2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 979-8-3503-1199-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPSW59300.2023.00020

Remote Execution of OpenCL and SYCL Applications via rOpenCL

Rui Alves
Instituto Politécnico de Bragança
Campus de Santa Apolónia
5300-253 Bragança, Portugal
rui.alves@ipb.pt,
ORCID 0000-0003-4128-8779

José Rufino
Research Centre in Digitalization and Intelligent Robotics (CeDRI)
Laboratório para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC)
Instituto Politécnico de Bragança, Campus de Santa Apolónia
5300-253 Bragança, Portugal
rufino@ipb.pt, ORCID 0000-0002-1344-8264

***Abstract*—With the increasing computational demands of modern applications, heterogeneous systems continue to have an important role in accelerating computationally intensive tasks, a trend confirmed by the most recent HPC architectures. Efficiently exploiting these systems implies the use of specific programming paradigms, such as the classic OpenCL model, or modern single-source alternatives, like SYCL. However, the original execution model of these approaches does not provision for the use of co-processors other than those directly attached to the host system where the heterogeneous application starts. Over time, several solutions emerged to cope with this limitation, both at the hardware and software level, allowing to exploit remote/distributed co-processors. In this paper, a representative set of seminal OpenCL API Forwarders is revisited and their performance compared with rOpenCL (a recently introduced platform of the same kind), using the classical matrix multiplication case study. In addition, given the importance of SYCL, which has been steadily gaining traction, this paper also exploits the potential of rOpenCL in supporting SYCL applications that use remote accelerators. To that end, another set of benchmarks is used, with both OpenCL and SYCL implementations, allowing not only to gather insight into the performance trade-offs of local versus remote (via rOpenCL) execution, but also about the current performance differential between the two programming models.**

***Index Terms*—HPC, Heterogeneous Computing, API Forwarders, OpenCL, SYCL**

I. Introduction

Nowadays, technologies such as Cloud Computing and IoT allow to optimize many day-to-day routines [1], [2], making it possible to transform complex and time-consuming activities into increasingly simple and faster tasks. Being one of the technological trends for the next decade [3], a part of this optimization is derived from the use of Artificial Intelligence (AI) techniques (e.g., Machine Learning, Deep Learning) [4], making AI an important pillar in many areas of modern life and science, such as autonomous navigation, cibersecurity, data science, medicine and physics, to name a few [5], [6].

However, the level of practicality and sophistication of all these technologies typically builds on a considerable amount of backend computing power. Efficient training of complex AI models, for instance, often requires substantial computing resources [7], a requirement that has been growing in the construction of AI-based solutions and tools [8]. Fundamental and applied research also depend, increasingly, on the availability of high performance computing technologies that allow to simulate or solve hard problems (some even previously considered untractable), and to shorten the time-to-market.

Meanwhile, this continuous demand for computational power solidified the role of heterogeneous architectures as a way to accelerate processing beyond the limits of traditional multi-core systems. In heterogeneous architectures, co-processors like GPUs, FPGAs and DSPs, cooperate with the host CPUs, assuming the execution of certain hot-spot tasks (kernels) that are both highly parallelizable and match accelerators architectural features (e.g., SIMD operation).

Since its emergence, more than a decade ago, these architectures became widespread, being present not only in the HPC space (having a crucial role in the path leading to the 1st exascale system [9]), but also on cloud computing datacenters [10] (augmenting virtual machines performance via attached GPUs or FPGAs), personal computers (where CPU-integrated or discrete GPUs accelerate everyday tasks), handheld devices (with heterogeneous SoCs that include Audio/Video and AI co-processors) and embedded systems [11], [12].

From a programmatic point of view, exploiting heterogeneous computing systems implies that applications are developed using specific programming paradigms, like those specified by the vendor-agnostic standards OpenCL [13] and SYCL [14] (see Section II). However, the original application model of these approaches assumes that heterogeneous applications are meant to use only local co-processors, i.e., accelerator devices directly attached to the host system. Thus, no provision was made, when designing those models, to transparently take advantage of remote accelerators, installed in other network accessible hosts (e.g., a common scenario in modern HPC clusters). Granted, a performance penalty will be introduced by the network layer. However, depending on the Compute/IO-bound nature of the heterogeneous application, this impact may be counterbalanced by the speedup provided by the remote co-processors, specially when there are none

The authors are grateful to the Foundation for Science and Technology (FCT, Portugal) for financial support through national funds FCT/MCTES (PIDDAC) to CeDRI (UIDB/05757/2020 and UIDP/05757/2020) and SusTEC (LA/P/0007/2021).

979-8-3503-1199-0/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPSW59300.2023.00020

local (and even with local co-processors available, it may pay off to offload some of the workload to remote accelerators).

Preserving the original heterogeneous programming models, while introducing the necessary extensions and features to provide applications with access to remote accelerators, has been a research focus in the heterogeneous computing field, with several solutions having been proposed in the past few years (see Section III). The focus of this work, rOpenCL [15], is one of the most recent (and still in development), allowing not only OpenCL, but also SYCL applications, to use remote co-processors with a high level of transparency and a minimal performance impact for Compute-bound applications.

In this paper, a selection of seminal OpenCL API forwarders is revisited and its performance is compared with that of rOpenCL, using a classic matrix multiplication test. Additionally, and equally important (if not more), the ability of rOpenCL to support the remote execution of SYCL code is demonstrated. This is, perhaps, the main contribution of this paper, given the growing relevance of SYCL, as a high-level single-source approach to the programming of heterogeneous systems, that seems poised to become a *de facto* standard [16]. Noteworthy, that ability is shown using a set of reference benchmarks for which both OpenCL and SYCL implementations exist; this duality also allow to gain insight about the performance tradeoffs of using one programming approach versus the other, complementing other studies of the same kind, but which have focused only on local executions [17].

The rest of the paper is organized as follows: section II is a brief overview of the OpenCL and SYCL programming models for heterogeneous systems; section III revisits several approaches for the remote execution of OpenCL and SYCL code; section IV provides the results of a comparison between rOpenCL and other OpenCL forwarders, as well as the results obtained from the execution, on top of rOpenCL, of some OpenCL and SYCL benchmarks; section V concludes the paper and defines some directions for future work.

II. Open Heterogeneous Programming

Depending on the level of integration between the code that targets the host CPUs and the code to be run in co-processors, the programming models for heterogeneous applications may be classified in two broad categories: single-source and non single-source. In the 1st category, the host code and the co-processors code are tightly integrated in the same source code file; this single-source approach is considered high-level, offering a greater degree of abstraction, and depending exclusively on compilers to automatically generate all the low-level data structures and operations necessary to exploit the co-processors. In the 2nd category, an heterogeneous application explicitly unfolds into separated code components, that target the different architectures available in the heterogeneous system; this non single-source approach is considered low-level.

OpenCL [13] and SYCL [14] are paradigmatic examples of these categories: OpenCL is a low-level non single-source approach, and SYCL is a high-level single-source approach. Both were proposed by the Khronos Group, a non-profit consortium of IT organizations, a few years ago (OpenCL and SYCL initial releases were in 2009 and 2014, respectively).

1) OpenCL:

In OpenCL, co-processors are aggregated by *platform*, a particular implementation of the OpenCL standard, usually provided by co-processor vendors. Currently, the major OpenCL platforms are provided by NVIDIA [18] for its GPUs, AMD [19] for its GPUs and CPUs, and Intel [20] for its CPUs, GPUs and FPGAs. There are also vendor-independent platforms, like POCL [21], targeting CPUs, GPUs and other accelerators. OpenCL applications are developed in ISO C99 (or via C++ bindings), with some restrictions and extensions for the compute kernels. The typical application workflow includes low-level steps and operations like querying the *devices* (accelerators) of the local platforms, creating *contexts* associated with those devices, creating *command queues* to interact with those devices through those contexts, building the kernel-program for the devices, creating *buffers* to hold input/output data for the kernels, and en-queuing commands for kernel execution, synchronization, and data exchange. Moreover, adding up to these steps, the programmer needs to know and consider the specifics of the memory hierarchy of the accelerators used, in order to fully take advantage of their performance potential.

2) SYCL:

By opposition, SYCL provides a higher level of abstraction, being a framework based on C++ [22] and focused on productivity and ease of use. SYCL applications can use most of the techniques of conventional C++ applications (e.g., inheritance, templates, etc.). The reduction of programmatic effort and increased flexibility happens because the specification [23] extends the concepts found in OpenCL, in various ways, beyond the general use of C++ features: i) execution of parallel kernels is done in a more convenient way, with prioritization of common parallel patterns with simple syntax; ii) by using buffers and accessors, data access in SYCL is separate from data storage; some C++ features can identify data dependencies between blocks of device code, allowing data control, without the complexity of manually managing event dependencies between kernel instances and without the programmer having to move data explicitly; iii) a unified shared memory provides a mechanism for explicit allocation and movement of data; this approach allows for the use of pointer-based algorithms and data structures allowing increased reuse of code between host and device. Being SYCL a specification still in development, there are already several implementations, with different degrees of maturity. These include Codeplay ComputeCpp [24] (used in the experimental component of this workd), Intel oneAPI Data Parallel C++ (DPC++) [25] and hipSYCL [26], among some others.

3) oneAPI:

Another open standard for the programming of heterogeneous applications, which has gained some momentum, is oneAPI [27], promoted by Intel. Although it is primarily considered a high-level unified application programming interface, supporting the use of many different types of accelerators, it still offers the programmer the ability to resort to a lower-level

approach. An important feature [28] of oneAPI is the support for two portable heterogeneous programming methods: Data Parallel C++ with SYCL and OpenMP for C, C++ and Fortran.

III. Support for Remote Execution

One common characteristic shared by the the heterogeneous programming approaches discussed in the previous section is that their original execution model only allows local accelerators to be used, that is, co-processors directly connected to the host where the heterogeneous application is started, thus limiting the potential for acceleration. This is because the number of co-processors that can be linked to a single system, even in more sophisticated scenarios, where proprietary interconnects are used, is also limited [29].

To overcome this limitation, technologies at the driver and hardware levels were developed, to allow direct communication between co-processors (specially GPUs) on different networked nodes, bypassing the Main Memory of those nodes. This communication is based on the Remote Direct Memory Access protocol (RDMA), usually over an Infiniband network, with NVIDIA's GPUDirect-RDMA [30] and AMD's ROCn-RDMA [31] being well-known examples of this approach.

The use of existing HPX backends (see section III-B2), may also contribute to overcome this limitation. HPX, however, being an asynchronous many task (AMT) oriented approach, implies rewriting heterogeneous applications to make use of that paradigm and of the features of its runtime [32], making this option harder to be adopted outside the HPC domain.

Another alternative is to maintain the original programming model of the OpenCL and SYCL approaches, but provide to the applications, through the runtime and execution environment, transparent access to remote devices, as if they were local. Surely, redirecting API calls to remote co-processors will always imply a performance penalty, introduced by the network layer, but its magnitude will mostly depend on the Compute/IO-bound nature of the heterogeneous applications: compute-bound applications, with few data exchanges between the host-side and the accelerator devices, will be those that benefit most from remote co-processors; moreover, with the increasing bandwidth and decreasing latency of modern interconnects, the impact of communication tends to decrease.

Thus, despite the fact that forwarding heterogeneous API calls for remote execution is by no means a new approach, and several solutions were indeed proposed along the years (both for the OpenCL and CUDA domains), we argue that there is still room for improvement, by enhancing their performance and supporting recent models and respective APIs, like SYCL's. In this regard, rOpenCL is one of the very few OpenCL forwarders still in development [33], now also supporting the use of remote accelerators by SYCL applications.

Next, a small set of OpenCL forwarders is revisited, with the focus on those whose performance is later compared (see section IV-A). High-level approaches, some of which allow remote execution of SYCL code, are also briefly discussed.

A. Low-Level OpenCL Forwarders

1) VCL:

One of the first approaches developed to allow the remote execution of OpenCL calls was VCL [34]. This implementation aggregates all available accelerators on a global OpenCL platform, combining not only multi-node CPUs but also GPUs and other accelerators. The communication between the various components uses TCP/IP sockets. Due to the way it was originally designed, in conjunction with MOSIX, VCL doesn't integrate into the OpenCL ICD-Loader driver management mechanism, being less transparent than rOpenCL.

2) clOpenCL:

Another of the first wave OpenCL API forwarders was clOpenCL [35]. It consisted of a library of OpenCL wrappers, present in the host system where the OpenCL applications is launched, and services, running on the remote systems with the accelerators to be exposed to the application. OpenCL applications need to be recompiled (thus requiring its source code) and linked with the clOpenCL library, in order to take advantage of the remote devices. Aiming for a fast communication between with the remote services, clOpenCL network transactions relied on Open-MX [36], a low-level communications library, close to the Ethernet layer. Like VCL, clOpenCL also doesn't have an ICD-Loader compatible driver.

3) dOpenCL:

dOpenCL [37] was the most complete and mature OpenCL forwarder of its time, considering its OpenCL API coverture and features like support for callbacks. In dOpenCL, a client driver installed on the launching host of an OpenCL application makes possible to access remote co-processors, interfaced by dOpenCL services running on their hosting nodes. However, the client driver is not fully transparent: it is not compatible with the OpenCL ICD-Loader driver manager, operating separately. The communication between the client driver and the remote services is TCP-based, but asynchronous, via the Asio C++ library [38]; this made dOpenCL network transactions faster than contemporary forwarders.

4) rOpenCL:

rOpenCL [15], [33] was developed as a successor to clOpenCL, by adopting BSD sockets for message exchanges and being fully integrated with the ICD-Loader mechanism. This ensured communication portability (due to the ubiquity of BSD sockets and TCP/IP networks), and full transparency from the point of view of OpenCL applications, which do not require any recompilation, allowing binary OpenCL code to readily exploit remote accelerators from many different nodes.

Fig. 1 shows the components and interactions of the rOpenCL architecture in its late stage. The main components are the rOpenCL Driver and the rOpenCL Service(s). The rOpenCL Driver conforms to the Installable Client Driver (ICD) [39] mechanism, that allows the coexistence, in the same system, of multi-vendor OpenCL implementations, also known as OpenCL platforms. This way, rOpenCL appears to OpenCL applications as just another platform, but now giving access to remote devices. At each remote node, a multi-threaded instance of a rOpenCL Service attends and redirect multiple

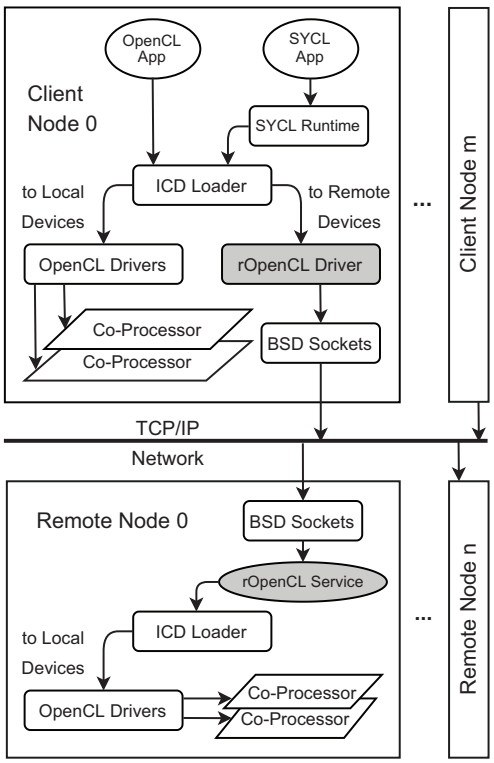


Fig. 1. rOpenCL components and interactions.

OpenCL calls, towards the underlying devices, coming from the same or different client nodes. For further details see [33].

Compared to its initial version [15], the main new feature of rOpenCL is the ability to support the execution of SYCL code, which is therefore able to take advantage of remote co-processors. This is possible because some implementations of SYCL (e.g. CodePlay ComputeCpp [24]) use, at a lower level, the multi-vendor OpenCL implementations configured on the host system. The rOpenCL Driver appears as another of them, and thus SYCL applications readily make use of it.

Supporting the remote execution of SYCL code implied some enhancements in rOpenCL, namely full support for callbacks (pertaining to the *clSetEventCallback* function), and for asynchronous reading/writing (related to the *clEnqueueWriteBuffer* and *clEnqueueReadBuffer* functions). Moreover, many optimizations were applied in the network transactions code, with tangible gains, especially for OpenCL/SYCL applications that involve transfers of large amounts of data between the host-side of the application and the co-processors.

5) Other OpenCL Forwarders:

Other significant contributions to the OpenCL forwarders ecossystem have emerged over time, like Hybrid OpenCL [40], SnuCL [41] and, more recently, clusterCL [42] and POCL-R [43]. Still, none has its source code or executable currently available in the public domain, thus precluding its inclusion in the performance comparison presented in section IV-A.

POCL-R, though, deserves a special mention, once it has been in active development. It shares with rOpenCL a similar architecture, but appears focused on the distribution of workloads in edge clusters, which reflects on its design. For instance, to reduce the time spent in network transactions, POCL-R enables data exchange between existing remote servers, avoiding round-trips back to OpenCL clients whenever synchronization or data transfers are required between remote devices. Additionally, it provides some extensions of the OpenCL standard (e.g *clSetContentSizeBufferPOCL*), optimizing the exchange of data between its components.

B. SYCL and Other High-Level Approaches

Although SYCL is a high-level (single-source) approach, depending on the implementation used, SYCL applications build on low-level data structures and operations often expressed in CUDA or OpenCL. In principle, this should allow forwarders, like rCUDA [44] or those discussed in section III-A, to be used as SYCL backends that enable access to remote co-processors. However, in opposition to rOpenCL, not all forwarders provide the required level of transparency to be readily used as SYCL backends. Thus, to allow SYCL applications to exploit remote co-processors, other types of approaches have been pursued. Some are next presented.

1) Celerity:

Celerity [45] is an API which extends the SYCL specification, allowing parallelization of SYCL applications in heterogeneous computing clusters. Briefly, the runtime system of Celerity can be described as a multithreaded application, built on MPI and SYCL, which uses the master/worker execution model. The master node is responsible for scaling and distributing work to the worker nodes. Each node worker encapsulates the available accelerators and executes the commands it receives from the master asynchronously. Despite the performance gains, developing SYCL applications using Celerity implies using its own specific SYCL wrappers; this prevents the direct execution of pre-compiled SYCL applications.

2) HPX:

HPX [46] is a library of C++ patterns for concurrency and parallelism, that follows the asynchronous many task paradigm, using C++ language resources and libraries to support various types of parallelism. It has a modular architecture, interconnecting with the heterogeneous computing domain through different computing backends. Currently, it is possible to develop HPX heterogeneous applications that target CUDA [47], OpenCL [48] and SYCL [49]. In order to integrate with HPX, SYCL applications must be properly modified and recompiled. Thus, like Celerity, it is not a fully transparent solution.

3) PHAST:

PHAST [50] is a high-level C++ library for single-source programming of parallel applications that exploit multi-core systems and NVIDIA GPUs (one of these targets at a time, not both). Code developed with this library is written once, using high-level constructs based on the classic STL "containers, iterators, algorithm" approach. The choice of the target architecture is made during the build phase, via a single preprocessor macro. The roadmap includes support for

code generation for OpenCL (on top of NVIDIA and CPU compilable code) and, relevant to this discussion, for multi-node operation. PHAST development, however, seems stalled.

IV. Experimental Evaluation

This section presents the results of two different sets of experiments. In the first, the performance of rOpenCL is compared with that of VCL, clOpenCL and dOpenCL, using the classic matrix multiplication example. In the second, rOpenCL is used as an OpenCL backend for several SYCL benchmarks, allowing to assess the impact of remote execution of SYCL code, and also the performance differential against the OpenCL version of the same benchmarks.

Each experiment, with a specific combination of parameters, was repeated 5 times (with a pause of 5s between consecutive runs) and the values shown are their arithmetic average. All execution times presented are *turnaround times*, covering the full execution of the benchmark program, and not only of its core. On the other hand, the bandwidth values shown in section IV-B are the ones returned by the respective benchmarks.

GPUs were used as co-processors in all experiments. The computational systems used were virtual machines of a KVM-based cluster, but the cluster was monitored to ensure the tests were conducted under lightly-loaded conditions in order to minimize the influence of the shared nature of the runtime environment (the relative standard deviation associated with each average was always low, bellow 5%).

A. Comparison with other OpenCL Forwarders

The comparison between rOpenCL and other OpenCL forwarders is based on the well-known matrix multiplication scenario (for dense matrixes), specifically in an OpenCL application already used to validate clOpenCL [51]. This example multiplies two $n \times n$ square matrices of single-precision floating point numbers. The input matrices are decomposed in slices (sub-matrices) of dimension $s \times n$ (horizontal slices of the first matrix) and $n \times s$ (vertical slices of the second matrix), which are evenly distributed by the remote GPUs (of the same model) to be multiplied there. The code of the example is multi-threaded, with one thread created to interact with each different GPU by means of the appropriate OpenCL calls.

The <*n,s*> combinations tested were <8K,1K>, <16K, 2K> and <32K,4K>, implying increasing matrix sizes (256 MB, 1 GB and 4 GB, respectively) and correspondly increasing slice sizes (32 KB, 128 KB and 512 KB, respectively), but always the same number of slices (24 slices per matrix). These values were defined to fit within the constrains of our testbed, allowing for the node where the host-side of the example runs to always be able to hold the three matrices involved in main memory, and to have enough slices to ensure an even distribution of the workload by the co-processors available.

It should be noted that because the number of slices is fixed, this evaluation scenario does not ensure the best performance achievable; in order to do so, the slices size should be defined accordingly with the number of remote co-processors and with their local memory, in order to minimize the number of slices and, at the same time, maximize the number of co-processors used (basically, with g remote GPUs, with r GB of RAM each, the slice size should be $\approx (r/3)/g$ GB, so that both input slices and the output slice could fit in the GPU memory, and all GPUs were used). However, once the same workload distribution is ensured for all OpenCL forwarders tested, their relative ranking will be preserved by the benchmark.

The test-bed for this round of experiments used 8 virtual machines (VMs), hosted in 8 different nodes of a KVM-based cluster. All VMs shared the same characteristics: 8 SMT CPU-cores of an AMD EPYC 7351 CPU; 16 GB of RAM; virtio NIC attached to a vlan segment with MTU 9000 (provided by a Mellanox ConnectX-5 100Gbps Ethernet physical NIC); Linux Ubuntu 18.04 64 bit OS with kernel 4.8 (to ensure a compatible environment with clOpenCL, that requires a specific kernel for its Open-MX communication layer). Four of the VMs were attached, via PCIe pass-through, to one NVIDIA RTX 2080 Ti GPU (11 GB, driver 430.50) each.

Two different scenarios were tested: i) one client using a single local GPU; ii) one client using a varying number of remote GPUs. Their results are next presented and discussed.

1) One client using a single local GPU:
The goal of this first scenario is to provide an evaluation baseline to measure the impact of the access to remote GPUs in the second scenario. Thus, in this scenario, a single local GPU is loaded by a single-threaded matrix multiplication OpenCL client. The average execution times for the three <*n,s*> combinations tested are given in Table I.

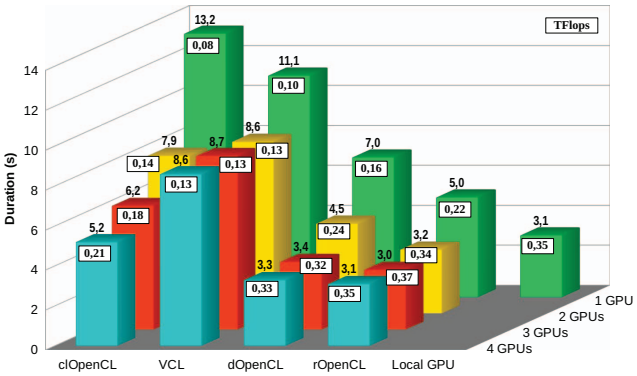
TABLE I
Matrix Multiplication benchmark - one client using one local GPU: Execution Time (s) and Data Transferred (MB)

<*n,s*>	<8K,1K>	<16K,2K>	<32K,4K>
Execution Time (s)	3.1	11.7	59.9
Data Transferred (MB)	768	3072	12288

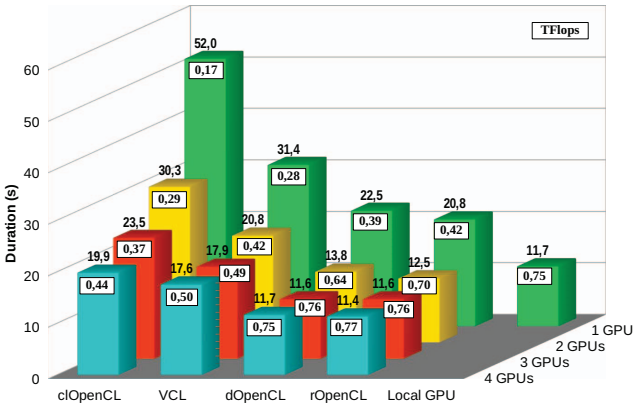
As already stated, the slice size is not optimized to minimize the data exchanges, meaning the times shown in Table I are not the best attainable in the testbed used. But once this contingency also affects remote executions, a comparison with this baseline is considered fair. The same table also shows the overall amount of data transferred for each <*n,s*> combination. This amount and the overall execution time increase in similar proportions, as would be expected.

2) One client using many remote GPUs:
In this scenario, the goal is to assess how the matrix multiplication benchmark scales with the OpenCL forwarders clOpenCL, VCL, rOpenCL and dOpenCL, when a single OpenCL client executes the benchmark with an increasing number of remote GPUs (from 1 to 4, with each GPU in a separate host), for the 3 combinations of matrix sizes and slices previously defined.

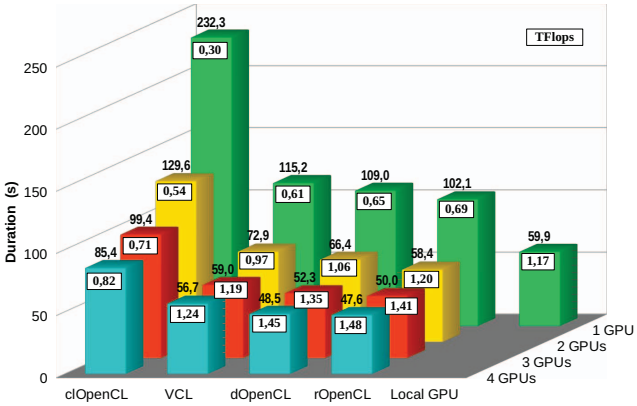
The results are shown in the charts of Figure 2 (one chart per <*n,s*> combination). The execution time values are shown on top of each column and are represented in the z axis, the OpenCL platforms tested (forwarders and local) are represented in the x axis, and the different number of GPUs



(a) <*n,s*> = <8K,1K>



(b) <*n,s*> = <16K,2K>



(c) <*n,s*> = <32K,4K>

Fig. 2. Matrix Multiplication benchmark - one client using one local GPU vs many remote GPUs: Duration and FLOPS.

used varies along the y axis. The same charts also include the baseline scenario ({Local GPU} $\times$ {1 GPU} combination), thus providing a perspective on the performance impact brought by the remote execution compared to a purely local execution (albeit with a single local GPU). The corresponding number of floating point operations per second (measured in TFlops) is also shown (numbers inside white rectangles).

An analysis of the charts identifies some general trends and specific situations that deserve to be mentioned: i) as expected, with more GPUs used, there is a trend for the execution time to decrease with all forwarders, although with progressively diminishing gains; ii) the biggest performance gain happens, in all forwarders, when moving from 1 to 2 remote GPUs used; iii) with $\geq$3 remote GPUS, dOpenCL and rOpenCL match or outperform the Local GPU scenario; this also happens with VCL, but only for the biggest problem size tested (<*n,s*> = <32K,4K>) which hints at a good VCL scalability; iv) rOpenCL outperforms the other forwarders in all scenarios.

Focusing only on rOpenCL and dOpenCL, which consistently exhibited the best performance, Table II shows their speedups over the baseline provided by the single local GPU scenario. Thus, replacing one local GPU by a single remote one (see speedups marked with (*)) roughly implies doubling the execution time. Also, only with the biggest problem size (<*n,s*> = <32K,4K>) did it pay off to use remote GPUs, and it was necessary to use at least 3 GPUs, to achieve 15% to 26% extra performance (see speedups in bold). Noteworthy, this analysis is only valid for our matrix multiplication scenario, and the conclusions will vary with other benchmarks.

TABLE II
MATRIX MULTIPLICATION BENCHMARK - ROPENCL AND DOPENCL SPEEDUP AGAINST ONE LOCAL GPU

<*n,s*>	remote GPUs	rOpenCL	dOpenCL
<8K,1K>	1	0,62 (*)	0,44 (*)
	2	0,97	0,69
	3	1,03	0,91
	4	1	0,94
<16K,2K>	1	0,56 (*)	0,52 (*)
	2	0,94	0,85
	3	1,008	1,008
	4	1,03	1,00
<32K,4K>	1	0,57 (*)	0,55 (*)
	2	1,03	0,90
	3	**1,20**	**1,15**
	4	**1,26**	**1,24**

It is also possible to compare rOpenCL with the other forwarders by considering the overall execution time resulting from the sum of the execution time of the 12 tests performed for each forwarder (three different problem sizes, combined with four different remote GPU configurations, yields 12 different tests). Table III allows for that comparison, by providing the overall execution times of all forwarders, and the speedups of rOpenCL against them. The performance advantage of rOpenCL over clOpenCL is considerable, relatively modest over VCL, and minor versus dOpenCL ($\approx$8%). The later observation points, again, to the use of asynchronous communication, which may be responsible by this short gap.

Another possible test scenario would be to use one more local GPUs and couple them with one or more remote GPUs. This is perhaps a more realistic scenario: if one launches an heterogeneous application in one system, it is probable that there is already some local accelerators and, in this case, the goal would be to complement them with remote accelerators.

TABLE III
MATRIX MULTIPLICATION BENCHMARK - OVERALL PERFORMANCE COMPARISON OF ALL THE FORWARDERS TESTEED

Metric	rOpenCL	clOpenCL	VCL	dOpenCL
Execution Time (s)	328,62	704,86	428,36	354,15
rOpenCL Speedup	-	2,145	1,304	1,078

The case for using remote accelerators may also be justified for situations when there is simply none local co-processors and thus using remote ones is the only option to try to accelerate a workload compared to running it in a local multi-core CPU. Past experiments [52], in which both these scenarios were exploited, showed tangible gains in both situations.

B. rOpenCL as a SYCL backend

The results of a preliminary evaluation of rOpenCL that demonstrates its compatibility with the execution of SYCL applications are analyzed in this section. The tests performed prove that rOpenCL is able to sustain the complete execution of well-known OpenCL and SYCL benchmarks. In addition, the tests also allow to understand the kind of overheads introduced by having API calls forwarded and executed remotely.

The benchmarks used were BabelStream [53], Heat [54], XSBench [55] and RSBench [56]. BabelStream is a well known benchmark based on 5 kernels, allowing to measure the memory bandwidth of an accelerator. Heat is a simple benchmark that solves a simple differential equation using explicit finite differences; its main kernel applies a 5-point stencil in a 2D domain and, like in BabelStream, memory bandwidth is the main performance limiting factor. XSBench is a small application that executes a key computational kernel of the Monte Carlo neutron transport algorithm (the continuous energy macroscopic neutron cross-section lookup kernel). RSBench also targets the same Monte Carlo neutron transport algorithm as XSBench, but with an improved kernel (multipole method) that requires orders of magnitude less memory storage than the kernel used by XSBench.

These benchmarks were selected for the following reasons: i) they have comparable OpenCL and SYCL versions, allowing for a performance comparison of both implementations; ii) they stress different device subsystems (memory vs compute bound); iii) they are open-source (some instrumentation code was needed to automate the benchmarks execution); vi) they run on Linux (rOpenCL is only supported in this OS).

For this evaluation, the test-bed consisted on two virtual machines from the same KVM-based cluster used to compare the OpenCL forwarders, this time with different characteristics: 16 SMT CPU-cores of an AMD EPYC 7442 CPU; 64 GB of RAM; the same kind of 100Gbps virtual NIC; Linux Ubuntu 22.04 64 bit OS with kernel 5.15; CodePlay ComputeCPP CE 2.11.0 SYCL platfom. One of the VMs was assigned, via PCIe pass-through, either one NVIDIA RTX 4090 GPU or one NVIDIA A100 GPU (NVIDIA driver 525.60 used for both).

Although there are several SYCL platforms, and with variable performance [57], the community version of ComputeCPP was used due to its support for the use of NVIDIA GPUs as co-processors, and the stability shown in our test-bed. In the future, these tests will be repetead with other SYCL implementations, to verify if the conclusions derived still hold.

1) Memory-Bound Benchmarks: The results of the memory-bound benchmark (BabelStream and Heat) may be observed in the charts of Figs. 3 and 4. These charts show the bandwidth, as measured by each benchmark, when using a local GPU (Local Bandwidth), and when using a remote GPU through rOpenCL (Remote Bandwidth). They also present the bandwidth decrease of the last regime (Remote Deceleration, given by the ratio Local Bandwidth / Remote Bandwidth). The bandwidth is represented against the left vertical axis; the deceleration is represented against the right vertical axis.

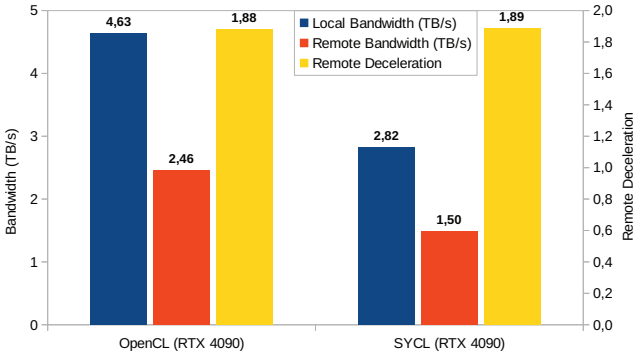
As expected, there is a noticeable decrease in the bandwidth available when accessing a remote GPU. Moreover, such decrease is more pronounced in the Heat benchmark than in the BabelStream one. A possible explanation is that Heat, being a benchmark that executes very fast ($\lesssim$ 0,25s in the OpenCL version and $\lesssim$ 1s in the SYCL version) in comparison to BabelStream ($\lesssim$ 2s in the OpenCL version and $\lesssim$ 10s in the SYCL version), it is much more sensitive to any transient conditions in the execution environment and, more important, to the impact of the network transactions of rOpenCL.

The differential in terms of the memory bandwidth capabilities of the two GPU models used in the tests is also evident, with the A100 GPU roughly showing double the bandwidth o the 4090 model in the BabelStream test, which is in agreement with the manufacturer specifications of both GPUs.

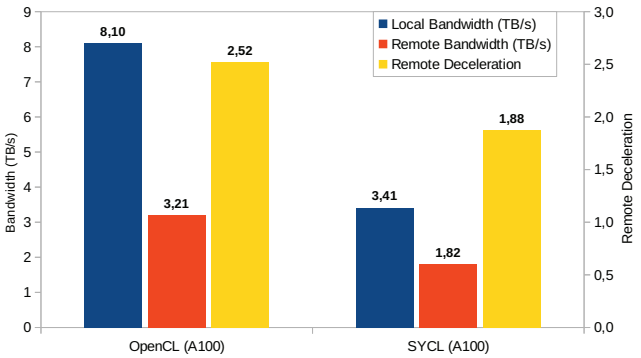
2) Compute-Bound Benchmarks: The charts of Figs. 5 and 6 show the results of the compute-bound benchmark (XSBench and RSBench). For each benchmark the charts represent the average execution time when exploiting a single local GPU ($\overline{T_l}$), and when exploiting a single remote GPU ($\overline{T_r}$). The charts also provide the speedup of the remote execution, given by $\overline{T_l}/\overline{T_r}$. The executio time is represented against the left vertical axis; the speedup is represented against the right vertical axis. An alternative metric to the speedup, used in the discussion bellow, is the overhead (in %) introduced by the remote execution, given by $(\overline{T_r}/\overline{T_l} - 1) \times 100$.

In all compute-bound tests, the remote execution was slower than the local one, as already expected, once only one remote GPU is being used. Complementary to the speedups (all bellow 1.0, typical of a deceleration), the corresponding percentual overheads of the remote execution are presented in Table IV. The overheads of the OpenCL version of the tests, in the same GPU, are always larger than the SYCL overheads (at a short distance in RSBench, but by one order of magnitude in the BabelStream and XSBench tests).

However, at the same time, the OpenCL versions of the benchmarks are always faster than the SYCL versions, as also shown in Table V. This suggests that the OpenCL versions, being low-level (with less software layers involved), are faster by nature and thus are more sensitive (in relative terms) to the penalty brought by the network transactions, which contributes to the higher overheads shown in Table IV.

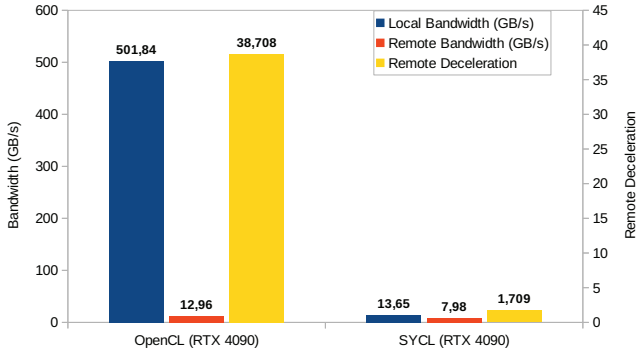


(a) OpenCL vs SYCL (RTX 4090 GPU)

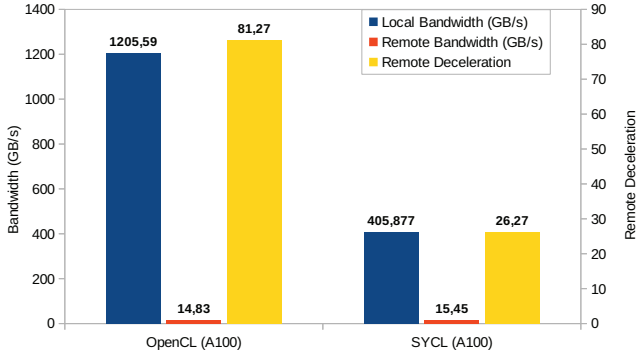


(b) OpenCL vs SYCL (A100 GPU)

Fig. 3. BabelStream results (1 local vs 1 remote GPU)



(a) OpenCL vs SYCL (RTX 4090 GPU)



(b) OpenCL vs SYCL (A100 GPU)

Fig. 4. Heat results (1 local vs 1 remote GPU)

TABLE IV
OVERHEAD OF REMOTE EXECUTION

Test	Variant	4090	A100
XSBench	OpenCL	290,2%	331,8%
	SYCL	92,7%	62,4%
RSBench	OpenCL	1,8%	8,9%
	SYCL	1,5%	7,1%

Moreover, the overheads for XSBench are much higher that those registered for RSBench; this is somehow expected, once, as already stated, the RSBench main kernel requires a much smaller amount of memory and so the impact of the buffers exchange through the network will be much less significant.

TABLE V
SPEEDUP OF OPENCL OVER SYCL

Test	GPU	Local Execution	Remote Execution
XSBench	4090	3,83	1,89
	A100	5,21	1,96
RSBench	4090	1,69	1,68
	A100	2,04	2,01

Finally, and without any intent of generalization (due to the limited scope of this study), this discussion ends by identifying which of the GPUs used (which have different targets - consumer vs datacenter) performed best in the different scenarios evaluated, and how faster performed in comparison with the other. Thus, Table VI shows that the recent RTX 4090 GPU was able to match and sometimes surpass the A100 unit, in the XSBench benchmark. However, Table VII tells the opposite story for the RSBench test, where the A100 GPU was considerably faster (up to 5x) than the RTX 4090 unit.

TABLE VI
SPEEDUP OF RTX 4090 OVER A100

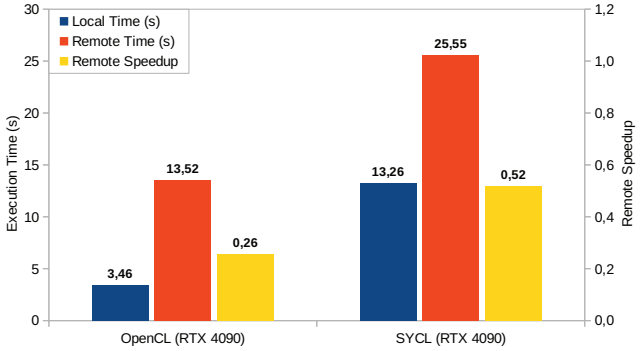
Test	Variant	Local Execution	Remote Execution
XSBench	OpenCL	0,993	1,099
	SYCL	1,352	1,139

TABLE VII
SPEEDUP OF A100 OVER RTX 4090

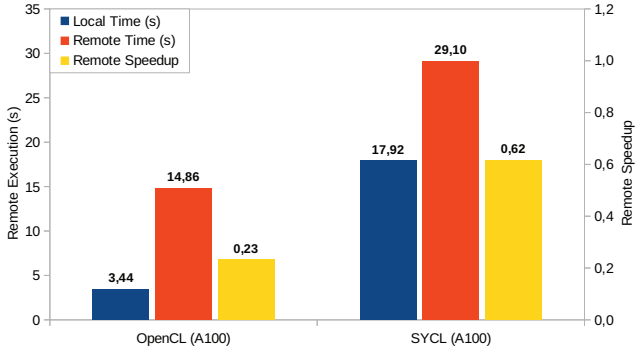
Test	Variant	Local Execution	Remote Execution
RSBench	OpenCL	5,119	4,787
	SYCL	4,235	4,012

V. CONCLUSION

Having the ability to transparently forward OpenCL calls for remote execution, via normal TCP/IP transactions, as provided by rOpenCL, opens up a wide range of possibilities for heterogeneous applications: in an HPC cluster, typically with

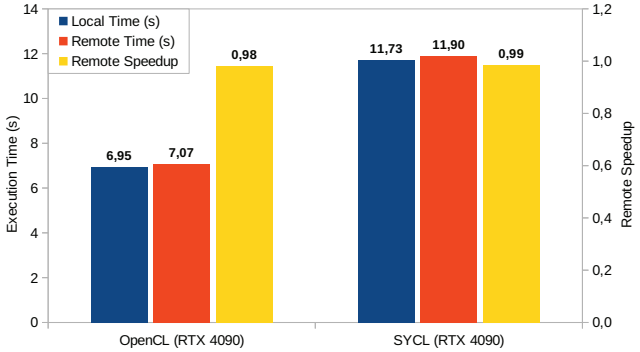


(a) OpenCL vs SYCL (RTX 4090 GPU)

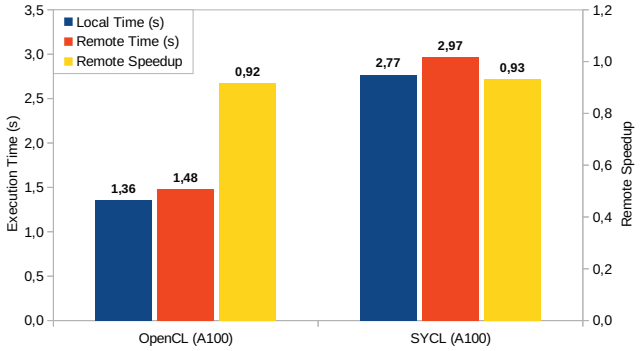


(b) OpenCL vs SYCL (A100 GPU)

Fig. 5. XSBench results (1 local vs 1 remote GPU)



(a) OpenCL vs SYCL (RTX 4090 GPU)



(b) OpenCL vs SYCL (A100 GPU)

Fig. 6. RSBench results (1 local vs 1 remote GPU)

very fast interconnects, applications may efficiently exploit the set of accelerators that are scattered by the cluster nodes, without the need to resort to hybrid programming techniques, like integration with MPI; in virtualized scenarios, a host GPU may be shared by several local virtual machines, without the need to partition the GPU using proprietary non-free drivers; edge applications, using 5G or faster next-generation mobile networks, can offload heavy tasks to accelerators in the cloud. Granted, remote execution only pays off for compute-bound applications, but many real-world use cases fit this category.

Supporting the remote execution of OpenCL calls brings with it yet another possibility, exploited in this paper: providing SYCL applications with the ability to use remote accelerators; to that end, it suffices that the underlying SYCL platform offers an OpenCL back-end, which is mostly expected.

Performance-wise, the results of the first set of benchmarks executed show that rOpenCL outperforms other forwarders. Within the second benchmark set, all OpenCL variants outperformed their SYCL counterparts (sometimes by a considerable margin), a trend already observed in other studies. The results of the various benchmarks executed also hint at the kind of overheads to be expected from remote execution, although the simplified evaluation scenario used (with only two nodes, and two GPUs) prevents any generalization. Though not yet fully compliant with the OpenCL 1.2 specification (covering around 70% of the overall API, and more than 90% of the compute-related primitives), the current rOpenCL implementation is nevertheless already able to sustain the execution of many more OpenCL and SYCL non-graphical applications and benchmarks, beyond the ones exploited in this paper.

Besides continuing to improve the performance of rOpenCL (specially at the network level) and its coverage of the OpenCL specification, in the future we plan to do tests with other SYCL platforms, perform a comparison with other OpenCL forwarders (and also solutions for remote execution of SYCL code), and tackle further (edge and cloud) and larger-scale (with more accelerators and nodes involved) scenarios.

References

[1] L. Haji, O. Ahmed, S. Zeebaree, H. Dino, R. Zebari, and H. Shukur, "Impact of cloud computing and internet of things on the future internet," *Tech. Reports of Kansai University*, vol. 62, pp. 2179–2190, 06 2020.

[2] P. D. Baruah, S. Dhir, and M. Hooda, "Impact of iot in current era," in *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, 2019, pp. 334–339.

[3] dac.digital. Top 10 tech trends for the next ten years: Is your business prepared? [Online]. Available: https://dac.digital/top-10-tech-trends-for-the-next-ten-years-is-your-business-prepared/

[4] P. R. CENTER. Artificial Intelligence and the Future of Humans. [Online]. Available: https://www.pewresearch.org/internet/2018/12/10/artificial-intelligence-and-the-future-of-humans/

[5] G. Biggi and J. Stilgoe, "Artificial intelligence in self-driving cars research and innovation: A scientometric and bibliometric analysis artificial intelligence in self-driving cars research and innovation: A scientometric and bibliometric analysis," 04 2021.

[6] S. G. Kumar, S. J. Corrado, T. G. Puranik, and D. N. Mavris, "Classification and analysis of go-arounds in commercial aviation using ads-b data," *Aerospace*, vol. 8, no. 10, 2021.

[7] K. Hao. The computing power needed to train AI is now rising 7x faster than ever before. [Online]. Available: https://www.technologyreview.com/2019/11/11/132004/the-computing-power-needed-to-train-ai-is-now-rising-seven-times-faster-than-ever-before

[8] K. Wiggers. MIT researchers warn that deep learning is approaching computational limits. [Online]. Available: https://venturebeat.com/ai/mit-researchers-warn-that-deep-learning-is-approaching-computational-limits/

[9] TOP500.org. June 2022 — TOP 500. [Online]. Available: https://www.top500.org/lists/top500/2022/06/

[10] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, August 2016, pp. 1–10.

[11] A. Ben Abdallah, "3 - heterogeneous computing: An emerging paradigm of embedded systems design," in *Computational Frameworks*, M. K. Traoré, Ed. Elsevier, 2017, pp. 61–93.

[12] W. Carballo-Hernández, M. Pelcat, and F. Berry, "Why is FPGA-GPU Heterogeneity the Best Option for Embedded Deep Neural Networks?" Feb. 2021, presented at DATE Friday Workshop on System-level Design Methods for Deep Learning on Heterogeneous Architectures (SLOHA 2021) (arXiv:2102.00818).

[13] T. K. Group. Opencl overview. [Online]. Available: https://www.khronos.org/opencl/

[14] T. K. Group. Sycl overview. [Online]. Available: https://www.khronos.org/sycl/

[15] R. Alves and J. Rufino, "Extending heterogeneous applications to remote co-processors with ropencl," in *Proceedings - Symposium on Computer Architecture and High Performance Computing*, vol. 2020-September, 2020, pp. 305–312.

[16] J. Reinders and M. Wong. Why SYCL: Elephants in the SYCL Room. [Online]. Available: https://www.hpcwire.com/2022/02/03/why-sycl-elephants-in-the-sycl-room/

[17] T. Deakin and S. McIntosh-Smith, "Evaluating the performance of hpc-style sycl applications," in *IWOCL '20*. United States: Association for Computing Machinery (ACM), Apr. 2020, international Workshop on OpenCL, IWOCL ; Conference date: 27-04-2020 Through 29-04-2020.

[18] NVIDIA. Opencl nvidia developer. [Online]. Available: https://developer.nvidia.com/opencl

[19] A. M. Devices. Amd rocm open ecosystem. [Online]. Available: https://www.amd.com/en/graphics/servers-solutions-rocm

[20] Intel. Intel sdk for opencl applications. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/opencl-sdk/overview.html

[21] P. Jääskeläinen, C. Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable opencl implementation," *International Journal of Parallel Programming*, 08 2014.

[22] C. S. Michael Wong, VP R&D. Iso c++ and sycl join for the future of heterogeneous programming. [Online]. Available: https://www.computer.org/publications/tech-news/research/iso-c-and-sycl-join-for-the-future-of-heterogeneous-programming

[23] T. K. Group. Sycl™ 2020 specification (revision 6). [Online]. Available: https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html

[24] CodePlay. Code play computecpp. [Online]. Available: https://developer.codeplay.com/home/

[25] Intel. Data parallel c++: the oneapi implementation of sycl*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/data-parallel-c-plus-plus.html#gs.nu48ba

[26] Illuhad. hipsycl - a sycl implementation for cpus and gpus. [Online]. Available: https://github.com/illuhad/hipSYCL

[27] oneAPI. onepai - collaboration encouraged. [Online]. Available: https://www.oneapi.io/spec/

[28] I. oneAPI Programming Guide. Programming guide. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top/oneapi-programming-model.html

[29] run:ai. NVIDIA DGX: Under the Hood of DGX-1, DGX-2 and A100. [Online]. Available: https://www.run.ai/guides/nvidia-a100/nvidia-dgx

[30] NVIDIA. Developing a Linux Kernel Module using GPUDirect RDMA. [Online]. Available: https://docs.nvidia.com/cuda/pdf/GPUDirect_RDMA.pdf

[31] Advanced Micro Devices. ROCnRDMA: ROCm Driver RDMA Peer to Peer Support. [Online]. Available: https://github.com/rocmarchive/ROCnRDMA

[32] T. Heller, P. Diehl, Z. D. Byerly, J. Biddiscombe, and H. Kaiser, "Hpx – an open source c++ standard library for parallelism and concurrency," 2017.

[33] R. Alves and J. Rufino, "rOpenCL," 12 2020. [Online]. Available: https://github.com/ruialves7/rOpenCL

[34] A. Barak and A. Shiloh. The VirtualCL (VCL) Cluster Platform. [Online]. Available: http://www.mosix.cs.huji.ac.il/vcl/VCL_wp.pdf

[35] A. Alves, J. Rufino, A. Pina, and L. Santos, "clopencl - supporting distributed heterogeneous computing in hpc clusters," 01 2013, pp. 112–122.

[36] B. Goglin, "High-Performance Message Passing over generic Ethernet Hardware with Open-MX," *Parallel Computing*, vol. 37, no. 2, pp. 85–100, Feb. 2011. [Online]. Available: https://hal.inria.fr/inria-00533058

[37] P. Kegel, M. Steuwer, and S. Gorlatch, "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, pp. 174–186.

[38] C. M. Kohlhoff. Asio C++ library. [Online]. Available: http://think-async.com/Asio/

[39] T. K. Group, "Opencl™ icd installation guidelines," https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_ICD_Installation.html, (accessed Junho 22, 2019).

[40] R. Aoki, S. Oikawa, T. Nakamura, and S. Miki, "Hybrid opencl: Enhancing opencl for distributed processing," in *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, 2011, pp. 149–154.

[41] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12, New York, NY, USA, June 2012, pp. 341–352.

[42] V. Raca and E. Mehofer, "clustercl: comprehensive support for multi-kernel data-parallel applications in heterogeneous asymmetric clusters," *The Journal of Supercomputing*, 03 2020.

[43] J. Solanti, M. Babej, J. Ikkala, and P. Jääskeläinen, "Pocl-r: Distributed opencl runtime for low latency remote offloading," 04 2020, pp. 1–2.

[44] C. Reaño, F. Silla, and J. Duato, "Enhancing the rcuda remote gpu virtualization framework: From a prototype to a production solution," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 695–698.

[45] P. Thoman, P. Salzmann, B. Cosenza, and T. Fahringer, "Celerity: High-level c++ for accelerator clusters," in *Euro-Par*, 2019.

[46] S. Group. Hpx- the c++ standard library for parallelism and concurrency. [Online]. Available: https://hpx.stellar-group.org/

[47] P. Diehl, M. Seshadri, T. Heller, and H. Kaiser, "Integration of CUDA processing within the c++ library for parallelism and concurrency (HPX)," in *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, nov 2018.

[48] M. Schupikov. Opencl-hpx integration. [Online]. Available: https://elib.uni-stuttgart.de/bitstream/11682/11866/1/schupikov-michael-bachelor-thesis-2021.pdf

[49] M. Copik and H. Kaiser, "Using sycl as an implementation framework for hpx.compute," in *Proceedings of the 5th International Workshop on OpenCL*, ser. IWOCL 2017. New York, NY, USA: Association for Computing Machinery, 2017.

[50] B. Peccerillo and S. Bartolini, "Phast - a portable high-level modern c++ programming library for gpus and multi-cores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 1, pp. 174–189, Jan 2019.

[51] IPB. source code of a distributed Matrix Multiplication:. [Online]. Available: http://www.ipb.pt/~rufino/clopencl/matrix-v13-split-v1_2012-11-10.tgz

[52] R. Alves, "ropencl: uma ferramenta para acesso de aplicações heterogéneas a co-orocessadores remotos," Master's thesis, Polytehcnic Institute of Braganca, Portugal, https://bibliotecadigital.ipb.pt/handle/10198/23222, 2020.

[53] UoB-HPC, "Babelstream," https://github.com/UoB-HPC/BabelStream.

[54] UoB-HPC, "Heat," https://github.com/uob-hpc/heat_sycl.

[55] ANL-CESAR, "Xsbench," https://github.com/ANL-CESAR/XSBench.

[56] ANL-CESAR, "Rsbench," https://github.com/ANL-CESAR/RSBench.

[57] P. Thoman, F. Molina Heredia, and T. Fahringer, "On the compilation performance of current sycl implementations," in *International Workshop on OpenCL*, ser. IWOCL'22. New York, NY, USA: Association for Computing Machinery, 2022.