International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 8, Number 2, pages 282–300, July 2018

CloudCL: Single-Paradigm Distributed Heterogeneous Computing for Cloud Infrastructures

Max Plauth, Florian Rösler and Andreas Polze
Operating Systems and Middleware Group
Hasso Plattner Institute for Digital Engineering
University of Potsdam
PO Box 990460, 14440 Potsdam, Germany

Received: January 30, 2018
Revised: April 18, 2018
Accepted: June 6, 2018
Communicated by Shinya Takamaeda-Yamazaki

Abstract

The ever-growing demand for compute resources has reached a wide range of application domains, and with that has created a larger audience for compute-intensive tasks. In this paper, we present the *CloudCL* framework, which empowers users to run compute-intensive tasks without having to face the total cost of ownership of operating an extensive high-performance compute infrastructure. *CloudCL* enables developers to tap the ubiquitous availability of cloud-based heterogeneous resources using a single-paradigm compute framework, without having to consider dynamic resource management and inter-node communication. In an extensive performance evaluation, we demonstrate the feasibility of the framework, yielding close-to-linear scale-out capabilities for certain workloads.

Keywords: Cloud, Heterogeneous Computing, Single-Paradigm, Dynamic Resource Management

1 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 [12, 14].

Even though hardware accelerators such as Field-Programmable Gate Arrays (FPGAs) or Graphics Processing Units (GPUs) are a popular approach for satisfying these demands, operating a heterogeneous compute infrastructure is still expensive [15] and involves a high degree of application complexity [7]. 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 dynamic scale-out scenarios 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 [13].

Here, we present the *CloudCL*[1] framework, which empowers users to run compute-intensive tasks on demand without having to face the total cost of ownership of operating an extensive high-performance compute infrastructure. The presented framework hides the complexity of distributed programming for dynamic cluster configurations, enabling developers to tap the ubiquitous availability of cloud-based heterogeneous resources on demand using a single-paradigm compute framework. At the same time, the single paradigm approach helps in improving productivity, as developers and domain experts may focus their implementation efforts on application logic without having to consider inter-node communication and dynamic resource management. As illustrated in Figure 1, *CloudCL* combines the capabilities of *dOpenCL* [7] and *Aparapi* [10] and augments these enabling technologies with a job scheduling system that can handle the dynamic properties of cloud-based resource-provisioning 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*, using Amazon EC2-based *public cloud* resources as well as *hybrid* configurations.

This paper is structured as follows: Section 2 provides background about *MPI*, *Opencl*, *dOpenCL* and *Aparapi* for two purposes. On one hand, the goal is to provide a quick walk-through of the enabling technologies behind *CloudCL*. On the other hand, re-iterating over the basics of both distributed as well as heterogeneous computing should remind readers about the many layers of complexity that are concealed by the *CloudCL* framework. Subsequently, Section 3 reviews related *OpenCL* API forwarding approaches and compares them with *dOpenCL* – the library of our choice. Section 4 proceeds with elucidating the major components making up *CloudCL*. Afterwards, Section 5 provides an extensive performance evaluation of the scale-out behavior of *CloudCL*, before a conclusion is reached in Section 6.

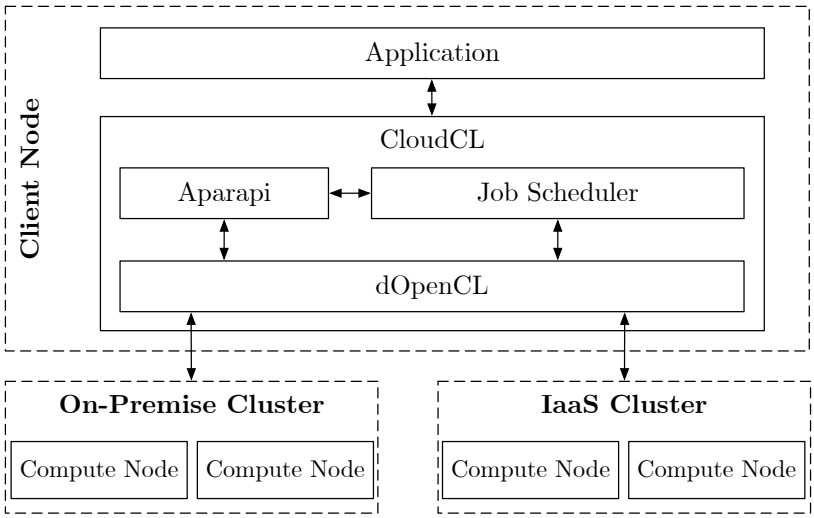


Figure 1: The *CloudCL* framework hides compute device management (*Aparapi*) and inter-node communication (*dOpenCL*), allowing developers to focus on compute kernel development.

2 Background

CloudCL is built on top of multiple base technologies, combining their functionality in order to offer more than just the sum of its parts. First, this section explicates the basics of distributed programming based on the Message Passing Interface (MPI), stressing the benefits of *CloudCL* as a single-paradigm approach for distributed heterogeneous computing. Further emphasizing the increased productivity offered by *CloudCL*, this section provides background about *Opencl*, *dOpenCL* and *Aparapi* as enabling technologies behind *CloudCL*.

[1] `https://github.com/osmhpi/cloudcl`

2.1 Distributed Programming with MPI

In order to implement software that can leverage the resources of many compute nodes, communication across multiple machines is necessary. The de-facto standard for facilitating inter-node communication in distributed setups is the MPI. Many MPI implementations are available, optimized for various hardware types and programming languages, enabling applications to identify a process instance, so called ranks, as well as sending and receiving messages[5] among them. While the provided mechanisms may be used to run multiple ranks on a single machine, their main purpose it to distribute workloads across cluster nodes.

```c
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char **argv){
 char messageBuffer[64];
 int rank, processCount;

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &processCount);

 if(rank == 0){
  for(int i = 1; i<processCount;i++){
   sprintf(messageBuffer, "This␣is␣a␣message␣from␣Process␣0.");
   MPI_Send(messageBuffer, sizeof(messageBuffer), MPI_CHAR, i, 0,
       ↪ MPI_COMM_WORLD);
  }
 }else{
  MPI_Recv(messageBuffer, sizeof(messageBuffer), MPI_CHAR, 0, 0,
      ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  printf("Process␣%d␣receives:␣%s\n", rank, messageBuffer);
 }

 MPI_Finalize();
 return 0;
}
```

Listing 1: Simple message passing example implemented in C using MPI.

MPI programs can be run with a specified number of processes. Executing the code reproduced in Listing 1 with 4 processes initiates a simple communication from the first process to all other processes. The given code is executed in every process but each process identifies its own rank, which is used to define whether to send or to receive a message.

2.2 Heterogeneous Programming with OpenCL

OpenCL is a compute framework for executing parallel compute kernels on heterogeneous hardware. OpenCL supports various device types and vendors [17]. In fact, OpenCL programs may be executed on CPUs, GPUs, *field-programmable gate arrays* (FPGAs) and *digital signal processors* (DSPs). A prerequisite for the execution of OpenCL code on a targeted device is the availability of an OpenCL *Installable Client Driver* (ICD) on the host system.

OpenCL kernels are written in OpenCL C, which is a superset of C99, containing additional keywords that are related to memory access and synchronization. Kernels have to be started from so-called host code, which runs on the machine that contains the targeted devices. The host code is usually written in C or C++, however other language bindings exist as well.

Listing 2 presents a minimal OpenCL kernel implementing the addition of two vectors. Each parameter is a pointer to an array of double values residing in the global memory of the accelerator. As the kernel is executed for every work-item, each kernel instance identifies its work-item by querying *get_global_id*.

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void vec_add(__global double *a, __global double *b, __global double *c){
 int i = get_global_id(0);
 c[i] = a[i] + b[i];
}
```

Listing 2: Examplary OpenCL vector addition kernel.

While this simplistic kernel amounts to very few lines of kernel code, OpenCL requires a lot of host code in order to choose an appropriate device, initiate data transfers, execute the kernel, and perform other auxiliary tasks. Requiring more than 50 lines of code, the verbosity of the host code is exemplified in Listing 3. It should be noted that the demonstrated host code is a minimal example that assumes the availability of only one OpenCL platform and only one device. Furthermore, the example has been stripped of error handling code.

```
int main(){
 // Initialize arrays on host
 double array_a[10] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
 double array_b[10] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
 double* array_result = new double[10];
 size_t size = 10 * sizeof(double);

 cl_mem d_a, d_b, d_c;        // Device input/output buffers
 cl_platform_id cpPlatform;   // OpenCL platform
 cl_device_id device_id;      // device ID
 cl_context ctx;              // context
 cl_command_queue queue;      // command queue
 cl_program program;          // program
 cl_kernel kernel;            // kernel

 // Bind to platform
 clGetPlatformIDs(1, &cpPlatform, NULL);
 // Get ID for the device
 clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_ALL, 1, &device_id, NULL);
 // Create a context
 ctx = clCreateContext(0, 1, &device_id, NULL, NULL, NULL);
 // Create a command queue
 queue = clCreateCommandQueue(ctx, device_id, CL_QUEUE_PROFILING_ENABLE, NULL);
 // Create the compute program from the source buffer
 program = clCreateProgramWithSource(ctx, 1, (const char**) &source, NULL, NULL);
 // Build the program executable
 clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
 // Create the compute kernel in the program we wish to run
 kernel = clCreateKernel(program, "vec_add", NULL);

  // Create the input and output arrays in device memory for our calculation
 d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, size, NULL, NULL);
 d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, size, NULL, NULL);
 d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, size, NULL, NULL);

 // Write our data set into the input array in device memory
 clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0, size, array_a, 0, NULL, NULL);
 clEnqueueWriteBuffer(queue, d_b, CL_TRUE, 0, size, array_b, 0, NULL, NULL);

 // Set the arguments to our compute kernel
 clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
 clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
 clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);

 // Execute the kernel over the entire range of the data set
 clEnqueueNDRangeKernel(queue, kernel, 1, NULL, {10} , NULL, 0, NULL, NULL);

 // Wait for the command queue to get serviced before reading back results
 clFinish(queue);

```

```
51   // Read the results from the device
52   clEnqueueReadBuffer(queue, d_c, CL_TRUE, 0, size, array_result, 0, NULL, NULL);
53
54   // release OpenCL resources
55   clReleaseMemObject(d_a); clReleaseMemObject(d_b); clReleaseMemObject(d_c);
56   clReleaseProgram(program); clReleaseKernel(kernel);
57   clReleaseCommandQueue(queue); clReleaseContext(context);
58
59   for(int i = 0; i < 10; i++){
60    cout << array_result[i] << endl;
61   }
62  }
```

Listing 3: Minimal OpenCL host code necessary to execute the vector addition kernel.

2.3 Distributed Heterogeneous Programming with dOpenCL

During the development of software that is intended to be executed on clusters made up from heterogeneous systems, software developers have to consider both distribution aspects (e.g. using MPI as discussed in Section 2.1) as well as heterogeneous programming environments (e.g. using OpenCL as discussed in Section 2.2). Both programming paradigms add a lot of complexity that may be handled by experienced software developers, but which is extremely hard to manage for domain experts (e.g. biologists, physicists, etc.). Fortunately, API forwarding approaches enable developers to interact with remote accelerators while hiding the aspects of distributed programming. For the OpenCL compute language, one such approach is *dOpencl*, which enables developers to transparently utilize *OpenCL* devices installed in remote machines [7]. 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 exemplary compute cluster based on *dOpencl* is shown in Figure 2.

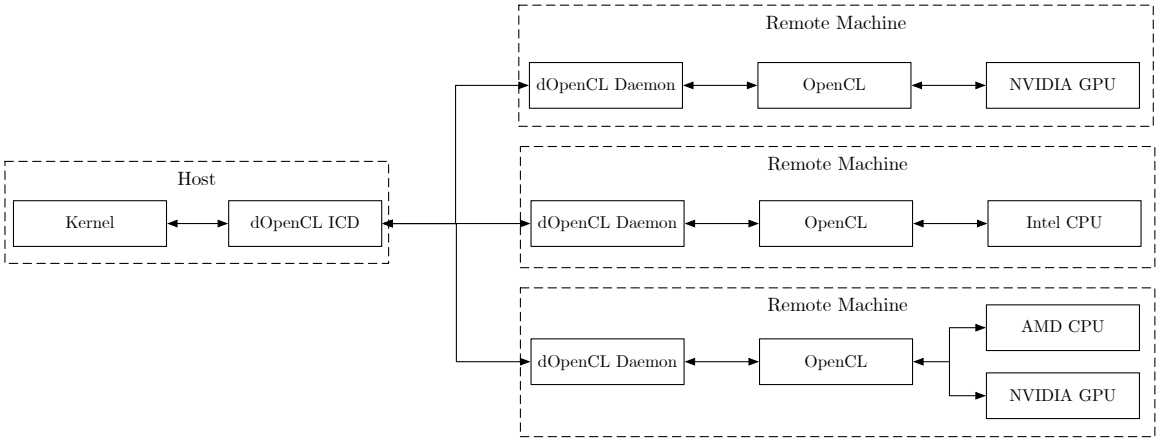


Figure 2: *dOpenCL* ties in remote resources in a local ICD.

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.

2.4 Aparapi

Aparapi is a library that drastically simplifies the usage of the *OpenCL* API and that minimizes development efforts of *OpenCL* kernels [10]. 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, as illustrated in Figure 3. 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. The library also takes care of all tedious setup-tasks demonstrated in Listing 3. For a more fine grained approach, developers can also define explicitly which data should be written or read in order to improve performance.

```
1  final double[] a = new double[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
2  final double[] b = new double[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
3  final double[] c = new double[10];
4
5  Kernel kernel = new Kernel() {
6    @Override
7    public void run() {
8       int i = getGlobalId();
9       c[i] = a[i] + b[i];
10   }
11 };
12
13 kernel.execute(10);
14 System.out.println(Arrays.toString(c));
```

Listing 4: Vector addition kernel and host code based on *Aparapi*.

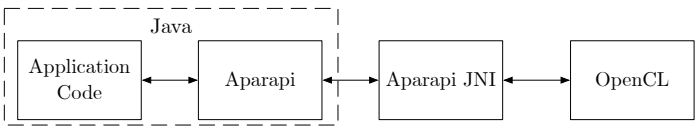


Figure 3: *Aparapi* translates Java code to *OpenCL* kernels.

The framework enables developers to express the same functionality of native OpenCL with much fewer lines of code, reducing code complexity significantly as demonstrated in Listing 4. This enables developers to program algorithms considerably faster and offers beginners and domain experts easy access to *OpenCL* features without profound knowledge of low-level mechanisms.

3 Related Work

With API-forwarding techniques for *OpenCL* providing one of the base technologies enabling *CloudCL*, this section offers an overview of related *OpenCL* API forwarding approaches and frameworks for aggregating heterogeneous cluster resources. Theoretically, all approaches could be used as replacements for *dOpenCL*.

3.1 SnuCL

SnuCL [9] may 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.

3.2 VirtualCL

Similarly to *SnuCL* and *dOpenCL*, *VirtualCL* forwards *OpenCL* API calls to remote machines within a cluster [2]. The authors evaluated various applications comparing the local versus remote runtimes via *VirtualCL*. *VirtualCL* also offers an extension, called *SuperCL*, which aims at reducing network transfers by allowing multiple kernels to be submitted to a remote node for consecutive execution. In this case, the results do not have to be transferred back and forth. It also supports more complex use cases like alternating iterative kernels on the same data set.

3.3 DistCL

DistCL aims at fusing multiple GPUs into a single virtual *OpenCL* device [4]. 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. However unlike *CloudCL*, *DistCL* is not capable of managing dynamic resources that are being added or removed during the lifetime of the cluster.

3.4 MultiCL

MultiCL is an extension of *SnuCL* and schedules kernels across multiple heterogeneous devices in a cluster [1]. 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.

4 Approach

With *CloudCL* being built on top of many other frameworks and libraries, this Section explicates how *CloudCL* offers more than just the sum of its parts. Here, we provide an overview of the major characteristics of *CloudCL*, including the *job design*, the *resource manager* the *job scheduling* mechanisms, and the *dynamic scale-out capabilities* offered by *CloudCL*.

4.1 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 the following strategies for dividing jobs into job partials:

Manual Splits Hadoop MapReduce enforces saving input data for the executable tasks in split files on the Hadoop Distributed File System. Each split file is executed independently and software developers have to define how to interpret a split during the map phase. Ultimately, the individually programmed reduce algorithm merges the splits together to form a final result.

Naive Buffer Replication A simple and safe way to ensure that every split has all necessary data available is to transfer all data to the employed devices as done by De La Lama et al. [3]. The suggested method is straightforward for transferring input data to remote devices. In contrast, merging the individual output buffers back into the final result requires developers to specify an algorithm for merging the partial results back to the global output buffer.

Intelligent Buffer Replication Kim et al. [8] have developed a mechanism for automatically determining how to split input buffers among subtasks in OpenCL so that only minimal data has to be transferred. Their method is based on sampling memory access patterns in a test run in order to identify lower and upper memory bounds for the job partials.

Meta Functions Li et al. [11] as well as Diop et al. [4] employ meta functions to determine data splits independently of the problem size. These meta functions have to be supplied by the programmer, describing the memory access patterns of the employed algorithms. Based on these meta functions, the bounds of the job partials can be identified.

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 additional execution 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. In this initial version, *CloudCL* employs the *Manual Splits* strategy. However, future versions of *CloudCL* will also support more elaborate strategies for distributing work. Listing 5 demonstrates how the manual splitting approach is integrated into *CloudCL* and that writing a split and merge algorithm can be intuitive and does not require much code. Using the variable *partialCount*, developers can specify an upper bound for the number of job partials that are created for each kernel. Higher numbers may enable a higher degree of parallelization due to wider distribution, however lower numbers might be more suitable for tasks where input data is hard to split, resulting in a behavior similar to *Naive Buffer Replication*.

```
1  public class AdditionKernel extends Kernel{
2   int[] a, b, result;
3
4   public AdditionKernel(int[] a, int[] b) {
5    this.a = a;
6    this.b = b;
7    this.result = new int[a.length];
8   }
9
10  @Override
11   public void run() {
12   int i = getGlobalId();
13   result[i] = a[i] + b[i];
14 }}
15
16 public class Addition {
17  public static void main(String[] args) {
18   final int partialCount = 2;
19   int[] a = new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
20   int[] b = new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
21   int[] result = new int[a.length];
22
23   int partWidth = a.length / partialCount;
24   for(int i = 0; i < partialCount; i++){
25    int[] aPartial = Arrays.copyOfRange(a, i * partWidth, (i + 1) * partWidth);
26    int[] bPartial = Arrays.copyOfRange(b, i * partWidth, (i + 1) * partWidth);
27
28    AdditionKernel additionKernel = new AdditionKernel(aPartial, bPartial);
29    additionKernel.execute(partWidth);
30    System.arraycopy(additionKernel.result, 0, result, i * partWidth, partWidth);
31 }}}
```

Listing 5: 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. Partials 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.

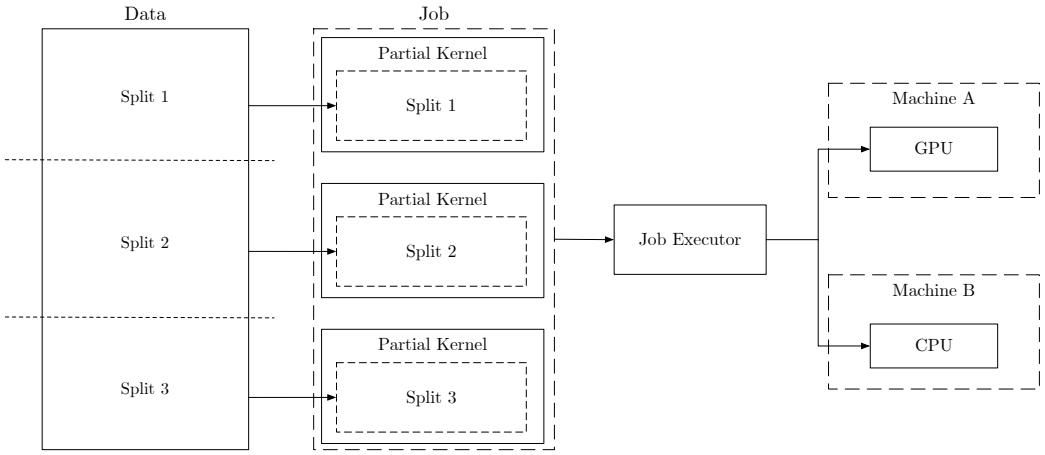


Figure 4: The *CloudCL* execution model splits kernel invocations into job partials, which are distributed across the available cluster resources.

Job Dependencies In practice, most applications are comprised of multiple jobs. Hence, it is important to talk about concurrency and synchronization aspects of the current job design. At its core, a *CloudCL* job resembles a kernel in *Aparapi* or *OpenCL*, with the main difference being that jobs invoke the same kernel concurrently, resulting in partial kernels that operate on different splits (i.e. subranges) each. The manual splitting approach covered previously assumes that each split can be executed independently from other splits, prohibiting any inter-split communication and thus sparing the necessity for synchronization mechanisms on the inter-split/intra-job level.

Regarding concurrency and synchronization on the inter-job level, *CloudCL* inherits the behavior of *Aparapi*. Per default, *Aparapi* does not use multiple command queues, resulting in jobs to be executed sequentially in the order in which their execution has been issued. While this behavior is guaranteeing sequential execution order of jobs, it is also limiting applications that would otherwise benefit from leveraging an increased level of concurrency for jobs that do not depend on each other and thus could be executed concurrently. For future iterations of *CloudCL*, enabling concurrent execution of jobs is a top priority, as well as providing the means necessary to express inter-job dependencies, including inter-job synchronization mechanisms.

4.2 Resource Manager

The *Resource Manager* component enables *CloudCL* to utilize both on-premise as well as cloud-based compute resources, as long as they provide support for *OpenCL*. While the amount of local compute resources is usually quite static, the resources provided by cloud-based instances may vary rapidly. The dynamic properties are taken into consideration by the *Resource Manager* interface (see Listing 6. The interface is used by *CloudCL* to manage resources from various sources. Arbitrary resources can be managed by providing corresponding implementations of the *Resource Manager* interface. In our first version of *CloudCL*, the we provide implementations for static on-premise resources as well as for dynamic cloud-based resources offered by Amazon EC2. However, EC2 serves as a mere example and the *ResourceManager* interface has been designed with interchangeability in mind so that it is simple to implement support for arbitrary providers as well.

```
public abstract class MachineManager {
  protected abstract List<CloudCLInstance> bookInstance(int count, String type);
  protected abstract void terminateInstances(List<CloudCLInstance> instances);
  ...

  public List<CloudCLInstance> book(int count, String type){
    List<CloudCLInstance> instances = bookInstance(count, type);
    ...
    return instances;
  }

  public void release(List<CloudCLInstance> instances){
    terminateInstances(instances);
    ...
  }
}
```

Listing 6: The *Machine Manager* interface enables managing resources from various providers.

4.3 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.

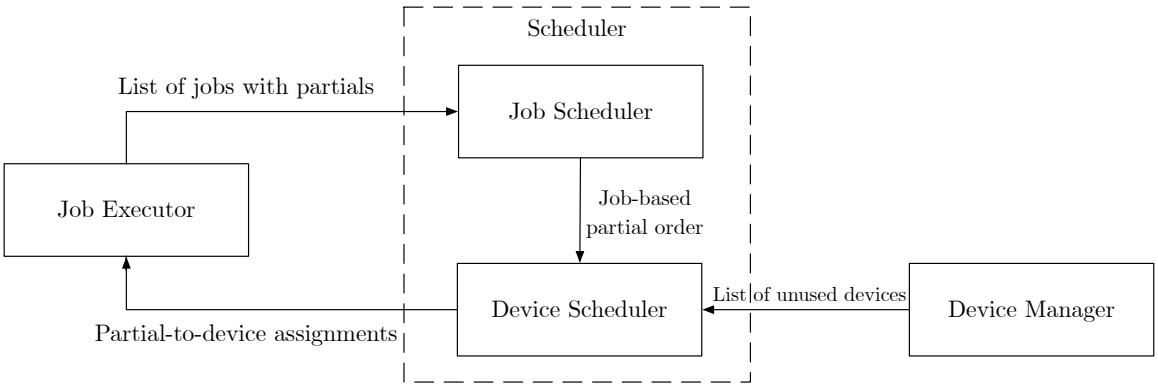


Figure 5: The two-tier scheduling strategy of *CloudCL* considers the job abstraction level on its first tier and compute device characteristics on the second tier.

4.3.1 Job Scheduler (First Tier)

The first scheduling tier considers only the system state at the 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.

4.3.2 Device Scheduler (Second Tier)

The second scheduling tier works on the output of 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:

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.

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.

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.

4.4 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.

Table 1: Specifications of on-premise nodes.

	HPE ProLiant m710p Server Cartridge
CPU	1 × Intel Xeon E3-1284L v4 (8 logical cores)
Memory	4 × 8GB PC3L-12800
NIC	Mellanox Connect-X3 Pro (10 GBit/s)
OS	Ubuntu 16.04.1 64 Bit
OpenCL	1.2.0.25

Table 2: Network performance of on-premise deployment.

	HPE ProLiant m710p Server Cartridge
iperf3	9.409 Gbit/s ($\sigma = 0.051$)
ping	0.14 ms ($\sigma = 0.012$)

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.

5 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.

5.1 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

Table 3: Specifications of public cloud nodes.

	c4.8xlarge	g2.2xlarge
CPU	Intel Xeon E5-2666 v3, 36 cores	Intel Xeon E5-2670, 8 cores
GPU	-	NVIDIA GRID K520
Memory	60GB	15GB
NIC	10 GBit/s	1 GBit/s
OS	Ubuntu 14.04.4 64 Bit	Ubuntu 14.04.4 64 Bit
OpenCL	1.2.0.25	1.2 CUDA 367.57

Table 4: Network performance of public cloud deployment.

	c4.8xlarge	g2.2xlarge
iperf3	9.44 Gbit/s ($\sigma = 0.068$)	992.6 MBit/s ($\sigma = 4.8$)
ping	0.158 ms ($\sigma = 0.021$)	1.29 ms ($\sigma = 0.047$)

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 [6] for both compute nodes and the application host, with the detailed specifications denoted in Table 1. The practical network performance is documented Table 2. For the *public cloud deployment scenario*, *g2.2xlarge* instances are used to asses the performance of GPU-equipped compute nodes, whereas *c4.8xlarge* instances are employed to asses 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 3 and 4, 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 5.

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. [16], 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. For the job scheduling tier, the *round-robin* approach has been used. On the device scheduling level, the history-based *performance based device scheduler* has been utilized, which assigns partials to the best suitable device in case multiple devices are available.

To demonstrate the scaling capabilities of *CloudCL*, we evaluate the performance of the framework for two different workloads: a *single-job workload* where all resources are utilized to complete a single job, and a *mixed-jobs workload*, where a suite of jobs with varying characteristics (e.g. compute intensive, memory intensive) is executed on the *CloudCL* cluster.

Single-Job Workload For the single-job case, we use naïve 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 naïve matrix multiplication can not be perfectly divided into partials. Instead, one of the matrices has to be a part in its entirety in each partial,

Table 5: Network performance of hybrid deployment.

	HPE ProLiant m710p & c4.8xlarge
iperf3	169.26 Mbit/s ($\sigma = 11.93$)
ping	18.9 ms ($\sigma = 0.17$)

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.

Mixed-Jobs Workload The mixed-jobs case aims at evaluating the performance of *CloudCL* in shared cluster environments. For this purpose, we compiled job suite comprised of six workloads of various complexity and type. Not only should the runtime per partial differ between jobs but also the amount of data that needs to be transferred. The employed workloads as well as the respective problem sizes are documented in Table 6.

Table 6: The mixed-jobs workloads comprises multiple workloads and problem sizes.

Workload	Problem Size	Iterations	Partials
GEMM	8000x8000	1	5
GEMM	6000x6000	1	5
Mandelbrot	1000x1000 (1000000 iterations)	1	5
Mandelbrot	2000x2000 (600000 iterations)	1	5
K-means	6000000 objects, 200 clusters	10	1
N-body	64000 object	1	1

5.2 Single-Job Performance

5.2.1 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.

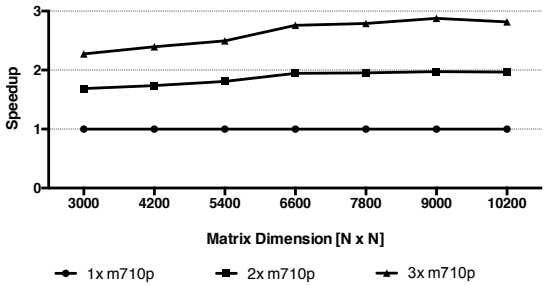
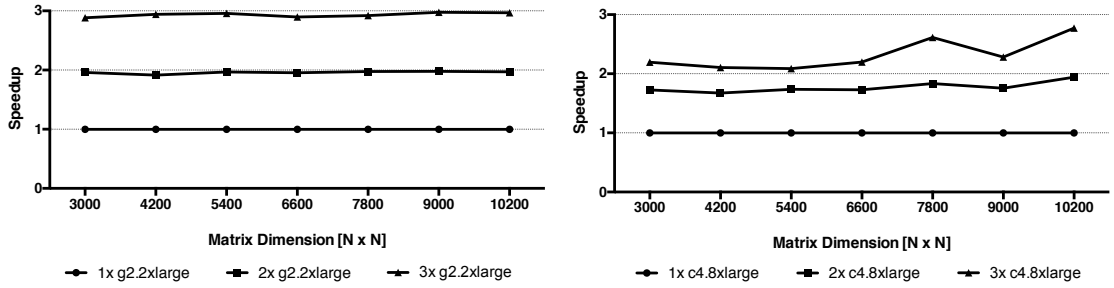


Figure 6: *CloudCL* achieves close-to-linear speedup in the on-premise deployment scenario.

5.2.2 Public Cloud Deployment Scenario

The performance results for the *public cloud deployment scenario* are illustrated in Figure 8. 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.



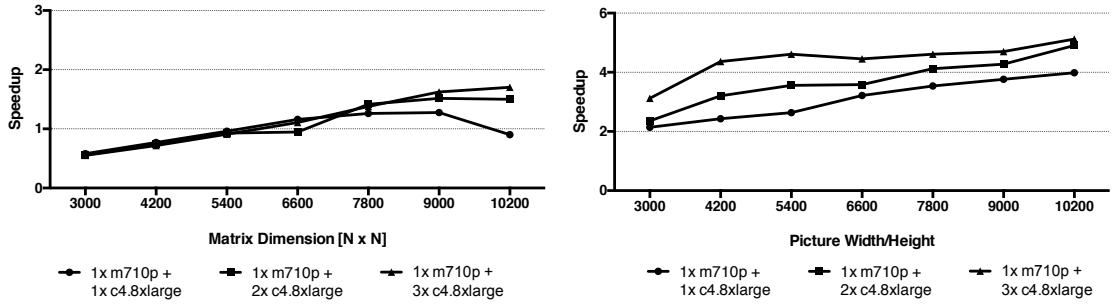
(a) GPU-based compute nodes (b) CPU-based compute nodes

Figure 7: Using public cloud resources, GPU-based nodes achieve linear speedup (a), whereas CPU-based nodes offer close-to-linear speedup for larger problem sizes (b).

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 4). 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.

5.2.3 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-intense 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.



(a) Matrix Multiplication (b) Mandelbrot Set

Figure 8: Hybrid clouds are not feasible for data-intensive workloads (a), but perform well for compute-intensive workloads (b).

5.3 Mixed-Jobs Performance

5.3.1 On-Premise Private Cloud Deployment Scenario

The performance measurements for the mixed workloads case reported in Figure 9 indicate that employing additional remote machines for mixed workloads can produce significant performance benefits. Adding a second and third machine yields a 1.8x and 2.4x speedup respectively. Hypothetically, the additional workloads are less data-intensive and thus should be less sensitive to an increased number of compute nodes. Since the data-intensive single-job workload yielded better performance, it might be possible that the job splitting strategies implemented in the additional workloads deliver sub-par performance. This hypothesis needs to be investigated in additional experiments.

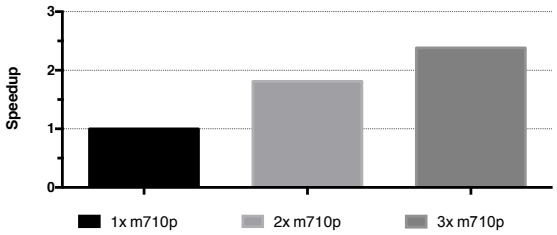
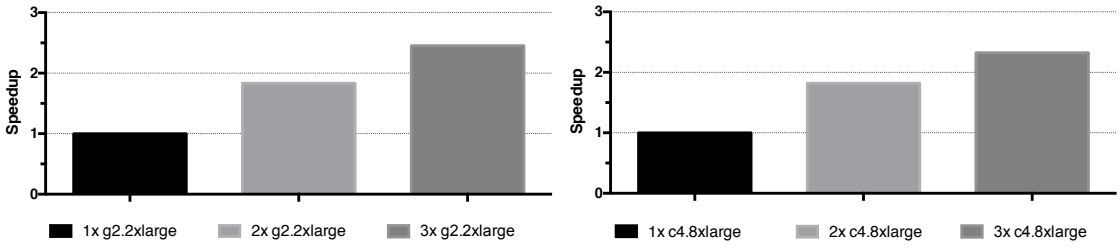


Figure 9: The mixed workload does not scale optimally in the private cloud environment.

5.3.2 Public Cloud Deployment Scenario

The evaluation results illustrated in Figure 11 draw a similar picture like the *on-premise private cloud deployment scenario* for mixed workloads: adding a second and third instance provides 1.8x and 2.3x speedup respectively when using CPU resources, and 1.8x and 2.5x speedup respectively when GPU instances are employed. In both cases, the performance gains of adding a third instance did not match those of the second instance, which contradicts the results retrieved for the data-intensive single-job workload. Even more so, the single-job workload yielded distinct differences between CPU- and GPU instances, as well as between private and public cloud deployment. Having yielded homogeneous results on very different hardware might indicate deficiencies in the implementation of the additional jobs.



(a) GPU-based compute nodes

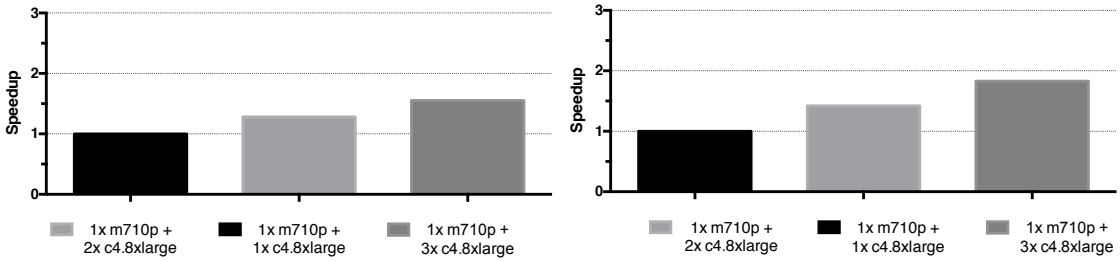
(b) CPU-based compute nodes

Figure 10: Regardless of the employed instance types, the public cloud environment yields performance levels for the mixed workload similar to the private cloud environment.

5.3.3 Hybrid Cloud Deployment Scenario

As reported in Figure 11a, the meagre performance retrieved for the *hybrid cloud deployment scenario* suffers from the relatively limited capacities of our wide area network link. Adding one and two remote *c4.8xlarge* instances in addition to the *m710p* cartridge resulted in a mere 1.28x and 1.55x speedup respectively. Considering the computational capabilities of the *c4.8xlarge* instance types, running mixed workloads in the *hybrid cloud deployment scenario*is not feasible, yet.

In an attempt to mitigate the impact of the slow network connectivity, we implemented an additional *network aware device scheduler*, which incorporates knowledge about the current cluster topology. More specifically, the *network aware device scheduler* distinguishes between local and remote resources. Based on this information, the scheduler classifies partials based on the amount of data they comprise and assigns them correspondingly: remote devices receive partials with small data volumes, whereas local devices are assigned with partials requiring large data volumes.



(a) Default Performance Based Device Scheduler

(b) Network Aware Device Scheduler

Figure 11: Hybrid clouds are not feasible for mixed workloads as well (a), however improved scheduling mechanisms can mitigate the impact of the network bottleneck (b).

With the new scheduler in place, a repeated benchmark run yielded the performance results reported in Figure 11b. The new scheduler introduces noticeable benefits compared to the approach based on performance history. Adding one *c4.8xlarge* instance results in 1.42x speedup, compared to 1.28x for the vanilla scheduler. Similarly, a second remote *c4.8xlarge* instance results in 1.83x speedup, compared to 1.55x for the vanilla scheduler. Whereas the measured performance gains are still far from linear speedup, the performance gains yielded by the *network aware device scheduler* support the hypothesis, that there is still a lot of potential for further optimizations, especially for mixed workloads.

5.4 Overarching Evaluation

Our performance evaluation has demonstrated that *CloudCL* scales well for single-job workloads, both in *on-premise* and *public cloud* deployments, using matrix multiplication as a data-intensive worst case workload. For *hybrid cloud* deployments, network connectivity remains as the predominating bottleneck. Additional benchmarks using mixed workloads demonstrated that *CloudCL* is capable of handling a variety of different job types. However, the overall scaling capabilities for mixed workloads remain on subpar levels, however the results also revealed several opportunities for action in order to improve the performance of *CloudCL* for such workloads. Regardless of the workload types, network connectivity is a common limitation. Hence, to mitigate the impact of the network capacity both for single-job as well as for mixed workloads, we are currently investigating potential implementation strategies for transparent light-weight compression in the underlying *dOpenCL* API forwarding mechanism. Furthermore, additional performance gains can be expected from improved job partitioning strategies as well as for improvements in the job scheduling mechanisms.

6 Conclusion

In this paper, we introduced the *CloudCL* framework, which empowers a wider audience of users to run compute-intensive tasks without having to face the total cost of ownership of operating an extensive high-performance compute infrastructure. To achieve this goal, *CloudCL* offers two major contributions: The presented framework hides the complexity of distributed programming for dynamic cluster configurations, enabling developers to tap the ubiquitous availability of cloud-based heterogeneous resources on demand using a single-paradigm compute framework. Furthermore, the single paradigm approach helps in improving productivity, as developers and domain experts may focus their implementation efforts on application logic without having to consider inter-node communication and dynamic resource management.

In an extensive performance evaluation, we demonstrated that the framework provides close-to-linear scale-out performance for optimized single-job workloads in *on-premise private cloud deployment scenarios* and in *public cloud deployment scenarios*, using matrix multiplication as a data-intensive worst case workload. For data-intensive workloads, *hybrid cloud deployment scenarios* suffer from insufficient wide area network connectivity. However, for compute-intense but less data-sensitive workloads such as the computation of the *Mandelbrot* set, the *hybrid cloud deployment scenario* also revealed good performance. Additional tests using a bouquet of different job types in a mixed workload scenario did not match the scaling capabilities of the single-job workload scenario, however the benchmarks also revealed several opportunities for action in order to improve the performance for mixed workloads in future versions of *CloudCL*.

References

[1] Ashwin Mandayam Aji, Antonio J. Peña, Pavan Balaji, and Wu-chun Feng. Automatic Command Queue Scheduling for Task-Parallel Workloads in OpenCL. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing*, CLUSTER '15, pages 42–51, Washington, DC, USA, 2015. IEEE Computer Society.

[2] A. Barak and A. Shiloh. The VirtualCL (VCL) Cluster Platform. White paper, Rachel and Selim Benin School of Computer Science, 2014.

[3] Carlos S. de la Lama, Pablo Toharia, Jose Luis Bosque, and Oscar D. Robles. Static Multi-device Load Balancing for OpenCL. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA '12, pages 675–682, Washington, DC, USA, 2012. IEEE Computer Society.

[4] Tahir Diop, Steven Gurfinkel, Jason Anderson, and Natalie Enright Jerger. DistCL: A Framework for the Distributed Execution of OpenCL Kernels. In *Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, MASCOTS '13, pages 556–566, Washington, DC, USA, 2013. IEEE Computer Society.

[5] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1, June 2015.

[6] Hewlett Packard Enterprise. HPE ProLiant m710p Server Cartridge QuickSpecs. `https://goo.gl/0dV579`, 2015.

[7] Philipp Kegel, Michel Steuwer, and Sergei Gorlatch. dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 174–186, Washington, DC, USA, 2012. IEEE Computer Society.

[8] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a Single Compute Device Image in OpenCL for Multiple GPUs. *SIGPLAN Not.*, 46(8):277–288, February 2011.

[9] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA, 2012. ACM.

[10] Ryan LaMothe and Gary Frost. Official AMD Aparapi Repository. `https://github.com/aparapi/aparapi`.

[11] Pei Li, Elisabeth Brunet, Francois Trahay, Christian Parrot, Gael Thomas, and Raymond Namyst. Automatic OpenCL Code Generation for Multi-device Heterogeneous Architectures. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*, ICPP '15, pages 959–968, Washington, DC, USA, 2015. IEEE Computer Society.

[12] M Johnson et al. Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation. *CoRR*, abs/1611.04558, 2016.

[13] Sebastian Nanz, Scott West, and Kaue Soares da Silveira. *Examining the Expert Gap in ParallelProgramming*, pages 434–445. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[14] Ladislav Rampasek and Anna Goldenberg. TensorFlow: Biologys Gateway to Deep Learning? *Cell Systems*, 2(1):12–14, 2016.

[15] Federico Silla, Sergio Iserte, Carlos Reaño, and Javier Prades. On the benefits of the remote GPU virtualization mechanism: The rCUDA case. *Concurrency and Computation: Practice and Experience*, 29(13), 2017.

[16] Karsten Tausche, Max Plauth, and Andreas Polze. dOpenCL – Evaluation of an API-Forwarding Implementation. In *Proceedings of the Fourth HPI Cloud Symposium "Operating the Cloud" 2016*, Potsdam, Germany, 2017.

[17] The Khronos Group. List of OpenCL Conformant Companies. `https://www.khronos.org/conformance/adopters/conformant-companies`, 2018.