PoCL-R: A Scalable Low Latency Distributed OpenCL Runtime

Jan Solanti[1], Michal Babej[1], Julius Ikkala[1], Vinod Kumar Malamal Vadakital[2], Pekka Jääskeläinen[1]

[1] Faculty of Information Technology and Communication Sciences (ITC)
Tampere University
Tampere, Finland
{jan.solanti,michal.babej,julius.ikkala,pekka.jaaskelainen}@tuni.fi
[2] Nokia Technologies
Tampere, Finland
vinod.malamalvadakital@nokia.com

Abstract. Offloading the most demanding parts of applications to an edge GPU server cluster to save power or improve the result quality is a solution that becomes increasingly realistic with new networking technologies. In order to make such a computing scheme feasible, an application programming layer that can provide both low latency and scalable utilization of remote heterogeneous computing resources is needed. To this end, we propose a latency-optimized scalable distributed heterogeneous computing runtime implementing the standard OpenCL API.
In the proposed runtime, network-induced latency is reduced by means of peer-to-peer data transfers and event synchronization as well as a streamlined control protocol implementation. Further improvements can be obtained streaming of source data directly from the producer device to the compute cluster. Compute cluster scalability is improved by distributing the command and event processing responsibilities to remote compute servers. We also show how a simple optional dynamic content size buffer OpenCL extension can significantly speed up applications that utilize variable length data.
For evaluation we present a smartphone-based augmented reality rendering case study which, using the runtime, receives 19x improvement in frames per second and 17x improvement in energy per frame when offloading parts of the rendering workload to a nearby GPU server. The remote kernel execution latency overhead of the runtime is only 60 microseconds on top of the network roundtrip time. The scalability on multi-server multi-GPU clusters is shown with a distributed large matrix multiplication application.

1 Introduction

End-user applications are increasingly moving to battery-powered devices, and at the same time, the computational complexity of their functionalities increase. Offloading parts of applications to an edge node that resides within a short network round-trip from the user device is a solution that is becoming more feasible

with low-latency next-gen networking technologies such as 5G and WiFi6. The overall concept of utilizing edge cluster resources across low latency network links, called *Multi-access Edge Computing (MEC)* [29] is now an active field of research and development.

In the application layer, the MEC paradigm calls for a solution that both minimizes end-to-end latency overheads and allows utilizing all the heterogeneous compute resources in the remote edge cluster in a scalable and portable manner. To this end, we propose a scalable low-latency distributed heterogeneous computing runtime that implements the standard OpenCL API [15] and is targeted for usage by the application layer either directly or transparently as a backend for higher level interfaces with OpenCL backends such as SYCL [16] and oneAPI [10].

Unlike the previous distributed OpenCL projects, the proposed runtime called *PoCL-R* focuses on latency and the edge cluster side scalability at the same time. Furthermore, *PoCL-R* also provides support for low latency distributed streaming applications where data is read from a remote input device to the end user (client) device, which then needs to be further processed to produce the output. With *PoCL-R*, the input data can be streamed directly to the remote compute node, reducing the client's bandwidth use and overall latency. Overall, a key benefit of *PoCL-R* is that the whole edge cluster workload distribution can be orchestrated from the client application logic side without application-specific server-side software, thanks to the generality and power of the heterogeneous OpenCL API.

We identify the following novel aspects in the runtime presented in this paper:

- Utilization of edge cluster compute resources with *peer-to-peer (P2P)* communication and synchronization for improved compute scalability.
- Capability of supporting applications with both high performance and low latency demands to support distributed compute offloading scenarios of MEC with a wide complexity range.
- Enable transfers of input data straight from a producer server to the edge cluster before passing it to the client device while still only utilizing the standard OpenCL API's features.
- A minimal (optional) OpenCL API extension that can improve transfer times of dynamic-size buffers dramatically. This is especially useful for taking advantage of buffers with compressed data.
- The first distributed OpenCL runtime that is integrated to a long-maintained widely used open source OpenCL implementation framework *PoCL* [13] and is thus usable and extensible for anyone freely in the future.[3]

In order to test the latency of the runtime in a real-time context, we present a real-time augmented reality mobile case study, which receives significant improvements in both *frames per second (FPS)* and *energy per frame (EPF)* by offloading parts of the object rendering workload to a remote GPU server.The edge compute cluster side performance is reported separately with a remote

[3] **The source code is available at http://code.portablecl.org/**

kernel execution latency overhead measurement and a multi-server multi-GPU cluster scaling experiment.

The paper is organized as follows. Section 2 gives an overview of the *PoCL-R* top level design and its usage aspects. Section 3 describes the most relevant techniques in the proposed runtime to achieve the low latency while retaining scalability. Section 4 lays out the results in terms of latency and throughput measurements, and Section 5 presents the MEC offloading case study. Section 7 describes some plans for future work and concludes the paper.

2 Architecture

The focus of *PoCL-R* is on minimizing the end-to-end latency to enable high quality user experience in responsive real-time edge cluster use scenarios as well as enable scalable use of diverse compute resources in the cluster.

The whole application logic is defined in a single host application, as specified by the OpenCL standard. The application includes both the main program running in the local device as well as the kernel programs that are executed on local, or in the case of *PoCL-R*, remote OpenCL devices. The OpenCL standard allows the kernel programs to be defined in a portable source code or an intermediate language, and alternatively using target-specific binary formats. This can be used to bypass long synthesis steps at application runtime when using FPGA-based accelerators.

PoCL-R runtime is implemented as a standard client-server architecture. The *client* is implemented as a special *remote driver* in *Portable Computing Language (PoCL)* [13], an open source implementation of the OpenCL API with flexible support for custom device backends. The remote driver acts as a "smart proxy" that exposes compute devices on a remote server through the OpenCL platform API the same way as local devices, making the use of remote devices in OpenCL applications identical to using local devices at the program logic level. Features of the remote devices depend on what their native drivers support.

A *host application* using the OpenCL API can use *PoCL-R* as a drop-in implementation without recompilation. When the *host application* is linked against *PoCL-R*, OpenCL calls are made to the *PoCL-R client* driver, which in turn connects to one or multiple *remote* servers, each providing one or more *remote compute devices*. The remote servers can form interconnected *clusters* visible and controlled by *PoCL-R* as *peers* to avoid round-trips back to the client whenever synchronization or data transfers are needed between the remote devices. The application can identify remote devices by the device name string that contains an additional "pocl-remote". This allows optimising choices of command queues and kernel implementations.

The server side is a daemon that runs on the remote servers and receives commands from the client driver, and dispatches them to the OpenCL driver of the server's devices accompanied with proper event dependencies.

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

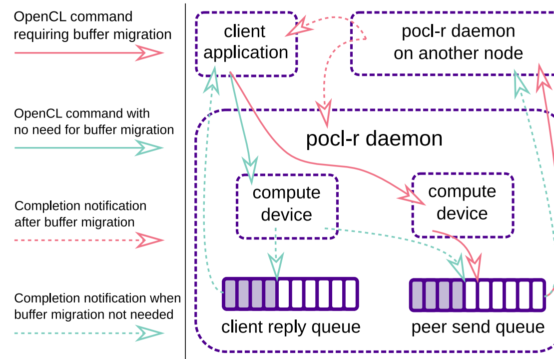


Fig. 1: The information flow from an application to the *PoCL-R* daemon and between remote servers. Two different commands are illustrated, one that transfers buffer contents from one remote node to another and one that doesn't.

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 server writer thread iterates through commands 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 work otherwise similar to the server writer.Fig. 1 illustrates this architecture and the flow of commands and data through it.

3 Latency and Scalability Optimizations

The following subsections describe the essential latency and scalability optimization techniques of *PoCL-R*.

3.1 Peer-to-Peer Communication

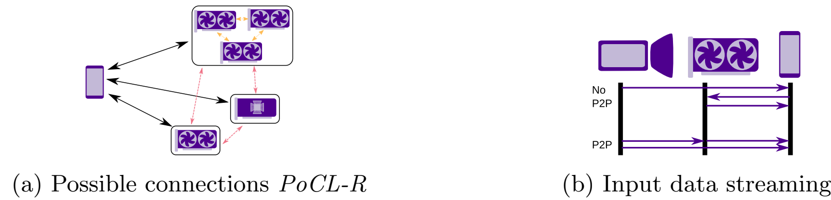
PoCL-R supports transferring buffers directly between devices on the same remote server (provided that the server's OpenCL implementation supports it), P2P transfers of buffers between servers, as well as distributed event signaling.

Fig. 2a illustrates the various possible links between the host application running in the client device that communicates with remote servers and devices. In a typical edge cluster use case, the client connection to the remote servers is much slower than the interconnect between servers in the 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 server.

3.2 Distributed Data Sourcing

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

Fig. 2b 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 a slight latency and throughput overhead.



(a) Possible connections *PoCL-R* (b) Input data streaming

Fig. 2: Various connections between devices in a *PoCL-R* context. Roundtrips to the client device are avoided when possible.

3.3 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* [9] and *MessagePack* [7] 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. The only added data is a fixed-size integer indicating the length of the next command structure.

The trade-off of this approach is that all remote servers 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. 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 [15]:

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.

3.4 Decentralized Command Scheduling

OpenCL provides command completion events as a synchronization mechanism between commands. *PoCL-R* relies heavily on these for keeping execution in sync across nodes with minimal overhead. Commands are pushed to the remote servers immediately when OpenCL enqueue API calls are made by the client. Event dependencies are mapped to platform-local events on each server and events for commands running on other servers are substituted with user events. This way the heterogeneous task graph based on event dependencies defined by the application stays intact on the remote servers and the runtime can apply optimisations utilizing the dependency rules outlined in [12].

In addition to the control and data connections to the client, each remote server keeps a direct connection to each of its peers. This is used for peer-to-peer buffer migrations and to signal event completions to other servers for use in command scheduling as illustrated in Fig. 1. 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 to the source server, which then signals other servers as needed.

3.5 Dynamic Buffer Content Size Extension

OpenCL allows applications to allocate memory in the form of buffers whose size is fixed once they are created. However, for many applications the amount of data actually produced or consumed varies greatly over time.As a means to improve performance when dealing with kernels dealing with varying size data, we propose a simple yet powerful OpenCL extension named *cl_pocl_content_size*. The extension provides an optional way to signal the actual used portion of an OpenCL buffer to the runtime as well as the consuming kernels. It works by designating a separate buffer, just large enough to hold a single unsigned integer, that holds the number of bytes actually being used by the buffer for valid data. *PoCL-R* runtime reads the content size buffer as an hint to only transfer the meaningful portion of buffers when migrating them between remote servers.

An example of using the extension is shown in the code snippet of Fig. 3. The only addition to the standard OpenCL API calls is the call which associates a content size buffer with a data buffer (`clSetContentSizeBufferPOCL`), and the addition of this “size buffer” to the kernels’ arguments.

4 Latency and Scalability Results

The following subsections describe the experiments performed to measure the latency and scalability of the *PoCL-R* runtime and the results obtained. In order

```
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();
```

Fig. 3: 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.

to more accurately measure the performance overhead of *PoCL-R*, wired network connections were preferred. In real-world use, client connections would generally be wireless and introduce network-dependent latency and jitter.

4.1 Command Overhead

Since low latency is a key priority of *PoCL-R*, we constructed a synthetic benchmark to measure the overheads imposed by the runtime itself using a kernel that simply exists. Some runtimes don't handle this well but it is a good indicator for command handling overhead. 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 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. Time stamps 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.

Two machines with a Ryzen Threadripper 2990wx CPU and two Geforce 2080 Ti GPUs each were used for testing. The machines were connected to a 100Mbit LAN.

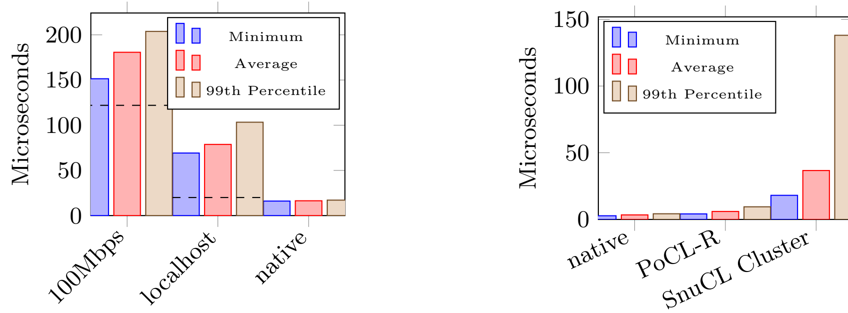
The results of this test are shown in Fig. 4a. For reference, the ICMP round-trip latency as reported by the `ping` utility fluctuates around 0.122 ms. On localhost the ICMP round-trip latency was measured to average at 0.020 ms. The average command duration was observed to be consistently around 60 microseconds more than ping. We consider this to be 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 our 100-Mbps LAN 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. Running the application and server on the same machine confirms that the overhead is constant.

The closest related work that we could successfully make run and benchmark against was the latest version (1.3.3) of SnuCL [18] (released in 2015). SnuCL has a similar idea to *PoCL-R* but seems to focus more on datacenter-side throughput scalability. In order to compare *PoCL-R* imposed minimum runtime latencies to SnuCL, a simple passthrough kernel that simply copies its single integer input to an output buffer was implemented. Kernel runtimes as reported by the OpenCL event profiling API were measured for three setups of interest: The proprietary NVIDIA driver used without any distribution layer, the SnuCL Cluster implementation and *PoCL-R*. The runtime differences here are indicative of internal command management overhead of the respective frameworks on top of the native driver and the additional overhead imposed by the MPI runtime in case of SnuCL. The results of this benchmark as shown in Fig. 4b put *PoCL-R* noticeably ahead of SnuCL with the average command duration in *PoCL-R* being only around $\frac{1}{6}$ of SnuCL's. In comparison to running without a distribution layer, *PoCL-R* takes almost twice as long, indicating some room for improvement.

4.2 Data Migration Overhead

The authors of SnuCL report data movement being the bottleneck in some of their benchmarks [18]. 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 servers communicate directly with each other in a P2P fashion: the host application only has to send a migration command to the source server which in turn forwards the command to the destination server.

The test triggers 1000 migrations between remotes and averages the durations at the end. A buffer of 4 bytes is used to minimize the effect of transfering the actual contents and better measure runtime overhead. 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



(a) No-op command on different network connections. Dashed line represents ping.

(b) Passthrough kernel on different OpenCL runtimes.

Fig. 4: Comparison of runtime duration of a no-op command in various network conditions.

timing information was analyzed. The results are shown in Fig. 5. When using a 100-Mbps ethernet connection between the remote servers 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 server, to the second server and back to the host) with extra buffer management on the intermediate hops.

Using an 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 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 native OpenCL implementation used by the daemon turned out to exhibit a notable performance regression when using two GPUs simultaneously instead of just one, making this configuration impossible to compare. A comparison with SnuCL was attempted, but calling *clEnqueueMigrateMemObjects* consistently resulted in a segmentation fault.

Two machines with an AMD Ryzen Threadripper 2990wx CPU and two NVIDIA Geforce 2080 Ti GPUs each were used for testing. The machines were connected to a 100Mbit LAN and had an additional direct 40Gbit infiniband link between them.

4.3 Distributed Large Matrix Multiplication

For a non-trivial throughput scalability benchmark, we constructed a distributed matrix multiplication application. 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

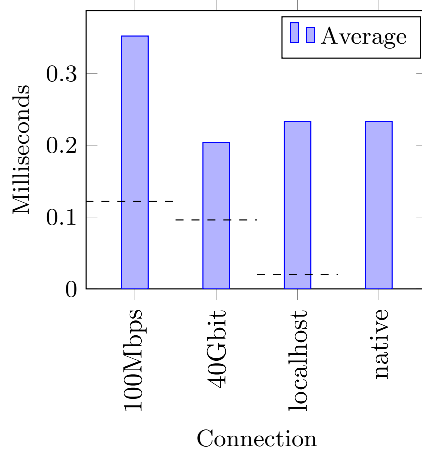


Fig. 5: Duration of a migration of a 4-byte buffer between two devices using different connectivity between servers, as well as using the native NVIDIA driver for reference. Numbers are averaged across 1000 migrations. The dashed line represents the average ICMP ping for the given connection.

rows of the output matrix. Five independent multiplications are run in parallel in order to keep all GPUs saturated and demonstrate total throughput. While the actual calculations are 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 [18] 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 [18] only measures the duration of the actual compute kernel invocations, which corresponds to the device-measured timings in our benchmark. It is unknown if the time to combine the partial results was accounted for in the SnuCL benchmark, but given that they report scalability problems it likely was part of the measurements.

Benchmarking was done on a cluster with three servers with an *Intel™ Xeon™ E5-2640 v4* CPU and four *NVIDIA Tesla P100* GPUs. An additional server 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 servers 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 6. We observe logarithmic speedups compared to using a single GPU up to slightly below 6x with 16 GPUs. This is roughly in line with the results reported in [18] with the version of SnuCL that

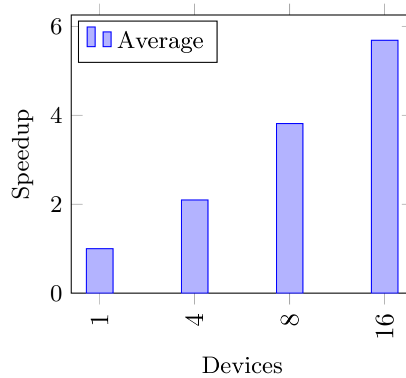


Fig. 6: Relative speedup when multiplying two 8192x8192 matrices using 1 to 16 remote devices spread across 4 servers.

uses their proposed MPI collective communication extensions. Our implementation also doesn't exhibit the performance regression suffered by the unextended P2P version of SnuCL when using more than 8 devices.

5 Real-time Point Cloud Augmented Reality Rendering Case Study

In this section, we describe a full application task offloading case study, a smartphone application [21] that renders a streamed animated point cloud in augmented reality (AR). Fig. 7 shows the application in action. The point cloud is received as an HEVC-encoded [22, 28] VPCC (Video-based Point Cloud Compression) stream [8] which is decompressed using the mobile device's hardware decoder and reconstructed using OpenGL [30] shaders [26]. A more in-depth explanation of this process is given in [24].

Visual quality can be greatly improved by using alpha blending to hide point boundaries, but this requires sorting the points by distance to the viewer, which is a costly operation and a prime candidate for remote offloading. When offloading is enabled, the VPCC stream is sent to both the device and directly to the remote compute server and decoding and point reconstruction are performed on both. However the point sorting is only done on the remote and the sorted point indices are sent back to the mobile device for rendering.

The remote daemon makes use of the OpenCL 1.2 custom device type feature to provide a virtual device that exposes the server's video decoding capabilities using VDPAU 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 proposed dynamic buffer size extension can optionally be used to speed up transfers of the

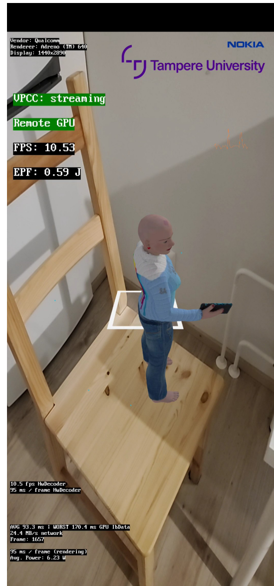


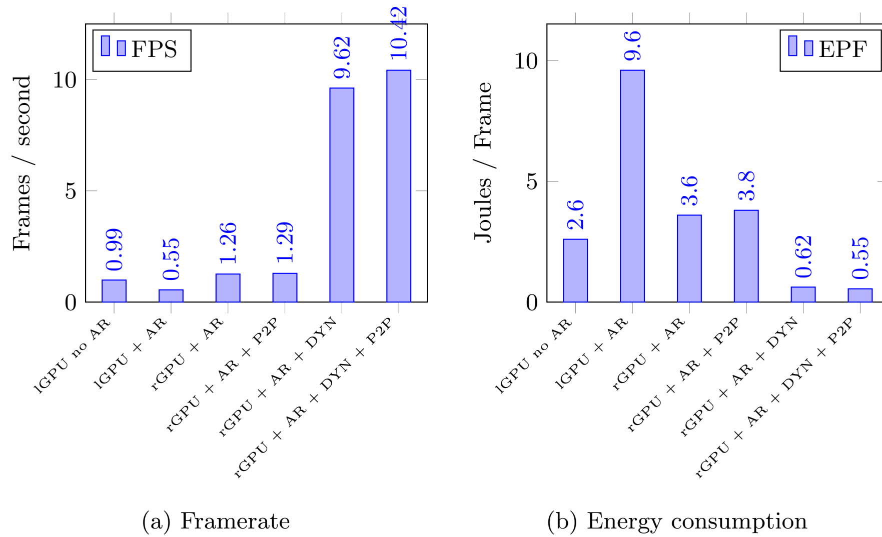
Fig. 7: Screenshot of the AR application used to measure the effect of offloading heavy computation. A streamed animated point cloud of a person holding a small tablet device is displayed in augmented reality on top of a real-world chair.

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 worst case.

Framerates measured from the application are shown in Fig. 8a. The first two measurements are obtained using the local (mobile) GPU only for reconstruction sorting and AR positioning. The next two measurements offload point sorting to a GPU on a *PoCL-R* remote server 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 an almost 19x speedup when using the dynamic buffer size extension.

Fig. 8b shows energy consumption per frame (EPF) measured on the mobile device in the same offloading configurations.The power usage of the smartphone was retrieved using Android's Power Stats HAL interface. Offloading the sorting of the point cloud compensates for most of the added energy consumption from AR positioning even without further optimizations.Enabling P2P buffer transfers and the content size extension further cuts energy consumption per frame to a mere fifth of the non-AR case. Overall the results point to *PoCL-R* being a powerful enabler for rendering advanced content on handheld devices.

Testing was done on a PC with an Intel Core i7-6700 CPU and a NVIDIA GeForce 1060 3GB GPU that was connected to an ASUS ROG Rapture GT-AX11000 WiFi6 router via gigabit ethernet. The mobile device used was a Sam-



(a) Framerate (b) Energy consumption

Fig. 8: Performance of the AR demo application in various offloading configurations. lGPU and rGPU refer to the mobile device's local GPU and the remote GPU exposed via *PoCL-R*. AR indicates live position tracking. 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.

sung Galaxy S10 SM-G973U1 with a Qualcomm® Snapdragon™ 855 chipset. The streaming data source emulated a camera feed by looping a prerecorded stream from a file.

6 Related Work

Multiple projects [14, 20, 23, 31] have expanded the scope of originally single server 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 [1–4, 14] also solely target HPC clusters with their existing library ecosystem and optimize purely for throughput.By contrast, our proposed runtime targets to support both compute clusters and realtime applications, and most interestingly, their combination.

Among the previous projects we found, the closest to *PoCL-R* is SnuCL [18]. It provides an implementation of the standard OpenCL API that enables execution of OpenCL commands on remote servers. 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.SnuCL also handles command scheduling on the host machine, whereas *PoCL-R* lets remotes do their scheduling autonomously.

Further work on SnuCL also exists in the form of SNUCL-D [17], which further decentralizes computation by duplicating the control flow of the entire host program on each remote server. This results in great scalability improvements in theory, but requires the host application to be fully replicable on all servers which is naturally not possible by default.

Another very close project in terms of the overall idea is rCUDA [5]. 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 portability.

There is also a recent open source project by the name RemoteCL [6] 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.

In a wider point of view, when used for accelerating graphics rendering of interactive content, *PoCL-R* could be thought of as an alternative to already commercialized *game streaming* services such as *Google Stadia*. The key difference is that when using *PoCL-R* for rendering acceleration, the use cases can be more flexible and adaptable to the available resources: A lightweight client device can render content using slower local resources and opportunistically exploit edge servers to improve quality instead of rendering exclusively on the server.

7 Conclusions and Future Work

In this paper we proposed a scalable low-latency distributed heterogeneous computing runtime *PoCL-R* which is based on the standard OpenCL API's features. We also proposed an API extension that significantly improves buffer transfer times for cases with varying data sizes. The unique latency and scalability enhancing features were tested with a distributed real-time augmented reality case study which reached 19x improvement in FPS and 17x in EPF by remote offloading a rendering quality enhancement kernel using the runtime. The remote kernel execution latency overhead was measured to be at 60 microseconds while the scalability at multi-server multi-GPU cluster level was shown with a logarithmic scaling of a distributed large matrix multiplication. These results indicate the significance of the proposed runtime as an enabler for high-performance low power distribution of computation and application deployment without needing additional distribution API layers.

In the future, we will research various low hanging fruits for improving the performance of the runtime further, e.g., by transparent and intelligent use of RDMA [25], GPUDirect [19] and similar technologies for improving cross-server

and cross-GPU data transfer latencies. We will also investigate improvements to dynamic multi-user scheduling and load balancing such as the approaches described in [27] and [11]. Wireless networks can be unreliable for various reasons, so we will add handling for network instability.

Acknowledgements 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.

References

1. Alves, A., Rufino, J., Pina, A., Santos, L.P.: clOpenCL-supporting distributed heterogeneous computing in HPC clusters. In: European Conference on Parallel Processing (2012)
2. Alves, R., Rufino, J.: Extending heterogeneous applications to remote co-processors with rOpenCL. pp. 305–312 (09 2020). https://doi.org/10.1109/SBAC-PAD49847.2020.00049
3. Barak, A., Shiloh, A.: The VirtualCL (VCL) cluster platform (2013)
4. Diop, T., Gurfinkel, S., Anderson, J., Jerger, N.E.: DistCL: A framework for the distributed execution of OpenCL kernels. In: 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (2013)
5. Duato, J., Peña, A.J., Silla, F., Mayo, R., Quintana-Ortí, E.S.: rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In: 2010 Int. Conf. on High Performance Computing Simulation (June 2010)
6. Ferreira, P.O.: RemoteCL. https://github.com/silverclaw/RemoteCL, accessed: 2020-10-16
7. Furuhashi, S.: Messagepack: It's like json. but fast and small. https://msgpack.org/, accessed: 2020-10-19
8. Group, D., et al.: Text of ISO/IEC CD 23090-5: Video-based point cloud compression. ISO/IEC JTC1/SC29/WG11 Doc. N18030
9. Inc., G.: Flatbuffers. https://google.github.io/flatbuffers/, accessed: 2020-10-19
10. Intel: oneAPI Specification. https://spec.oneapi.com/versions/1.0-rev-1/oneAPI-spec.pdf, accessed: 2020-10-16
11. Iserte, S., Prades, J., Reaño, C., Silla, F.: Improving the management efficiency of gpu workloads in data centers through gpu virtualization. Concurrency and Computation: Practice and Experience **33**(2), e5275 (2021)
12. Jääskeläinen, P., Korhonen, V., Koskela, M., Takala, J., Egiazarian, K., Danielyan, A., Cruz, C., Price, J., McIntosh-Smith, S.: Exploiting task parallelism with OpenCL: A case study. Journal of Signal Processing Systems **91** (2019)
13. Jääskeläinen, P., de La Lama, C.S., Schnetter, E., Raiskila, K., Takala, J., Berg, H.: pocl: A performance-portable OpenCL implementation. Int. Journal of Parallel Programming **43**(5) (2015)

14. Kegel, P., Steuwer, M., Gorlatch, S.: dOpenCL: Towards a uniform programming approach for distributed heterogeneous multi-/many-core systems. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (2012)
15. 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
16. Khronos® SYCL™ Working Group: SYCL™ Specification. https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf, accessed: 2020-10-16
17. Kim, J., Jo, G., Jung, J., Kim, J., Lee, J.: A distributed OpenCL framework using redundant computation and data replication. SIGPLAN Not. **51**(6) (Jun 2016)
18. Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., Lee, J.: SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In: Proceedings of the 26th ACM international conference on Supercomputing. pp. 341–352 (2012)
19. Li, A., Song, S.L., Chen, J., Li, J., Liu, X., Tallent, N.R., Barker, K.J.: Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. IEEE Transactions on Parallel and Distributed Systems **31**(1) (2020)
20. Liang, T.Y., Lin, Y.J.: JCL: an OpenCL programming toolkit for heterogeneous computing. In: International Conference on Grid and Pervasive Computing. pp. 59–72. Springer (2013)
21. Nokia Technologies Ltd.: Video point cloud coding (V-PCC) AR demo. https://github.com/nokiatech/vpcc/, accessed: 2020-10-16
22. Rec, I.: H. 265 and ISO/IEC 23008-2: High efficiency video coding (HEVC) (2013)
23. Reynolds, C.J., Lichtenberger, Z., Winter, S.: Provisioning OpenCL capable infrastructure with infiniband verbs. In: 2011 10th International Symposium on Parallel and Distributed Computing. IEEE (2011)
24. Schwarz, S., Pesonen, M.: Real-time decoding and AR playback of the emerging MPEG video-based point cloud compression standard. IBC 2019, Helsinki, Finland.
25. Shpiner, A., Zahavi, E., Dahley, O., Barnea, A., Damsker, R., Yekelis, G., Zus, M., Kuta, E., Baram, D.: RoCE rocks without PFC: Detailed evaluation. In: Proceedings of the Workshop on Kernel-Bypass Networks. KBNets '17 (2017). https://doi.org/10.1145/3098583.3098588
26. 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
27. Soldado, F., Alexandre, F., Paulino, H.: Execution of compound multi-kernel opencl computations in multi-cpu/multi-gpu environments. Concurrency and Computation: Practice and Experience **28**(3), 768–787 (2016)
28. Sullivan, G.J., Ohm, J.R., Han, W.J., Wiegand, T.: Overview of the high efficiency video coding (HEVC) standard. IEEE Transactions on circuits and systems for video technology **22**(12), 1649–1668 (2012)
29. Taleb, T., Samdanis, K., Mada, B., Flinck, H., Dutta, S., 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)
30. 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
31. Xiao, S., Balaji, P., Zhu, Q., Thakur, R., Coghlan, S., Lin, H., Wen, G., Hong, J., Feng, W.c.: VOCL: An optimized environment for transparent virtualization of graphics processing units. In: 2012 Innovative Parallel Computing (InPar) (2012)