Extending heterogeneous applications to remote co-processors with rOpenCL

Rui Alves*, **José Rufino**†*
{rui.alves,rufino}@ipb.pt

†Research Centre in Digitalization and Intelligent Robotics (CeDRI),
*Polytechnic Institute of Bragança, Bragança, Portugal

WAMCA 2020
10th International Workshop on Applications for Multi-core Architectures
September 8, 2020, Porto, Portugal

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

Outline

1. **Context and Motivation**
   - Heterogeneous Systems
   - Scaling-Out Heterogeneous Applications
   - The rOpenCL Approach
2. **Architecture and Implementation**
   - Architecture
   - Connections Management
   - Objects Management
   - Concurrency Management
   - Context Management and Extensions
3. **Preliminary Evaluation**
   - API Coverage
   - Experimental Setup
   - Results
4. **Related Work**
5. **Conclusions**

Heterogeneous Systems (1/2)

- for some time now computing systems have become **heterogeneous**
- these are systems where **different architectures co-exist**
- traditional multi-core CPUs are coupled with auxiliary **co-processor devices**: Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Digital Signal Processors (DSPs)
- co-processors are exploited beyond their primary original goal (i.e., graphics/signal processing), in order to **accelerate the execution** of general purpose computationally demanding applications

Heterogeneous Systems (2/2)

- heterogeneous systems require special **programming models**
- the vendor-specific NVIDIA CUDA Driver API, and the OpenCL open standard, are the main **low-level programming** approaches
  - heterogeneous applications are **non-single source**, unfolding explicitly into different code components per architecture
  - **host code** creates the data structures to interact with the **co-processors** (code issuing/data exchanges/synchronization)
  - co-processors execute **kernels** of specialized code

Scaling-Out Heterogeneous Applications

- originally, both CUDA and OpenCL are **single-node**:
  - an heterogeneous application can only exploit the co-processors directly attached to the node where is started
  - these local devices are necessarily of a limited number, which limits the potential for application acceleration
- scaling-out execution of heterogeneous applications to external co-processors has been done using different approaches
  - HW/Driver-level: data exchanges between GPU memory in different nodes over Infiniband networks, bypassing the main memory subsystem (NVIDIA GPUDirect-RDMA and AMD ROCn-RDMA)
  - SW/Service-level: extend the original programming models of CUDA and OpenCL to expose remote accelerators in their runtimes (rCUDA and several OpenCL-focused approaches like rOpenCL)

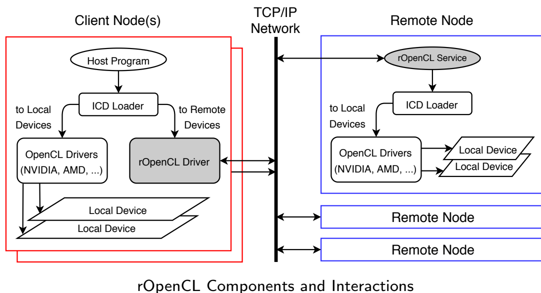
The rOpenCL Approach (1/2)

- rOpenCL (remote OpenCL) allows scaling-out OpenCL applications to co-processors within nodes interconnected by a TCP/IP network
- it uses BSD sockets as a portable communication layer; it does not depend on niche network technologies (e.g., Infiniband), but takes advantage of them when available (provided they support TCP/IP)
- it allows unmodified pre-compiled binary OpenCL applications to take advantage of remote co-processors (no source code necessary)
- why to extend OpenCL (vs CUDA) to a distributed environment ?
  - OpenCL offers i) standard openness, ii) application portability, iii) support for a wide range of co-processors (not only GPUs)
  - despite its popularity, CUDA does not fit into this requisites

The rOpenCL Approach (2/2)

- it is also important to clarify a crucial difference between rOpenCL and similar approaches, related to the OpenCL **platform** concept
- an OpenCL **platform** is an implementation of the OpenCL specification
- a platform from a HW vendor (e.g., NVIDIA, AMD, Intel) targets **device** accelerators from that vendor, via specific **drivers** supplied by the vendor
- rOpenCL **explicitly exposes remote platforms individually** to an OpenCL application, which then chooses the ones to use (possibly combined with local platforms) and is also responsible for doing their load-balancing
- in turn, other related approaches unify all remote platforms into a **single virtual platform**, hiding individual remote platforms from the applications; this is done to allow automatic load-balancing of the distributed device set

Components



rOpenCL Components and Interactions

- two main components: **rOpenCL Driver** and remote **rOpenCL Services**
- rOpenCL Driver: an installable client driver (ICD), in the host/client node
  - `/etc/OpenCL/vendors/rOpenCL.icd` points to an rOpenCL shared library
  - the OpenCL runtime (ICD Loader) locates all ICDs and loads their libraries
  - the runtime redirects OpenCL calls to the appropriate vendor shared libraries
  - the rOpenCL Driver forwards OpenCL calls to rOpenCL Services

Connections Management (1)

- currently, exchanges between the driver and services are restricted to **TCP**
- the driver keeps a **record of previous connections** so they may be reused (a new connection for each OpenCL call would severely harm performance)
  - every time the driver receives an OpenCL call, it checks the unique OS kernel thread ID (TID) of the requester; the TID, together with a certain parameter (*) of the call, map onto the service IP address
  - then, the TID and the IP address map onto an open socket descriptor or, if absent, trigger the creation and connection of a new one

Connections Management (2)

- at each rOpenCL Service, POSIX threads handle the clients connections
  - a **frontend thread** accepts connection requests from client threads (at client/host nodes), and assigns each new connection to a new worker thread
  - a **worker thread** forwards the client thread requests to the proper underlying OpenCL platform, and returns the results of the OpenCL call to the client
- **connections cleanup**: 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
- **services discovery**: done at the startup of an OpenCL application in the client node; the first application thread that tries to query the platforms exposed by rOpenCL will learn the IP addresses of the services from a *hostfile*; the connections established with the services will be later reused

Objects Management

- OpenCL applications at the host node only deal with **local object pointers**
- when using rOpenCL, local pointers must be converted to **remote pointers**
- other approaches just create **local fake pointers** for each remote pointer (e.g., by joining/merging the remote pointer and the service IP address)
- in rOpenCL, the tight integration with the ICD loader prevents fake pointers
- instead, for every remote object, there's a **twin local object**; this local object **is not a replica**: it's value is irrelevant, only its address is of use
- local to remote pointers mappings are registered by the rOpenCL Driver in a data structure that maps <TID, local pointer (*)> pairs into <remote pointer, IP address> pairs; (*) this is the "certain parameter" in slide 9

Caveat: this only concerns OpenCL API object pointers; data from/to user-level pointers is transfered between the client application, the driver and the services.

Concurrency Management

- in the rOpenCL Driver
  - concurrent access to data structures that register connections and map pointers; one MultipleReaderOneWriter (MROW) lock per data structure
- in rOpenCL Services
  - many worker threads issuing OpenCL calls to the underlying platform(s)
  - under OpenCL 1.2 almost all OpenCL API functions are thread-safe
  - single exception: `clSetKernelArg`, when called from multiple threads on the same `cl_kernel` object; dealt with a single global MROW lock
    - future work: register `cl_kernel` objects, have one lock per object

(*)Context Management and OpenCL extensions

- **context management**:
  - in OpenCL, **contexts** are used by the runtime to manage several interrelated objects (command-queues, memory buffers, program and kernel objects), and also to execute kernels on the devices specified for that context
  - in OpenCL, devices of a context must belong to the same OpenCL platform; in turn, an OpenCL platform object is specific to a certain computing node
  - thus, contexts are specific to a certain node, and so rOpenCL doesn't need to replicate contexts (and subordinated objects) among different nodes
- rOpenCL **extends** the OpenCL `clGetPlatformInfo` primitive to support the new value **`CL_PLATFORM_IPADDR`** for the parameter `param_name`
  - by knowing at which node resides a platform and its devices, an OpenCL application may conduct some form of load-balancing

API Coverage

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

OpenCL 1.2 API coverage of rOpenCL

- currently, rOpenCL supports $\approx$ 71% of the OpenCL 1.2 API
- image/graphic primitives left out, as usual in similar approaches
- computing primitives coverage is $\approx$ 94%, with the following exceptions:
  - `clEnqueueCopyBuffer`: if in the same node; `clEnqueueCopyBufferRect`, `clEnqueueMigrateMemObjects`, `clEnqueueNative Kernel`: none support
- found to be enough for most computing benchmarks and applications

Experimental Setup (1/2)

- **preliminary evaluation**:
  - execute some reference OpenCL benchmarks (test compliance)
  - execute at least one real-world application (show usefulness)
  - gain insight on the overhead of remote execution
- **tests selected**:

OpenCL Test	Test Profile				
	Memory	Compute	Multi-Device	Synthetic	Real-World
BabelStream [1]	✓			✓	
cl-mem [2]	✓			✓	
clpeak [3]	✓	✓		✓	
FinanceBench [4]		✓		✓	✓
Hashcat [5]		✓	✓		✓

  - stress different sub-systems (compute vs memory)
  - open-source (easy instrumentation and automation)
  - actively maintained/developed in the last few years
  - all mono-device (use only one GPU) except Hashcat
  - mix of synthetic benchmarks and real-world tests

Experimental Setup (2/2)

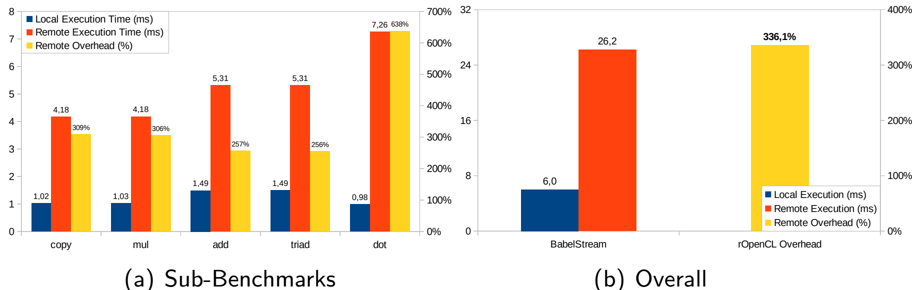
- **computing nodes (2)**:

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

HW and SW specifications of each node

- **methodology**:
  - test results are averages of 5 runs
  - 3s pause between consecutive runs
  - most results are execution times, such that remote execution overhead is given by $(\overline{T_R} - \overline{T_L})/\overline{T_L} = \overline{T_R}/\overline{T_L} - 1$, where
    - $\overline{T_R}$ is the remote execution average time
    - $\overline{T_L}$ is the local execution average time
  - NVIDIA Persistence Daemon used to eliminate driver load latency

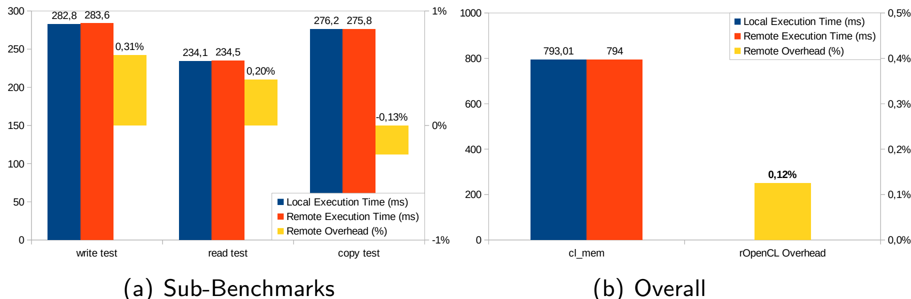
Babel Stream



(a) Sub-Benchmarks (b) Overall

- memory bandwidth for 5 kernels (copy, mul, add, triad, dot)
- specific overheads between 256% and 638%, average of 353%
- overall remote execution overhead of 336% (3,36 times slower)

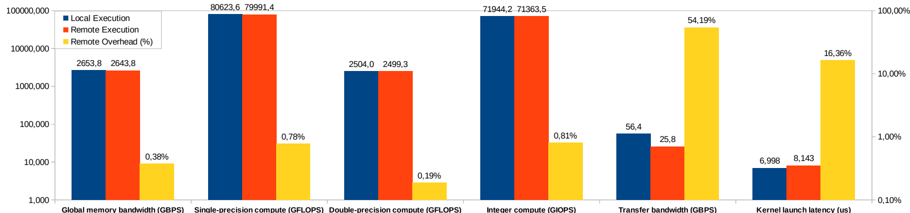
cl-mem



(a) Sub-Benchmarks (b) Overall

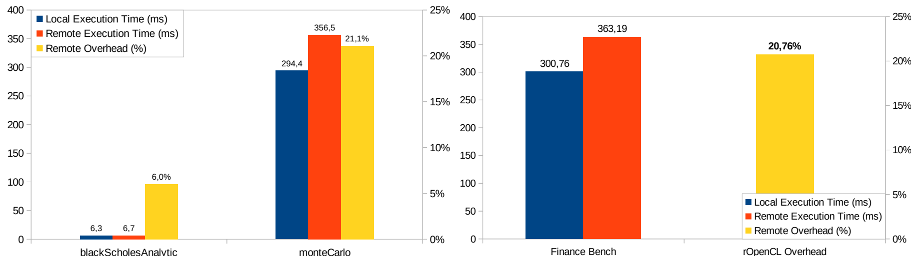
- sequential write, read and copy of 128 GB of data, by groups of threads
- data generated on-the-fly in the GPUs (not transferred from the host)
- minimal network intervention $\implies$ negligible overhead (specific and overall)
  - negative overhead in copy test (acceleration); still under investigation (possibly due to NUMA or caching effects in the rOpenCL service)

clpeak



- stresses device memory and processing elements; measures peak capabilities
- note: for bandwidth and compute tests, remote overhead is given by $(\overline{P_L} - \overline{P_R})/\overline{P_L}$, where $\overline{P_L}$ ($\overline{P_R}$) is the local (remote) execution peak value
- tests with very few network usage exhibit negligible overhead ($< 1\%$)
- for the others, the overhead becomes noticeable (54,19% and 16,36%)

FinanceBench



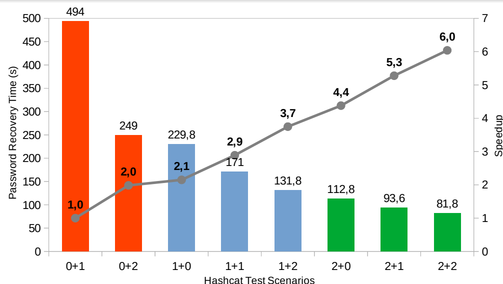
(a) Sub-Benchmarks (b) Overall

- only two OpenCL sub-benchmarks: Black-Scholes and Monte-Carlo
- minimal overhead for the 1st (6%), and moderate for the 2nd (21%)
- overall remote execution overhead of $\approx 21\%$

Hashcat

Test Scenario	#Local GPUs	#Remote GPUs
0+1	0	1
0+2	0	2
1+0	1	0
1+1	1	1
1+2	1	2
2+0	2	0
2+1	2	1
2+2	2	2

GPU combinations for the Hashcat test



- dictionary attack with rules to recover passwords from 28 MD5 hashes [6]
- simple scalability test using up to 4 GPUs (mix of local and remote)
- when using only remote GPUs or only local GPUs, it pays off to add an extra GPU of the same kind (speedup brought by the second GPU is ≈ 2)
- combining local GPUs with remote GPUs always pays off, but the performance gains from remote GPUs diminish with more local GPUs

General Remarks on the Tests Results

- As expected, remote execution introduces an overhead
- The order of magnitude of the overhead depends on the tests specifics
  - compute-bound (io-bound) tasks, with very little (significant) data transfers between host and devices, will take better (worse) advantage of rOpenCL
- Results are also affected by other factors:
  - **load time of GPU drivers in every test run**; affects local tests (only noticeable on short duration tests), not remotes (rOpenCL services already force the loading of the driver); solution: NVIDIA Persistence Daemon
  - **standby time between consecutive runs of a test**; increasing this time allows GPU temperatures to decrease more, improving the test results
  - **NUMA topology of the hosts** (may affect GPU latency) and **placement of the GPU in the chassis** (may affect cooling); mono-device tests that allowed the choice of the GPU to be used were executed separately, for each of the two local and the two remote GPUs, and their results were averaged

Related Work

- **dOpenCL** (distributed OpenCL) [7]: 2012-2013
  - runs binary OpenCL apps; unifies remote platforms in one virtual OpenCL platform; automatic load-balancing; assumes exclusive devices access (not shared); TCP transfers using Boost.Asio; no ICD loader integration
- **clOpenCL** (cluster OpenCL) [8]: 2012
  - OpenCL apps must be recompiled; exposes remote OpenCL platforms separately; low-level transfers using Open-MX; no ICD loader integration
- **SnuCL** (Seoul National University CL) [9] : 2012-2015
  - OpenCL source code needed; unifies/partitions a cluster OpenCL device set; only CPUs or GPUs; OpenCL-C-to-CUDA-C translator; schedules commands from OpenCL command queues to cluster nodes; MPI communication
- **VCL** (VirtualCL) [10]: 2011-2017
  - OpenCL apps must be recompiled; unifies separate node platforms in cluster-wide OpenCL platforms; hides devices location (automatic selection); integrates with cluster job managers; no ICD loader integration; TCP/IP

Conclusions

- rOpenCL is a novel implementation of an API-forwarding layer that allows OpenCL applications to use remote network-accessible OpenCL devices
  - able to run binary OpenCL applications (no recompilation needed)
  - integrates perfectly in the OpenCL runtime (ICD loader integration)
  - uses TCP/IP: network-portable, not restricted to local networks
  - exposes remote devices separately: simple and flexible approach; no transparent/automatic load-balancing; remote devices are shareable
  - small/acceptable remote execution overhead for many use cases
- **future work:**
  - scalability studies with many clients and services (in-progress)
  - performance comparison with related approaches (in-progress)
  - comparison of local CPU-only executions vs remote GPU-based
  - increase the OpenCL API coverage
  - evaluate overhead in virtualized GPU-sharing scenarios
  - conduct tests with TCP/IP routing involved
  - improve the performance of network transfers; Async IO ? UDP ?

Bibliography

[1] 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.

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

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

[4] 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

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

[6] 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/

[7] 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.

[8] 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.

[9] 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.

[10] 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.