: end function
```

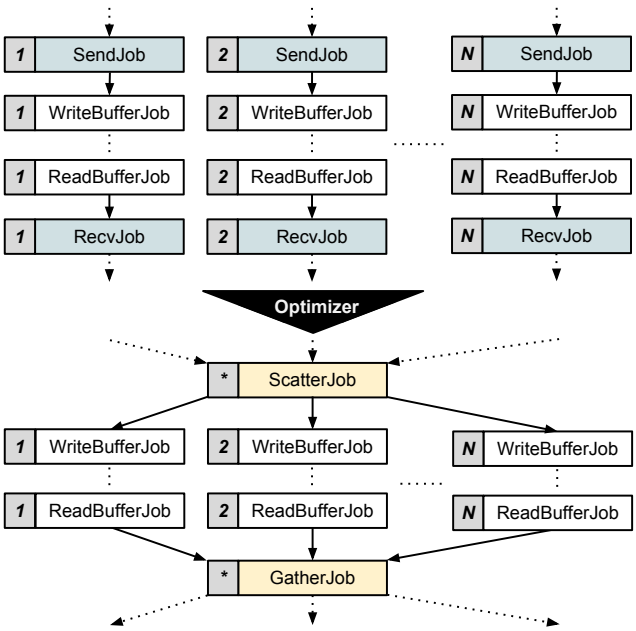

Figure 3: Dynamic collective communication pattern replacement (`DCR`) optimization.

a similar completion event internally at node 0. In such a way, commands in the global queue waiting for the completion of c can be scheduled – depending on the targeted device – either to a local device or to a remote node.

This multi-level scheduling allows the runtime system to hide the costs of the scheduling, as well as data transfers, with the actual work being done by the devices in the background. The main idea is to use non-blocking semantics when OpenCL commands are scheduled in the corresponding devices. In this way, the `WTRScheduler` can continuously dispatch commands to other devices or move data from and to the root node. In the example in Figure 2, commands `0|1` and `0|2` can be executed in parallel. Events at addresses $e + 0$ and $e + 1$ are handled by the root `WTRScheduler` since the OpenCL standard does not allow non-blocking semantics for these operations. The remaining commands (i.e. `0|3`, `0|4` and `0|5`) are inserted asynchronously into the OpenCL device queue of node 0, upon completion of commands `0|1` and `0|2`. Events `e+2` and `e+3` are therefore handled directly by the OpenCL runtime system. Following the same logic, `wtr_commands` addressing the second OpenCL device (i.e. $1|*$) are sent to the node with rank 1. The blocking function `wtr_wait_for_events` stops the execution of the host until the release operations on both nodes have completed.

5. THE DYNAMIC COLLECTIVE REPLACEMENT (DCR) OPTIMIZATION

The underlying architecture of the *libWater* runtime system and the emphasis on events, promoted by its interface, enables several runtime optimizations which are transparent to the user. This capability is a direct consequence of adhering to the OpenCL queuing semantics. Indeed, while commands are being enqueued into the system, a command DAG (as shown in Figure 2) is internally created. Since OpenCL issues commands to the appropriate device only when an explicit flush is invoked by the programmer, the runtime system can analyze large portions of the application DAG and optimize it for improving scalability.

An optimization which has been implemented in the *libWater* runtime system is the dynamic detection and replacement of collective communication patterns (`DCR`). Whenever the addressed device is not hosted in the root node, a call to `wtr_write_buffer` and `wtr_read_buffer` respectively generates an MPI send and receive operation. When an OpenCL application is distributed among all available devices, input buffers are usually either split or replicated between compute nodes. This parallelization strategy is common and it results in a DAG containing several send/receive transfer operations for every device of the cluster. An example is depicted in Figure 3 which represents a realistic DAG resulting from the splitting of an input and output buffer among a set of N OpenCL devices.

Point-to-point data transfers performed by the *libWater* runtime system imply an increased latency when compared with the native MPI send or receive routines. The reason for that is the polling mechanism implemented by the *libWater* runtime system – mainly employed to save node resources – which replaces the spin-lock mechanism commonly used by MPI libraries. Additionally, the number of required data transfers is directly proportional to the cluster nodes (and thus devices). This results in a large number of commands being dispatched by the runtime system and consecutively negatively impacts the overall scalability. MPI offers a large set of communication patterns called *collective operations* [25]. These routines are highly efficient since nearly all modern supercomputers and high-performance networks provide specialized hardware support for collective operations [23]. Additionally, the implementation of such collective operations employs dynamic runtime tuning techniques which choose, among a set of semantically equivalent algorithms, which best fit the underlying network topology and architecture [9, 27, 28].

Algorithm 3 DCR pattern recognition

```
1: function REPLACE_COLLECTIVE_PATTERNS(G(V, E), root)
2:   for t in BFS(G(V, E), root) do
3:     if t.type ∈ {SendJob, RecvJob} then
4:       jobs[t.node_id].append(t)                    ▷ Orders transfer jobs
5:     end if
6:   end for
7:   for i ← 0 to min({jobs[k].length : ∀ k | 0 ≤ k < N}) do
8:     pattern ← 0
9:     for j ← 1 to N do
10:      if jobs[j][i].type ≠ jobs[j − 1][i].type then break
11:      if jobs[j][i].buf = jobs[j − 1][i].buf ∧
12:        jobs[j][i].size = jobs[j − 1][i].size then
13:        if pattern = 2 then break
14:        pattern ← 1              ▷ Sequence recognized as a broadcast
15:      end if
16:      if jobs[j][i].buf = jobs[j − 1][i].buf + jobs[j − 1][i].size then
17:        if pattern = 1 then break
18:        pattern ← 2       ▷ Sequence can be either scatter or gather
19:      end if
20:    end for
21:    if j ≠ N ∨ pattern = 0 then continue
22:    if pattern = 1 ∧ jobs[0][i].type = SendJob then
23:      replace_with_broadcast(jobs, i)
24:    else
25:      if jobs[0][i].type = SendJob then replace_with_scatter(jobs, i)
26:      else replace_with_gather(jobs, i)
27:    end if
28:  end for
29: end function
```

Related work analyzed the problem of automatic detection of collective patterns from a set of point-to-point communications. This technique is common in MPI performance tools which are capable of detecting such patterns via post-mortem analysis of program traces [21]. The general problem of collective communication pattern detection is NP-hard, however, under particular restrictions the problem can be solved in polynomial time. A more recent work [15] proposed a fast solution, with a complexity of $O(n \log n)$, which makes the approach more suitable for runtime systems.

The goal of our `DCR` optimization algorithm is to analyze the command DAG isolating point-to-point data transfers and detect whether a subset of those resembles one of the collective patterns supported by MPI. This is possible since – if the application is carefully written using events for command synchronization – the command DAG will be available to the runtime system scheduler before the first blocking command is invoked (e.g. `wtr_wait_for_event`(s)). Since data transfers in our environment have all the same root (the node 0), the analysis for patterns is simplified. The pattern recognition is presented in Algorithm 3. The command DAG is traversed once in breadth-first order (lines 1–6), transfer commands are collected into N separate lists (i.e. variable *jobs*), one per device. On the extracted N lists, pattern analysis is performed (lines 7–28). The check is done by considering elements having the same position within the transfer job lists. Furthermore, the check is simplified by the fact that every send and receive `wtr_command` carries information of the buffer location (*buf*) and the amount of bytes being transfered (*size*). The pattern analysis starts by taking the first transfer `wtr_command` from the N lists and by checking against a supported pattern, i.e. *broadcast*, *scatter* or *gather*. For instance, in a broadcast N send operations are expected where $\forall\ i \mid 0 \leq i < N-1,\ buf_i = buf_{i+1} \vee size_i = size_{i+1}$. If the check fails, the transfer jobs are tested against a scatter or gather pattern $\forall\ i \mid 0 \leq i < N-1,\ buf_i + size_i = buf_{i+1}$.

Once a pattern is recognized, single point-to-point transfers are removed from the command DAG and replaced by the corresponding collective communication operation (lines 23, 25 and 26). A visual example of this optimization is depicted in Figure 3, where multiple send operations are collapsed into a single scatter operation and correspondingly, receives are rewritten as a gather operation. By doing so, dependencies between successive commands are updated in order to keep the semantics of the input program unchanged.

Since collective operations must involve all the processes in a communicator, the current implementation of the `DCR` optimization works when all the initialized devices participate in the computation. Therefore, the analysis is limited to regular applications which *must* involve all OpenCL devices in data transfers. This is important to keep the pattern recognition algorithm simple and fast, since this optimization is applied during runtime. In the future, we plan to improve this mechanism by extending the pattern recognition also to sub-groups of devices.

Vienna Supercomputing Cluster 2 (VSC2)

Max # of nodes	1.314
Processors	2 x AMD Opteron 6132 HE
Cores per node	2 x 8
Clock Frequency	2.2 GHz
Memory per Node	32 GB DDR3
Interconnection	Infiniband 4x QDR
Open MPI version	1.6.1
OpenCL version	AMD APP 2.6

Table 2: The VSC2 experimental target architecture.

6. EXPERIMENTAL EVALUATION

We used *libWater* to encode 6 computational kernels, some of them taken from various OpenCL benchmarking suites (i.e. AMD and IBM), and studied their scalability. In four of them, the kernels were optimized for local memory, i.e. `PerlinNoise` (from IBM), `NBody` (from AMD), `Floyd` and `kNN` manually written by us. For the remaining two codes, `MatrixMul` and `LinReg` we used a naive implementation unoptimized for what concern local memory. The table shows, for each kernel, the number of input and output buffers used by the kernel. We define a buffer as *splittable* when its content can be distributed among the devices. The nature of a buffer is strictly related to the algorithm being implemented within the OpenCL kernel, and thus the application. Non splittable buffers are always replicated on every device.

For the strong scalability analysis we used a real production cluster, the Vienna Supercomputing Cluster 2 (VSC2) [3], ranked 162th in the Top500 list [4], whose details are summarized in Table 2. A second study was conducted to test the suitability of *libWater* to exploit the computational capabilities of a heterogeneous cluster configuration. For this purpose we used a cluster, composed of 3 compute nodes (i.e. `mc1`, `mc2` and `mc3`), custom made with *off-the-shelf* GPU accelerators. The hardware details are depicted in Table 3.

The six applications utilized for our study are listed in Table 4. We started from a pure OpenCL implementation and

	mc1	mc2	mc3
CPUs	2 x AMD Opteron(tm) 6168 @1.9GHz	2 x AMD Opteron(tm) 6168 @1.9GHz	2 x Intel(R) Xeon(R) X5650 @2.67GHz
GPUs	2 x ATI Radeon HD 5870	1 x NVIDIA GTX 480	1 x NVIDIA GTX 460
RAM	24 GB DDR3		
Interconn.	Infiniband QDR		
Open MPI	1.6.1		
OpenCL	AMD APP 2.6	CUDA 5.0	CUDA 5.0

Table 3: The architecture of `mc1`, `mc2` and `mc3` heterogeneous compute nodes.

rewrote them using *libWater*. In Table 4, we show the reduction, in terms of lines of code, achieved when the application is written using our library. It is worth mentioning that while the original OpenCL applications were single device codes, the *libWater* based implementation is instead multi-device code. On average, we were able to reduce the lines of the host code by approximately a factor of 2 due to the higher level abstractions provided by *libWater*.

6.1 Homogeneous CPU cluster

The applications shown in Table 4 were executed on the VSC2 homogeneous CPU cluster. We were able to access up to 64 nodes with a total of 1024 CPU cores. Since the 2 AMD CPUs which are hosted per node are considered by the OpenCL driver as a single device, the speedup was computed based on the number of compute nodes (and thus OpenCL devices) instead of single CPU cores. The workload partitioning is implemented, for each test case, by assigning to each OpenCL device an equal amount of work.

The scalability tests were performed in the following way: the original OpenCL version of the applications were executed in a single node and their execution times used as a reference measurement. *libWater* was then used for node numbers ranging from 2 to 64. The main differences between the original version of the application codes and the one written using *libWater* are mainly in the host code. The kernel code was slightly modified only to forward the *offset* value used by the workload partitioning (as shown in Listing 1). We computed the ideal scaling for each application using the reference execution time and dividing it by the number of nodes. We conducted experiments with *libWater* by using two different settings: the first, named *baseline*, uses the runtime system without dynamic optimizations enabled; the second, `DCR`, uses the collective pattern replacement mechanism as described in Section 5. The results of our experiments are depicted in Figure 4.

For each of the six applications, we show the execution time (in seconds) for up to 64 devices and the corresponding speedup with respect to a single node. Overall, we observe that our approach scales almost linearly, especially for those codes using few input/output buffers. `PerlinNoise`, Figure 4(a), is an example of those, since it has no dependencies on input buffers and the data produced by the kernel is distributed between the devices. For such code, the baseline configuration of our runtime system achieves a speedup of 53 for 64 nodes, and thus an efficiency of 83%. When the number and size of the input/output buffers increases, the efficiency of our system decreases. The worst case is represented by the `LinReg` application, Figure 4(f), which stops scaling after 32 nodes. This kernel has 4 input buffers, 2 of them are not splittable (because of dependencies within the kernel code) and therefore must be replicated on every node. The remaining 2 input and output buffers are instead splittable. For such code we have an immediate decrease (75% on two nodes) of the efficiency. This is because the kernel execution is delayed due to the fact that several `wtr_commands` are executed (and transfered to the target nodes) to create and initialize the input/output buffers. However this delay is a constant and system efficiency remains almost unvaried up to 16 nodes. On 32 and 64 nodes the efficiency of the baseline runtime system starts decreasing significantly.

This problem is largely addressed by the dynamic collective pattern replacement, i.e. `DCR`, optimization which was introduced in Section 5. This optimization reduces the load on the scheduler since it replaces several single transfer jobs with one collective operation. In `LinReg` this optimization improves the scalability of the system by a factor of 2 achieving an efficiency of 55%. A small effect of this optimization can be observed for smaller node configurations because collective operations are optimized for a large number of nodes. An interesting result is the effect of the `DCR` optimization on the `PerlinNoise` test case. In such a case, the `DCR` optimization fails to improve performance over the baseline. The reason is that collective operations are blocking while point-to-point communications in the runtime system are non-blocking thereby allowing overlapping of multiple transfers. The synchronization costs introduced by the gather operation is therefore not properly compensated by the amount of exchanged data. We believe that this problem can be eliminated by using *non-blocking collective* routines which have been introduced in the latest MPI standard [25] and will soon be available in mainstream MPI libraries. Additionally, since this optimization is done dynamically, and therefore the amount of data being transfered is known by the scheduler, heuristics can be integrated to decide when such optimization should be applied.

On average, *libWater* achieves an efficiency of 80% on 32 nodes and 64% when 64 nodes are used. Without the `DCR` optimization the system has an efficiency of 47% on 64 nodes. This means that the `DCR` optimization improves the system efficiency by 17% on 64 nodes and we expect this value to increase proportionally with the number of nodes.

6.2 Heterogeneous GPU cluster

Since OpenCL allows access to heterogeneous devices we conducted a second experiment which demonstrates *libWater* on a heterogeneous GPU cluster as described in Table 3. In order to run applications on such environment, the input code was rewritten so that the workload distribution was controllable via command line arguments. It is worth mentioning that workload partitioning for heterogeneous architectures is an active research problem [13, 22, 17, 14]. However, this aspect is completely orthogonal to our library and for the sake of this experiment, we derive workload partitionings in an empirical way.

We ran the `MatrixMul` and the `Floyd` test cases using different combinations of devices. For each device configuration, several different workload splittings were tested and the fastest one was chosen. The partitionings and their corresponding execution times, are shown in Table 5. For exam-

Application	OpenCL LOC	*libWater* LOC	Input size	Input/Output buffers (*splittable*)	Short Description
`PerlinNoise`	412	**301**	20K x 20K	0(0) / 1(1)	Gradient noise generator
`Nbody`	450	**324**	600K bodies	2(0) / 2(2)	N-body simulation
`kNN`	234	**101**	*ref*: 8M, *query*: 80K	2(1) / 2(1)	k-nearest neighbor
`Floyd`	222	**113**	Vertices 8K, Adjacency matrix 64K	1(0) /1(0)	Floyd-Warshall
`MatrixMul`	219	**104**	7K x 7K ($A = B = C$)	2(1) / 1(1)	Matrix Multiplication
`LinReg`	298	**149**	1000K	4(2) / 1(1)	Linear regression

Table 4: Application codes used for *libWater* evaluation.

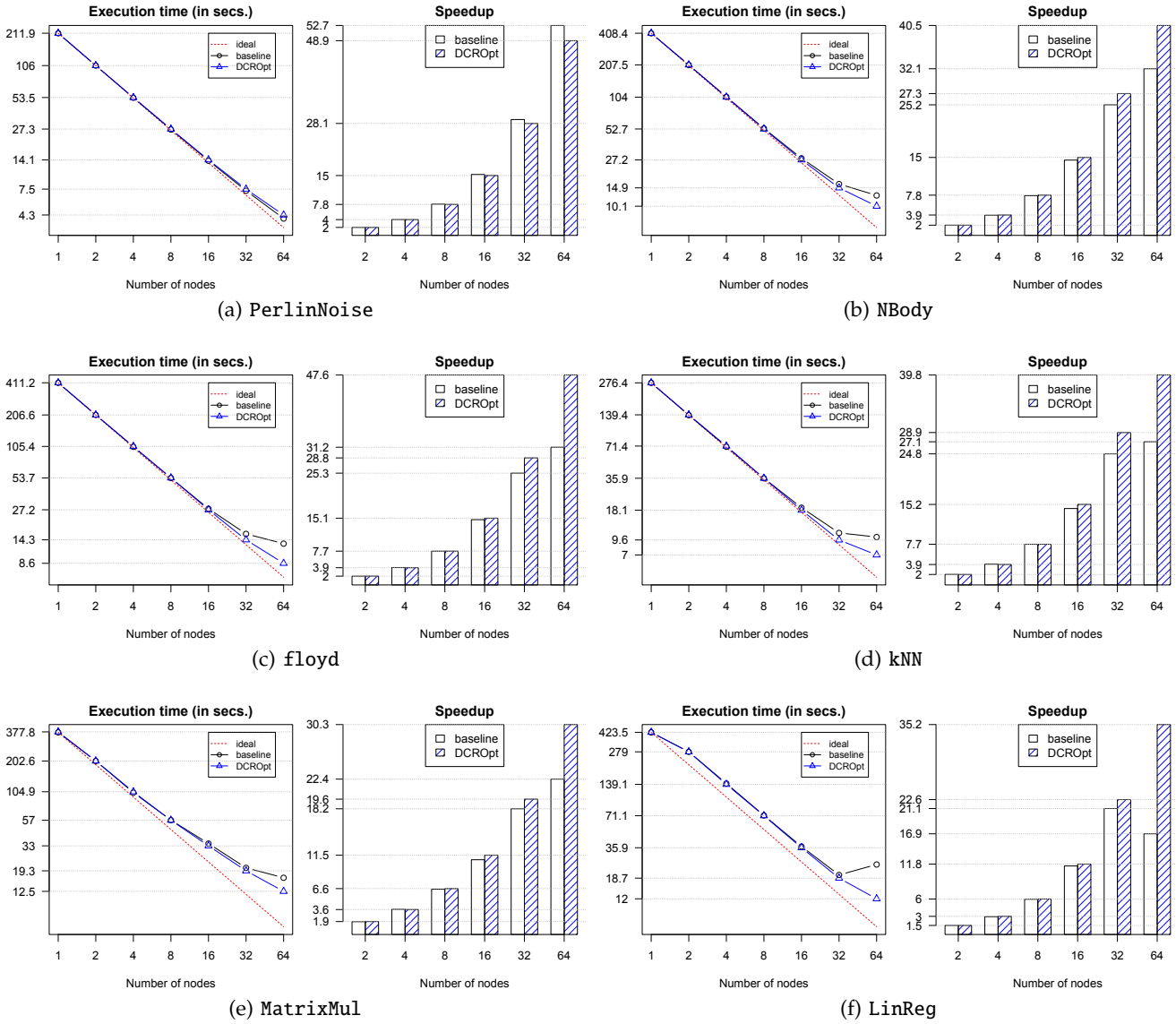

Figure 4: Strong scaling of *libWater* on the VSC2

ple, in `MatrixMul`, configuration `C1` assigns all the workload to the first GPU of node `mc1`. The execution time for this configuration is 63.3 seconds. By equally splitting the workload between the two accelerators on the same node, i.e. `C2`, we double the performance. Between the GPUs, the NVidia GTX 480 is the fastest device requiring only 29.4 seconds to complete the work. However *libWater* can be used to improve the execution time even further. The overall execution time can be reduced by 50% by using the workload partition as described by configuration `C8` which assigns 22% to each GPU in `mc1`, 44% to the NVidia GTX 480 and the remaining 12% to the NVidia GTX 460 accelerator. For `Floyd` results are different. Its execution on the GTX 480 is 8 times faster than the AMD GPU and 4 times better than the GTX 460. However, performance can still be improved by splitting the workload between the two NVidia accelerators by assigning

	Device	Workload Partition Configurations							
		C1	C2	C3	C4	C5	C6	C7	C8
MatrixMul	mc1-GPU1	100%	50%	-	-	35%	25%	-	22%
	mc1-GPU2	-	50%	-	-	-	25%	-	22%
	mc2-GPU3	-	-	100%	-	65%	50%	75%	44%
	mc3-GPU4	-	-	-	100%	-	-	25%	12%
	Exec. time (in secs.)	63.3	32.5	29.4	68.4	23.7	19.0	26.6	**17.3**
Floyd	mc1-GPU1	100%	50%	-	-	2%	1%		0.5%
	mc1-GPU2	-	50%	-	-	-	1%	-	0.5%
	mc2-GPU3	-	-	100%	-	98%	98%	99%	98%
	mc3-GPU4	-	-	-	100%	-	-	1%	1%
	Exec. time (in secs.)	101.6	51.3	14.9	58.3	17.3	16.0	**13.1**	16.3

Table 5: Performance of `MatrixMul` and `Floyd` on the heterogeneous cluster for different combination of GPUs.

99% of the work to the faster GTX 480 and 1% to the GTX 460. This experiment demonstrates, despite higher latencies caused by additional data transfers between host and device memory, non-blocking communication yields good scalability behaviour even for heterogeneous architectures. However, scalability on such environments depends on several factors and we plan to investigate this issues in future work.

7. RELATED WORK

In recent years, heterogeneous systems have received a great amount of attention from the research community. Although several projects have been recently proposed to facilitate the programming of clusters with heterogeneous nodes [20, 8, 6, 5, 18, 12, 26, 31], none of them combines support for high performance inter-node data transfer, support for a wide number of different devices and a simplified programming model. Our work takes into account all this aspects through the development of the *libWater* library.

Kim et al. [20] proposed the *SnuCL* framework that extends the original OpenCL semantics to heterogeneous cluster environments. Their work is closely related to ours. SnuCL relies on the OpenCL language with few extensions to directly support collective patterns of MPI. Indeed, in SnuCL is the programmer responsibility to take care of the efficient data transfers between nodes. In that sense, end users of the SnuCL platform need to have an understanding of MPI collective calls semantics in order to be able to write scalable programs. This deeply differs from our system where such optimizations are transparently applied by the *libWater* runtime system. Furthermore, SnuCL poses a limit to the number of devices which can be addressed by their runtime system, i.e. NVidia accelerators and CPUs. Differently, *libWater* can interface with every OpenCL driver making the list of supported platforms virtually unlimited.

Also other works have investigated the problem of extending the OpenCL semantics to access a cluster of nodes. The Many GPUs Package (*MGP*) [8] is a library and runtime system that using the MOSIX VCL layer enables unmodified OpenCL applications to be executed on clusters. *Hybrid OpenCL* [6] is based on the FOXC OpenCL runtime and extends it with a network layer that allows the access to devices in a distributed system. The *clOpenCL* [5] platform comprises a wrapper library and a set of user-level daemons. Every call to an OpenCL primitive is intercepted by the wrapper which redirects its execution to a specific daemon at a cluster node or to the local runtime. *dOpenCL* [18] extends the OpenCL standard, such that arbitrary compute devices installed on any node of a distributed system can be used together within a single application. *Distributed OpenCL* [12] is a framework that allows the distribution of computing processes to many resources connected via network using JSON RPC as communication layer. *OpenCL Remote* [26] is a framework which extends both OpenCL's platform model and memory model with a network client-server paradigm. *Virtual OpenCL* [31], based on the OpenCL programming model, exposes physical GPUs as decoupled virtual resources that can be transparently managed independent of the application execution.

While the objectives of these approaches are similar to ours, none of them provides an abstraction layer to reduce the complexity associated with the OpenCL development and, furthermore, they show a very limited scalability in clusters of 4 to 8 compute nodes. In particular, none of them employs dynamic communication optimizations as we do.

Besides OpenCL-based approaches, also CUDA solutions have been proposed to simplify distributed systems programming. *CUDASA* [29] is an extension of the CUDA programming language which extends parallelism to multi-GPU systems and GPU-cluster environments. *rCUDA* [11] is a distributed implementation of the CUDA API that enables shared remote GPGPU in HPC clusters. *cudaMPI* [24] is a message passing library for distributed-memory GPU clusters that extends the MPI interface to work with data stored on the GPU using the CUDA programming interface. All of these approaches are limited to devices that support CUDA, i.e. NVidia GPU accelerators, and therefore they cannot be used to address heterogeneous systems which combines CPUs and accelerators from different vendors.

Other projects have investigated how to simplify the OpenCL programming interface. Sun et. al [30], proposed a task queueing extension for OpenCL that provides a high-level API based on the concepts of work pools and work units. *Intel CLU* [16], *OCL-MLA* [1] and *SimpleOpencl* [2] are lightweight API designed to help programmers to rapidly prototype heterogeneous programs. Beside the simplified interface, *libWater* provides fine-grained control over device selection (i.e. DQL) and an improved device synchronization based on events.

A more sophisticated approach was proposed in [10]. OmpSs relies on compiler technologies to generate host and kernel code from a sequential program annotated with pragmas. The runtime of OmpSs internally uses a DAG similar to ours with the scope of scheduling. However, to our knowledge, the DAG is not dynamically optimized like in our approach. Additionally, by relying on the user for kernel code, *libWater* allows for fine-grained performance tuning.

8. CONCLUSIONS

In this paper, we introduced *libWater*, a library for simplifying the programming of heterogeneous distributed systems.

The proposed interface demonstrates that raising the abstraction level of the OpenCL programming model is possible without losing control over performance. We showed, with an example, how a multi-device distributed host program can be written using approximately 25 lines of code. By defining a simple, but powerful, device query language (DQL), *libWater* simplifies the management and discovery of a large number of OpenCL devices. The simple API makes the library a perfect target for automatic code generation tools, thus it can be easily integrated in compilers.

libWater's interface is tightly bound to a lightweight distributed runtime system which is designed from scratch for high scalability and low resource usage. Because of the non-blocking semantics promoted by the library interface, commands can be organized by the runtime system into a DAG to be used for dynamic analysis and optimizations.

We studied the strong scalability on a homogeneous production cluster using up to 64 nodes, for which our system achieves an efficiency of 64%. We also demonstrated the effectiveness of how *libWater* addresses highly heterogeneous configurations by running two test cases on a cluster system composed by 4 different GPU accelerators.

libWater will be released as an open-source project with the goal of becoming a research platform to investigate performance aspects of heterogeneous and distributed HPC architectures.

Acknowledgments

This project was funded by the FWF Austrian Science Fund as part of project TRP 220-N23 "Automatic Portable Performance for Heterogeneous Multi-cores" and by the FWF Doctoral School CIM Computational Interdisciplinary Modelling under contract W01227. The computational results presented have been achieved in part using the Vienna Scientific Cluster 2 (VSC2).

9. REFERENCES

[1] OCL-MLA. `http://tuxfan.github.com/ocl-mla/`.

[2] Simple-opencl. `http://code.google.com/p/simple-opencl/`.

[3] The Vienna Scientific Cluster 2. `http://www.vsc.ac.at/`, 2012.

[4] Top 500 Supercomputer sites. `http://www.top500.org/`, 2012.

[5] A. Albano, R. Jose, P. Antonio, and S. L. Paulo. clOpenCL - Supporting Distributed Heterogeneous Computing in HPC Clusters. In *10th International Workshop HeteroPar*, 2012.

[6] R. Aoki, S. Oikawa, T. Nakamura, and S. Miki. Hybrid OpenCL: Enhancing OpenCL for Distributed Processing. In *ISPA*, pages 149–154, 2011.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.

[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 PPAC*, pages 224–231, 2010.

[9] J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby. Efficient Algorithms for All-to-All Communications in Multi-Port Message-Passing Systems. In *SPAA*, pages 298–309, 1994.

[10] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive Programming of GPU Clusters with OmpSs. In *IPDPS*, pages 557–568, 2012.

[11] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *HPCS*, pages 224–231, 2010.

[12] B. Eskikaya and D. T. Altilar. Distributed OpenCL Distributing OpenCL Platform on Network Scale. In *IJCA*, volume ACCTHPCA 2, pages 26–30, 2012.

[13] I. Grasso, K. Kofler, B. Cosenza, and T. Fahringer. Automatic problem size sensitive task partitioning on heterogeneous parallel systems. In *PPoPP*, 2013.

[14] D. Grewe and M. F. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *CC*, 2011.

[15] T. Hoefler and T. Schneider. Runtime Detection and Optimization of Collective Communication Patterns. In *PACT*, pages 263–272, 2012.

[16] Intel Corporation. Computing Language Utility. `http://software.intel.com/`.

[17] M. Kai, L. Xue, C. Wei, Z. Chi, and W. Xiaorui. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. In *ICPP*, 2012.

[18] P. Kegel, M. Steuwer, and S. Gorlatch. dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems. In *IPDPS Workshops*, pages 174–186, 2012.

[19] Khronos OpenCL Working Group. The OpenCL 1.2 specification. `http://www.khronos.org/opencl`, 2012.

[20] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *ICS*, pages 341–352, 2012.

[21] A. Knüpfer, D. Kranzlmüller, and W. E. Nagel. Detection of Collective MPI Operation Patterns. In *PVM/MPI*, pages 259–267, 2004.

[22] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *ICS*, 2013.

[23] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer. The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer. In *ICS*, pages 94–103, 2008.

[24] O. S. Lawlor. Message passing for GPGPU clusters: CudaMPI. In *CLUSTER*, pages 1–8, 2009.

[25] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3. `http://www.mpi-forum.org`, 2012.

[26] R. Özaydin and D. T. Altilar. OpenCL Remote: Extending OpenCL Platform Model to Network Scale. In *HPCC-ICESS*, pages 830–835, 2012.

[27] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[28] P. Sanders, J. Speck, and J. L. Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

[29] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl. CUDASA: Compute Unified Device and Systems Architecture. In *EGPGV*, pages 49–56, 2008.

[30] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli. Enabling task-level scheduling on heterogeneous platforms. In *Workshop GPGPU*, pages 84–93, 2012.

[31] S. Xiao and W. chun Feng. Generalizing the Utility of GPUs in Large-Scale Heterogeneous Computing Systems. In *IPDPS Workshops*, pages 2554–2557, 2012.