Ivan Grasso, Simone Pellegrini, Biagio Cosenza, Thomas Fahringer
LibWater: heterogeneous distributed computing made easy

Conference Object, Postprint version
This version is available at http://dx.doi.org/10.14279/depositonce-6344

Suggested Citation
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

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 manycore 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[i, ...]]
```

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 cGetDeviceInfo function. A DQL use case is shown and discussed in Section 4.2. DQL
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 POSIX 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 ≠ 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 model 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
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 are 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
Listing 1: A complete multi-device program example using libWater's routines

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 xly where x represents the node rank, cnode_id, on which the targeted device, cdev_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 hierarchical 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 C2 depends on a second wtr_command C1, scheduled in another context (of the same node), the local WTRScheduler ensures that C1 is not enqueued into the OpenCL device queue before C2 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
shown in Figure 2) is internally created. Since OpenCL issues to the OpenCL queuing semantics. Indeed, while commands the user. This capability is a direct consequence of adhering and the emphasis on events, promoted by its interface, en-

Algorithms for these operations. The remaining commands (i.e. the OpenCL standard does not allow non-blocking seman-

5. THE DYNAMIC COLLECTIVE REPLACE-

Algorithm 1: The WTR Scheduler’s algorithm

Algorithm 2: Update wtr_command dependencies

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 REPLACE-

M (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 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].

![Figure 3: Dynamic collective communication pattern replacement (DCR) optimization.](image-url)
Algorithm 3 DCR pattern recognition

1: function replace_collective_patterns(G(V,E), root)
2:  for i in BFS(G(V,E), root) do
3:      if type ∈ {SendJob, RecvJob} then
4:          job[i].node_id.append(i) // 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:      end if
12:      if pattern = 1 then break
13:      pattern ← 1 // Sequence recognized as a broadcast
14:      end if
15:      if jobs[j][i].buf = jobs[j − 1][i].buf ∧
16:         jobs[j][i].size = jobs[j − 1][i].size then
17:          if pattern = 2 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[j][i].type = SendJob then
23:         replace_with_broadcas(jobs, i)
24:      else
25:         if jobs[j][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

Table 2: The VSC2 experimental target architecture.

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

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), libWater (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 m3), 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
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 code, 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 wr commands are executed (and transferred 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 transferred is known by the scheduler, heuristics can be integrated to decide whether 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-
Table 4: Application codes used for libWater evaluation.

<table>
<thead>
<tr>
<th>Application</th>
<th>OpenCL LOC</th>
<th>libWater LOC</th>
<th>Input size</th>
<th>Input/Output buffers (splittable)</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PerlinNoise</td>
<td>412</td>
<td>301</td>
<td>20K x 20K</td>
<td>0(0) / 1(1)</td>
<td>Gradient noise generator</td>
</tr>
<tr>
<td>NBody</td>
<td>450</td>
<td>324</td>
<td>600K bodies</td>
<td>2(0) / 2(2)</td>
<td>N-body simulation</td>
</tr>
<tr>
<td>kNN</td>
<td>234</td>
<td>101</td>
<td>ref: 8M, query: 80K</td>
<td>2(1) / 2(1)</td>
<td>k-nearest neighbor</td>
</tr>
<tr>
<td>Floyd</td>
<td>222</td>
<td>113</td>
<td>Vertices 8K, Adjacency matrix 64K</td>
<td>1(0) / 1(0)</td>
<td>Floyd-Warshall</td>
</tr>
<tr>
<td>MatrixMul</td>
<td>219</td>
<td>104</td>
<td>7K x 7K (A = B = C)</td>
<td>2(1) / 1(1)</td>
<td>Matrix Multiplication</td>
</tr>
<tr>
<td>LinReg</td>
<td>298</td>
<td>149</td>
<td>100K</td>
<td>4(2) / 1(1)</td>
<td>Linear regression</td>
</tr>
</tbody>
</table>

Figure 4: Strong scaling of libWater on the VSC2

Table: Application codes used for libWater evaluation.

- **PerlinNoise**: Gradient noise generator. 412 LOC, 301 LOC, input size: 20K x 20K, input/output buffers: 0(0) / 1(1).
- **NBody**: N-body simulation. 450 LOC, 324 LOC, 600K bodies: 2(0) / 2(2).
- **kNN**: k-nearest neighbor. 234 LOC, 101 LOC, ref: 8M, query: 80K: 2(1) / 2(1).
- **Floyd**: Floyd-Warshall. 222 LOC, 113 LOC, vertices 8K, adjacency matrix 64K: 1(0) / 1(0).
- **MatrixMul**: Matrix Multiplication. 219 LOC, 104 LOC, 7K x 7K (A = B = C): 2(1) / 1(1).
- **LinReg**: Linear regression. 298 LOC, 149 LOC, 100K: 4(2) / 1(1).

Figure 4: Strong scaling of libWater on the VSC2.

The execution time for PerlinNoise is 211.9 seconds, and by using DCROpt configuration, the execution time is reduced to 52.7 seconds, achieving a speedup of 3.94. For NBody, the execution time is 211.9 seconds, and by using DCROpt configuration, the execution time is reduced to 51.1 seconds, achieving a speedup of 4.17. For kNN, the execution time is 411.2 seconds, and by using DCROpt configuration, the execution time is reduced to 78.6 seconds, achieving a speedup of 5.27. For Floyd, the execution time is 377.8 seconds, and by using DCROpt configuration, the execution time is reduced to 65.3 seconds, achieving a speedup of 5.71. For MatrixMul, the execution time is 423.5 seconds, and by using DCROpt configuration, the execution time is reduced to 71.1 seconds, achieving a speedup of 5.97. For LinReg, the execution time is 207.5 seconds, and by using DCROpt configuration, the execution time is reduced to 35.9 seconds, achieving a speedup of 5.76.
Table 5: Performance of MatrixMul and Floyd on the hetero-

geneous cluster for different combination of GPUs.

<table>
<thead>
<tr>
<th>Device</th>
<th>Workload Partition Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C1</td>
</tr>
<tr>
<td>mc1-GPU</td>
<td>100%</td>
</tr>
<tr>
<td>mc2-GPU</td>
<td>100%</td>
</tr>
<tr>
<td>mc3-GPU</td>
<td>100%</td>
</tr>
<tr>
<td>Exec. time (in secs.)</td>
<td>63.3</td>
</tr>
</tbody>
</table>

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 variety of different devices and a simplified programming model. Our work takes into account all these 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 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 cloOpenCL [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 Interdisciplinary Modelling under contract W01227. The Vienna Scientific Cluster 2 (VSC2).

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 Interdisciplinary Modelling under contract W01227. The Vienna Scientific Cluster 2 (VSC2).

9. REFERENCES


