Tampere University

Jan Solanti

DISTRIBUTED LOW LATENCY COMPUTING WITH OPENCL

A Scalable Multi-Access Edge Computing Framework

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Assistant Prof. Pekka Jääskeläinen, M.Sc. Joonas Multanen
December 2020

ABSTRACT

Jan Solanti: Distributed Low Latency Computing with OpenCL
Master of Science Thesis
Tampere University
Master's Programme in Information Technology
December 2020

The ever increasing computational complexity of applications requires increasing amounts of processing power, yet users are increasingly moving to resource and power constrained mobile devices for their computational needs. This calls for creative solutions that provide increased processing capabilities without impacting battery life or degrading the user experience. Multi-Access Edge Computing is a standardization effort to provide consistent cloud edge environments for optimizing applications on low-power devices by enabling developers to offload parts of the application to networked computing infrastructure that is located physically close to the device running the application.

This master's thesis describes pocl-r, a framework for transparently offloading computation in applications that use the OpenCL API for heterogeneous computation. The implementation performs comparably to previous work in synthetic benchmarks while offering greater flexibility to application developers by not depending on 3rd party communication frameworks and not requiring the application to be aware of any particular OpenCL API extensions. In addition to synthetic benchmarks, the impact of offloading heavy computation is measured in a case study of a mobile application that renders a streamed animated point cloud. The resulting energy consumption when offloading was measured to be roughly half of what it was without offloading. When additionally making the application aware of a minimal extension to the OpenCL API, energy consumption per frame was cut to a roughly a 20th of the original while also increasing the framerate tenfold.

Keywords: MEC, OpenCL, pocl, parallel computing, distributed computing

TIIVISTELMÄ

Jan Solanti: Laskennan hajautus matalalla latenssilla käyttäen OpenCL:ää
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Joulukuu 2020

Tietokonesovellusten alati kasvava laskentakompleksiteetti vaatii jatkuvasti suurempaa laskentatehoa, mutta käyttäjät ovat siirtyneet enenevässä määrin mobiililaitteisiin joita rajoittavat rajalliset laskentaresurssit sekä virrankulutus. Ongelma on ratkaistava tavalla joka ei lyhennä käyttäjien laitteiden akunkestoa eikä huononna käyttökokemusta. MEC-standardi pyrkii luomaan yhtenäisen ympäristön ns. pilvenreunalaskennalle, joka on käyttäjän laitteella ja pilvessä tehtävän laskennan välimuoto. Se tarjoaa kehittäjille keinon hajauttaa sovelluksen raskasta laskentaa vaativat osat käyttäjän päätelaitetta fyysisesti lähellä olevalle infrastruktuurille.

Tämä diplomityö esittelee pocl-r:n, kirjaston joka mahdollistaa laskennan heterogeenisen hajauttamisen OpenCL-rajapintaa hyödyntäen. Kirjaston toteutus suoriutuu keinotekoisissa tehokkuusmittauksissa kilpailukykyisesti aiempien vastaavien kirjastojen kanssa mutta tarjoaa kehittäjille enemmän joustavuutta. Erona aiempiin kirjastototeutuksiin, pocl-r ei vaadi sovellukselta OpenCL-rajapinnan laajennosten käyttöä eikä muiden ulkopuolisten kirjastojen käyttöä kommunikaatiota varten. Keinotekoisten suorituskykytestien lisäksi hajautuksen vaikutusta suorituskykyyn ja virrankulutukseen mitattiin mobiilisovelluksella, joka piirtää suoratoistettua pistepilvidataa. Pelkällä hajautuksella sovelluksen energiankulutus saatiin puolitettua. Ottamalla lisäksi käyttöön minimaalinen laajennos OpenCL-rajapintaan, energiankulutus piirrettyä kuvaa kohden laski kahdeskymmenesosaan alkuperäisestä ja piirtonopeus kymmenkertaistui.

Avainsanat: MEC, OpenCL, pocl, rinnakkainen laskenta, hajautettu laskenta

PREFACE

This master's thesis was written as part of my work at Tampere University under supervision from Pekka Jääskeläinen. The implementation of the pocl-r framework was done jointly by the Customized Parallel Computing group that the Author was part of. The basis for pocl-r was mainly implemented by Michal Babej and peer-to-peer signaling and command protocol optimizations were performed by the Author as part of this thesis.

The Author would like to thank his co-workers Julius Ikkala and Michal Babej for their endless patience and invaluable insights during the whole project. Additionally the Author would like to thank his friends, in particular Julius Ikkala and Jesper Hjorth for helping proofread this thesis and putting up with him during the entire writing process.

This project has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 783162. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Netherlands, Czech Republic, Finland, Spain, Italy. It was also supported by European Union's Horizon 2020 research and innovation programme under Grant Agreement No 871738 (CPSoSaware) and a grant from the HSA Foundation.

Tampere, 7th December 2020

Jan Solanti

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CAN	Controller Area Network
CPU	Central Processing Unit
ETSI	European Telecommunications Standards Institute
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
HEVC	High Efficiency Video Coding
MEC	Multi-Access Edge Computing
MIMD	Multiple Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
MPI	Message Passing Interface
MPSD	Multiple Program, Single Data
OpenCL	Open Computing Language
OpenGL	Open Graphics Layer
PoCL	Portable Computing Language
pocl-r	PoCL Remote
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SoC	System on Chip
SPMD	Single Program, Multiple Data
TAU	Tampere University
TUNI	Tampere Universities

1 INTRODUCTION

In recent years, computing has increasingly moved to handheld and other mobile devices which, while constantly improving, are still very limited in processing power and even further constrained by energy use limits imposed by battery capacity. While heavy computational workloads are clearly not suitable for such devices, improvements to network connectivity promised by WiFi6 and 5G standards have sparked the development of Mobile Edge Computing (MEC), a virtual platform standardized by the European Telecommunications Standards Institute (ETSI). MEC is described as a natural evolution of cellular and other mobile base stations and at its core it brings together the computational power of data centers and the flexibility of mobile networks by providing a standardized way to access compute devices, storage and other resources that are located at or physically very close to the base station for minimum access latency.[1]

For applications to be able to make use of both the local resources of the terminal device and, depending on their availability, the heavy-duty resources at the cloud edge a consistent application programming interface (API) capable of exposing a heterogeneous set of compute resources is needed. One such API is the Open Compute Language (OpenCL) which is an open specification maintained by the Khronos group. OpenCL is a fairly low level API that caters heavily to general purpose computing on graphics processing units (GPGPU) but has explicit support for various types of parallel processing devices from CPUs to Field Programmable Gate Arrays (FPGA) and custom accelerator devices. It focuses on using heterogeneous resources present within a single system and does not have any provisions for distributed computing as is needed for MEC applications.[2]

This work describes *pocl-r*, a framework for transparently distributing OpenCL-based compute tasks across a network. The base offloading driver for *pocl* was implemented by Michal Babej and optimized by the Author to minimize the size of commands as they are transferred across the network. Peer-to-peer communication was also added by the Author. Julius Ikkala implemented synthetic benchmarks for basic overhead testing and the Author implemented a distributed matrix multiplication test for throughput testing. Nokia Technologies provided a base for an AR point cloud rendering demo that was extended by Michal Babej for a case study of pocl-r. This master's thesis documents the whole entity, since the parts are too tightly interwoven to sensibly be examined separately.

A paper about the pocl-r implementation has also been submitted and is currently under review. The figures in that paper were made by the Author and are reused here with the exception of the AR use case figures, which were created by Michal Babej and are reused here with his kind permission.

Chapter 2 of this work explains the underlying parallel and distributed computing concepts. Chapter 3 introduces the OpenCL API and its associated OpenCL C language that is used to define compute tasks. Chapter 4 details the implementation of *pocl-r* and the various means used to optimize its throughput and command latency. In Chapter 5 a set of both synthetic and real-world benchmarks are presented and their results reviewed. Offloading computation with OpenCL or similar APIs has been researched before by multiple independent parties. Their work is briefly described in Chapter 6. This work and the evaluation results are summarized and discussed in the concluding Chapter 7.

2 PARALLEL AND DISTRIBUTED COMPUTING

As computational complexity of applications increases, one way for computer hardware to keep up is to increase the speed at which it can serially execute machine instructions. Another method is to enable the hardware to execute commands in parallel. On the hardware level this *parallel computing* can be divided in to *data parallelism* where the same machine instruction is executed on multiple data items at once and *instruction level parallelism* where multiple independent instructions of a program are executed in parallel e.g. by means of pipelining [3].

In software architecture design, parallelism is generally implemented with *threads* that act a bit like independent processes, but share a virtual address space. Parallel work is often organized as logical *tasks* that can be performed independently as soon as their dependencies are met.

2.1 High Level Concepts

Parallelizing work on the hardware level is one part of making use of the increased capabilities of denser silicon. On a higher level, in software design particularly the Multiple Instruction, Multiple Data (MIMD) model can be utilized to enable the hardware to perform multiple unrelated tasks when such *task parallelism* exists in the program. In this model, multiple independent instruction streams that operate on independent data streams are executed within the same program. The simplest form of this is concurrent operation where multiple tasks "in flight" at the same time e.g. interleaved on one processor using time slicing to make use of time that would otherwise spent waiting for example for a response to a network request. When multiple independent execution units are available it is also possible to perform tasks truly at the same time, in *parallel*. This gives rise to two further terms: the slightly misnamed *Single Program, Multiple Data (SPMD)* where multiple parallel instances of the same program perform computations independently of each other, usually on separate parts of the input. *Multiple Program, Multiple Data (MPMD)* on the other hand extends this to completely separate programs operating in parallel.

2.1.1 Task Parallelism

On a higher level there is a corresponding concept: *task parallelism*, which refers to performing multiple tasks simultaneously, usually in the form of separate *threads*. This introduces a requirement for synchronization when multiple threads access the same data and at least one of them tries to modify it. Failing to synchronize access can result in inconsistent results or data corruption. Synchronization is commonly handled with a *mutual exclusion (mutex)* mechanism which is implemented using *atomic* operations that prevent other execution threads from accessing the same data until the command has finished executing.

Tasks can be split up across separate processors executing independent copies of the same program on different parts of the input as described by the SPMD model. In this case memory access operations need to be kept in known order by using *memory barriers* also known as *fences* to avoid non-deterministic behaviour as the processors, while executing the same program, may progress through it with a different control flow due to different input data and may further independently reorder execution on the instruction level. Memory barriers provide a known point in the program that is not affected by instruction reordering and can be used to ensure all processors have reached that point before allowing any processor to proceed.

2.1.2 Heterogeneous and Homogeneous Computing

Parallel computing platforms can be further divided into *heterogeneous* and *homogeneous* computing depending on the hardware configuration. A setup where all usable compute devices are identical at least in terms of instruction set such as cores within a single CPU can be considered homogeneous. Heterogeneous computing provides an alternative [4]. It makes use of various different types of compute units in the same application. These range from general purpose CPUs to more specialized hardware like GPUs or digital signal processing units (DSP) used for audio and video processing and sometimes even to FPGAs or Application Specific Integrated Circuit (ASIC) units tailored for a specific application. A common use for heterogeneous computing is *offloading* of a specific task to hardware specifically designed for that task. Another recent trend is combining low power cores and high performance cores into a single multi-core CPU such as in ARM's big.LITTLE architecture. This is sometimes also called an asymmetric homogeneous configuration [5].

2.1.3 Distributed Computing

In addition to splitting work between autonomous processor cores, it is possible to divide programs into completely separate applications that run on completely independent

computers while collaborating on the same task. A popular software architecture for *distributed computing* like this is the *client-server* architecture. In this type of setup, *client* applications communicate with a central *server* across a network. The server can perform tasks on the clients' behalf or relay information to other clients. One form of delegating tasks to a server is the remote procedure call (RPC) model where the application is structured such as to delegate certain procedures to the server. This may or may not involve parallel operation as the client can simply wait for the server to respond. The client simply waiting for a response is sometimes good solution e.g. when it is running on a device with heavy constraints on power use, such as a mobile phone or when reacting immediately once a response is received is necessary. This is called *synchronous* or *blocking* operation. By contrast, *non-blocking* operation refers to when the application continues with other tasks instead of waiting, letting the operation to be performed *asynchronously*. Another use case for this type of architecture is when multiple instances of the client application have to access a limited shared resource or a resource that is not feasible to replicate on each instance of the client application, such as a large database. [6]

An alternative to the client-server architecture is the peer-to-peer (P2P) architecture. This is primarily distinguished by the lack of a central server that handles all of the clients' communication. Instead, clients are called *peers* and communicate directly with each other across a network [7]. Depending on the system and the peer application, the network technology can vary but some common examples include ethernet, InfiniBand [8] and the Controller Area Network (CAN) bus. Peers perform small parts of a larger task independently. This necessitates coordinating to ensure peers collectively agree on the state of the data and data dependencies between subtasks taken on by different peers are honoured.

Especially in data centers one popular means of implementing distributed applications is the Message Passing Interface (MPI) framework [9]. Another relatively recent method for transferring data between peers with minimal intervention from the program is Remote Direct Memory Access (RDMA) which allows copying data directly from an auxiliary device's memory without the CPU having to perform the copy one word at a time. For example large buffers computed on a GPU or custom accelerator device in its local memory can be accessed via Direct Memory Access (DMA) across the machine-internal interconnect, commonly the Peripheral Component Interconnect Express (PCIe) or Advanced eXtensible Interface (AXI) bus. This capability extended to network interfaces, allowing them to send data directly from the local memory of such an accelerator without the application having to copy it to the main memory first. Similarly the network interface on the receiving end can place the received data directly in the compute accelerator's local memory without involving the host CPU or the main memory. [10, 11, 12]

2.1.4 Multi-Access Edge Computing

Multi-Access Edge Computing (MEC), formerly known as Mobile Edge Computing, is a standardized framework for cloud-computing at the edge of a network, in other words compute resources stationed physically close to the mobile base station. The MEC standard is produced by an Industry Specification Group (ISG) of the same name managed by the European Telecommunications Standards Institute (ETSI). It aims to ensure efficient use of the network for a low-latency end user experience.

The standard defines a framework for optimizing applications running on mobile devices by utilizing stationary infrastructure located at or near network base stations. Being physically close to the base station, the network delay when accessing these resources remains minimal. The 5G standard promises greatly lowered round-trip times from end-user devices to the internet. This lends itself well to user-facing applications that require short latency in order to be usable interactively in real time. Moving heavy application logic to cloud servers is something that is already done frequently but current solutions are of limited use for real-time applications due to large latency and unpredictable network environments causing problematic amounts of jitter. MEC relies heavily on the improvements to both latency and jitter the 5G promises to ensure a smooth user experience.

The platform defines facilities for running arbitrary applications in an isolated environment. This enables application developers to easily update or replace the offloaded portion of their applications without requiring potentially expensive modifications to the base station hardware. Furthermore, it opens up possibilities for additional 3rd party developers to move their computations either away from the end-user's device to reduce battery use or from far away servers closer to the user to improve response times. [1]

After its original introduction the standard was renamed from Mobile Edge Computing to Multi-Access Edge Computing and generalized to networks other than 5G cellular. On a smaller scale, a similar environment and service architecture can be built with the WiFi6 standard, which promises similar improvements over its predecessor as 5G. This is useful for infrastructure deployments that are limited to one location such as a campus or a convention hall. [13]

2.2 Performance Metrics

Measuring performance is an important part of evaluating a computer system. This section describes two impotant metrics frequently used for that purpose: *throughput* and *latency* as well as some of their derivative metrics that are useful to commonplace applications.

The more well known metric is *throughput* i.e. the rate at which whole compute tasks can be completed is one of the most prominent measures of the performance of a parallel or distributed system. This is of course also a key performance indicator of non-parallel systems but given the additional complexity of a parallel or distributed system over a more traditional serial implementation, the increased rate at which tasks can be completed is an important factor for evaluating that tradeoff. Throughput is a popular metric for performance due to its simple and intuitive nature and for many applications it is informative enough.

A very interesting derivative from throughput is *scalability* of applications, which reflects the increase in throughput as the total computing power of the system is increased. Computing power can be increased by increasing the *clock speed* of the compute units or by reducing the number of clock ticks needed to complete any given instruction. The average number of machine instructions that can be executed during one clock tick is called the *instructions per clock* metric. This is improved by making use of *instruction level parallelism (ILP)*, where consecutive instructions don't have direct data dependencies on each other. This allows for techniques such as pipelining, where multiple machine instructions are being processed in parallel. For example data for the operands of the next instruction can often be fetched while the previous instruction is still executing. If it can be shown that the order of operations does not affect the end result, the separate machine instructions may be reordered by the compiler or the processor for *out-of-order execution* to reduce the slack time between instructions. Some processor architectures also support executing multiple independent operations simultaneously by having multiple independent ALUs and FPUs.

Linear *scaling* is achieved when the number of tasks that can be completed in a given time frame increases proportionally with the increase in execution speed. On the level of a single task this is only practically possible by increasing the number of instructions a single compute unit (unit capable of processing an independent instruction stream) can perform per unit of time: when splitting computations among multiple units, it can trivially be seen that the amount of work that is left for one compute unit to perform is inversely proportional to the number of compute units available. Assuming that each compute unit operates at a fixed speed, the completion of each unit's partial tasks is directly proportional to the size of the task and thus the completion time of the whole task is also inversely proportional to the number of compute units. Amdahl's law provides an approximate formula for this and further accounts for the fact that virtually all programs have parts that are not possible to parallelize at all [14].

Gustafson however notes that as more processing power becomes available, the computational size of tasks also tends to be increased to achieve better accuracy or open up new application possibilities within the same total execution time of the application. The serial portion of various physical simulation workloads is found to remain largely

unaffected by the total task size. Additionally doubling the amount of parallelizable parts in the problem while also doubling the number of available compute units yielded no meaningful difference in the overall run time of the simulation. From this Gustafson concludes that relying solely on Amdahl's law and assuming a fixed application complexity is a flawed approach to evaluating the *performance cost* of a parallel system over a serial one. [15]

In addition to throughput, *latency*, is an important performance metric. It is a measure of how quickly the system can produce a corresponding output when given a new input and depending on the application it can be even more important than raw throughput.

Complex applications are increasingly being used in *real-time* systems, where the results of computation have to be available before a specific deadline. These systems are divided into *hard*, *firm* and *soft* real-time systems based on how badly the system is impacted if the deadline of a task is missed. In hard real-time systems missing the deadline results in total system failure whereas in soft real-time systems a result obtained past its deadline merely reduces the value of the result, usually degrading the system's quality of service by a non-fatal amount. Firm real-time systems lie between these two types: when a task in a firm real-time system exceeds its deadline, the result becomes unusable and quality of service degrades. However occasional failures to meet the deadline in a firm real-time system is tolerable and does not cause total system failure. [16]

In hard real-time systems there is naturally a limit on acceptable latency. Once this required latency is consistently reached there is little incentive to optimize the system further. Soft real-time systems however are often user-facing and while they have a desired maximum latency, the quality of service usually improves as the achieved latency is reduced. Thus latency is an important metric for real-time systems in addition to maximum throughput.

It is common for latency in a computer application to vary between inputs. This variance, called *jitter* can have many sources such as varying network conditions and interference from other applications running on the same computer. Jitter can have an even greater impact on the quality of service than the total amount of latency: a stable amount of latency can often be accounted for even if the total delay is large. On the other hand it can easily become impossible to compensate for even small amounts of latency if it varies a lot between inputs. For example many control automation applications rely on smoothing noisy input data to produce a new control value. The control value often needs to be adjusted depending on how long it takes to generate and how quickly the system can react to new control values. If the computation time varies a lot this adjustment becomes imprecise and behaviour of the system quickly becomes unreliable.

2.3 Parallelism in Hardware Architectures

According to Moore's law, the number of transistors that can be fit onto a single chip doubles roughly every two years [17]. This has held true for a long time and provided ample opportunity for developing processors that can perform multiple operations in parallel. As increases in clock speed have slowed down, parallelization of workloads has become an increasingly important method for improving overall performance. Based on whether or how this is is done, hardware architectures can be classified roughly into four types using Flynn's taxonomy [18].

Single Instruction, Single Data (SISD)

The traditional non-parallelized model is called Single Instruction, Single Data (SISD). In this model the program consists of only one command stream and only handles one data item at once. This is easy to design both hardware and software for.

However performance of a SISD system can only be improved by developing faster algorithms and improving the clock speed of the processing unit, both of which become very costly if not outright impossible due to physical or mathematical constraints. As such the *scalability* of such a system to greater workloads is limited.

Single Instruction, Multiple Data (SIMD)

A relatively simple extension of the SISD model is adding support for performing the same command on multiple data units simultaneously, e.g. by having multiple ALU that are run in lockstep and are controlled by the same command stream, while receiving different data inputs. This model, called Single Instruction, Multiple Data (SIMD) is extensively used in digital signal processors (DSP) as they continuously perform the same sequence of operations on large amounts of data. It has also found its way into consumer grade general-purpose CPUs and mobile systems on chip (SoC).

When such *data parallelism* is allowed by the program logic, processing can be done with special *vector operations* which operate on vectors of data instead of single scalar values. Vector and matrix arithmetic are commonly optimized using this type of architecture. Scaling an architecture of this type to handle even bigger amounts of data in parallel i.e. operating on wider vectors is fairly simple in theory but causes die space and energy use costs to grow at a problematic rate. Furthermore, as the width of the usable vectors grows, use cases that get the full benefit from them become fewer and fewer, leaving the vector engine underutilized or completely unused for a longer portion of time.

Multiple Instruction, Single Data (MISD)

The reverse of SIMD is also possible, i.e. multiple parallel ALUs receiving the same input data but separate command streams in a Multiple Instruction, Single Data (MISD) configuration. This is mainly useful for redundancy in applications where the computing units are expected to malfunction frequently and such malfunction has especially critical consequences. For example the space shuttle control computer system operated in a mode like this the during the ascent and re-entry phases of the shuttle's flights [19].

Multiple Instruction, Multiple Data (MIMD)

MIMD is combination of the previous configurations where multiple units receive both independent command streams and separate data streams. This is commonly seen in the form of multicore processors and multithreaded programs, but *distributed* computing where a program is ran collaboratively on multiple independent computers also falls into this category.

3 OPEN COMPUTING LANGUAGE

Open Computing Language (OpenCL) is an open specification of a vendor-neutral framework for performing heavily parallel computing tasks on heterogeneous systems by dispatching independent parallel tasks from a host application. The basic idea is illustrated in Figure 3.1. OpenCL supports dispatching tasks to CPUs and GPUs as well as custom hardware such as FPGAs or ASICs. The specification consists of two main parts: a variant of the C language for defining the tasks to be performed on heterogeneous hardware and a C API for managing memory allocations and dispatching commands to the available compute devices. The specification does not take into consideration whether or not these devices are distributed across multiple systems or part of the same system. [2, 20]

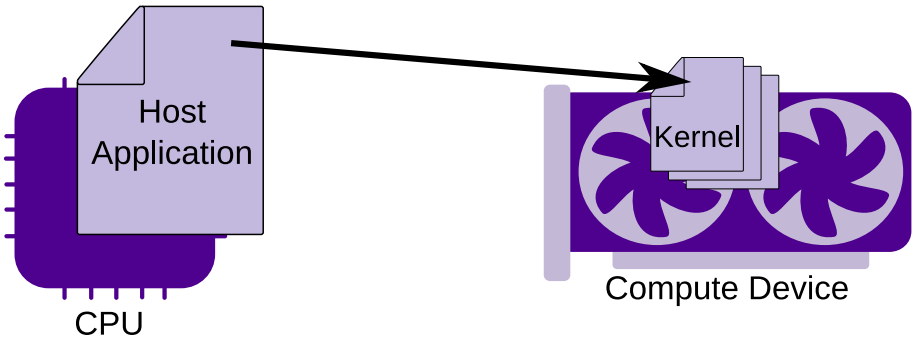

Figure 3.1. *A basic overview of the architectural concept of the OpenCL API. A host program running on the main CPU dispatches parallel workloads structured as kernels to heterogeneous compute devices.*

3.1 Runtime Environment

The OpenCL runtime is an implementation of the functions of the functionality outlined in the API specification. It consists of one or multiple Installable Client Drivers (ICD) that contain the individual implementations and a library loads those drivers and acts as a central location for accessing all OpenCL implementations available on the system. This *dispatcher* lists the available platforms and provides stubs for all API endpoints. These stubs then route the API calls to the correct function implementation in the ICD that corresponds to the platform that is referenced by the function parameters. The ICD then performs the necessary kernel compilation, bookkeeping and dispatches commands

to the actual hardware. [2]

All devices of the same platform can be requested to be available within an OpenCL *context* which represents a run time instance of the API implementation. Devices from different platforms can not be used together directly as their respective drivers have no knowledge of each other. On the other hand devices that are accessible within the same context can collaborate within the restrictions defined by event dependencies. This leaves ample room for the driver to optimize execution of parts of the whole application as explored by Korhonen et al. [21]

3.2 OpenCL API

The OpenCL application programming interface (API) consists of a *platform* layer and a *runtime* layer. The platform layer provides facilities for enumerating installed OpenCL implementations and creating a *context* for the desired platform. The runtime layer provides interfaces for allocating memory, creating and executing programs and handling data and control flow between tasks.

The API of the runtime layer is built around a model of *command queues* to which the host application submits *commands*. The runtime returns *events* that can be used to query the current state of their associated command as well as for declaring dependencies between commands. The runtime then asynchronously executes the commands. This can happen *in-order* or *out-of-order* depending on the way the queue was configured upon its creation. Commands submitted to an *in-order* queue appear to the application to be executed in the order in which they were submitted, but as long as this appearance is kept, the runtime is allowed to reorder commands if it sees fit. Conversely, commands submitted to an *out-of-order* queue may be executed in any order. The only way for the application to affect execution order of commands in an *out-of-order* queue is by using *event* dependencies or *command queue barriers*.

Every command generates an event handle that the application can store and use to define a list of dependencies for future commands. Defining dependencies this way allows the OpenCL runtime to schedule commands whatever way is best suited to the underlying hardware as long as the ordering guarantees of the command queue and the dependency chain as defined by events is kept intact.

Memory on the compute devices is allocated and freed by the host application in the form of *buffers*. One major limitation of buffers is that once allocated, their size can not be changed. The only way to resize a buffer is to create a new one of the desired size, copy the data to it and free the old buffer. For buffers that contain pixel data it is possible to define an image view overlay to make access to the data in a structured way more ergonomic and to allow the OpenCL implementation to make use of texture

sampling hardware to accelerate access and interpolation for example on GPUs, which commonly have texture units that provide these features.

Memory can be allocated from various *memory regions* depending on where the application needs to access it from. For example buffers that will be read directly by the host application are normally allocated from the main memory of the system or a memory region that can be mapped into the host application's virtual memory address space. On the other hand a buffer that is only accessed from within an offloaded compute task can be allocated from device-local memory that is much faster to access from tasks running on the device but may not be accessible to the host application at all. The various memory regions, shown in Figure 3.2, have their own *address spaces*. Pointers to data in any given address space are handled as relative offsets from that address space and are not valid in other address spaces. For example a pointer defined in the host application can not be used in a compute task that does not have access to the host memory region. For simplicity, the OpenCL specification combines the address spaces of private (to a single work item), local (to one work group) and global (shared among all work items on all devices in the system but not with the host application) memory regions into one *generic address space*. Shared Virtual Memory (SVM) is an optional feature in OpenCL 3.0 that adds a new address space that is shared by all devices and the host application.

The actual compute tasks are defined as *programs*, which correspond to a single binary that is compiled at the host application's runtime. The source for these programs can be supplied either as plaintext using a custom C-like language or from SPIR-V, a standard portable intermediate representation for machine code. SPIR-V allows developers to write OpenCL programs in any language and compile them to the binary intermediate format ahead of time. [22]

The *programs* have one or more entry points, called *kernels*, whose execution can be enqueued on the command queue of a supported device. Kernels can take any number of arguments that can be constants or pointers to memory buffers. These arguments have to be specified for each kernel before enqueueing the kernel's execution, but are stored for each given kernel so unchanged arguments do not need to be respecified for consecutive commands using the same kernel.

Kernels are launched in groups called *work groups*. One invocation of a kernel inside this group is called a *work item*. Sometimes multiple work items in the same work group need access to the same elements of a memory buffer and some of these accesses are mutable, necessitating some form of synchronization. For such situations OpenCL provides a fence function that can be called from within kernels. Upon reaching this function, the kernel will wait until all other work items in the same work group have done so, ensuring that there are no writes to or reads from that location in flight that could cause inconsistent results if not synchronized properly. For similar synchronization between separate tasks,

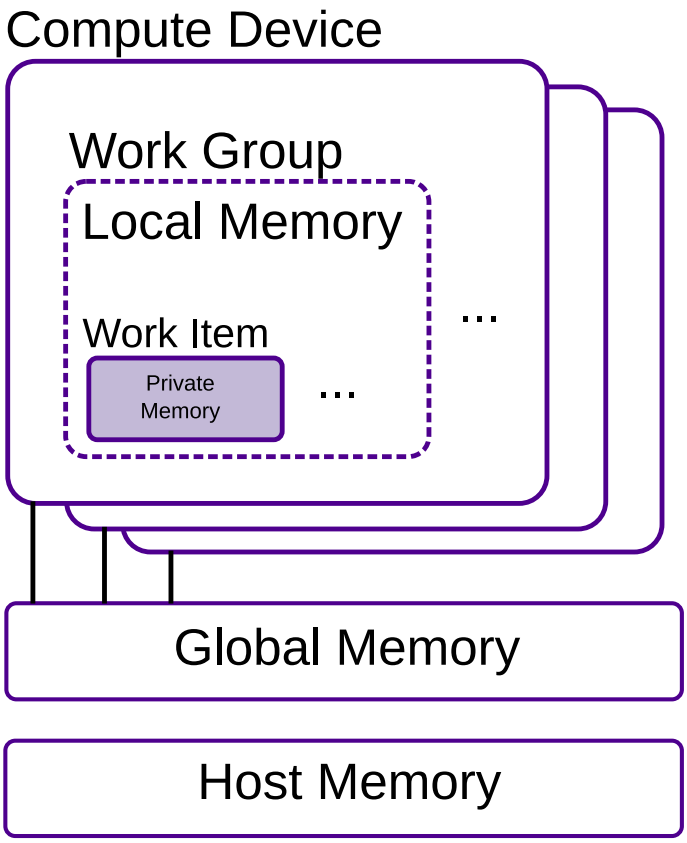

Figure 3.2. *An overview of the memory regions defined by the OpenCL specification. The global memory is shared among all compute devices. It also has a section for constant data that can't be changed during the run time of a kernel.*

a special barrier task type is provided that waits until all previously enqueued tasks or tasks listed as event dependencies are finished.

Figures 3.3 and 3.4 demonstrate the use of the OpenCL API in a basic application that creates a pair of input buffers and a result buffer, executes a kernel to compute the results and copies the results back to the main memory. Events are used to indicate that copying the results back should wait for the kernel execution to finish. [2]

3.3 OpenCL C Language

OpenCL C is a dialect of the C language used for defining programs that can be executed on compute devices exposed by OpenCL [20]. The most notable difference with commonplace variants of C are the existence of builtin functions and types specific to OpenCL, such as image reading and writing functions and associated image data types for hardware-accelerated image sampling. Another visible difference are the added attributes on variable and function declarations that specify the role of a function or the memory space that a variable is associated with.

Figure 3.5 shows a simple example kernel that performs element-wise addition of two

```c
#include <CL/cl.h>

#define ITEMS 512
const char* prog_src /*value omitted*/;
const float a[ITEMS], b[ITEMS] /*values omitted*/;

int main(int argc, char **argv)
{
    // Obtain a handle to the first available platform
    cl_platform_id platform;
    clGetPlatformIDs(1, &platform, NULL);

    // Obtain a handle for the first available CPU or GPU device in the platform
    cl_device_id dev;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU | CL_DEVICE_TYPE_CPU,
        1, &dev, NULL);

    // Create a *context*
    cl_context ctx = clCreateContext(NULL, 1, &device, NULL, NULL,
        NULL);

    // Create a command queue
    cl_command_queue q = clCreateCommandQueue(ctx, device, 0, NULL);

    // Create our example OpenCL C program from an array of source strings
    cl_program prog = clCreateProgramWithSource(ctx, 1, &prog_src,
        NULL, NULL);

    // Compile the program to executable form for the one device
    // for which a handle was obtained earlier
    clBuildProgram(prog, 1, &dev, NULL, NULL, NULL);

    // Obtain a handle to the kernel defined within the program
    cl_kernel kern = clCreateKernel(prog, "perform_add", NULL);


    // Create buffers for input data
    cl_mem in_a = clCreateBuffer(ctx,
        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
        ITEMS*sizeof(float), a, NULL);
    cl_mem in_b = clCreateBuffer(ctx,
        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
        ITEMS*sizeof(float), b, NULL);

    // Create output buffer
    cl_mem output = clCreateBuffer(ctx, CL_MEM_WRITE_ONLY,
        ITEMS*sizeof(float), NULL, NULL);

    /* continued... */
```

Figure 3.3. *Example of a C application using the OpenCL API to sum two arrays of floating point numbers in parallel*

```
    /* ...continued */

    // Assign buffers to kernel arguments by argument index
    clSetKernelArg(kern, 0, sizeof(in_a), (void*) &in_a);
    clSetKernelArg(kern, 1, sizeof(in_b), (void*) &in_b);
    clSetKernelArg(kern, 2, sizeof(in_c), (void*) &in_c);

    // Dispatch kernel invocations for each data item to the command queue
    // Grab an associated event handle
    size_t global_work_items = ITEMS;
    cl_event ev;
    clEnqueueNDRangeKernel(q, kern, 1, NULL, &global_work_items, NULL,
        0, NULL, &ev);

    // Read back results once kernels are done running, use event to synchronize
    // Tell clEnqueueReadBuffers to block execution of this program
    // until it has finished copying the entire buffer to 'results'
    float[ITEMS] results;
    cl_bool blocking = CL_TRUE;
    clEnqueueReadBuffers(q, output, blocking, 0, ITEMS*sizeof(float),
        results, 1, &ev, NULL);

    // Do something with the results (omitted)
}
```

Figure 3.4. *Example of a C application using the OpenCL API to sum two arrays of floating point numbers in parallel (cont'd)*

```
__kernel void perform_add(__global const float* const a, __global const
    float const *b, __global float *c)
{
    // obtain global ID for this invocation
    size_t idx = get_global_id(0);

    // use invocation ID to choose data to operate on
    c[idx] = a[idx] + b[idx];
}
```

Figure 3.5. *Example of an OpenCL kernel defined using OpenCL C.*

arrays of floating point buffers, storing the result in a 3rd array. Note the added *kernel* attribute that indicates that a function is an entry point of the program. One program may define multiple kernels. Another added attribute is *global*, which indicates to the compiler what kind of memory the data being pointed at is expected to reside in. In this example all pointers refer to memory buffers, thus the "global" memory designation. Kernels are dispatched in *work groups* that are defined as a dense one- to three-dimensional grid. The position of the current invocation, called *work item*, on each axis of this grid can be queried with the *get_global_id* command.

Furthermore, the specification defines an extension mechanism whereby vendor or implementation specific functionality can be added before it is considered for inclusion into

the main specification. Examples of such extension functionality range from data sharing with the Direct3D and OpenGL APIs and native support for planar color formats in image views to exposing hardware features such as full-width data types and providing more fine-grained control over kernel execution. [23][24][25]

4 POCL-REMOTE

The basis of this master's thesis is Portable Computing Language (pocl), a portable open source implementation of the OpenCL API with flexible support for custom device backends [26]. OpenCL itself does not provide any kind of functionality for *distributed* computing, its main focus being offloading massively parallel computation to dedicated hardware within the same system. This master's thesis aims to change that by introducing *pocl-r*, a new driver for pocl that exposes devices from other machines across a network. These appear in the same pocl-provided OpenCL context as if they were local devices, allowing for distributed offloading of compute tasks with minimal changes to application code and no dependency on 3rd party communication frameworks. This makes it an easy solution for e.g. adapting applications to MEC use cases.

The focus in *pocl-r* is on minimizing the end-to-end latency to enable a responsive high quality user experience in real-time edge cloud applications as well as enable scalable use of diverse compute resources in the cluster. A simplified view of a typical use case is shown in Fig. 4.1 where a mobile device running the main application offloads heavy computation across a low-latency wireless connection to a cluster of GPUs and potentially other accelerators, in this case FPGA devices.

4.1 Architecture

The *pocl-r* runtime is implemented as a standard client-server architecture. This is done for implementation simplicity compared to a peer-to-peer architecture and because it maps well to the OpenCL API design that is centered around the host application (client) submitting commands to a command queue that is consumed by the compute device (server). The *client* is implemented as a special *remote driver* of *pocl*. The remote driver acts as a "smart proxy" that exposes compute devices on a remote node through the OpenCL platform API the same way as local devices. It then receives the commands targeted at those remote devices and sends them to the daemon running on the corresponding remote node. This makes the use of remote devices appear to work identically to using local devices as far as application logic is concerned. The software stack is illustrated in Fig 4.2.

The server side is a daemon that runs on the remote nodes and receives commands

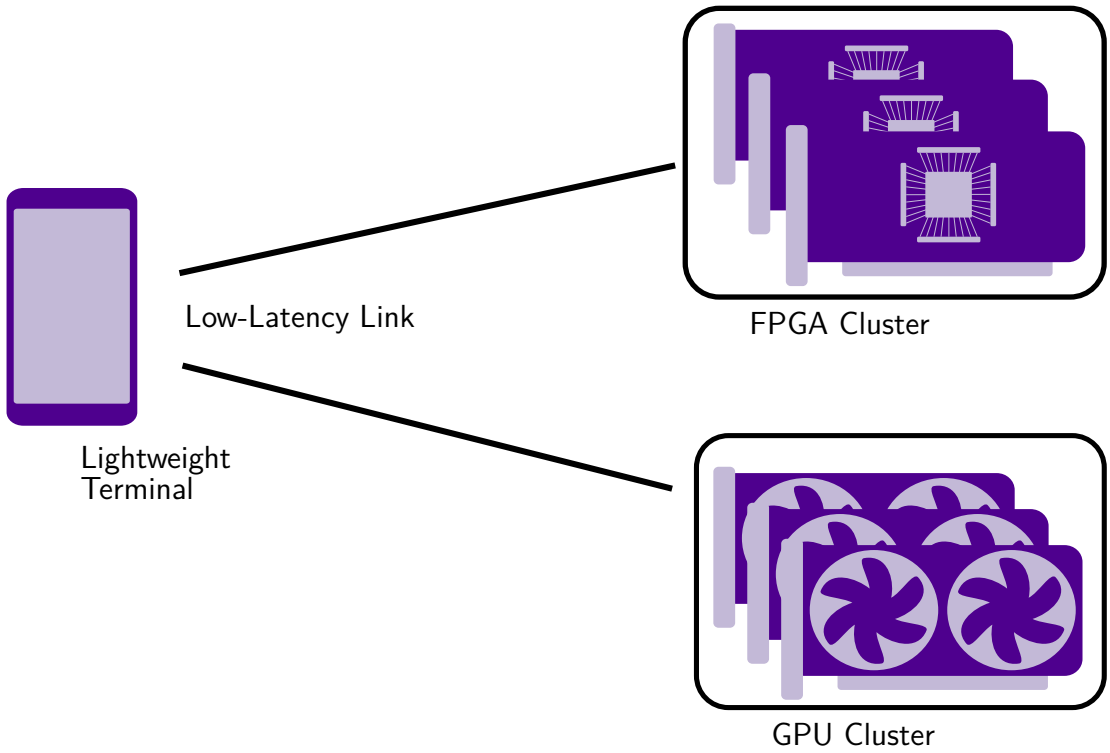

Figure 4.1. *High-level overview of a diverse distributed execution use case where multiple devices with varying types are utilized remotely by a lightweight terminal device.*

from the remote driver, and dispatches them to the OpenCL driver of the node's devices accompanied with proper event dependencies. The OpenCL devices can be controlled via a device-specific proprietary OpenCL driver by the daemon, or through, e.g., the open source drivers provided by *pocl*.

The daemon is structured around network sockets for the client and peer connections. Each socket has a reader thread and a writer thread. The readers do blocking reads on the socket until they manage to read a new command, which they then dispatch to the underlying OpenCL runtime, store its associated OpenCL event in a queue and signal the corresponding writer thread. The host writer thread iterates through events in the queue and when it finds one that the underlying OpenCL runtime reports as complete, writes its result to the socket representing the host connection. Peer writers have separate queues, but are otherwise similar to the host writer. The host writer adds events that peers need to be notified of to these queues and signals the peer writers. Fig. 4.3 illustrates this architecture and the flow of commands and data through it.

4.2 Latency and Scalability Optimizations

The following subsections describe the essential latency and scalability optimization techniques of *pocl-r*. A MEC usecase frequently has better connections between remote nodes than between a remote node and the client device. In this light reducing the need for

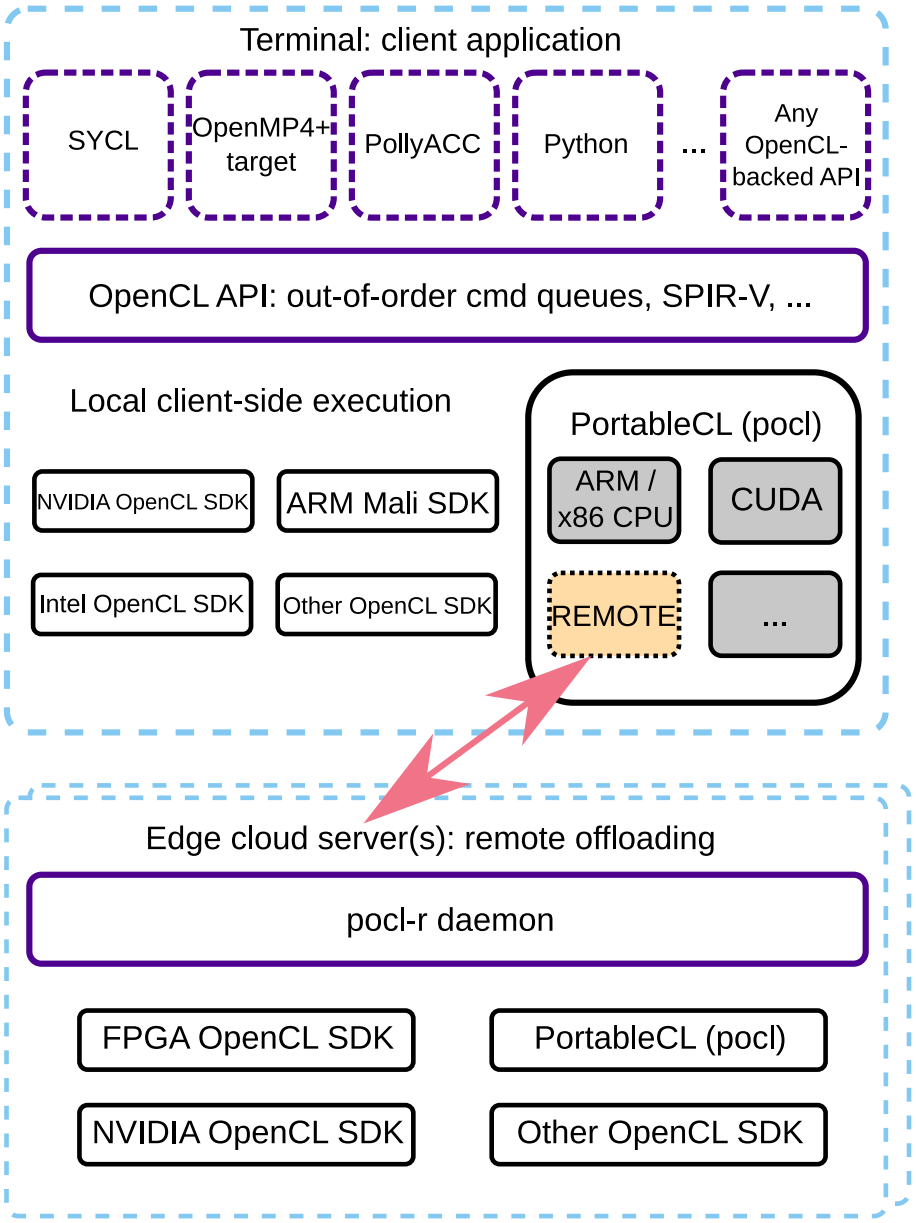

Figure 4.2. *Overview of the software stack for an application using pocl-r. The OpenCL API can be used directly for maximum efficiency, but also as a middleware for improved productivity APIs on top of it. In this stack, pocl is an OpenCL API implementation, a drop-in alternative to the other OpenCL implementations, with the special remote driver interfacing to the remote OpenCL-supported devices with distributed communication.*

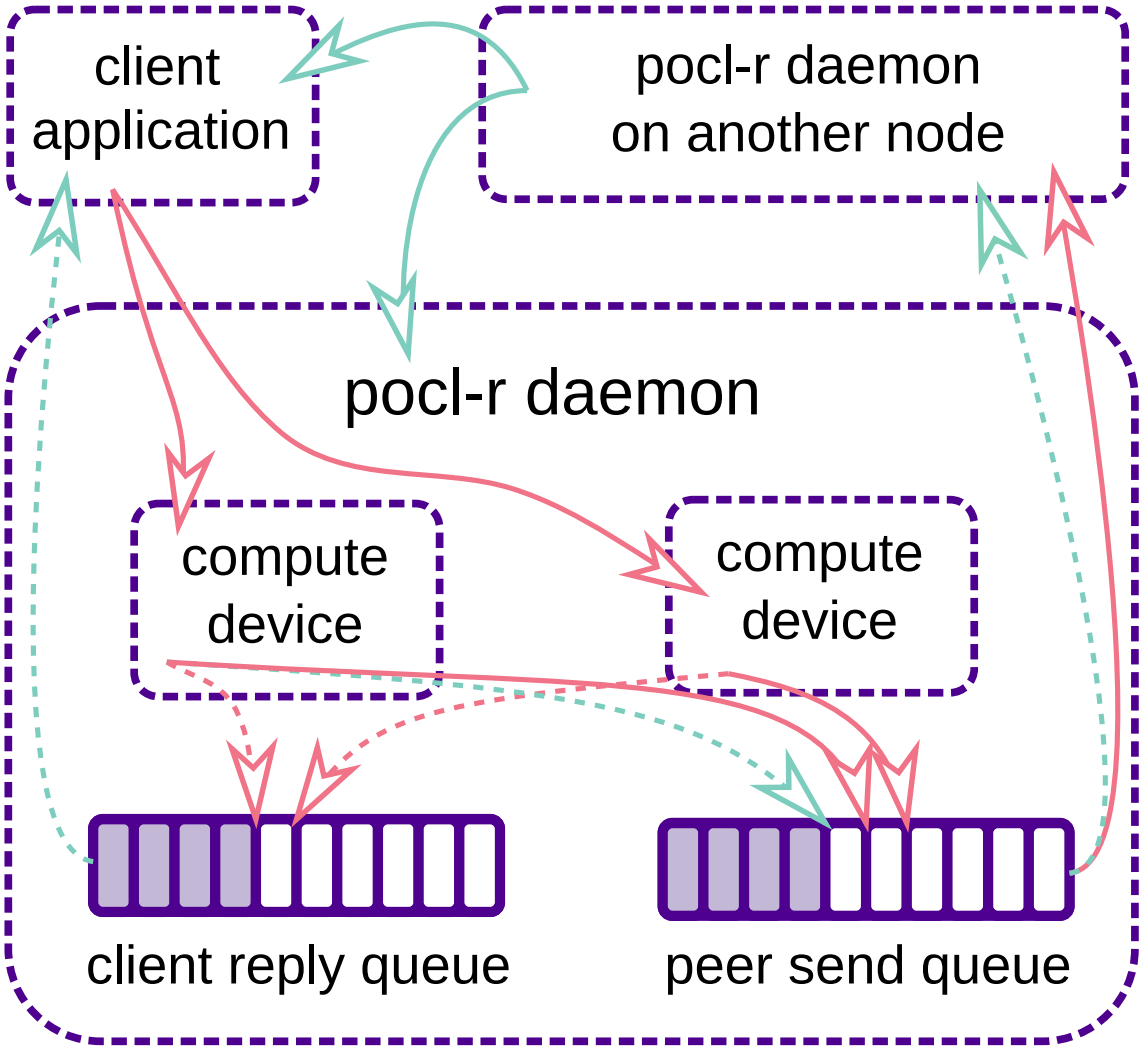

Figure 4.3. *The flow of data and commands from an application to the pocl-r daemon and between remote nodes. Red arrows represent OpenCL commands and green arrows indicate command completion notifications. Dashed arrows represent the control flow for a different command whose execution does not require migration to another node but whose completion is still signaled to other nodes.*

communication with the client is a top priority for improving throughput, latency and power consumption on the client device. One of the most important methods to do this is removing the client from the flow of data from a source and through the remote nodes where this is possible.

4.2.1 Peer-to-Peer Communication

As one of the improvements implemented by the Author as part of this master's thesis, *pocl-r* supports transferring buffers directly between devices on the same remote node (provided that the node's OpenCL implementation supports it), Peer-to-Peer (P2P) transfers of buffers between nodes, as well as distributed event signaling.

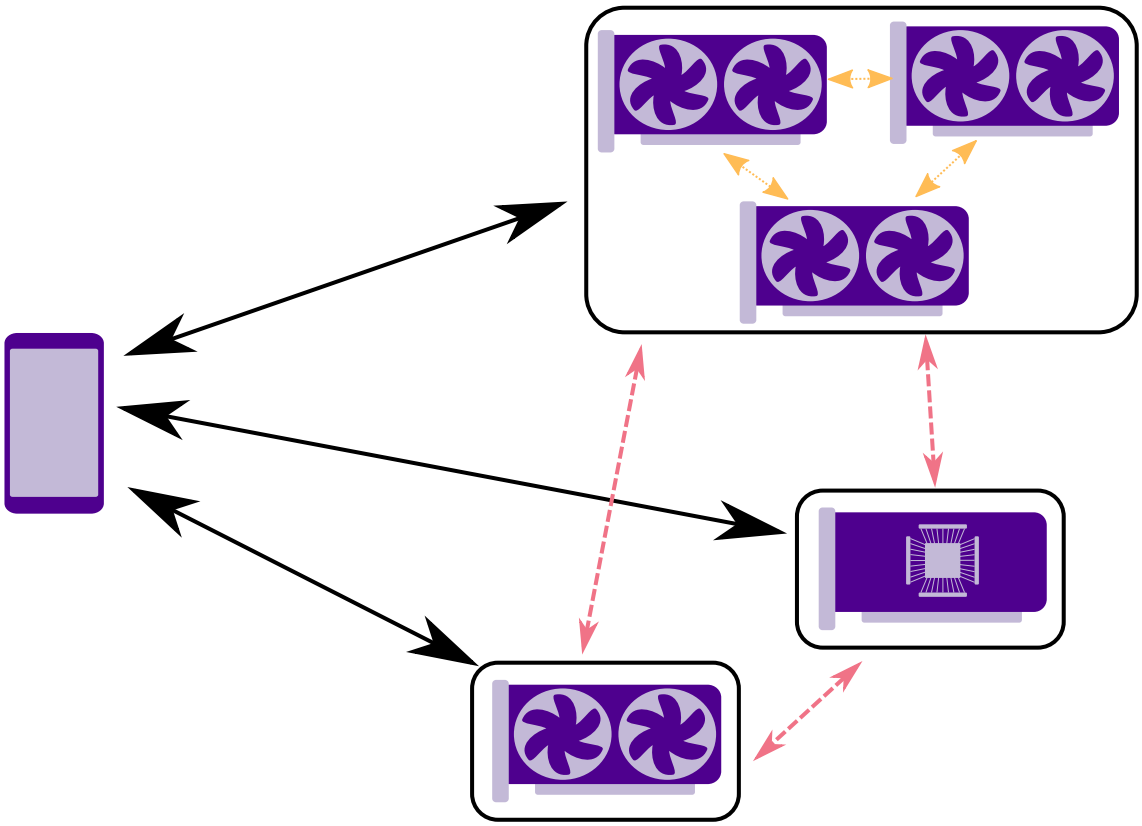

Figure 4.4. *Pocl-r with P2P network connections between remote nodes highlighted in red and direct transfers between devices within a node highlighted in yellow. The black arrows denote transfers between the client device and the remote nodes. The black groups indicate physical servers that are connected by some type of network. They may or may not be physically located in the same data center.*

Fig. 4.4 illustrates the various possible links between the host application running in the client device that communicates with remote nodes and devices. In a typical edge cloud use case, the client connection to the remote servers is much slower than the interconnect between nodes in the edge cloud cluster, thus the bandwidth savings versus transferring data always to the client application and back to another remote device can affect the overall performance dramatically. In addition, the number of network requests from the client are reduced drastically, since the host application only needs to send the migration command to the source node rather than requesting the data from the source node, awaiting a response and sending another request to the destination node. The faster interconnect between nodes may not be addressable directly from the network that the client application is running in, so the runtime supports specifying a separate IP address for nodes to use for their P2P connections.

4.2.2 Distributed Data Sourcing

When working with data that is not originally sourced from the client device, it would normally have to be transferred to the client first, and then distributed to compute devices

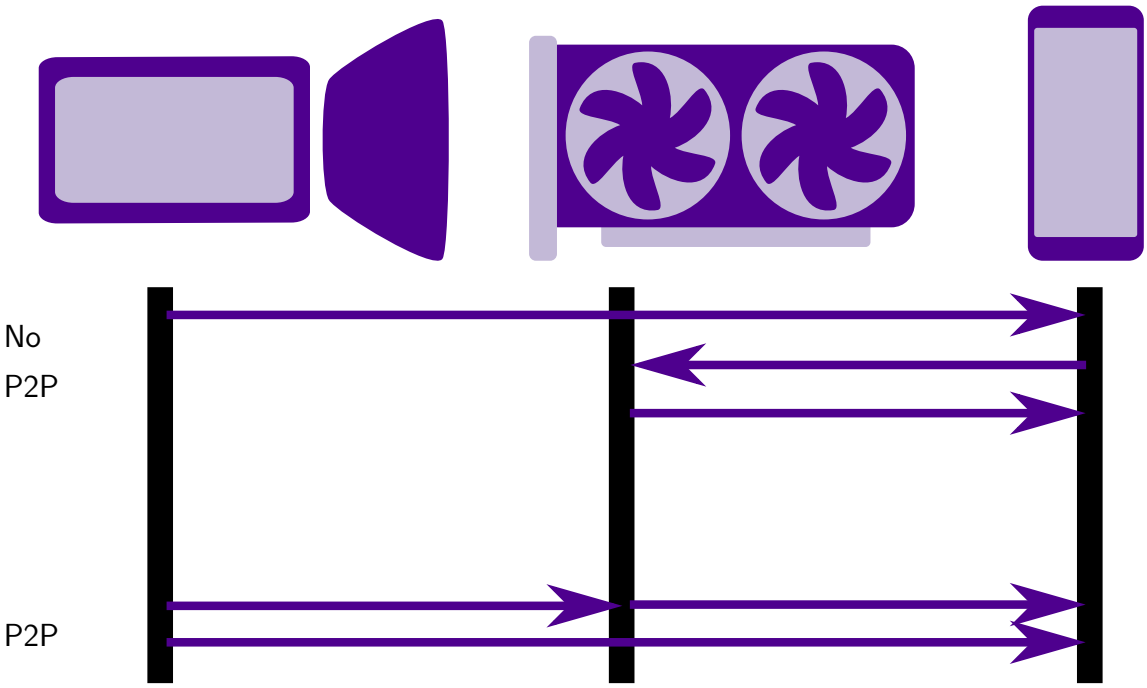

Figure 4.5. *Transferring intermediate results directly to the next device simplifies control flow and reduces the number of network round trips to the client that are needed to get the final result.*

from there. With OpenCL's custom devices feature it is possible to wrap arbitrary data sources to appear as devices in the OpenCL platform. Such devices can then utilize the P2P buffer migration functionality to transfer input data directly to the compute device that needs it, rather than making a round trip through the host application. Even in cases where both the compute device and the client application need access to the same data, the process can be streamlined as the compute device can produce its results simultaneously while the input data is still being sent to the host application, leading to the processed results being available much earlier than if the input had to be sent to the host in its entirety before computation could even begin.

Fig. 4.5 illustrates the difference between routing input data from a producer device through the host application and sending it directly to the compute device that needs it. In case the client application also needs the raw input data, some extra bandwidth use is naturally incurred. This can be mitigated by compressing the data in flight, at the cost of some latency and throughput overhead. With modern compression algorithms the bandwidth savings can easily outweigh the compression and decompression overhead, especially where lossy compression can be used, such as with video streams. Some data sources are also only available in compressed form to begin with, so transferring their data as is and decompressing it in parallel on any devices that need it is an obvious way to minimize latency as well as overall run time. One such remote data source was implemented by Michal Babej for the case study examined in this master's thesis.

4.2.3 Low-Overhead Communication

The base of the client-server communication is a pair of raw TCP sockets. One socket is dedicated to commands and the other to buffer data transfers, their send and receive buffer sizes tuned for their respective purposes. To minimize latency on the network level, TCP fast retransmission is enabled for both sockets.

While optimization of serialization protocols has been researched a lot and some extremely low-overhead protocols such as *FlatBuffers* [27] and *MessagePack* [28] have emerged, using a separate wire format for communication still adds overhead both on the sending and receiving side. *Pocl-r* uses the in-memory representation of commands as its wire format, avoiding this. Figure 4.6 shows a simplified definition of the command structure used. The in-memory layout of the command structure is kept consistent across platforms and compilers by forcing the fields of the structure to be aligned to addresses that are a multiple of eight bytes as shown, which is large enough to accommodate any of the used types on any tested platform.

As an optimization done by the Author as part of this master's thesis, commands are prefixed by an integer value indicating the actual meaningful size of the command structure. This is done based on the notion that there is only a small number of commands generated from OpenCL API calls that require a massive amount of data, while most commands can fit all information they need in a few dozen bytes. If the command has events in its wait list, their IDs are sent immediately after the meaning part of the command structure as an array of integers. Some commands require variable amounts of additional data. Similarly to event dependencies, this is indicated with the extra_size and extra_size2 fields and the data is sent as plain byte arrays.

The trade-off of this approach is that all remote nodes as well as the client device running the host application need to have the same integer byte order. In practice we consider this not a noticeable limitation after successfully testing *pocl-r* across a range of devices, from commodity mobile SoCs to PC and server room hardware. Should mixed endianness become a problem in the future, adding byte swaps for commands' data fields to the runtime is fairly straightforward and likely has negligible impact on performance. Additionally the swap can be eliminated entirely at compile time if the target machine's endianness is already the same as the host application's. A bigger hurdle is the OpenCL C application code itself, as OpenCL has no knowledge about buffer contents' endianness and makes mixed endianness related swapping the application writer's responsibility [2]: Applications meant to work on platforms with mixed endianness need their kernels to be adapted to account for the difference and swap the byte order of multi-byte values stored in OpenCL buffers when crossing devices with different byte orders.

```
typedef struct __attribute__((packed, aligned(8))) RequestMsg_s {
  uint64_t msg_id;
  uint32_t pid;
  uint32_t did;
  uint32_t client_did;
  uint32_t waitlist_size;
  uint32_t extra_size;
  uint32_t extra_size2;

  uint32_t message_type;
  uint32_t obj_id;
  uint32_t cq_id;


  union {
    /* Some variants omitted for brevity */
    CreateKernelMsg_t create_kernel;
    FreeKernelMsg_t free_kernel;
    RunKernelMsg_t run_kernel;
  } m;
} RequestMsg_t;

// Actual waitlist
uint64_t waitlist;

// Command-dependent additional data
char[] extra_data;
char[] extra_data2;
```

Figure 4.6. *Simplified definition of the message type used for communication between the application and remote nodes.*

4.2.4 Decentralized Command Scheduling

Commands are pushed to the remote nodes immediately when OpenCL enqueue API calls are made on the client. Event dependencies are mapped to platform-local events on each node and events for commands running on other nodes are substituted with user events. This way the heterogeneous task graph based on event dependencies defined by the application stays intact on the remote nodes and the runtime can apply optimisations utilizing the dependency rules as outlined in [29].

In addition to the control and data connections to the client the Author added direct connections between separate remote nodes. These are used for peer-to-peer buffer migrations and to signal event completions to other nodes for use in command scheduling as illustrated in Fig. 4.3. Thanks to this setup, enqueuing a command that depends on a buffer produced by a command on a different device only requires two network requests from the host application: One request is needed to initiate the transfer of the relevant buffer from the source node and another request to enqueue the dependent command on the destination device. The destination node generates a placeholder event and marks it

as a dependency of this command. Once the buffer is received from the source node, the *pocl-r* daemon running on the destination node signals the event and the node's native OpenCL runtime can begin executing the command immediately.

4.2.5 Dynamic Buffer Size Extension

OpenCL allows applications to allocate memory in the form of buffers, whose size is fixed once they are created. For some applications the amount of data required varies wildly over time and fixed-size buffers have to be allocated according to the worst-case scenario. This often leads to large amounts of wasted space that is allocated only to meet the worst case requirements but in practice almost never gets used. As an optional means to improve performance when dealing with data of varying and unpredictable size, a minimal OpenCL extension named *cl_pocl_content_size* was drafted and implemented by Michal Babej. The extension provides a way to signal actual used portion of an OpenCL buffer. This works by providing a separate buffer, large enough to hold a single unsigned integer, that holds the number of bytes actually being used counting from the start of an associated content buffer. *Pocl-r* runtime uses this as an hint to only transfer the meaningful portion of buffers when migrating them between remote nodes.

An example use of the extension is shown in the code snippet of Fig. 4.7. The only addition to the standard OpenCL API calls is the `clSetContentSizeBufferPOCL` call to associate a content size buffer with a data buffer, and the addition of this "size buffer" to the kernels' arguments.

The extension is completely optional and does not alter the way the runtime behaves for applications that are without it in mind. However, when combined with compression algorithms that can squeeze the data to a small fraction of its original size with a dynamic ratio, the new functionality enabled by the extension can provide drastic savings in bandwidth when migrating data buffers between the remote nodes, between devices in the same node, and to and from the client device.

An obvious alternative approach to specifying used buffer capacity in a dynamic way would be to use the first word of the actual data buffer to hold the size. This poses its own problems as applications and hardware often have requirements regarding the memory addresses of data. Usually there is a requirement for the starting address of the data to be a multiple of some number of bytes. Adding an extra field to the beginning of the data buffer would offset the start of the actual data, sometimes by an amount that breaks this requirement. This can be avoided by adding enough padding between the size value and the start of the actual data to restore the required alignment. However even with this change this approach still requires adjusting any application code that deals with buffer contents to account for the start offset. The existence of the field would also have to be signaled by applications in some way to avoid breaking applications that

```
cl_mem data_buffer;
cl_mem data_size;
cl_event ev;
...
/* Attach data_size to data_buffer to hold
 * the content size. */
clSetContentSizeBufferPOCL(data_buffer, data_size);

/* Kernel writes an unknown amount of data to
 * data_buffer, and its size to the data_size
 * argument. */
clSetKernelArg(kernel1, 0, sizeof(cl_mem),
               &data_buffer);
clSetKernelArg(kernel1, 1, sizeof(cl_mem),
               &data_size);
clEnqueueNDRangeKernel(command_queue, kernel1, 1,
                       NULL, NULL, NULL,
                       0, NULL, &ev);

/* The second kernel uses information from data_size
 * to restrict its processing to the meaningful part
 * of data_buffer. */
clSetKernelArg(kernel2, 0, sizeof(cl_mem),
               &data_buffer);
clSetKernelArg(kernel2, 1, sizeof(cl_mem),
               &data_size);
clEnqueueNDRangeKernel(command_queue, kernel2, 1,
                       NULL, NULL, NULL,
                       1, &ev, NULL);
...
clFinish();
```

Figure 4.7. *Example of using the proposed dynamic buffer extension in a sequence of two kernels. The user defines a designated buffer where the kernel stores the size, which can be then used by the runtime to optimize the buffer transfers and migrations, as well as by the consumer kernels of the buffer to read the input size.*

have not been written with this extension in mind.

Another solution that would avoid the need for applications to account for the extra field at the beginning would be to place the size field at the end. This approach is even less feasible than the previous one, as the runtime would have to keep track of the known values of size fields for every buffer both on the client and on every remote node. When the used size is not known, the entire maximum size of the buffer would have to be transferred along with the new value for the size field. In practice buffers are frequently used in a streaming fashion where they get filled by one node and sent to others. This means that their used sizes will not be known to other peers ahead of any transfer and thus transfers would have to be done for the maximum size of the buffer.

5 EVALUATION

The performance of the *pocl-r* runtime was evaluated with a set of synthetic benchmarks covering latency, data migration between remote nodes and throughput. Finally the effect of offloading was measured in a case study where computational parts of a point cloud rendering application running on a mobile phone were offloaded to a powerful GPU server.

5.1 Synthetic Benchmarks

The following subsections describe the experiments performed to measure the latency and scalability of the *pocl-r* runtime and the results obtained. Latency is measured with a synthetic benchmark that repeatedly enqueues a no-op kernel on a remote device and measures the time until the kernel invocations are flagged as complete. The overhead from peer-to-peer data migration is measured by incrementing a single integer in a kernel that is enqueued repeatedly on multiple command queues corresponding to devices on separate remote nodes. Each invocation is enqueued on a different queue than the previous and consecutive invocations are marked as depending on each other by using events. Finally, throughput scalability is measured with a large matrix multiplication benchmark.

5.1.1 Command Overhead

Since low latency is a key priority of *pocl-r*, a synthetic benchmark was used to measure the runtime of a no-op kernel execution command to estimate the overhead imposed by the runtime. The kernel used for testing is a function that simply returns immediately, so any time spent executing it is negligible and the rest of the command duration is down to the network connection between the host application and the remote node, and the overhead from the runtime itself. We compare the numbers against the roundtrip time reported by the *ping* utility which is generally accepted as a good baseline for network latency.

This benchmark program implemented by Julius Ikkala creates a no-op kernel, enqueues it and waits for it to complete using `clFinish`. This is repeated 1000 times and the results

are averaged. The client is a desktop PC with a 100-Mbps wired connection to the server. Timestamps are taken in the application code before the `clEnqueueNDRangeKernel` and after a `clFinish` call to ensure the completion of the command has been registered by the client application. The duration between the two is used for the host-measured timings. Measurements were performend by the Author.

Two identical machines were used as the application host and as the remote node for benchmarking, with the following hardware configuration:

- **GPU:** 2x NVIDIA Geforce 2080 Ti
- **CPU:** AMD Ryzen Threadripper 2990wx
- **LAN:** 100 Mbit ethernet

The results of this test are shown in Fig. 5.1. For reference, the ICMP round-trip latency as reported by the `ping` utility fluctuates around 0.122 ms. On localhost the ICMP round-trip latency was measured to average at 0.020 ms. The average command duration was observed to be consistently around 60 microseconds more than ping. This can be considered a good result given that connections between consumer devices and application servers usually measure in tens to hundreds of milliseconds even in realtime applications and even on the 100-Mbps LAN between the workstations with a ping delay two to three orders of magnitude less than the aforementioned case, the overhead on top of ping is only a fraction of the full command duration. The absolute overhead on localhost remains roughly the same as on the 100-Mbps connection, which indicates that the overhead on top of ping delay is indeed constant and remains minor or even negligible.

5.1.2 Data Migration Overhead

The closest related work found in a review of existing research is SnuCL. Its authors report data movement being the bottleneck in some of their benchmarks [30]. In order to get a general idea of how much the runtime affects the communication overhead due to data movement, it is interesting to measure the minimum time a buffer migration between devices takes due to runtime overhead. This is done separately from the no-op command overhead measurements because *pocl-r* remote nodes communicate directly with each other in a P2P fashion: the host application only has to send a migration command to the source node. The source node then sends the buffer contents directly to the destination node, which in turn notifies the host application that the migration is done, while simultaneously proceeding to execute any commands it may have queued up that have an event dependency on the migration.

Memory migration overhead was measured by the Author with a small test program written by Julius Ikkala. The program requires at least two available devices in order

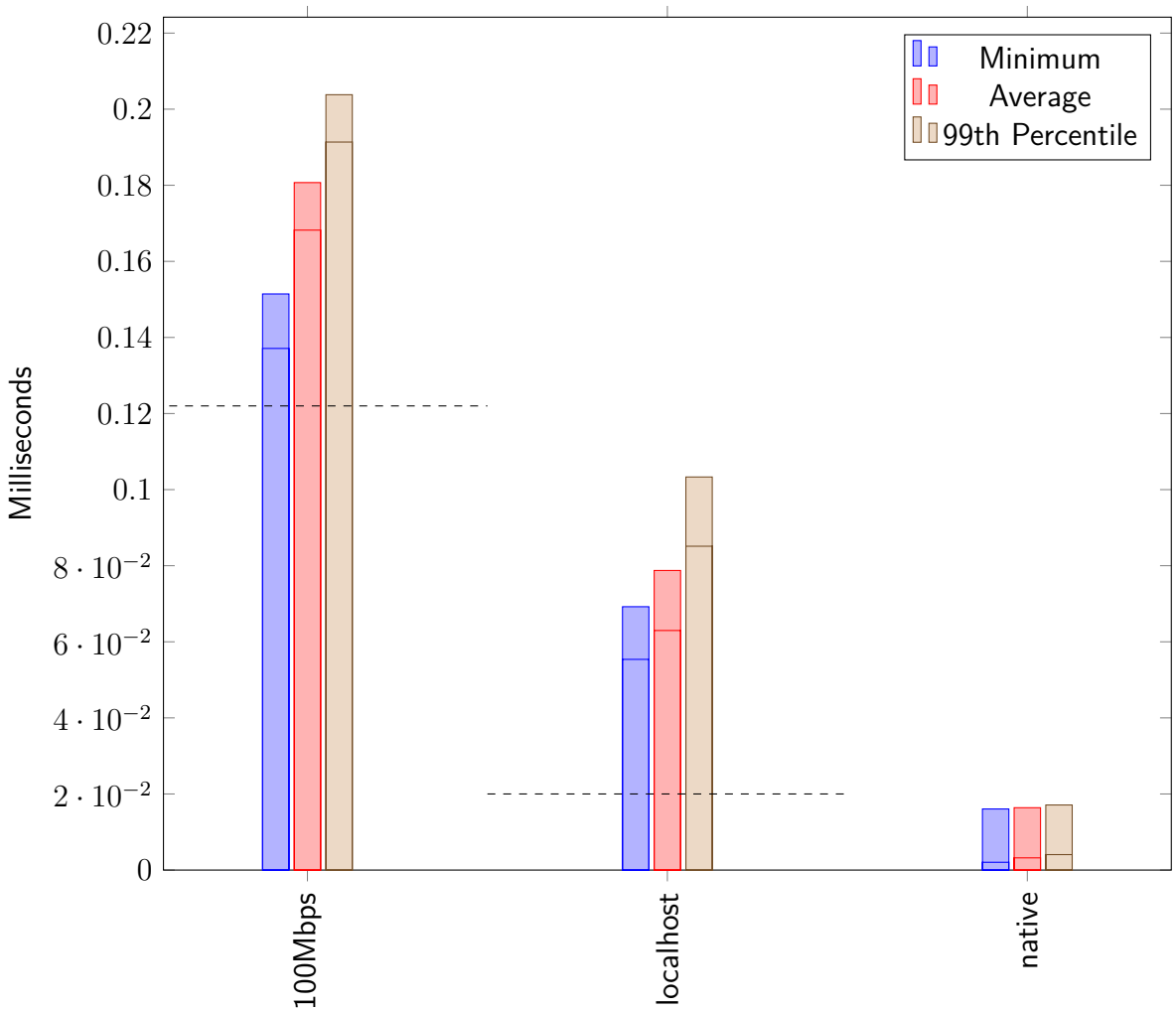

Figure 5.1. *Duration of a no-op command as measured by the client application using standard CPU timers (upper value) and as reported by OpenCL implementation-provided event timestamps (lower value). The dashed line represents the average ICMP ping for the scenarios that use TCP. Native refers to the native OpenCL implementation for the GPUs, provided by NVIDIA.*

to run. It creates a buffer of a given size and a kernel that increments the first value in the buffer. Then it cycles through all available devices in sequential or random order and enqueues an invocation of the kernel with an event dependency on the previous one on a different device. Sequential order was used for these measurements. This ensures an implicit migration of the buffer happens between kernel invocations. A workgroup of only one element is used to keep the runtime of the kernel itself to a minimum. All devices were cycled through 1000 times, and a buffer of 4 bytes, which makes the effect of the actual buffer content transfer negligible and allows for measuring the overhead of the remote offloading protocol itself.

All kernel invocations were enqueued in sequence, and after waiting for completion of all commands, the buffer migrations inserted by the *pocl-r* runtime were extracted and their timing information was analyzed. The timings from both servers were averaged together

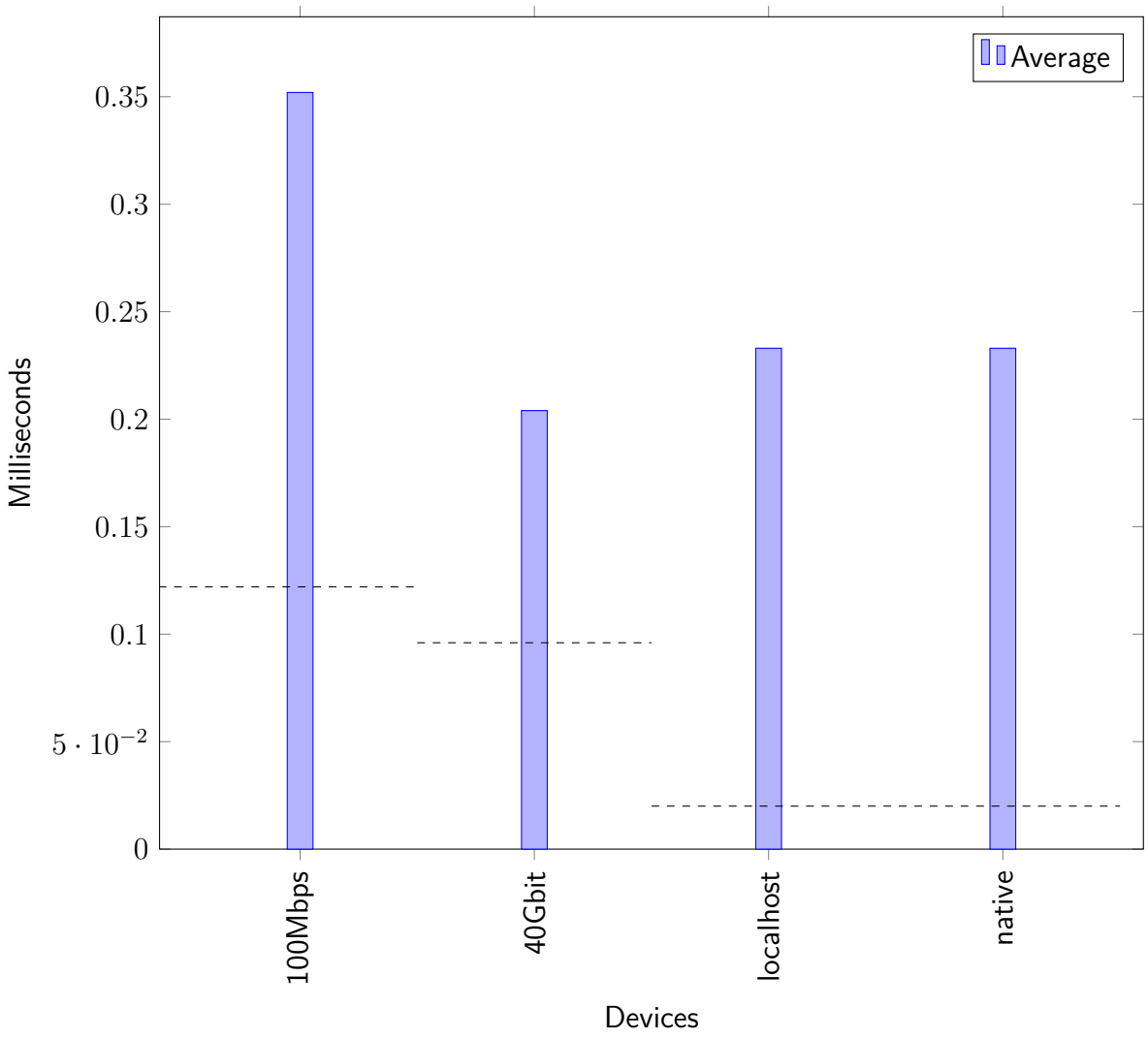

Figure 5.2. *Migration of a 4-byte buffer 1000 times between two devices using different connectivity between servers, as well as using the native NVIDIA driver for reference. Localhost refers to two pocl-r daemons running on the same machine as the host application and native refers to one daemon running on the same machine as the host application while providing access to two GPUs, letting the underlying native OpenCL implementation handle migrations internally. The dashed line represents the average ICMP ping for the given connection.*

and the results are shown in Fig. 5.2. When using a 100-Mbps ethernet connection between the remote nodes the average timings add up to around 3x the overhead of a no-op command on top of network ping, which seems reasonable for a 3-step roundtrip (from the host to the first node, to the second node and back to the host) with extra buffer management on the intermediate hops.

Using a 40-Gbps direct infiniband link shortens the total duration in comparison to the ping noticeably, mostly because this is a dedicated direct connection between the two machines with no switches or other network equipment on the way and no interference from other traffic from the operating system. The relative overhead ends up being only

around two thirds of that observed with the 100-Mbps connection. The benchmark was also run with two *pocl-r* daemons running on the same machine as well as one daemon migrating data between two GPUs installed on one machine. However, the drivers that provided the native OpenCL implementation used by the daemon appears to exhibit a notable performance regression when using two GPUs simultaneously instead of just one, making it impossible to obtain reliable numbers for these situations. This performance regression was also observed with applications that are not related to this master's thesis and used other APIs than OpenCL, granting further credibility to this observation.

Two machines with identical hardware were used for this benchmark:

- **GPU:** 2x Geforce 2080 Ti
- **CPU:** Ryzen Threadripper 2990wx
- **LAN:** 100Mbit ethernet, 40Gbit direct infiniband link

5.1.3 Distributed Large Matrix Multiplication

Non-trivial scalability was measured with a distributed matrix multiplication test implemented and measured by the Author. This benchmark multiplies two NxN matrices using as many devices as the OpenCL context has available. Every device gets the full data of both input matrices and calculates a roughly equal number of rows of the output matrix. While the actual calculations is an embarrassingly parallel task, the partial results from each device have to be collected into a single buffer for the final result, which makes the workload as a whole non-trivial to scale.

This is largely similar to the matrix multiplication used in the benchmarks of SnuCL [30] with the exception that here the parts of the output matrix are combined to a single buffer on one of the GPUs and this is included in the host timings. The NVIDIA example that is mentioned as the source for the benchmark in [30] only measures the duration of the actual compute kernel invocations, which corresponds to the device-measured timings in this benchmark. Whether the time to combine the partial results was accounted for in the SnuCL benchmark is unknown, but given that they report scalability problems, the time to combine the results was likely part of the measurements.

Benchmarking was done on a cluster with three nodes with an *Intel™ Xeon™ E5-2640 v4* CPU and four *NVIDIA Tesla P100* GPUs. An additional node with an *Intel™ Xeon™ Silver 4214* CPU and four *NVIDIA Tesla V100* GPUs was used to fill the number of usable compute devices to a total of 16 GPUs. All cluster nodes were connected to each other and to the machine running the host application with a 56-Gbps infiniband link.

The relative speedup when multiplying two 8192 by 8192 matrices with an increasing number of GPUs is shown in Fig 5.3. The results exhibit logarithmic speedups compared

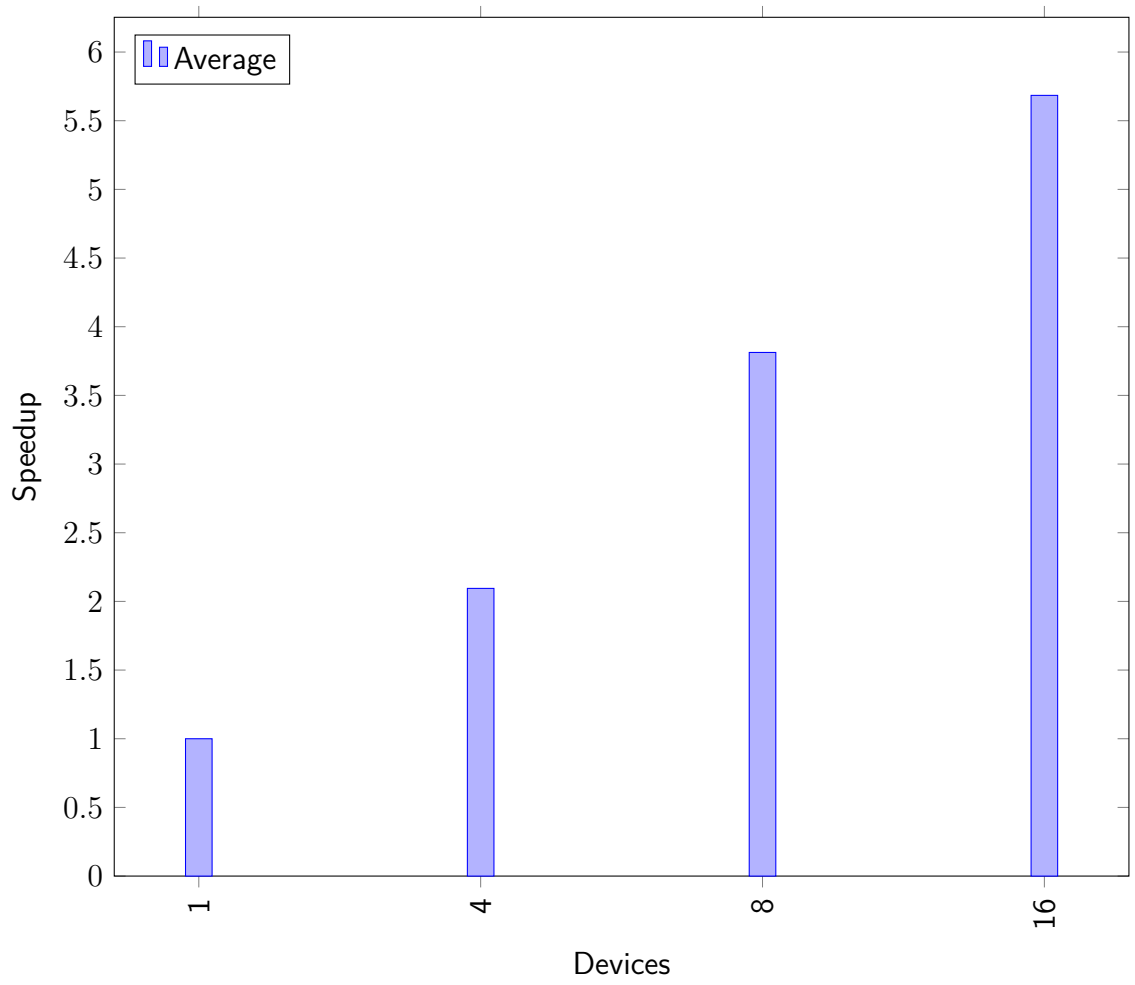

Figure 5.3. *Multiplication of two 8192x8192 matrices using 1 to 16 remote devices in servers with 4 GPUs each, averaged across 5 runs that were executed in parallel. Displayed is the speedup compared to using a single GPU.*

to using a single GPU up to slightly below 6x with 16 GPUs. This is expected, since the total computational workload stays the same while the number of GPUs used increases, making the workload for one GPU inversely proportional to the number of GPUs. After calculating the partial results on each GPU, they have to be combined into one single matrix, which is done in a hierarchical fashion and results in a logarithmic number of calls to *clCopyBufferRect*. This is also in line with the results observed in [30] with the version of SnuCL that uses their proposed MPI collective communication extensions. The implementation used here also doesn't exhibit the performance regression reported with the unextended P2P version of SnuCL when using more than 8 devices.

5.2 Real-time Point Cloud Augmented Reality Rendering Case Study

Real-world performance was measured with a full application task offloading case study implemented and measured by Michal Babej. The subject of the case study was an Android-based smartphone application [31] that renders a streamed animated point cloud in augmented reality (AR). Fig. 5.4 shows the application in action. The point cloud is received as an HEVC-encoded [32, 33] Video-based Point Cloud Compression (VPCC) stream [34] which is decompressed using the mobile device's hardware HEVC decoder and reconstructed using OpenGL [35] shaders [36]. A more in-depth explanation of this process is given in [37].

In order to enhance the rendering quality, alpha blending is used for a more visually pleasing, less pixelated end result compared to simply drawing the points as hard-edged squares. To facilitate this, the received cloud of points is sorted by distance to the viewer.

Computing the order of the points is a very computationally intensive task compared to reconstructing the individual point positions, so this can be optionally offloaded to a remote compute node. When offloading is disabled, all decoding, reconstructing and reordering happens on the mobile device's own GPU. When offloading is enabled, the VPCC stream is sent to both the device and directly to the remote compute node and decoding and point reconstruction are performed on both. However, the point sorting is only done on the remote, and the final sorted point indices are sent back to the mobile device for use when displaying the points. This frees up the mobile GPU for other tasks such as reconstructing its own viewpoint position from live data obtained from the builtin camera.

The remote daemon makes use of the OpenCL 1.2 custom device type feature to provide a virtual device that exposes the node's video decoding capabilities using VDPAU (Video Decode and Presentation API for Unix) and OpenGL; the decoder appears to the application as a fully conformant OpenCL device of type `CL_DEVICE_TYPE_CUSTOM` and thus does not require the use of any API extensions. The decoded result is made available as an OpenCL buffer with the OpenGL-OpenCL interoperation feature. The same event-based command scheduling as for remote OpenCL commands applies to commands targeting this device. Thanks to this, the host application does not need to be involved between the completion of the decoding command and the start of further computation (i.e. reconstruction and sorting of the points). The proposed dynamic buffer size extension can optionally be used to speed up transfers of the buffers between the OpenCL devices as their sizes vary wildly between frames – especially the compressed VPCC stream which on average has a much smaller chunk size than its

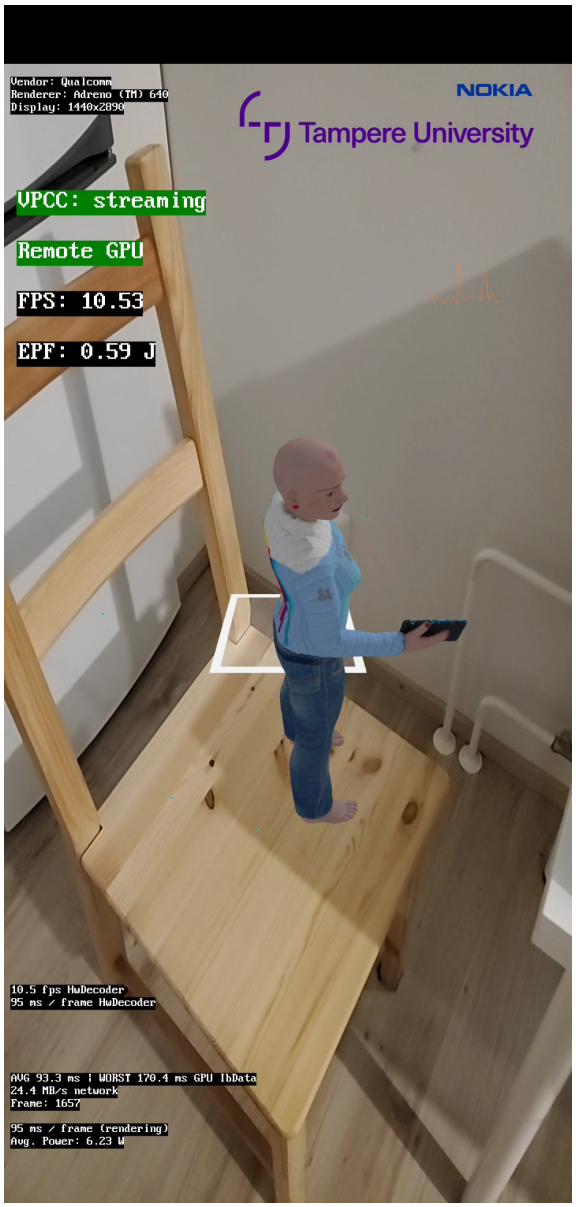

Figure 5.4. *Screenshot of the AR application used to measure the effect of offloading heavy computation. A streamed animated point cloud of a person holding a small tablet device is displayed in augmented reality on top of a real-world chair. Screenshot reprinted with kind permission from Michal Babej.*

worst case. A similar custom device is used as the data source when offloading is enabled. This *remote streaming device* reads a continuous VPCC-encapsulated HEVC video stream from a camera a prerecorded file and provides a builtin kernel (created with `clCreateProgramWithBuiltInKernels`) that places the next chunk of the stream in a buffer provided by the application.

Framerates measured from the application are displayed in Fig. 5.5. The first two values are measured with the local GPU performing the reconstruction, sorting and AR positioning tasks by itself. For the next two values the sorting task to a GPU on a *pocl-r* remote node with P2P buffer transfers disabled and enabled, for a roughly 2.3x speedup

over the full reconstruction, sorting and AR workload done on the mobile GPU. Finally, the figure shows the impact of the optional dynamic buffer size extension on the framerate. On average, the amount of data that has to be transferred when buffer content sizes are known is only a fraction of the buffers' worst-case size. Being able to omit the unnecessary portion while migrating buffers thus has a huge effect on performance, adding up to an almost 19x speedup compared to doing all the work on the mobile GPU.

Fig. 5.6 shows energy consumption per frame (EPF) measured on the mobile device in the same offloading configurations, with similar results. The power usage of the smartphone was retrieved using Android's Power Stats HAL interface. Most notably, offloading the sorting of the point cloud compensates for most of the added energy consumption from AR positioning even without any of the measured optimizations in place. Enabling P2P buffer transfers and the content size extension further cuts energy consumption per frame to a mere fifth of the first test that performs only decoding and rendering the point cloud from a fixed viewpoint with no AR positioning.

Hardware used for these measurements:

- **Remote GPU:** Geforce 1060 3GB, Lenovo M910t Intel with i7-6700
- **Remote custom device:** a virtual device implemented as a *pocl* device driver that serves as the data source for the application, simulating a point cloud camera by reading the stream from a file.
- **Mobile device:** Samsung Galaxy S10 SM-G973U1, Qualcomm® Snapdragon™ 855, Wifi 6
- **Wifi router:** ASUS ROG Rapture GT-AX11000
- **Wifi router to remote node connection:** 1Gbit ethernet

5.3 Discussion

Overall, the raw throughput measured with *pocl-r* is well in line with previous implementations of distributed compute frameworks on the same workload. Additionally, in a real-world use case performance was doubled by simply offloading sorting of the points to be rendered with *pocl-r* while still doing the actual rendering locally and only using plain unextended OpenCL in the application. When also making use of the proposed OpenCL extensions, performance improvements close to 10x compared to performing all computation on the local GPU of the mobile device used in testing were observed. Similarly, energy use in the augmented reality use case was halved with simply offloading work and when additionally making use of the proposed OpenCL extension to reduce network traffic, energy use dropped to a fraction of the original. Without the additional load from AR location tracking and live video compositing the energy use without the extension ended up being slightly worse. When the application was using the buffer

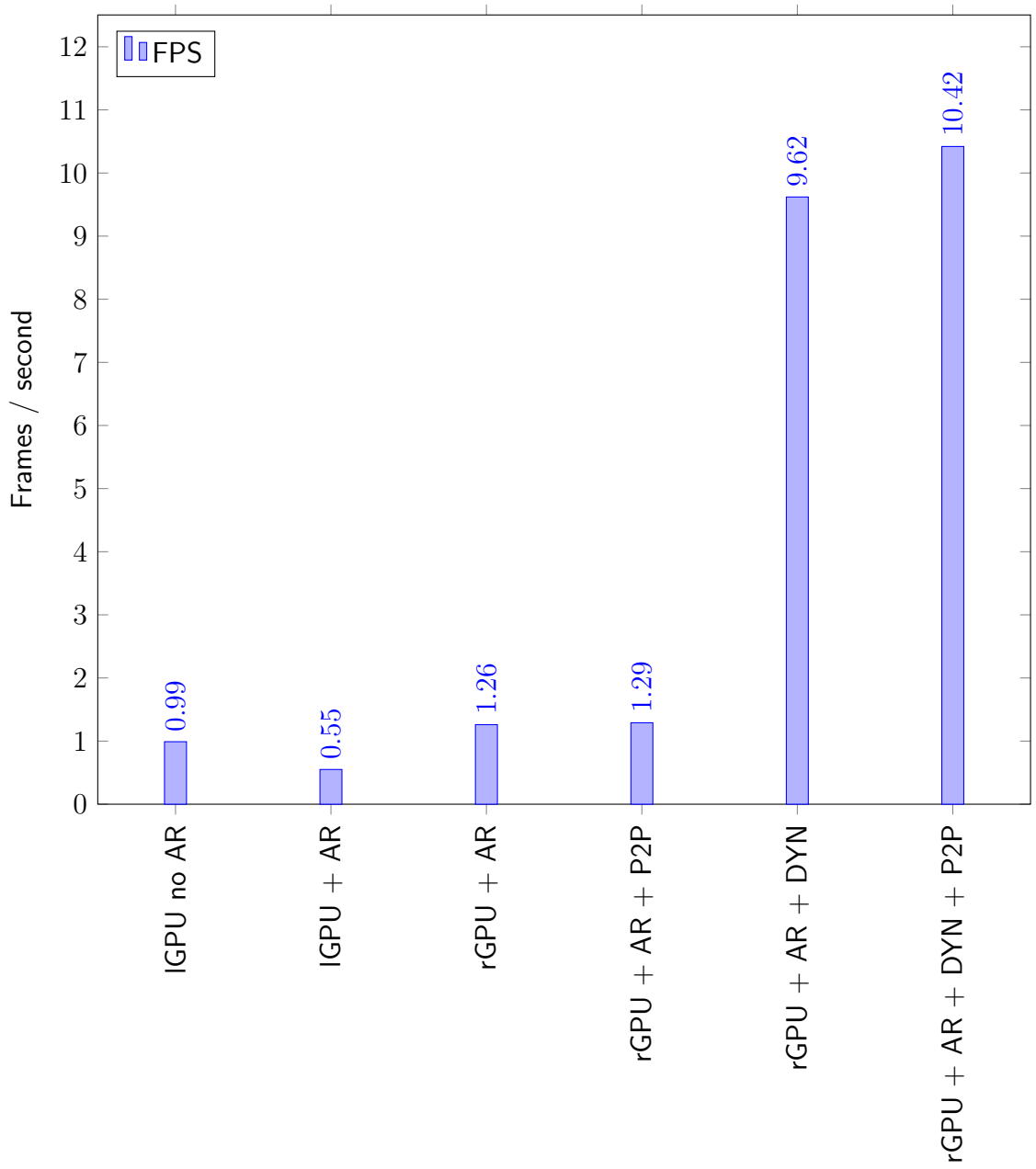

Figure 5.5. *Framerate of the AR demo application in various offloading configurations. IGPU and rGPU refer to the mobile device's local GPU and the remote GPU exposed via pocl-r. AR refers to whether the point cloud is rendered on top of a live camera view with AR position tracking or simply from a static perspective against a black background. P2P refers to transferring buffer data from the (remote) data source directly to the remote GPU and DYN indicates that the buffer content size extension is used to avoid sending unnecessary bytes. Figure reprinted with kind permission from Michal Babej.*

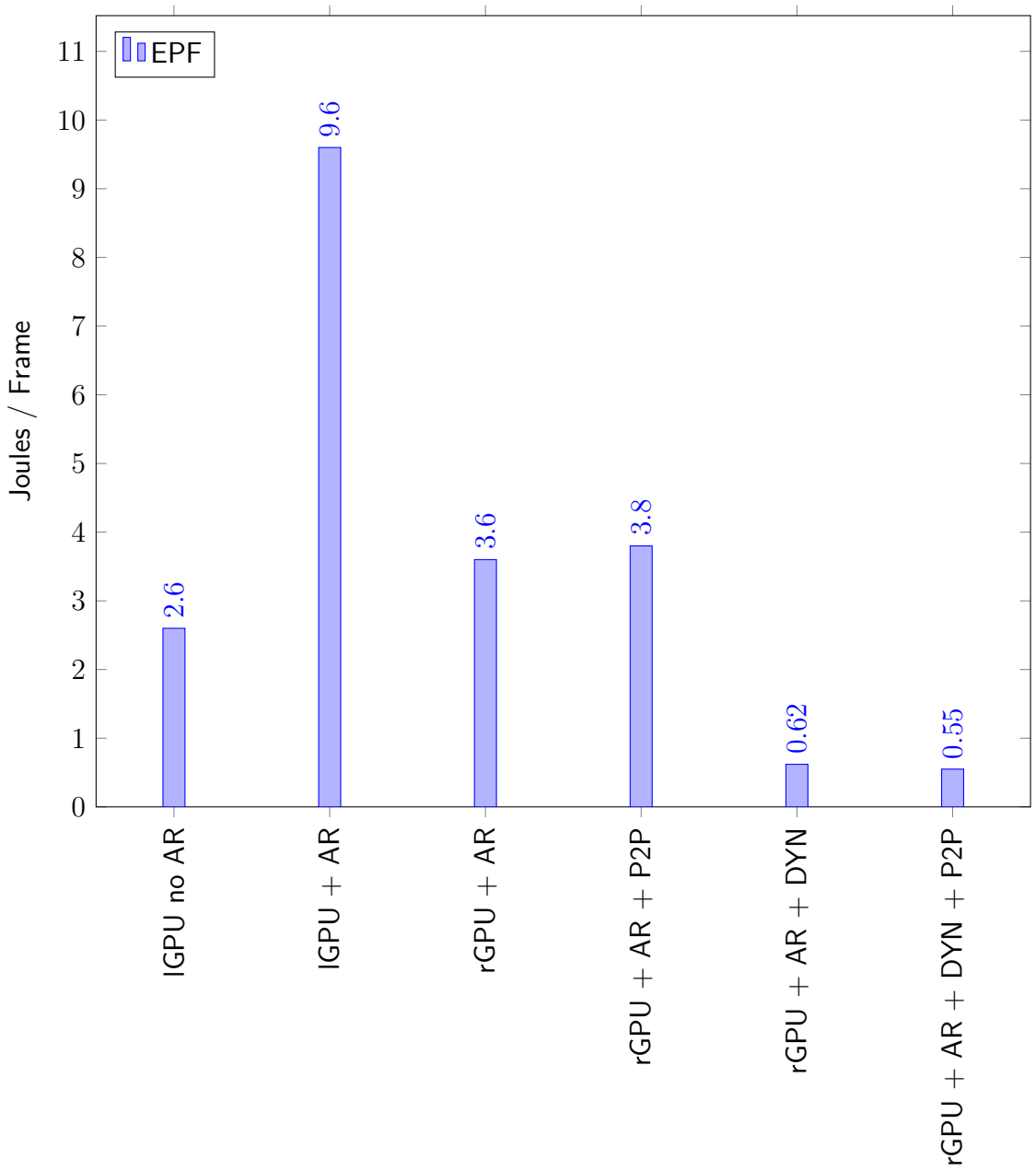

Figure 5.6. *The mobile device's energy consumption per frame in various offloading configurations. IGPU and rGPU refer to the mobile device's local GPU and the remote GPU exposed via pocl-r. AR refers to whether the point cloud is rendered on top of a live camera view with AR position tracking or simply from a static perspective against a black background. P2P refers to transferring buffer data from the (remote) data source directly to the remote GPU and DYN indicates that the buffer content size extension is used to avoid sending unnecessary bytes. Image reprinted here with kind permission from Michal Babej.*

size extension to reduce unnecessary network traffic, total energy use was reduced to roughly a fifth compared to sorting the point cloud locally on the mobile device. All in all the results show that great energy savings can be obtained from offloading and that *pocl-r* offers comparable performance to prior solutions without extending the standard OpenCL API and without relying on any specific network environment details.

6 RELATED WORK

Multiple projects [38, 39, 40, 41] have expanded the scope of originally single node targeting heterogeneous APIs for distributed use in the past, but most of them have long since faded into obscurity and their implementations are no longer available for use and comparison, let alone for further development. Various projects [41, 42, 43, 44] also solely target High Performance Computing (HPC) clusters with their existing library ecosystem and optimize for throughput even when it means that latency will suffer. By contrast, our proposed runtime targets to support both HPC clusters and realtime applications, and most interestingly, their combination.

Among the previous projects we found, the closest to *pocl-r* is SnuCL [30]. It provides an implementation of the standard OpenCL API that enables execution of OpenCL commands on remote nodes. However, it focuses solely on throughput in HPC cluster use cases with no consideration of latency. For communication it relies on the MPI framework. SnuCL supports peer-to-peer data transfers, but they report scaling problems in some tasks such as the matrix multiplication we used in our benchmarking. SnuCL solves these scaling issues with a proposed OpenCL extension that maps MPI collective operations to a set of new OpenCL commands. In contrast, *pocl-r* uses plain TCP sockets with a custom protocol and socket settings tuned for low latency and uses the native in-memory representation of commands as-is as its networking protocol to minimize serialization overheads. SnuCL also handles command scheduling on the host machine, whereas *pocl-r* sends commands to remote nodes as soon as the host application issues them, and relies on OpenCL events to let the remotes handle their internal command scheduling autonomously.

Further work on SnuCL also exists in the form of SNUCL-D [45], which further decentralizes computation by duplicating the control flow of the entire host program on each remote node. This results in great scalability improvements in theory, but requires the host application to be fully replicable on all nodes which is naturally not possible by default. For example, user-facing realtime applications have a single host program running the user interface. The host application can also require access to node-specific resources besides the compute devices that OpenCL provides access to, such as local files or connectivity to private networks. Functionality such as random number generation also has to be specially wrapped for consistency across all instances. Thus, user

software would have to be written specifically with this runtime in mind while *pocl-r* provides "drop-in compatibility" with existing OpenCL applications.

Another very close project in terms of the overall idea is rCUDA [46]. At the time of this writing, rCUDA is one of the most actively developed related projects, but being based on the proprietary CUDA API it is limited in hardware support and extensibility.

There is also a recent open source project by the name RemoteCL [47] that takes the same approach with plain network sockets as *pocl-r*. However, it only aims to fit the needs of the author and makes no attempt at providing a full conformant implementation of the OpenCL API. It also does not appear to support more than one remote server.

7 CONCLUSION

This work describes *pocl-r* an implementation of the standard OpenCL API that allows for offloading and distribution of heavy computation across a network with a focus on low latency and scalability. For scalability direct peer-to-peer communication between remote nodes is implemented to reduce the host application's involvement in buffer transfers to just sending the initial command to the source device. Low latency is achieved by eliminating the serialization step that is usually done in network communication.

Synthetic benchmarks place *pocl-r* at least on par with previous distributed offloading frameworks in terms of throughput. Latency of a no-op compute command and migration of a tiny memory buffer was compared to a basic network ping round-trip time for a lack of comparable measurements in prior work. Latency benchmark results across a network connection scaled well with connection speed, but were inconclusive on localhost due to driver behaviour when using multiple GPUs on the same machine.

Real-world impact was measured with a case study of a streaming AR point cloud renderer running on a mobile phone. Offloading parts of the point cloud reconstruction yielded notable speed and energy usage improvements. Further improvements were observed when enabling optional features of *pocl-r* to allow the remote node to fetch the data stream by itself without routing traffic through the host application as well as by extending the OpenCL API to allow attaching information about the used size to fixed-capacity buffers that hold compressed data of an unpredictable size.

All in all *pocl-r* performs comparably to previous compute offloading solutions, but offers greater interoperability by being based on the vendor-independent OpenCL rather than a proprietary API. Compared to the closest prior work, SnuCL, *pocl-r* also provides greater flexibility by not relying on data center -focused networking frameworks. This makes it viable both in the traditional data center applications as well as consumer applications targeting mobile platforms, where offloading to cloud edge servers is desired.

Future optimizations might be possible by using Remote Direct Memory Access (RDMA) for buffer transfers between remotes, and potentially between the application and remotes as well. One notable concern is the overhead from the RDMA API that requires registering potential buffer transfers beforehand, and signaling of the transfer's start and completion. Should the setup phase prove to be prohibitively slow it might be possible

to mitigate it by having the application specify buffer residency hints to allow the runtime to set up the necessary transfers ahead of time. This could be automated by using information from task graph analysis as outlined in [29].

Improvements to the flexibility of this offloading and distribution approach would be useful. In particular mobile applications where the application is running on a device whose network connectivity and latency to remote nodes can change frequently across the runtime of the application would benefit greatly from the ability to dynamically roam between remote nodes based on which nodes are accessible and which provide the best latency-throughput tradeoff. This will require rather invasive extensions to the OpenCL API. On the other hand, such extensions would likely also make it feasible to harness compute resources of other nearby compute devices for collaborative *swarm computing*.

It might also be possible to extend this approach to other APIs, such as Vulkan. Compute shaders in Vulkan are very similar to OpenCL kernels and the specification technically allows devices that do not have any graphics functionality, so the design of *pocl-r* should be easy to adapt for a compute-only Vulkan implementation. Furthermore, there is an extension that enables hardware-accelerated ray tracing, which could be exposed remotely [48]. Antwerpen et al explore ways to divide ray tracing tasks among multiple GPUs and describe an effective load balancing scheme that makes distribution across a network an interesting research question [49].

REFERENCES

[1] Hu, Y. C., Patel, M., Sabella, D., Sprecher, N. and Young, V. Mobile edge computing—A key technology towards 5G. *ETSI white paper* 11.11 (2015), 1–16.

[2] Khronos® OpenCL Working Group. *The OpenCL™ Specification*. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf. accessed: 2020-10-16.

[3] Tjaden, G. and Flynn, M. Detection and Parallel Execution of Independent Instructions. eng. *IEEE transactions on computers* C-19.10 (1970), 889–895. ISSN: 0018-9340.

[4] Shan, A. Heterogeneous processing: a strategy for augmenting moore's law. *Linux Journal* 2006.142 (2006), 7.

[5] Greenhalgh, P. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper* 17 (2011).

[6] Reese, G. *Database Programming with JDBC and JAVA*. " O'Reilly Media, Inc.", 2000, 128–136.

[7] Saroiu, S., Gummadi, P. K. and Gribble, S. D. Measurement study of peer-to-peer file sharing systems. *Multimedia Computing and Networking 2002*. Vol. 4673. International Society for Optics and Photonics. 2001, 156–170.

[8] Shanley, T. *InfiniBand network architecture*. Addison-Wesley Professional, 2003.

[9] The MPI Forum, C. MPI: A Message Passing Interface. *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing '93. Portland, Oregon, USA: Association for Computing Machinery, 1993, 878–883. ISBN: 0818643404. DOI: 10.1145/169627.169855. URL: https://doi.org/10.1145/169627.169855.

[10] Culley, P., Garcia, D., Hilland, J., Metzler, B. and Recio, R. A remote direct memory access protocol specification. *IETF RFC-5040* (2007).

[11] Li, A., Song, S. L., Chen, J., Li, J., Liu, X., Tallent, N. R. and Barker, K. J. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31.1 (2020).

[12] Shpiner, A., Zahavi, E., Dahley, O., Barnea, A., Damsker, R., Yekelis, G., Zus, M., Kuta, E. and Baram, D. RoCE Rocks without PFC: Detailed Evaluation. *Proceedings of the Workshop on Kernel-Bypass Networks*. KBNets '17. Los Angeles, CA, USA, 2017. ISBN: 9781450350532. DOI: 10.1145/3098583.3098588.

[13] Taleb, T., Samdanis, K., Mada, B., Flinck, H., Dutta, S. and Sabella, D. On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration. *IEEE Communications Surveys Tutorials* 19.3 (2017).

[14] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, 483–485.

[15] Gustafson, J. L. Reevaluating Amdahl's Law. *Commun. ACM* 31.5 (May 1988), 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: https://doi.org/10.1145/42411.42415.

[16] Martin, J. *Programming real-time computer systems*. Tech. rep. 1965.

[17] Moore, G. E. Cramming more components onto integrated circuits. *Proceedings of the IEEE* 86.1 (1998), 82–85.

[18] Flynn, M. J. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers* C-21.9 (1972), 948–960. DOI: 10.1109/TC.1972.5009071.

[19] Spector, A. and Gifford, D. The Space Shuttle Primary Computer System. *Commun. ACM* 27.9 (Sept. 1984), 872–900. ISSN: 0001-0782. DOI: 10.1145/358234.358246. URL: https://doi-org.libproxy.tuni.fi/10.1145/358234.358246.

[20] Khronos® OpenCL Working Group. *The OpenCL™ C Specification*. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_C.pdf. accessed: 2020-10-16.

[21] Jääskeläinen, P., Korhonen, V., Koskela, M., Takala, J., Egiazarian, K., Danielyan, A., Cruz, C., Price, J. and McIntosh-Smith, S. Exploiting Task Parallelism with OpenCL: A Case Study. *Journal of Signal Processing Systems* 91.1 (Jan. 2019). ISSN: 1939-8115.

[22] Kessenich, J., Ouriel, B. and Krisch, R. *SPIR-V Specification*. https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf. accessed: 2020-12-01.

[23] Khronos® OpenCL Working Group. *The OpenCL™ Extension Specification*. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_Ext.pdf. accessed: 2020-10-16.

[24] Mark Segal and Kurt Akeley. *The OpenGL® Graphics System: A Specification (Version 4.6 (Core Profile) - October 22, 2019)*. https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf. accessed: 2020-11-19.

[25] Microsoft. *Direct3D 12 graphics*. https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics. accessed: 2020-11-19.

[26] Jääskeläinen, P., La Lama, C. S. de, Schnetter, E., Raiskila, K., Takala, J. and Berg, H. pocl: A Performance-Portable OpenCL Implementation. English. *Int. Journal of Parallel Programming* 43.5 (2015). ISSN: 0885-7458.

[27] Google Inc. *FlatBuffers*. https://google.github.io/flatbuffers/. accessed: 2020-10-19.

[28] Furuhashi, S. *MessagePack: It's like JSON. But fast and small.* https://msgpack.org/. accessed: 2020-10-19.

[29] Jääskeläinen, P., Korhonen, V., Koskela, M., Takala, J., Egiazarian, K., Danielyan, A., Cruz, C., Price, J. and McIntosh-Smith, S. Exploiting task parallelism with OpenCL: A case study. *Journal of Signal Processing Systems* 91 (2019).

[30] Kim, J., Seo, S., Lee, J., Nah, J., Jo, G. and Lee, J. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. *Proceedings of the 26th ACM international conference on Supercomputing*. 2012, 341–352.

[31] Nokia Technologies Ltd. *Video Point Cloud Coding (V-PCC) AR Demo*. https://github.com/nokiatech/vpcc/. accessed: 2020-10-16.

[32] Rec, I. *H. 265 and ISO/IEC 23008-2: High Efficiency Video Coding (HEVC)*. 2013.

[33] Sullivan, G. J., Ohm, J.-R., Han, W.-J. and Wiegand, T. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on circuits and systems for video technology* 22.12 (2012), 1649–1668.

[34] Group, 3. et al. Text of ISO/IEC CD 23090-5: Video-based Point Cloud Compression. *ISO/IEC JTC1/SC29/WG11 Doc. N18030* (2020).

[35] The Khronos Group Inc. *OpenGL® ES Version 3.2 (October 22, 2019)*. https://www.khronos.org/registry/OpenGL/specs/es/3.2/es_spec_3.2.pdf. accessed: 2020-10-19.

[36] Simpson, Robert J. and Baldwin, Dave and Rost, Randi. *OpenGL ES® Shading Language Version 3.20.6*. https://www.khronos.org/registry/OpenGL/specs/es/3.2/GLSL_ES_Specification_3.20.pdf. accessed: 2020-10-19.

[37] Schwarz, S. and Pesonen, M. Real-time decoding and AR playback of the emerging MPEG video-based point cloud compression standard. *IBC 2019, Helsinki, Finland.* (2019).

[38] Liang, T.-Y. and Lin, Y.-J. JCL: an OpenCL programming toolkit for heterogeneous computing. *International Conference on Grid and Pervasive Computing*. Springer. 2013, 59–72.

[39] Xiao, S., Balaji, P., Zhu, Q., Thakur, R., Coghlan, S., Lin, H., Wen, G., Hong, J. and Feng, W.-c. VOCL: An optimized environment for transparent virtualization of graphics processing units. *2012 Innovative Parallel Computing (InPar)*. 2012.

[40] Reynolds, C. J., Lichtenberger, Z. and Winter, S. Provisioning OpenCL capable infrastructure with infiniband verbs. *2011 10th International Symposium on Parallel and Distributed Computing*. IEEE. 2011.

[41] Kegel, P., Steuwer, M. and Gorlatch, S. dOpenCL: Towards a uniform programming approach for distributed heterogeneous multi-/many-core systems. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 2012.

[42] Alves, A., Rufino, J., Pina, A. and Santos, L. P. clOpenCL-supporting distributed heterogeneous computing in HPC clusters. *European Conference on Parallel Processing*. 2012.

[43] Barak, A. and Shiloh, A. *The VirtualCL (VCL) cluster platform*. 2013.

[44] Diop, T., Gurfinkel, S., Anderson, J. and Jerger, N. E. DistCL: A framework for the distributed execution of OpenCL kernels. *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. 2013.

[45] Kim, J., Jo, G., Jung, J., Kim, J. and Lee, J. A Distributed OpenCL Framework Using Redundant Computation and Data Replication. *SIGPLAN Not.* 51.6 (June 2016). ISSN: 0362-1340.

[46] Duato, J., Peña, A. J., Silla, F., Mayo, R. and Quintana-Ortí, E. S. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. *2010 Int. Conf. on High Performance Computing Simulation*. June 2010.

[47] Ferreira, P. O. *RemoteCL*. https://github.com/silverclaw/RemoteCL. accessed: 2020-10-16.

[48] The Khronos® Vulkan Working Group. *Vulkan® 1.2.162 - A Specification*. https://www.khronos.org/registry/vulkan/specs/1.2/html/. accessed: 2020-11-24.

[49] Antwerpen, D. v., Seibert, D. and Keller, A. A Simple Load-Balancing Scheme with High Scaling Efficiency. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Ed. by E. Haines and T. Akenine-Möller. Berkeley, CA: Apress, 2019, 127–133. ISBN: 978-1-4842-4427-2. DOI: 10.1007/978-1-4842-4427-2_10. URL: https://doi.org/10.1007/978-1-4842-4427-2_10.