Extending heterogeneous applications to remote co-processors with rOpenCL

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

José Rufino
Research Centre in Digitalization and Intelligent Robotics (CeDRI),
Instituto Politécnico de Bragança,
Campus de Santa Apolónia,
5300-253 Bragança, Portugal
rufino@ipb.pt

***Abstract*—In heterogeneous computing systems, general purpose CPUs are coupled with co-processors of different architectures, like GPUs and FPGAs. Applications may take advantage of this heterogeneous device ensemble to accelerate execution. However, developing heterogeneous applications requires specific programming models, under which applications unfold into code components targeting different computing devices. OpenCL is one of the main programming models for heterogeneous applications, set apart from others due to its openness, vendor independence and support for different co-processors.**

In the original OpenCL application model, a heterogeneous application starts in a certain host node, and then resorts to the local co-processors attached to that host. Therefore, co-processors at other nodes, networked with the host node, are inaccessible and cannot be used to accelerate the application. rOpenCL (remote OpenCL) overcomes this limitation for a significant set of the OpenCL 1.2 API, offering OpenCL applications transparent access to remote devices through a TPC/IP based network. This paper presents the architecture and the most relevant implementation details of rOpenCL, together with the results of a preliminary set of reference benchmarks. These prove the stability of the current prototype and show that, in many scenarios, the network overhead is smaller than expected.

***Index Terms*—OpenCL, heterogeneous computing, API forwarding, remote execution, parallel and distributed computing**

I. Introduction

The last decade saw the emergence of *heterogeneous* computing systems, where different architectures co-exist, in addition to the main architecture embodied by the familiar general purpose CPU. In such systems, traditional multi-core CPUs are coupled with auxiliary co-processor devices, like Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Digital Signal Processors (DSPs). These devices are exploitable beyond their primary original goal (i.e., graphics/signal processing), in order to accelerate the execution of computationally demanding applications [1], [2].

For heterogeneous systems to be efficiently exploited, applications must be developed using special programming models. With low-level programming approaches, like the vendor-specific NVIDIA CUDA Driver API [3] or the OpenCL open standard [4], heterogeneous applications are non-single source, unfolding explicitly into different code components per architecture. These components are written in C (with some extensions) or in C++ (using the bindings provided), and target separately the co-processor devices and the host system where devices are attached to; typically, the host-side code creates the necessary data structures to interact with the devices and triggers the relevant operations (data exchanges with / issuing code to / synchronization with the devices); sometimes, host code solves parts of the problem, in parallel with the devices.

Another common denominator to CUDA and OpenCL is that their original application model is intra-node centric, that is, an heterogeneous application launched on a certain host can only exploit the co-processor devices directly attached to that host. This necessarily implies a limited number of locally usable co-processors, due to physical and logical constraints (power, cooling, chassis, maximum number of devices supported by the drivers and the system BIOS, etc.).

However, scaling-out execution of heterogeneous applications to co-processors in multiple nodes has been possible, from some time now, with NVIDIA GPUDirect-RDMA [5] or AMD ROCn-RDMA [6], that rely on inter-node GPU communication over Infiniband. Basically, Infiniband RDMA-enabled network cards have direct access (via the PCIe bus) to GPU memory, allowing for data exchanges between GPU memory in different nodes, bypassing the main memory subsystem. At a higher-level, these RDMA-based approaches, namely NVIDIA GPUDirect-RDMA, have been taken advantage of by popular MPI-compliant [7] message passing libraries that can be combined with CUDA in developing distributed parallel hybrid applications [8]; in essence, with CUDA-aware MPI implementations, the MPI library can send and receive GPU buffers directly, thus optimizing data exchanges (notably, opposed to the availability of several CUDA-aware MPI implementations, there's currently no OpenCL equivalent).

An alternative to exploit inter-node multi-accelerator configurations, while being able to use the original programming models of CUDA and OpenCL, is to provide, through its runtime systems, access to remote accelerator devices (that is, devices attached to other nodes than the one in which the host-side of the heterogeneous applications is launched), as if they were local, effectively creating a unified view of all the available accelerators. This even allows to start heterogeneous

This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the Project Scope: UIDB/05757/2020.

2643-3001/20/$31.00 ©2020 IEEE
DOI 10.1109/SBAC-PAD49847.2020.00049

applications in nodes where accelerators are absent, provided nodes with co-processors are networked with the starting node. Of course, issuing CUDA/OpenCL API calls and kernels for remote execution, or exchanging data with or between remote devices, is expected to exhibit less performance, compared to using only local co-processors. However, this may not always be the case: depending on the computing capabilities differential between local and remote devices, and also on the behavior of the heterogeneous application (namely its workload distribution algorithm), it may pay off to offload certain tasks to remote devices, despite the penalty introduced by the network. Moreover, if enough transparency is ensured, such transparency may be regarded as a key advantage to counteract the expected performance limitations (ideally, pre-compiled binary code should be transparently executed in the overall set of devices exposed to the application by the unified runtime, irregardless of the devices location). On the other hand, when developing new applications, or when the source code is available, if the unified runtime offers extensions that allow to determine whether a device is local or remote, such information (together with the results of previous benchmarks) may be used during the distribution of the workload.

This paper introduces rOpenCL (remote OpenCL), as an approach that tackles the previous scenario for the specific case of distributed OpenCL heterogeneous applications and TCP/IP networks where hosts support the traditional BSD sockets mechanism (meaning virtually any networked host). Thus, rOpenCL does not depend on niche network technologies (e.g., Infiniband), but takes advantage of them when available (provided they support TCP/IP). Also, rOpenCL (and similar approaches, including CUDA-related – see section IV), is not meant to compete with dense intra-node multi-accelerator systems, or multi-node multi-accelerator clusters built on specialized communication fabrics; it is a complementary multi-node multi-accelerator approach that may be deployed on commodity hardware and networks, or on high-performance infrastructures when available. Finally, in opting to extend OpenCL to a distributed environment, in detriment of alternatives like CUDA, the key factors considered were i) standard openness, ii) application portability, and iii) support for a wide range of co-processors (and not only GPUs).

The remainder of the paper is organized as follows: section II covers the software architecture of rOpenCL and important implementation details; section III provides results of a preliminary evaluation of the stability and performance of the rOpenCL prototype; section IV offers a brief survey of the most relevant approaches comparable to rOpenCL; section V concludes and defines directions for future work.

II. Architecture and Implementation

A. Overview

OpenCL is a specification that allows applications to take advantage of different architecture devices [4]. Such *heterogeneous applications* are composed of *host code* and a set of *kernels*. A kernel is a specific core function of an OpenCL application. Typically, a kernel is meant to be executed in a co-processor that is faster (regarding the specific kernel operations) than the processor(s) where the host code runs; exceptionally, a kernel may also execute on the same processor(s) as the host code, e.g., if no co-processors are available.

The OpenCL programming model is currently supported by four major implementations of the official specification, targeting different device types: NVIDIA GPU drivers [9]; ROCm [10] OpenCL runtime for AMD CPUs and GPUs; Intel OpenCL SDK [11] for Intel CPUs, GPUs and FPGAs; POCL [12], vendor-agnostic and mainly CPU-oriented, though with some GPU support. In OpenCL parlance, an implementation of the specification is a *platform*. A platform typically supports a limited set of devices, accessible by the platform runtime.

However, despite all the features offered by the OpenCL model for the building of heterogeneous applications, and the variety of platforms, OpenCL applications that run on top of conventional OpenCL platforms (those conforming to the official specification) can only use devices from the machine where they run. To circumvent this limitation, allowing network-reachable devices from other machines to be used, several alternatives were developed (see section IV for a brief survey), including rOpenCL, introduced in this paper.

The main feature that sets rOpenCL apart, when compared with similar approaches, is its transparency: OpenCL applications do not need to be recompiled in order to use rOpenCL to reach out and exploit remote platforms and their devices; this is because rOpenCL exposes remote platforms to applications as if they were local platforms, that is, rOpenCL is an *aggregator* of remote platforms; also, rOpenCL doesn't expose any local devices (such concerns only to the local conventional platforms); thus, with rOpenCL installed in its host node, an OpenCL application is able to use any mix of local platforms/devices and remote platforms/devices.

TABLE I
OpenCL 1.2 API coverage of rOpenCL.

Function Categories	Implemented	Not Implemented
OpenCL Platform Layer	13	0
OpenCL Runtime	4	0
Buffer Objects	13	3
Program Objects	11	0
Kernel and Event Objects	22	1
Image Objects	7	3
Sampler Objects	4	0
OpenGL Sharing	0	9
Direct3D 10 Sharing	0	6
DX9 Media Surface Sharing	0	4
Direct3D 11 Sharing	0	6

In its current stage, rOpenCL supports $\approx$ 71% of the OpenCL 1.2 API [13] – see Table I. Most image/graphic processing primitives (categories in *italic*) were left out, as usual in other distributed OpenCL implementations, where the main focus is also pure computing. Thus, considering only the computing-related primitives, rOpenCL coverage of the OpenCL 1.2 API is $\approx$ 94%, with the following functions yet to be supported: `clEnqueueCopyBuffer` (partially implemented; doesn't support copies between de-

vices of different machines); `clEnqueueCopyBufferRect`, `clEnqueueMigrateMemObjects` and `clEnqueueNativeKernel` (none support). Nevertheless, the primitives already supported by rOpenCL are enough to conduct a comprehensive range of OpenCL benchmarks (see section III). Moreover, by targeting OpenCL 1.2, rOpenCL is in line with the recently released OpenCL 3.0 specification [14], which only mandates full support for OpenCL 1.2 (though with more focus in C++).

B. Architecture

rOpenCL consists of two main components: a client driver (rOpenCL Driver) and a set of remote services (rOpenCL Services) responsible for executing OpenCL requests in the underlying devices. Figure 1 provides a representation of these components and their relations with the OpenCL environment.

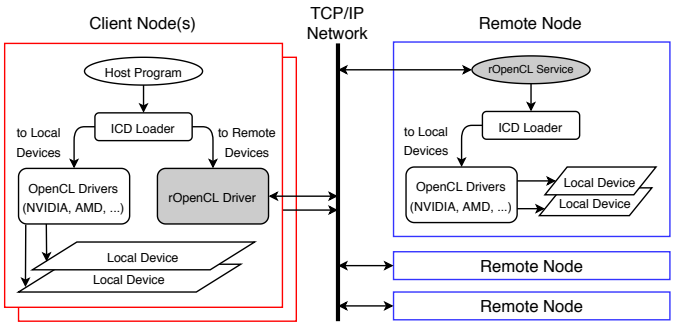

Fig. 1. rOpenCL architecture

At a host node, where an OpenCL application starts, the rOpenCL Driver is installed, like if it were another vendor driver; such drivers are known as installable client drivers (ICDs), and so rOpenCL has its own ICD. This way, the OpenCL runtime (the ICD Loader), is able to locate the ICD for rOpenCL and redirect OpenCL calls to it. In a Linux system, where rOpenCL was developed, this means there is a `/etc/OpenCL/vendors/rOpenCL.icd` file containing the path to an rOpenCL shared library where OpenCL calls are redirected to. Thus, by providing its own ICD, rOpenCL offers OpenCL applications full transparent access to the remote platforms and devices beneath the rOpenCL Services.

An OpenCL application may, however, want to know about the location of a specific remote OpenCL platform (and corresponding devices). This knowledge may be used, for instance, to implement client-side load balancing mechanisms, where an OpenCL application does not blindly assign requests to remote platforms that may happen to co-exist in the same remote node (thus overloading its devices); instead, it may decide to spread the requests among platforms in different nodes, by following a round-robin distribution, or other algorithms. rOpenCL makes this possible by extending the OpenCL `clGetPlatformInfo` primitive to support the new value `CL_PLATFORM_IPADDR` for its parameter `param_name`.

Another important architectural consideration is how rOpenCL manages OpenCL *contexts* in a distributed environment. A context is used by the OpenCL runtime to manage several interrelated objects (like command-queues – used to submit commands to devices –, memory buffers, program and kernel objects), and also to execute kernels on the devices specified for that context; these devices may even be of different types (e.g., CPUs and GPUs); however, they all must belong to the same OpenCL platform; in turn, an OpenCL platform object is, per the OpenCL standard, specific to a certain node, and rOpenCL complies with this; as such, rOpenCL doesn't replicate contexts (and subordinated objects) among different nodes; avoiding the burden of replica management simplifies the implementation; however, it prevents the creation of inter-node contexts, based on virtual platforms encompassing devices from different nodes; such objects, with a broader view of the nodes device set, could perform some form of automatic load-balancing; instead, with rOpenCL, that task is left to the programmer, by exploiting the `CL_PLATFORM_IPADDR` extension, as already stated.

Figure 1 also reveals that all network communication between the rOpenCL components happens via TCP/IP; such is because one of the goals for rOpenCL was network portability; this, coupled with the need for maximal network efficiency, lead to the choice of C-based BSD sockets as the programming framework for network communication; there are lower-level alternatives [15], but usually they do not support routing and so can only be used in the same LAN segment (thus limiting the remote device set that can be reached); moreover, the rOpenCL code base was designed from the very beginning to support the choice between TCP and UDP; currently, only the TCP code path is considered sufficiently robust, with the advantage of being usable with any network topology; UDP support is still experimental and meant to be used only on local network segments in order to minimize the probability of packet loss.

C. Connections Management

When an OpenCL application starts at the host node, OpenCL requests begin to be forwarded to the various OpenCL drivers configured, including the rOpenCL Driver. At minimum, this driver will be queried for platforms (and devices), in accordance with the usual workflow of OpenCL applications; after this inquiry phase, the application decides which platforms (and devices) to use, and further OpenCL requests will follow, directed to the drivers behind the platforms chosen.

To honor the platforms and device queries, as well as any other subsequent OpenCL request, the rOpenCL Driver will forward those requests to the appropriate rOpenCL Services. In order to make an efficient use of the communication layer, the driver minimizes the number of TCP connections and reuses them as much as possible. Every time the driver receives an OpenCL request, it checks the unique OS kernel thread ID (TID) of the requester process/thread, as well as a particular OpenCL parameter of the request (the *main* parameter – see section II-D) that, together with the TID, is mappable on the IP address of a rOpenCL service, using a Red-Black tree (RBT_1); it then uses the TID and the remote IP address to find out if it is necessary to create a new socket descriptor and establish a connection between the requester and the remote rOpenCL

Service; subsequent requests from the same process/thread to the same remote service will reuse the previously established connection; the inventory of TCP connections, by local TID and remote IP address, is hold on another Red-Black tree (RBT_2). To understand the importance of connection reuse, it was found, at an early stage of the development of rOpenCL, that establishing a TCP connection per each OpenCL call would consume around 30% of the execution time of the call (for OpenCL functions that involved small network transfers).

At each rOpenCL Service, POSIX threads are used. A frontend thread accepts connection requests from client threads (at host nodes), and assigns each new connection to a new worker thread. A worker thread will be responsible to forward the OpenCL requests received from a client thread, to the proper underlying OpenCL platform. When an OpenCL application at the host node ends, all open sockets descriptors will be closed by the operating system; this makes all remote worker threads that were previously paired with the application to unblock from the closed connections and to self terminate.

The startup of an OpenCL application in a host node also involves the discovery of rOpenCL Services. This is done simply, by reading a hostfile, that may be user-specific (default hostfile) or system-wide (in the absence of the former). The file contains the IP addresses and ports of the rOpenCL Services, as well as the local interface that the host node should use to communicate with those services (useful if the host node is a multi-homed system). The initial discovery process is conducted by a certain thread of the OpenCL application, that is usually (but not necessarily) the main thread; that discovery triggers the creation of an initial set of connections to each rOpenCL Service; at this initial stage, they are used to inquiry the remote platforms and devices; latter, those connections may be reused by the same client thread, if it needs to conduct new OpenCL transactions with the same rOpenCL Services.

D. Objects Management

OpenCL applications at the host node are supposed to deal with local pointers, referencing local memory areas where the various OpenCL objects, necessary to the application, are stored. However, when an OpenCL application makes use of the rOpenCL Driver, all local pointers must be mapped into remote pointers, for the same kind of objects, referencing memory zones of the nodes where rOpenCL Services run.

Every time an OpenCL primitive is called, the ICD Loader redirects that call to the proper driver. It does so by taking advantage of function pointers that the ICD Loader learned from the driver. That redirection submits and receives back certain OpenCL objects that must be well-formed (with an internal valid structure), once they may be passed along to other primitives by the ICD Loader. This means that, at the driver level, is risky to return fake pointers to the ICD Loader, once they may have and unpredictable effect. Fake pointers are used used by other distributed OpenCL implementations as equivalents to remote pointers; they can do so because their approach to the forwarding of OpenCL primitives to remote nodes is higher-level (wrapper-based), and not as lower level (driver level) as the one followed by rOpenCL in order to attain maximum transparency. Thus, instead of returning fake pointers, the rOpenCL Driver does indeed create real local OpenCL objects; however, it treats them as "hollow" objects, without any further use than misleading the ICD Loader. The real usefulness lies on the twin remote object that rOpenCL Services create for each local object. Once both objects are created, the rOpenCL Driver must register their equivalence. It does so in the RBT_1 Red-Black tree, that maps <TID, local pointer> keys into <remote pointer, IP address> values.

When the rOpenCL Driver receives a OpenCL request, one of its parameters (usually the 1st) is considered the *main* parameter, meaning its local address is already valid and registered in the RBT_1 tree; the local address of the main parameter, together with the requester TID, are used to search this tree, for the remote pointer and its IP address; local addresses of other parameters are also searched in the RBT_1 tree, for their remote addresses (some local addressees may already have been mapped into remotes, and others may not); the TID and IP address for the main parameter are used to search the RBT_2 tree for a previously open connection to the remote service; then, a message is sent to this service, carrying all necessary remote addresses; that message may result in the creation of new remote objects; the addresses of these objects are returned to the client thread; the twin local objects are then created and the new mappings registered in the RBT_1 tree.

E. Concurrency Management

At the client side, in the rOpenCL Driver, there's concurrent access to the RBT_1 and RBT_2 trees by different OS kernel threads; a MultipleReaderOneWriter (MROW) lock (`rw_semaphore`) is used, per tree, to ensure its consistency.

At the services side, there are several worker threads, each one dealing with a separate connection from a client thread. A worker thread unmarshalls the requests received from its client thread, triggers appropriate actions in the OpenCL layer, and replies accordingly to the client thread. Thus, at any time, there may be multiple worker threads issuing OpenCL calls. This rises the question of thread-safety. Per the OpenCL 1.2 specification [16], all OpenCL API calls are thread-safe except `clSetKernelArg` and even this function is not thread-safe only when called from multiple threads on the same `cl_kernel` object at the same time. Even though this is a corner case, it is dealt with, at cost of some overhead. Currently, a single MROW lock (a POSIX `pthread_rwlock_t` object), shared by all worker threads, serializes all calls to `clSetKernelArg`; an alternative, with less contention, planned for future work, is to register all `cl_kernel` objects in a tree-like structure (like the Red-Black trees used in the rOpenCL Driver), together with a MROW lock per object.

III. Preliminary Evaluation

This section provides results of a preliminary evaluation of rOpenCL. The tests conducted prove that rOpenCL is able to sustain the full execution of a set of well-know OpenCL

reference benchmarks, in various experimental conditions. Besides asserting the compliance of rOpenCL with the OpenCL standards, the tests also measure the overhead of the remote execution over local execution. One of the tests, using many devices, provides an initial insight into the scalability of rOpenCL. The tests selected are listed in Table II.

TABLE II
OPENCL BENCHMARKS USED TO EVALUATE ROPENCL

OpenCL Benchmark	Benchmark Profile		
	Memory	*Compute*	*Multi-Device*
BabelStream [17]	✓		
cl-mem [18]	✓		
clpeak [19]	✓	✓	
FinanceBench [20]		✓	
Hashcat [21]		✓	✓

The choice of this particular set of benchmarks is due to the following reasons: i) they stress different device subsystems (compute vs memory), ii) they are all open-source (some instrumentation code was necessary to automate the benchmarks), iii) they have been actively maintained in the last few years, iv) they run in Linux (the OS environment targeted by rOpenCL), v) at least one of them (Hashcat) is able to issue OpenCL calls to multiple devices in parallel.

A. Experimental Conditions

The tests were conducted between two nodes of an HPC cluster at CeDRI/IPB, each with the HW & SW of Table III.

TABLE III
SPECIFICATIONS OF EACH TEST NODE

CPUs	2 x AMD EPYC 7351 16-Core 2.4/2.9GHz
RAM	256 GB ECC DDR4 2666 MHz
Network	10Gbps Ethernet
GPUs	2 x NVIDIA RTX 2080 Ti
OS	Linux Ubuntu 18.04.3 LTS 64 bits
OpenCL	NVIDIA Driver 430.50

All benchmarks were repeated 5 times. The times presented in the charts of the next section are averages of the times measured in all 5 runs (the time of the 1st run was observed to be similar to the others). The execution overhead (deacceleration), in percentage, is given by $(\overline{T_r}/\overline{T_l} - 1) \times 100$, where $\overline{T_r}$ is a remote execution average time and $\overline{T_l}$ is its corresponding local execution average time. Also, to remove the start latency caused by the loading of the GPU driver at the beginning of each run, the NVIDIA Persistence Daemon [22] was configured on the two test nodes (the absence of this service would only affect the client-side of the benchmark – and would be only noticeably on short-duration benchmarks – once an rOpenCL Service forces the pre-load of the driver).

B. Results

The results for the BabelStream benchmark are shown in Figure 2. BabelStream provides a measure of what memory bandwidth performance can be attained when executing five kernels (copy, mul add, triad and dot). Although the benchmark is quick to execute, it is the one tested with the highest remote execution overhead, varying between 256% and 638%, depending on the sub-benchmarks, and with an overhead average of 353%. So, on average, BabelStream is 3,5 times slower when using a remote GPU via rOpenCL.

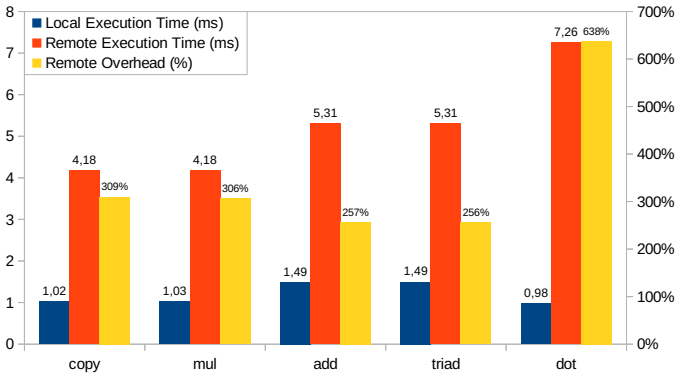

Fig. 2. BabelStream results (copy, mult, add, triad, dot).

cl-mem is a very simple benchmark with three different OpenCL kernels, to test the speed of sequential write, read and copy operations (of 128 GB of data, by default); these operations are executed in parallel, by groups of threads, in the device being tested; also, it does not require the previous transfer of the data to the device memory, once the data used in the test is generated on-the-fly; thus, cl-mem makes very little use of the network connection between the rOpenCL Driver in the client node and the rOpenCL Service in the remote node. This is clearly visible in the results presented in Figure 3.

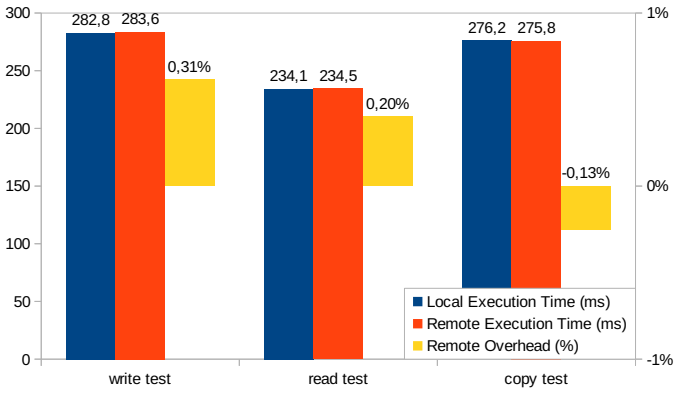

Fig. 3. cl-mem results (write, read and copy).

These results show very little differences between the local and the remote execution. And, in fact, for the copy kernel test, the remote execution turns out to be slightly faster. We have tried to uncover the reason for this, but that investigation was inconclusive; a possible explanation may lie in the fact that the nodes used for benchmarking have a NUMA architecture and the attachment of GPUs to the PCIe bus may be different.

Another benchmark conduced was clpeak. This is a simple hybrid benchmark, stressing both device memory and processing elements. It measure peak capabilities achieved using

vector operations and does not represent a real-world use case. Thus, we restrain from showing the peak memory bandwidth and compute power; instead, Figure 4 presents only the kernel launch latency; for the remote execution, it is interesting to see the impact introduced by the network on such a small operation; the overhead measured was modest: just an excess of 16%, compared to the latency of a local kernel launch.

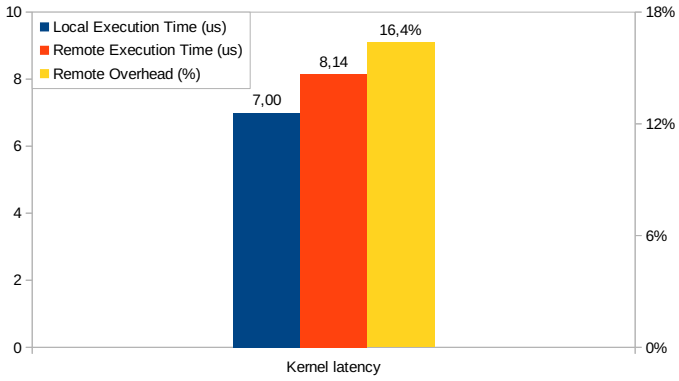

Fig. 4. clpeak results (kernel launch latency).

The last mono-device benchmark executed was FinanceBench, more compute focused. Even though four specific tests are available (Black-Scholes, Monte-Carlo, Bonds, Repo), all of which are financial applications, only two have an OpenCL version (Black-Scholes and Monte-Carlo), and so results are only presented for these two – see Figure 5. For one of the test (Black-Scholes) the remote overhead is minimal; for the other (Monte-Carlo), it is rather modest (21% more).

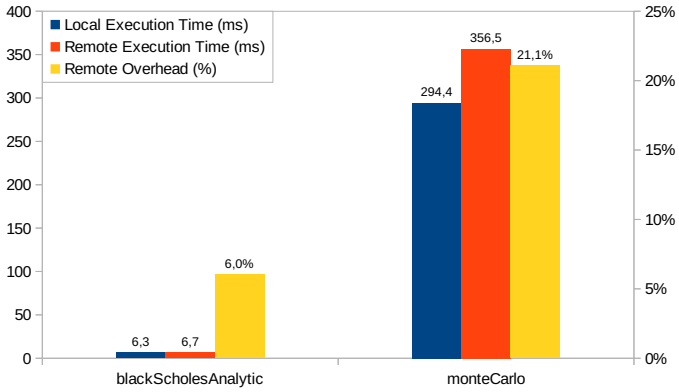

Fig. 5. FinanceBench results (Black-Scholes and Monte-Carlo).

The last benchmark results presented are for the Hashcat application, a well-known advanced password recovery tool [21]. It contains a built-in benchmark option – a single hash brute force attack; however, this benchmark involves very little data transfers between the host program and the devices, once the hash space is known a prior and is evenly divided by the used GPUs; instead, a dictionary attack with rules [23] (recovering 28 password from their MD5 hash) was chosen as the Hashcat benchmark. Moreover, once Hashcat is able to use multiple GPUs, this created the opportunity to put rOpenCL to the test with 2 GPUs (the maximum number of remote GPUs available in our test bed). Table IV shows all GPU combinations tested, and Figure 6 shows the respective execution times; depending on the specific combination of local and remote GPUs, these times are for pure local cracking, pure remote cracking, or include both alternatives.

TABLE IV
GPU COMBINATIONS FOR THE HASHCAT TEST.

Test Scenario	Local GPUs	Remote GPUs
T1	0	1
T2	0	2
T3	1	0
T4	1	1
T5	1	2
T6	2	0
T7	2	1
T8	2	2

The test results show that: i) when working only with remote GPUs, it certainly pays off to add an extra GPU (speedup is 494/249 = 1,98 $\approx$ 2); ii) the moment a local GPU is added, adding 1 remote GPU has little benefit (speedup is 249/229,8 = 1,08), and only by adding a 2nd remote GPU does the gain become noticeable (speedup is 249/171 = 1,45); iii) with 2 local GPUs, the benefit of 1 or 2 remote GPUs is again noticeable (speedup with 1 remote GPU is 112,8/93,6 = 1,20, and speedup with 2 remote GPUs is 112,8/81,8 = 1,38). The general conclusion is that the more GPUs, the better, and having local GPUs together with remote GPUs is always beneficial. Moreover, rOpenCL shows to be a viable tool to improve the performance of real-world OpenCL applications.

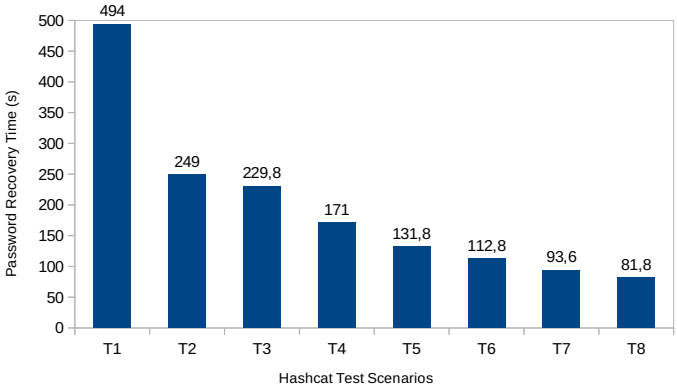

Fig. 6. Hashcat results (dictionary attack with rules on 25 MD5 hashes).

IV. Related Work

Vendors released the first public OpenCL implementations in 2009. Soon after, distributed variants started to be developed [24], [25]. This section surveys the most relevant, and includes comparisons with correlated rOpenCL features.

dOpenCL [26] is probably the most thorough distributed OpenCL implementation to date, and to which a comparison with rOpenCL is more meaningful. Both approaches build on a client driver at the host system and a set of services deployed

in the nodes of the distributed system with co-processors. Transparency is also a primary goal that both share, with the intent of allowing to run existing OpenCL applications in a heterogeneous distributed environment without any modifications. There are, however, important differences.

dOpenCL is represented by one dOpenCL platform object that gathers all the nodes and respective devices; this allows for a device management mechanism with transparent load balancing and exclusive access to specific remote devices by client applications; a single global unified platform implies, however, the need to maintain consistency of several shared distributed objects, like OpenCL contexts built on devices from different nodes (and such need extends to objects subordinated to contexts, like command queues and buffers). In turn, rOpenCL exposes the remote platforms and devices separately; so, it is up to the client application to choose which ones to use and to do its own load balancing; also, because different applications may share the same remote devices, rOpenCL services must perform some basic concurrency control in the particular situations where thread-safety is not ensured; moreover, because OpenCL contexts are restricted to use devices from the same platform, and platforms are separate per node, there's no need in rOpenCL to do replica management as in dOpenCL; this implies, though, that an application wanting to exploit devices from different nodes will have to use as many contexts at minimum (there may be more than one platform per node), while in dOpenCL one context suffices.

dOpenCL doesn't support the ICD loader mechanism; this implies that before running an OpenCL application on top of dOpenCL, the OpenCL system loader must be replaced by the dOpenCL loader, or the dOpenCL ICD must be explicitly preloaded; in this regard, rOpenCL is more transparent, once its ICD is fully integrated with the OpenCL system loader. In both, the client driver only exposes remote devices (unless a service to expose local nods is also installed in the host node).

Finally, in dOpenCL the client driver and services communicate using a Generic Communication Framework (GFC), part of a Real-Time Framework (RTF) developed for high-performance communication in distributed real-time applications, namely massively multi-player online computer games [27]. Though GFC supports TCP and UDP communication, dOpenCL opts for TCP. Likewise, rOpenCL also uses TCP communication, but via BSD sockets, a more portable and lighter approach (though arguably less feature-rich) than GFC.

Another approach to a distributed OpenCL implementation is clOpenCL (*cluster* OpenCL) [28]. It's architecture is similar to rOpenCL's, separately exposing the remote platforms, but it was not as transparent: legacy OpenCL applications needn't to be modified, but they needed to be recompiled in order to be linked with a library of wrapper functions that intercepted the OpenCL API calls. Moreover, in an effort to increase the performance of message passing between client applications and remote services, it relied on the low-level Open-MX library [15], directly above the Ethernet layer, thus restricting its use to non-routed LAN scenarios (e.g., HPC clusters).

Also tailored to heterogeneous cluster environments, SnuCL [29] is another distributed OpenCL framework that provides a single unified view of all the cluster compute devices and, at the same time, supports different partitions of the cluster device set: a compute device may be a GPU, or any sub-set of the CPU cores of a cluster node (including the host node, where the host program is launched); thus, other compute devices besides CPUs and GPUs are not supported. SnuCL relies on its own implementation of OpenCL: it uses a OpenCL-C-to-CUDA-C translator for kernels that target GPUs (NVIDIA only) and an OpenCL-C-to-C translator for kernels that target CPU devices; these translations are performed in the host node, the translated code is sent to compute nodes and there it's compiled with the native compiler for each compute device; thus, SnuCL requires the original OpenCL source code of applications to be available, although no modifications to it. Notably, SnuCL is not an API-forwarding approach: the OpenCL primitives of an OpenCL application are executed in the host node up to the point in which commands are enqued to command-queues; commands are then schedule across compute devices in the cluster. Communication between the components of the SnuCL framework is done via MPI and OpenCL is enriched with collective communication operations between buffers. For large scale clusters, the centralized task scheduling model of SnuCL degrades performance; SnuCL-D [30], a recent evolution of SnuCL, tackles this problem.

API-forwarding and source-to-source translation (thus requiring the original source code of OpenCL applications) were combined in Hybrid OpenCL [24], one of the first distributed OpenCL proposals. In Hybrid OpenCL, OpenCL code is translated to readable C code that uses embedded IA-32 SSE functions, thus targeting only multi-core x86 devices and excluding GPUs or other co-processors. At the host node, OpenCL applications link with the Hybrid OpenCL runtime, a version of the FOXC (Fixstars OpenCL Cross Compiler) OpenCL runtime, modified to include a network layer (though the communication protocol used is not disclosed). The Hybrid OpenCL runtime is able to submit request to the underlying OpenCL x86 devices or to forward them to remote nodes. Each remote node runs a networked bridge service on top of a local OpenCL runtime (that may be other than FOXC, as long as it allows to use the local CPUs as devices), to which it relays requests coming from the host node. Hybrid OpenCL offers to client applications a single OpenCL platform with all the x86 devices of the involved nodes, and applications may use any combination of those devices.

The abstraction of a global OpenCL platform combining all compute devices (CPUs, GPUs and other accelerators) available in a set of networked nodes is also provided by the VirtualCL (VCL) cluster platform [31], originally a MOSIX [32] layer. With VCL, most OpenCL applications run unmodified, starting in a single node, and take advantage, transparently, of the cluster device set; applications may create OpenCL contexts that mix devices from different nodes, or are restricted to intra-node devices (the default); the real location of the devices in the cluster is hidden from applications, but environment variables allow to tune the device allocation policies. VCL

includes i) a front-end thread-safe wrapper library to which OpenCL applications must link (thus requiring its source code to be available), ii) a broker daemon for cluster resource monitoring, reporting and allocation, that runs on the node where the application starts, and iii) a back-end daemon, per each cluster node with compute devices, that relays OpenCL requests from the client applications to local vendor-specific OpenCL libraries (with each device being exclusively locked by one application at a time). Communication between the cluster nodes involved standard TCP/IP sockets. VCL is still used and maintained (but only binaries are available).

V. Conclusions

This paper introduced rOpenCL, a novel implementation of an API-forwarding layer that allows OpenCL applications to take advantage of OpenCL devices accessible via a TCP/IP network. This puts the co-processor devices of virtually any networked system (even if routing is involved) within the reach of an OpenCL application. Moreover, the way rOpenCL was architected and implemented allows for pre-compiled OpenCL applications to take immediate advantage of it; this level of transparency, achieved by the perfect integration of the rOpenCL Driver in the OpenCL runtime, is a feature that we believe will make rOpenCL an attractive choice for scenarios where a distributed OpenCL layer is necessary. The stability of the current implementation, and its usefulness for a real-world application, were also shown by the benchmark results presented (a subset of the benchmarks already conducted).

In the future, we will enhance rOpenCL in several areas, including: further optimizing the network transfers, making UDP a robust alternative for local networks, and increasing the OpenCL 1.2 API coverage. We also intend to do tests with more computing nodes, thus supporting more simultaneous clients and rOpenCL services, with the goal of assessing the scalability of the later. Finally, a comparison with previous distributed OpenCL implementations is also planned (this should be viable at least with clOpenCL, dOpenCL and VCL).

References

[1] A. Cano, "A survey on graphic processing unit computing for large-scale data mining," *WIREs Data Mining and Knowledge Discovery*, vol. 8, no. 1, p. e1232, 2018.

[2] H. Wang, H. Peng, Y. Chang, and D. Liang, "A survey of gpu-based acceleration techniques in mri reconstructions," *Quantitative Imaging in Medicine and Surgery*, vol. 8, pp. 196–208, March 2018.

[3] NVIDIA. CUDA Driver API :: CUDA Toolkit Documentation. [Online]. Available: https://docs.nvidia.com/cuda/cuda-driver-api/index.html

[4] The Khronos Group. OpenCL Overview. [Online]. Available: https://www.khronos.org/opencl/

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

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

[7] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard - Version 3.1. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[8] NVIDIA. An Introduction to CUDA-Aware MPI. [Online]. Available: https://devblogs.nvidia.com/introduction-cuda-aware-mpi/

[9] NVIDIA. OpenCL NVIDIA Developer. [Online]. Available: https://developer.nvidia.com/opencl

[10] Advanced Micro Devices. AMD ROCm Open Ecosystem. [Online]. Available: https://www.amd.com/en/graphics/servers-solutions-rocm

[11] Intel. Intel SDK for OpenCL Applications. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/opencl-sdk.html

[12] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable opencl implementation," *Int. Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10766-014-0320-y

[13] The Khronos Group. OpenCL 1.2 Overview. [Online]. Available: https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf

[14] The Khronos Group. Khronos Group Releases OpenCL 3.0. [Online]. Available: https://www.khronos.org/news/press/khronos-group-releases-opencl-3.0

[15] B. Goglin, "Design and implementation of open-mx: High-performance message passing over generic ethernet hardware," in *2008 IEEE Int. Symposium on Parallel & Distributed Processing*, April 2008, pp. 1–7.

[16] The Khronos Group. The OpenCL Specification 1.2. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

[17] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via BabelStream," *International Journal of Computational Science and Engineering*, vol. 17, no. 3, pp. 247–262, 10 2018.

[18] nerdralph. CL-Mem. [Online]. Available: https://github.com/nerdralph/cl-mem

[19] krrishnarraj. CL-Peak. [Online]. Available: https://github.com/krrishnarraj/clpeak

[20] S. Grauer-Gray, W. Killian, R. Searles, and J. Cavazos, "Accelerating financial applications on the gpu," in *6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6)*, 03 2013, pp. 127–136. [Online]. Available: https://github.com/cavazos-lab/FinanceBench

[21] hashcat. hashcat overview. [Online]. Available: https://hashcat.net/hashcat/

[22] NVIDIA. Driver Persistence. [Online]. Available: https://docs.nvidia.com/deploy/driver-persistence/index.html/

[23] Jake Miller. Hashcat Tutorial – The basics of cracking passwords with hashcat. [Online]. Available: https://laconicwolf.com/2018/09/29/hashcat-tutorial-the-basics-of-cracking-passwords-with-hashcat/

[24] R. Aoki, S. Oikawa, R. Tsuchiyama, and T. Nakamura, "Hybrid OpenCL: Connecting Different OpenCL Implementations over Network," in *2010 10th IEEE International Conference on Computer and Information Technology*, June 2010, pp. 2729–2735.

[25] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh, "A package for OpenCL based heterogeneous computing on clusters with many GPU devices," in *2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*, October 2010, pp. 1–7.

[26] P. Kegel, M. Steuwer, and S. Gorlatch, "dopencl: Towards uniform programming of distributed heterogeneous multi-/many-core systems," *Journal of Parallel and Distributed Computing*, vol. 73, p. 1639–1648, December 2013.

[27] F. Glinka, A. Ploss, J. Müller-Iden, and S. Gorlatch, "Rtf: A real-time framework for developing scalable multiplayer online games," in *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games, NetGames '07*, January 2007, pp. 81–86.

[28] A. Alves, J. Rufino, A. Pina, and L. P. Santos, "clOpenCL: Supporting Distributed Heterogeneous Computing in HPC Clusters," in *Proceedings of the 18th Int. Conference on Parallel Processing Workshops*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 112–122.

[29] 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.

[30] J. Kim, G. Jo, J. Jung, J. Kim, and J. Lee, "A distributed OpenCL framework using redundant computation and data replication," in *PLDI '16: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2016, pp. 553–569.

[31] A. Barak and A. Shiloh, "The MOSIX Virtual OpenCL (VCL) Cluster Platform," in *Proceedings of the Intel European Research and Innovation Conference*, October 2011, p. 196.

[32] A. Barak and A. Shiloh. The MOSIX Cluster Management System for Distributed Computing on Linux Clusters and Multi-Cluster Private Clouds. [Online]. Available: http://www.mosix.cs.huji.ac.il/pub/MOSIX_wp.pdf