IEEE Copyright Notice

Copyright © IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

This work has been published in “2017 Fifth International Symposium on Computing and Networking (CANDAR)”, 9 - 22 November, 2017, Aomori, Japan. DOI: 10.1109/CANDAR.2017.49

http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=8345455
Abstract—The ever-growing demand for computing resources has reached a wide range of application domains. Even though the ubiquitous availability of cloud-based GPU instances provides an abundance of computing resources, the programmatic complexity of utilizing heterogeneous hardware in a scale-out scenario is not yet addressed sufficiently. We deal with this issue by introducing the CloudCL framework, which enables developers to focus their implementation efforts on compute kernels without having to consider inter-node communication. Using CloudCL, developers can access the resources of an entire cluster as if they were local resources. The framework also facilitates the development of cloud-native application behavior by supporting dynamic addition and removal of resources at runtime. The combination of a straightforward job design and the corresponding job scheduling framework make sure that cluster resources are used efficiently and fairly. In an extensive performance evaluation, we demonstrate that the framework provides close-to-linear scale-out capabilities in multi-node deployment scenarios.

I. INTRODUCTION

In the age of big data, compute tasks are gaining complexity and data volumes are growing by the day. Unlike the High-Performance Computing (HPC) domain, this ever-growing demand for computing resources is not restricted to selected applications and algorithms, but it concerns a wide range of application domains. As a result, an increasing number of everyday use cases are developing a demand for computing resources drawing near to that of certain HPC use cases [1, 2].

Even though hardware accelerators such as Graphics Processing Units (GPUs) or Field-Programmable Gate Arrays (FPGAs) are a popular approach for satisfying these demands, operating a heterogeneous compute infrastructure is still expensive [3] and involves a high degree of application complexity [4]. To a certain degree, the economic concerns are alleviated by the wide availability of cloud-based accelerator instances equipped with GPUs, FPGAs, or other accelerators. However, the programmatic complexity of utilizing heterogeneous hardware in a scale-out scenario is not yet addressed sufficiently. Implementing applications for such massively parallel, distributed systems is already a challenging task for software engineers that are well-trained in parallel implementation strategies. For domain experts without deeper software expertise, it is very hard to write code that efficiently utilizes heterogeneous resources, especially in a distributed environment [5].

We address this issue by introducing the CloudCL framework, which enables developers and domain experts to focus their implementation efforts on compute kernels without having to consider inter-node communication and management tasks for heterogeneous compute devices (see Figure 1). Based on the API-forwarding capabilities of dOpenCL [4], CloudCL enables developers to access the resources of an entire cluster as if they were local resources. The complexity of kernel development is further reduced by employing Aparapi [6] as a high-level, Java-based programming interface for domain experts. CloudCL combines the capabilities of these base technologies and augments them with a scalable job design concept and a corresponding job scheduling system, enabling cloud-native application behavior by supporting dynamic addition and removal of resources at runtime. In an extensive performance evaluation, we demonstrate that the framework provides close-to-linear scale-out capabilities both in an on-premise hosting environment and using Amazon EC2-based public cloud resources.

This paper is structured as follows: Section II provides background about the employed base technologies. Subsequently, Section III reviews related work in the field. CloudCL and its fundamental concepts are introduced in Section IV. Lastly, Section V evaluates the scale-out behavior of the framework before a conclusion is reached in Section VI.
II. BACKGROUND

To provide background about the base technologies employed by CloudCL, this section outlines the properties of dOpenCL and Aparapi.

A. dOpenCL

dOpenCL is a wrapper library that enables users to transparently utilize OpenCL devices installed in remote machines based on API forwarding techniques [4]. The library provides its own Installable Client Driver (ICD), which forwards the API calls to specified remote machines in the network, which run a dOpenCL daemon. The calls are received by the daemon and are executed using the available native OpenCL ICDs on the remote machine with the results being returned via network. This allows utilizing remote devices as if they were installed locally in the host machine. For example, a remote GPU appears to the application as if it was installed in the local machine’s PCI Express slot. Therefore OpenCL kernels do not require changes to run remotely as dOpenCL hides network transfers behind the standard OpenCL API. An overview of the architecture of an example cluster is shown in Figure 2.

![Fig. 2: dOpenCL ties in remote resources in a local ICD.](image)

dOpenCL supports shared cluster environments, in which multiple OpenCL programs run concurrently, by employing a device manager. The device manager handles the assignment of devices to specific kernels and keeps track of device utilization within the cluster. Thus, it ensures that a device is used only by a single kernel at each point in time.

B. Aparapi

Aparapi is a library that offers extensive functionality to ease the usage of the OpenCL API and minimize programming effort when developing OpenCL kernels [6]. Firstly, it contains bindings to access OpenCL functions through a Java Native Interface (JNI), abstracting and bundling multiple low-level calls within Java high-level functions. Secondly, Aparapi is able to translate Java code to valid OpenCL kernels. Input data may be defined in Java code itself and the results are again available in Java after execution. This is possible as Aparapi automatically copies the participating data back and forth between the host code and the executing device. For a more granular approach, developers can also define implicitly which data should be written or read in order to improve performance.

![Fig. 3: Aparapi translates Java code to OpenCL kernels.](image)

The framework drastically reduces code complexity, taking care of tasks such as device selection and data handling when no explicit control is required. This enables developers to program algorithms considerably faster and offers beginners a simplified access to OpenCL features without knowledge of low-level mechanisms.

III. RELATED WORK

In this section, we provide an overview of related OpenCL API forwarding approaches and frameworks for aggregating heterogeneous cluster resources.

A. SnuCL

Instead of simply forwarding API calls to remote machines, SnuCL [7] heavily transforms kernels depending on the available runtimes of a machine. For example, it translates OpenCL to CUDA when only the CUDA platform is provided by the machine. Additionally, it can transform OpenCL code to C, which is then executed within a thread for each core of a CPU. Furthermore, SnuCL introduces a virtual global memory, in which buffers may be shared among devices. It manages the consistency among shared buffers and attempts to minimize copy operations throughout the execution. Buffers that are not written can therefore remain on a device without the requirement to be copied back to the host.

B. DistCL

DistCL aims at fusing multiple GPUs into a single virtual OpenCL device [8]. To achieve this, it abstracts the devices by representing them as one unified device while handling kernel distribution and data transfers transparently. In order to enable parallelization of a kernel, DistCL automatically splits it into multiple kernels with their respective required data, called subranges. For this, programmers have to supply a meta-function that determines the memory access pattern. Based on the given function, DistCL can transfer only relevant data to a device that executes a subrange.

C. MultiCL

MultiCL is an extension of SnuCL and schedules kernels across multiple heterogeneous devices in a cluster [9]. It offers a round robin approach as well as an autofit mechanism. When queuing a kernel, a flag can be attached to it, which labels the assumed execution type. The available flags comprise compute-intensive, memory bound, I/O bound or iterative. The scheduling mechanism is able to employ a static or a dynamic algorithm. In the static scheduling approach, the approach profiles all available devices in regards to memory bandwidth and instruction throughput. Based on these measurements, the best fitting device is selected with respect to the kernel flag.
IV. APPROACH

Here, we provide an overview of the major characteristics of the job design, the job scheduling mechanisms, and the dynamic scale-out capabilities in CloudCL.

A. Job Design

In order to fully utilize as many remote computing resources as possible, CloudCL has to provide a mechanism for splitting tasks. It is mandatory to ensure that each split has all the necessary data for its correct computation all while keeping the amount of memory transfers at a minimum level. Based on literature research, we identified Manual Splits, Naive Buffer Replication [10], Intelligent Buffer Replication [11] and Meta Functions [8, 12] as potential splitting strategies. All methods come with a tradeoff between performance and programming complexity: Naive Buffer Replication is inefficient and does not scale with increasing split counts. Intelligent Buffer Replication requires an extra step for sampling memory accesses, which includes intermediate code translation. Although Meta Functions try to unburden developers, they are still required to define the access patterns manually. Manual Splits put the entire responsibility into the hands of the developer. While it is the least automated method, it also grants full control over performance. Thus, CloudCL employs the Manual Splits strategy. Listing 1 demonstrates how the splitting approach is integrated into OpenCL and that writing a split and merge algorithm can be intuitive and does not require much code. In order to grant higher parallelization, the partialCount can be dynamically changed into a divisor of the array length.

```java
public class AdditionKernel extends Kernel{
    int[] a, b, result;
    public AdditionKernel(int[] a, int[] b) {
        this.a = a;
        this.b = b;
        this.result = new int[a.length];
    }

    @Override
    public void run() {
        int i = getGlobalId();
        result[i] = a[i] + b[i];
    }

    public static void main(String[] args) {
        final int partialCount = 2;
        int[] a = new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
        int b = new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
        int result = new int[a.length];

        int partialWidth = a.length / partialCount;
        for(int i = 0; i < partialCount; i++){
            int i = getPartialId();
            int[] aPartial = Arrays.copyOfRange(a, i*partialWidth, (i+1)*partialWidth);
            int[] bPartial = Arrays.copyOfRange(b, i*partialWidth, (i+1)*partialWidth);
            System.arraycopy(additionKernel.result, 0, result, i*partialWidth, partialWidth);
        }
    }
}
```

Listing 1: Implementation of Manual Splits based on Aparapi. This example portrays the serial execution of the partials for the sake of simplicity.

The execution model of CloudCL requires developers to define Jobs, which are comprised of at least one Partial, which is an extended Aparapi kernel. Partial tasks contain the data that is split manually. After a job is constructed it can be submitted to the Job Executor that assigns and executes the partials across the available devices of its cluster. The Job partial execution model is illustrated in Figure 4.

![Figure 4](image-url)

**Fig. 4:** The CloudCL execution model splits job into partials, which are distributed across the cluster resources.

B. Job Scheduling

In order to allow the scheduler to serve specific cluster requirements, a pluggable two-tiered architecture is proposed. As such the scheduler consists of two modules, called Job Scheduler and Device Scheduler. CloudCL allows switching the modules during runtime in order to enable clusters to adapt to varying situations and usage scenarios. The overall scheduling architecture is depicted in Figure 5. In the following sections, the Job Scheduler and Device Scheduler are explained in detail.

![Figure 5](image-url)

**Fig. 5:** The two-tier scheduling architecture of CloudCL considers the job abstraction level on its first tier and compute device characteristics on the second tier.

1) Job Scheduler (First Tier): The first scheduling tier considers only the system state on a job abstraction level without having any knowledge about the actual partial structure or available devices in the cluster. Instead, this tier is mainly concerned about ensuring predefined fairness policies and as such controls the order in which jobs are eligible for consideration during a scheduling round. Algorithms available in CloudCL for this tier are First-In First-Out and Round-Robin. Still, the first tier hands over the partial order to the second tier, which may choose to ignore the given order based on the nature of the underlying algorithm.
2) Device Scheduler (Second Tier): The second scheduling tier gets its input from the preceding tier and has no concerns of the high-level job abstraction. Instead, it focuses on the actual assignment of partials to devices. Executing OpenCL code in a heterogeneous environment can lead to drastically different performance behavior based on the algorithm implementation and the executing device. For example, code that may perform exceptionally well on a GPU may run poorly on a CPU because of varying computational capabilities. Therefore, CloudCL enables developers to express preferences in terms of the device types that their algorithm should be computed on. The mechanism is implemented through a defined Device Preference attribute that can be attached to every partial. Valid values for the attribute include: None, CPU only, CPU preferred, GPU only and GPU preferred.

For CloudCL, not only the performance of the execution device matters, but also the speed of the network connection is a crucial factor. This becomes especially important when remote resources are tied in via a wide area network, which inherently can not offer the same bandwidth and latency as a local network. A naïve approach that combines measuring device performance and networking capabilities is the usage of historical data. With the assumption that jobs are split into many equally sized partials, one can identify the best-suited device for a partial given the history of previous executions within the cluster.

In order to provide scheduling algorithms on the second tier with meaningful metrics, CloudCL provides the following metrics per partial:

a) Kernel Data Approximation: Kernel classes report the actual size of the overall kernel including its data. The size is gained by employing Java reflection on an Aparapi kernel, identifying employed data types and data structures. However, this measure should only be treated as an approximation, as it does not consider the effects of explicit memory management.

b) Data Transfer History: To take explicit memory management into account, a history of transferred data volumes is maintained. For this purpose, the memory management calls of Aparapi were modified to accumulate the sizes of the corresponding data structures in a kernel attribute.

c) Performance History: CloudCL maintains a history of execution times of partials for every device. The observed timespan reaches from invocation to completion of a kernel from the application host, thus including network transfers.

In the current version of CloudCL, we implemented a Performance Based Device Scheduler based on the Performance History. It assigns a partial to the available device with the best performance history for the given kernel class. With the available metrics, more sophisticated and fine-grained schedulers can be implemented in the future. It should be noted that schedulers on the second tier may ignore the order of partials suggested by the first tier. For example, the first tier might hand over a list of partials in FIFO order. If this list is handled by a simplex-based scheduler on the second tier, the principle of FIFO may be violated in favor of finding the optimal distribution.

C. Dynamic Scale-Out Capabilities

The general idea behind scaling performance within CloudCL is that regardless of whether on-premise or cloud-based infrastructure as a Service (IaaS) resources are employed, you can add additional resources in times of high capacity demands. Similarly, in times of low demand, resources can be released again in order to reduce costs or yield resources to other workloads. Widely employed cluster resource management solutions include functionalities to add or remove nodes dynamically. Offering a similar functionality in a dOpenCL based solution differs substantially.

OpenCL and Aparapi are built to run on a single machine and as such include assumptions of limited capabilities concerning changing hardware during operation. For example, Aparapi caches device queries to OpenCL as devices are usually added or removed during maintenance procedures when a machine is powered down. dOpenCL overcomes such limitations and enables the host node to have devices virtually installed by adding nodes to the dOpenCL cluster at runtime. dOpenCL therefore extends the OpenCL standard by adding the custom methods clCreateComputeNodeWWU and clReleaseComputeNodeWWU.

As OpenCL itself has no multi-machine capabilities, it offers information only for the respective built-in devices through the standard API. Therefore all devices appear as if they were installed in the host node that runs the dOpenCL library. This makes identifying the owning machine per device impossible. Knowing the relation between devices and machines is crucial for providing basic functionality, such as releasing superfluous machines, where it has to be ensured that no partials are running anymore on its respective devices. In order to provide the machine-device relation, dOpenCL introduces another method called clGetDeviceIDsFromComputeNodeWWU.

The mentioned dOpenCL methods are available to C++ programmers when including the respective header files that contain the extensions. Thus, host code that wants to benefit from dynamic device management has to be modified appropriately. In its standard version, Aparapi is bound to the standard OpenCL specification and has no understanding of the offered dOpenCL extensions. Therefore it was necessary to modify the Aparapi JNI to enable dynamic resource scaling. In order to do so, two new JNI methods were implemented, addNode and removeNode. Both call the respective dOpenCL functions, with addNode also reporting back the available devices of the added node.

V. Evaluation

To evaluate the scale-out capabilities of CloudCL, this section starts with providing a detailed description of the test environment. This description includes the employed hardware as well as the general method of measurement. Afterwards, the results of our scale-out benchmarks are presented and discussed.
TABLE I: Specifications of on-premise nodes.

<table>
<thead>
<tr>
<th>HPE ProLiant m710p Server Cartridge</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>GPU</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>NIC</td>
</tr>
<tr>
<td>OS</td>
</tr>
<tr>
<td>OpenCL</td>
</tr>
</tbody>
</table>

TABLE II: Network performance of on-premise deployment.

<table>
<thead>
<tr>
<th>HPE ProLiant m710p Server Cartridge</th>
</tr>
</thead>
<tbody>
<tr>
<td>iperf3</td>
</tr>
<tr>
<td>ping</td>
</tr>
</tbody>
</table>

A. Test Environment & Method of Measurement

With CloudCL targeting dynamic deployment strategies on cloud infrastructure, the evaluation distinguishes between three scenarios: an on-premise private cloud deployment scenario, an Amazon EC2 based public cloud deployment scenario and a hybrid cloud deployment scenario. For each environment, we are utilizing well-defined hardware configurations and provide detailed hardware specifications. Since CloudCL heavily depends on network performance, we further provide network performance measurements based on iperf3 and the ping utilities, conducted between the application host and a compute node. For the on-premise private cloud deployment scenario, we utilize HPE ProLiant m710p server cartridges [13] for both compute nodes and the application host, with the detailed specifications denoted in Table I. The practical network performance is documented Table II. For the public cloud deployment scenario, g2.2xlarge instances are used to assess the performance of GPU-equipped compute nodes, whereas c4.8xlarge instances are employed to assess the performance of CPU-equipped compute nodes. In both cases, a c4.8xlarge instance is used for the application host. The detailed specifications and the network performance are reported in Table III and Table IV, respectively. The hybrid cloud deployment scenario uses one m710p on-premise compute node and up to three c4.8xlarge public cloud compute nodes. The network performance is documented in Table V.

TABLE III: Specifications of public cloud nodes.

<table>
<thead>
<tr>
<th>c4.8xlarge</th>
<th>g2.2xlarge</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-2666 v3, 36 cores</td>
</tr>
<tr>
<td>GPU</td>
<td>-</td>
</tr>
<tr>
<td>Memory</td>
<td>60GB</td>
</tr>
<tr>
<td>NIC</td>
<td>Mellanox Connect-X3 Pro (10 GBit/s)</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu 14.04.4 64 Bit</td>
</tr>
<tr>
<td>OpenCL</td>
<td>1.2.0.25</td>
</tr>
</tbody>
</table>

TABLE IV: Network performance of public cloud deployment.

<table>
<thead>
<tr>
<th>c4.8xlarge</th>
<th>g2.2xlarge</th>
</tr>
</thead>
<tbody>
<tr>
<td>iperf3</td>
<td>9.44 Gbit/s (σ = 0.068)</td>
</tr>
<tr>
<td>ping</td>
<td>0.158 ms (σ = 0.021)</td>
</tr>
</tbody>
</table>

TABLE V: Network performance of hybrid deployment.

<table>
<thead>
<tr>
<th>HPE ProLiant m710p &amp; c4.8xlarge</th>
</tr>
</thead>
<tbody>
<tr>
<td>iperf3</td>
</tr>
<tr>
<td>ping</td>
</tr>
</tbody>
</table>

Regarding the method of measurement, the application host and the compute nodes are always hosted on separate machines, hence kernel partials and the corresponding data always have to pass the network interface regardless of the number of compute nodes. This decision relies on the observation of Tausche et al. [14], who have demonstrated that using a fast network interconnect, the forwarding technique implemented by the dOpenCL framework incurs very little overhead compared to native execution.

To demonstrate the scaling capabilities of CloudCL, we decided to use naive matrix multiplication as a very data-intensive benchmark workload in order to test the scaling capabilities of CloudCL under the most challenging conditions. It should be noted that a naive matrix multiplication cannot be perfectly divided into partials. Instead, one of the matrices has to be a part in its entirety in each partial, while the second matrix can be split. This means that there is a growing overhead correlating to the number of splits. With every additional split, the input overhead is increased by the data size of the indivisible matrix, which is 50% of the overall input. Therefore, while a higher parallelization may grant faster computational capabilities of a cluster, it also requires more data transfers.

B. Benchmark Results

1) On-Premise Private Cloud Deployment Scenario: The performance for the on-premise deployment scenario is presented in Figure 6. Adding a second m710p compute node yields substantial performance improvements ranging from at least 1.65x speedup for small matrices up to 1.95x speedup for larger problem sizes. Adding a third m710p compute node pays off for larger problem sizes, where a speedup of 2.8x is achieved. Overall, CloudCL scales well for larger problem sizes in the on-premise deployment scenario, most likely due to the high-bandwidth and low-latency network connection between nodes.

![Fig. 6: Close-to-linear speedup in the on-premise deployment.](image-url)
2) Public Cloud Deployment Scenario: The performance results for the public cloud deployment scenario are illustrated in Figure 7a (GPU-based instances) and Figure 7b (CPU-based instances). Using GPU-based instances, CloudCL manages to achieve close to ideal scale-out performance with an average 1.96x speedup for using two g2.2xlarge compute nodes and an average 2.94x speedup when a third node is added. On the side of CPU-based instances, using a second c4.8xlarge compute node yields 1.94x speedup for large matrices. With up to 2.77x speedup for large matrices, adding a third node only exudes good scaling behavior for large matrices.

The discrepancy between the scaling behavior on GPU-based versus CPU-based compute nodes is likely to be caused by the differing network performance of employed instance types (see Table IV). In the case of the GPU-based setup, the application host node is equipped with a 10 Gbit/s Ethernet link which can easily saturate the 1 Gbit/s Ethernet links of the compute nodes. For the CPU-based setup, all nodes are equipped with a 10 Gbit/s Ethernet link, which no longer allows the application host node to fully saturate the network links of the compute nodes. To validate this hypothesis, additional experiments have to be conducted where the application host node is equipped with a faster network link.

3) Hybrid Cloud Deployment Scenario: Even though network connectivity is a massively limiting factor, we also conducted a benchmark using a hybrid cloud deployment scenario for the sake of completeness, where some compute nodes are hosted in the on-premise private cloud and other compute nodes are tied in from public cloud resources. Figure 8a presents the results of this experiment. As expected, using a data-intensive workload such as matrix multiplication, incorporating resources from the public cloud is not feasible at all. However, we conducted a second benchmark using Mandelbrot sets as a compute-intensive but data-insensitive workload in order to evaluate whether hybrid cloud deployment scenarios might be feasible for less data-dependent workloads. The results of the Mandelbrot set benchmark depicted in Figure 8b indicate much better scaling behavior for such workloads.

4) Overarching Evaluation: Our performance evaluation has demonstrated that CloudCL scales well both in on-premise and public cloud deployments, using matrix multiplication as a data-intensive worst case workload. With network connectivity being the predominating bottleneck, future versions of CloudCL might mitigate the impact of network performance by implementing lightweight compression or broadcast mechanisms in the underlying doOpenCL API forwarding mechanism.
VI. CONCLUSION

In this paper, we introduced the \textit{CloudCL} framework, which enables developers to focus their implementation efforts on compute kernels without having to consider inter-node communication. To achieve this goal, CloudCL removes two obstacles: First, by using Aparapi, developers can implement compute kernels in Java and no longer have to deal with tedious OpenCL boiler plate code for device management. For novice developers, this simplification greatly alleviates the hurdles towards getting started. Second, developers do not have to consider distributed aspects such as inter-node communication, since the underlying \textit{dOpenCL} library empowers developers to interact with remote compute devices as if they were installed locally. CloudCL extends both base technologies to enable the development of cloud-native application behavior by supporting dynamic addition and removal of compute nodes at runtime. Additionally, the combination of a straightforward job design and the corresponding job scheduling framework make sure that cluster resources are used efficiently and fairly.

In an extensive performance evaluation, we demonstrate that the framework provides close-to-linear scale-out performance in \textit{on-premise private cloud deployment scenarios} and in \textit{public cloud deployment scenarios}, using matrix multiplication as a data-intensive worst case workload. For data-intensive workloads, \textit{hybrid cloud deployment scenarios} suffer from insufficient wide area network connectivity. However, for compute-intensive but less data-sensitive workloads such as the computation of the Mandelbrot set, the \textit{hybrid cloud deployment scenario} also revealed good performance.

ACKNOWLEDGEMENT

This paper has received funding from the European Union’s Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866.

DISCLAIMER

This paper reflects only the authors’ views and the European Commission is not responsible for any use that may be made of the information it contains.

REFERENCES


