dOpenCL – Evaluation of an API-Forwarding Implementation

Karsten Tausche, Max Plauth and Andreas Polze

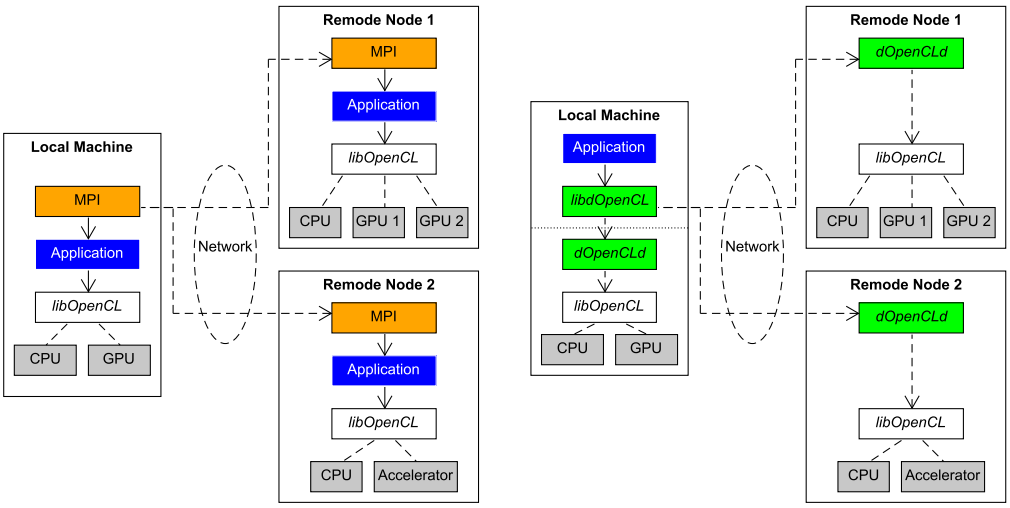
Operating Systems and Middleware Group
Hasso Plattner Institute for Software Systems Engineering, University of Potsdam
karsten.tausche@student.hpi.de,firstname.lastname@hpi.de

Parallel workloads using compute resources such as GPUs and accelerators is a rapidly developing trend in the field of high performance computing. At the same time, virtualization is a generally accepted solution to share compute resources with remote users in a secure and isolated way. However, accessing compute resources from inside virtualized environments still poses a huge problem without any generally accepted and vendor independent solution.

This work presents a brief experimental evaluation of employing *dOpenCL* as an approach to solve this problem. *dOpenCL* extends OpenCL for distributed computing by forwarding OpenCL calls to remote compute nodes. We evaluate the *dOpenCL* implementation for accessing local GPU resources from inside virtual machines, thus omitting the need of any specialized or proprietary GPU virtualization software. Our measurements revealed that the overhead of using *dOpenCL* from inside a VM compared to utilizing OpenCL directly on the host is less than 10% for average and large data sets. For very small data sets, it may even provide a performance benefit. Furthermore, *dOpenCL* greatly simplifies distributed programming compared to, e.g., MPI based approaches, as it only requires a single programming paradigm and is mostly binary compatible to plain OpenCL implementations.

1 Introduction

Since the emergence of big data in virtually all research and business fields, developments in high performance computing are focusing more and more on data parallel algorithms. For satisfying the resulting demand for processing power, GPUs and accelerators have become much more popular compared to traditional CPU-based approaches. This development is not yet reflected well in the field of parallel and distributed programming paradigms. Software developers are mostly forced to use combinations of techniques (Figure 1a). For example, MPI is used to distribute compute calls to multiple machines, whereas locally on each machine, APIs such as OpenCL are required to access compute devices. On the one hand, MPI itself has no means to directly access compute devices. On the other hand, compute APIs such as OpenCL and CUDA do not allow access to remote devices. However, using a combination MPI and a local compute APIs is unnecessarily complex and error prone [9]. Furthermore, MPI requires users to deploy their application code to all compute nodes, which might additionally introduce security risks.



(a) MPI/OpenCL combination **(b)** dOpenCL

Figure 1: Distributed computing using a traditional combination of MPI and OpenCL (1a) compared to the new dOpenCL implementation (1b). MPI requires explicit knowledge of all compute nodes and deployment of the application to all nodes. With dOpenCL, the application is deployed on a single machine. The distribution of OpenCL kernels to compute nodes is handle by the library.

dOpenCL[9] proposes a solution to these problems by extending the original OpenCL standard with means of distributed computing, without requiring any changes of the employed programming paradigm. Applications using dOpenCL are still deployed on a single machine only, but the underlying dOpenCL implementation allows to execute OpenCL kernels on any OpenCL device available in the network (Figure 1b). From a programmers point of view, all devices appear in a single OpenCL platform and are transparently usable as regular OpenCL devices.

When combining high performance computing with virtualization, accessing compute devices from within virtual machines (VMs) in a performant and flexible way is still quite difficult to realize. Virtualization solutions are available that directly assign compute devices to the VM[14] using PCI passthrough. However, these are highly specialized solutions that are limited to specific operating systems, platforms or vendors. Furthermore, such solutions generally exclusively lock compute devices for the entire lifetime of the VM, no matter if there is an application running that currently uses it. We found that dOpenCL is quite useful in this situation: The user's application code is deployed only to the VM, so that it is always isolated from the host system. OpenCL kernels, however, can be executed on the host or other compute nodes in the network with little overhead. At the same time, compute devices are only locked as long as they are actually used by an application. This enables utilizing compute devices on demand by a varying number of applications in VMs.

In this paper, we evaluate dOpenCL for accessing GPUs attached to the host from inside a VM. We found that dOpenCL currently is the only implementation that is entirely based on open standards and implementations. Furthermore, our evaluation shows that using dOpenCL leads to little overhead, both in terms of runtime and deployment, compared to a native OpenCL setup.

2 Related Work

Many GPU API-forwarding implementations have been proposed that enable the use of compute APIs from inside virtual machines (VMs). In this section, we compare implementations based on OpenCL and NVIDIA CUDA, as these are the most widely spread compute APIs. GPU API-forwarding implementations generally consist of a front-end and a back-end. The front-end provides access to a compute API within VMs and redirects API calls to the back-end running on the virtualization host.

2.1 CUDA Forwarding Implementations.

CUDA based implementations are only compatible with NVIDIA GPUs, and are thus not portable enough for our objectives. However, these approaches are still interesting for comparison. *GViM*[7] provides CUDA forwarding based on the Xen hypervisor. It additionally implements resource sharing between multiple VMs. Although its experimental implementation is limited to Xen, its approach might be generalization for other virtual machine monitors (VMMs). *vCUDA*[13] and *gVirtuS*[6] are VMM-independent implementations. Besides simple API-forwarding, vCUDA implements a full GPU-virtualization, including suspend-and-resume, and multiplexing capabilities. All three publications include experimental implementations that provide only a limited set of CUDA functions of outdated CUDA API versions. In contrasts, recent proprietary releases of *rCUDA*[5] provide complete CUDA 8.0 runtime API support. rCUDA was originally intended to provide remote GPU access in HPC clusters, but proved to be efficiently usable for local GPU access from within virtual machines, too[4].

2.2 OpenCL Forwarding Implementations

As OpenCL is less popular compared to CUDA, fewer publication exist that focus on OpenCL API-forwarding. However, the OpenCL API is completely open, which significantly alleviates creating portable implementations. *Hybrid OpenCL* [1] extends OpenCL by a network layer that enables to access remote OpenCL devices without using MPI or any additional RPC protocols. Its experimental implementation, based on a specific Linux-based OpenCL runtime for Intel CPUs, and demonstrated that its networking overhead amortizes for long running compute tasks. *SnuCL*[11] generalizes the OpenCL approaches for heterogeneous small and medium scale HPC clusters. Additionally, it extends the OpenCL API by collective operations, similarly to those included in MPI. The SnuCL runtime makes the com-

puting devices available in a set compute nodes accessible from a single compute host. Its implementation relies on OpenCL kernel code transformations to correctly execute data accesses on remote machines.

Finally, *dOpenCL*[8, 9] generalizes OpenCL for distributed, heterogeneous computing, comparable to SnuCL. However, it introduces a central device manager that distributes available compute devices (on compute nodes) in a network to multiple compute hosts in the network. Compute devices are exclusively locked on demand. This allows to flexibly share a large number of devices with a varying number of users in the network. Our experiments with dOpenCL are based on an unpublished prototype that replaces the proprietary *Real-Time-Framework* (RTF)[15] used in the original implementation by a *Boost.Asio*[12] based implementation. This dOpenCL version does not include the device manager, but it uses the same API-Forwarding implementation as the RTF-based version. In our experiments, we measure the performance of dOpenCL when used in a VM on a single machine. Thus, we do not need the device manager and can rely on the open Boost.Asio based prototype.

3 Concept

In this section we describe the main components, concepts and implemented OpenCL API features of dOpenCL. Furthermore, we present the Rodinia benchmark suite that we use to evaluate the performance of dOpenCL.

3.1 dOpenCL

We follow dOpenCL's original naming convention[9]: *Host* denotes the machine where user applications are running. These applications make use of computing hardware that is part of the *compute nodes*. In our case, the host is a virtual machine, whereas the local bare metal machine is the compute node.

Components. The dOpenCL middleware consists of three components as depicted in Figure 2. Client applications are linked against the dOpenCL *client driver*. This library provides binary compatibility with the OpenCL client libraries, so that any application linked against an OpenCL library can use the dOpenCL client driver without recompilation. Compute nodes in the network need to run the dOpenCL *daemon* in order to be accessible for dOpenCL clients. The daemon sets up an OpenCL context on its local machine to execute OpenCL calls on the actual hardware. Both client driver and daemon rely on the *communication library* that implements network transfers between host and compute node.

Remote Device Discovery. dOpenCL introduces a central *device manager* that dynamically assigns compute nodes available in the network to applications. Using a device manager for remote device discovery is called *managed mode* in dOpenCL (Figure 3). Initially, dOpenCL daemons register themselves and their local compute devices at the device manager. An application using dOpenCL is configured with a

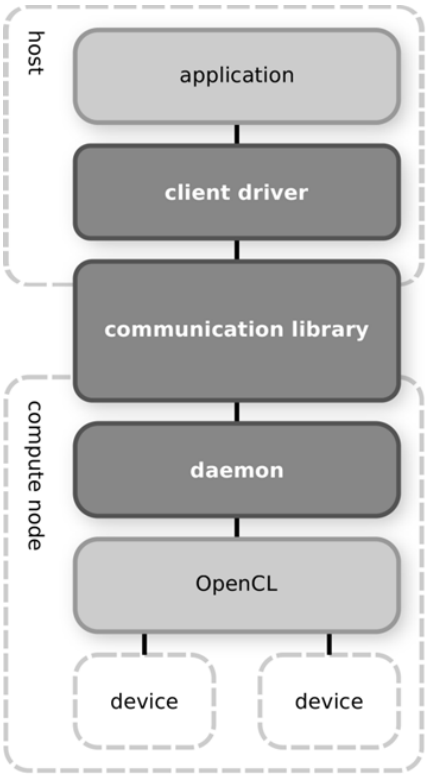


Figure 2: Stack of dOpenCL components. The user application is linked against the dOpenCL-library (client driver) that redirects all OpenCL calls through the communication library to dOpenCL daemons. Image source: [9]

set of properties that it requires for its OpenCL tasks. Based on this configuration, the client driver requests currently idle compute devices from the device manager (1). The device manager then assigns appropriate devices to the application (2). Relevant compute nodes are now informed of the device assignment (3a), and the list of nodes is sent back to the requesting application. In the last step, the client driver in the application requests its assigned devices from their respective compute nodes (4, 5). On the host side, this process is implemented transparently in the client driver. That way, contexts on remote OpenCL devices are set up in the same way as traditional local OpenCL contexts.

Besides the managed mode, dOpenCL provides a simpler setup where dOpenCL nodes are configured in a *dcl.nodes* file. This file is comparable to the *hosts* file that is used along with MPI. In this mode, no device manager is used, thus there is also no dynamic assignment of devices to multiple clients. However, for our current experiments, this simpler operation mode suffices. We run a single benchmark application at a time in a virtual machine, and its respective dcl.nodes file refers to the dOpenCL daemon running on the underlying bare metal machine (see also Section 2.2).

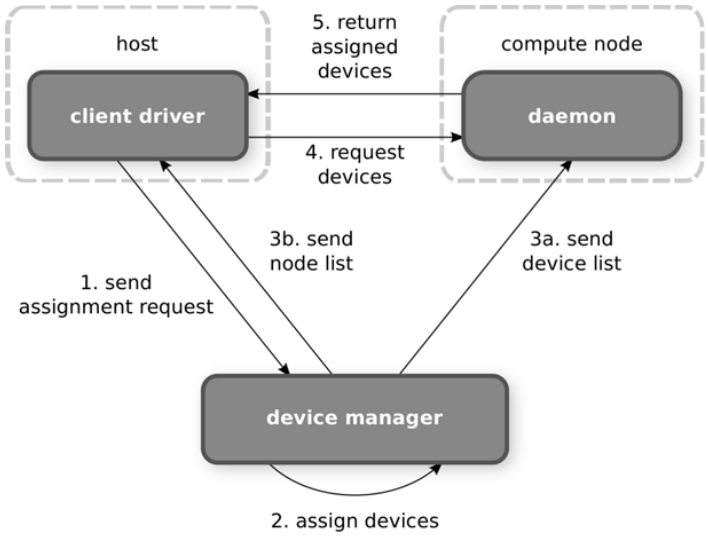


Figure 3: The managed mode allows the client driver to automatically discover available compute nodes in the network. For that, a central device manager exclusively assigns OpenCL devices to clients on demand. Image source: [9]

Comparison with the OpenCL API. dOpenCL implements a subset of the OpenCL API version 1.2[10]. Currently, it does not implement image and sampler APIs, sub-buffer APIs, vendor extensions and a few other functions. Besides, it is not meant to ever support interoperability with OpenGL or DirectX. Additionally to the OpenCL API, dOpenCL includes some experimental implementations for collective operations comparable to MPI. When using these function, application code is no longer compatible with OpenCL libraries anymore. However, especially for larger clusters, collective operations can potentially handle many operations more efficiently.

3.2 Rodinia Benchmark Suite

We use the *Rodinia Benchmark Suite*[2] for performance measurements. Rodinia is a set of benchmark applications designed for heterogeneous systems, and provides implementations for OpenCL, CUDA, and OpenMP. For our current tests, we only used the *Gaussian Elimination* benchmark included in the suite. Limiting to a single benchmark generally does not allow for a comparable performance evaluation [3]. However, in our case we only need to measure the overhead that is introduced by redirecting OpenCL calls – the executing hardware is the same, no matter whether the benchmark is started from inside a VM or directly on a bare metal machine.

During our experiments, we noticed a highly different degree of optimization in the set of benchmarks included in Rodinia. It also seems that most benchmarks are optimized primarily for specific NVIDIA GPUs, whereas we were using integrated and dedicated AMD GPUs. Furthermore, the Gaussian elimination benchmark we use for our evaluation shows irregularities that are probably not caused by the hardware or OpenCL/dOpenCL implementation, but rather by the benchmark

itself (see Section 4). For our purpose, however, these effects are not critical as we do not need to compare benchmarking results of different compute devices.

4 Evaluation

We evaluate the performance of dOpenCL on a desktop computer equipped with an integrated (AMD Radeon R7 "Spectre", APU: A10-7870K) and a dedicated (AMD FirePro W8100 "Hawaii") GPU. The detailed specifications of the test system are denoted in Table 1. We performed the Gaussian elimination benchmark included in Rodinia with different matrix sizes to measure the impact of task size on the resulting total runtime. We executed the benchmark on the bare metal machine using plain OpenCL to determine reference runtime. Additionally, we performed the same benchmark from within a KVM-based virtual machine using dOpenCL, with a dOpenCL daemon running on the bare metal machine. We did not succeed in running an applications using dOpenCL on the bare metal machine that is also running the dOpenCL daemon, as this caused deadlocks in the daemon.

Table 1: Specifications of the test systems.

CPU	AMD A10-7870K (Kaveri)
Memory	2 × 8GB PC3-17066U (DIMM)
Integrated GPU	AMD Radeon R7 Graphics (Spectre)
Dedicated GPU	AMD FirePro W8100 (Hawaii)
Operating system	Ubuntu Linux 15.10

In our benchmarks, we observed a generally small overhead introduced by dOpenCL compared to plain OpenCL (Figure 4). This behavior remains largely the same, irregardles of wether the integrated or the dedicated GPU are used. Also, both GPUs demonstrate an exceptional long runtime for a matrix size of $3200 \cdot 3200$ values, both using plain OpenCL and dOpenCL. We assume that this amplitude is issue caused by the interaction between the benchmark (see Section 3.2) and the GPU hardware we employed, but we did not investigate the issue any further.

Our measurements revealed, that for very small data set sizes, task execution through dOpenCL may lead to even faster execution times compared to plain OpenCL calls (Figure 5). Firstly, the benchmark running in a VM accesses hardware on the local physical machine, thus the networking latencies should be very low. Secondly, we assume that the dOpenCL daemon caches OpenCL contexts and states, so that consecutive executions of a benchmark may reuse a previously created OpenCL states. Consequently, the benchmark application effectively only initializes a dOpenCL platform and device, which transparently represents OpenCL objects. However, when running the benchmark application using plain OpenCL, it has to initialize the OpenCL platform and context in each run.

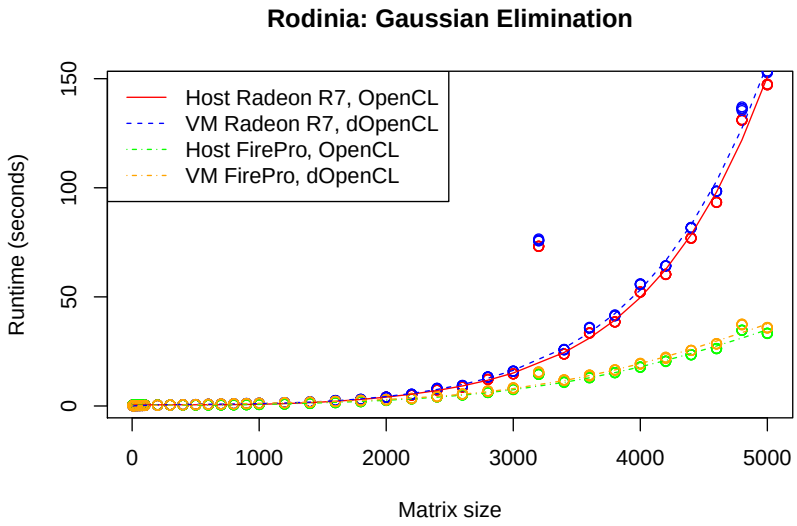


Figure 4: Runtimes of the Gaussian elimination benchmark on the integrated and dedicated GPU, using OpenCL (bare metal) and dOpenCL (VM). Note that the fitted lines do not take into account the high amplitudes at matrix size 3200.

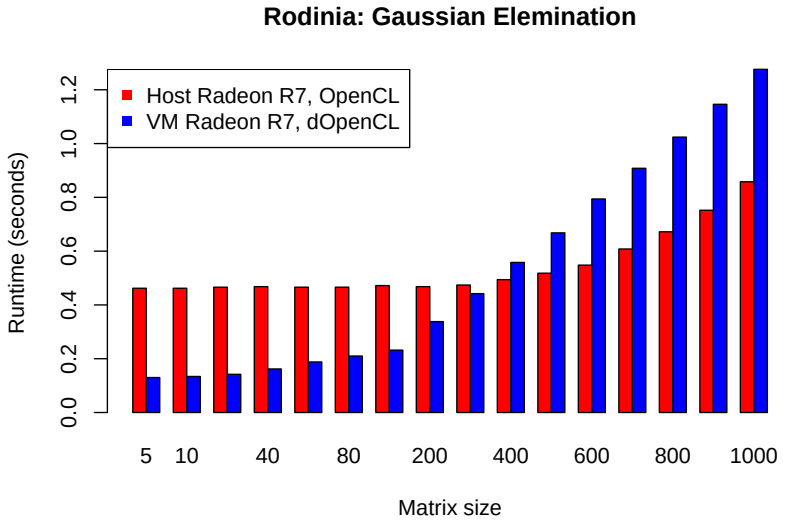


Figure 5: Detailed plot of runtimes for smaller matrices.

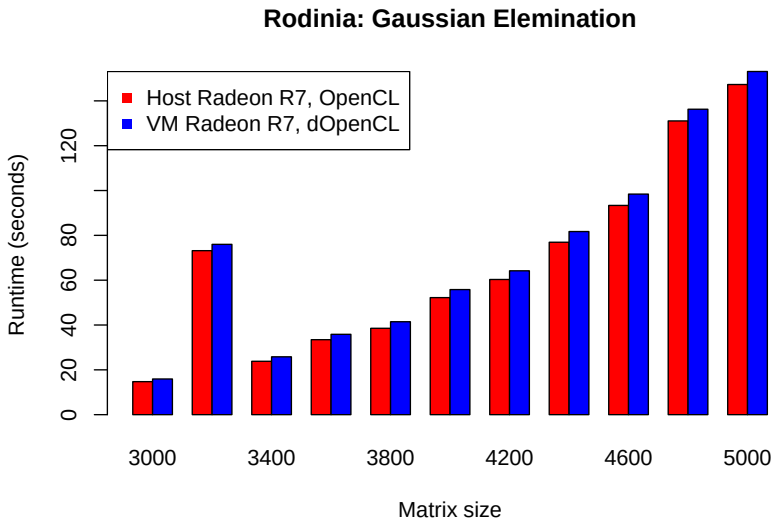


Figure 6: Even for larger data sets, forwarding OpenCL calls using dOpenCL is relatively inexpensive. The high amplitude for a matrix size of $3200 \cdot 3200$ is probably caused by the benchmark implementation.

For larger data sets, the overhead of dOpenCL compared to plain OpenCL remains constantly small (Figure 6). This is even the case for the high amplitude at a matrix size of 3200. Therefore, it can be assumed that the native execution time of an OpenCL task has little influence on the overhead induced by dOpenCL.

A relative comparison of the runtimes produced by using dOpenCL and plain OpenCL exposes three general trends (Figure 7). Firstly, as already noted, for very small data sets dOpenCL allows for shorter execution times compared to plain OpenCL. Secondly, for medium sized data sets, the overhead introduced by dOpenCL seems to exceed dOpenCL's initial speedup. In this case, using dOpenCL leads to an approximately doubled runtime in the worst case. Thirdly, for larger data sets, the asymptotic overhead of dOpenCL remains continuously smaller than 10%.

5 Conclusion

We evaluated dOpenCL as an OpenCL forwarding implementation to utilize OpenCL capable devices from within a local virtual machine. Compared to alternative approaches, dOpenCL is beneficial as it introduces little overhead, both in terms of deployment and in terms of execution time. It does not depend on a specific device vendor or virtualization product and is conceptually independent from a specific operating system. Also, dOpenCL allows for a dynamic assignment

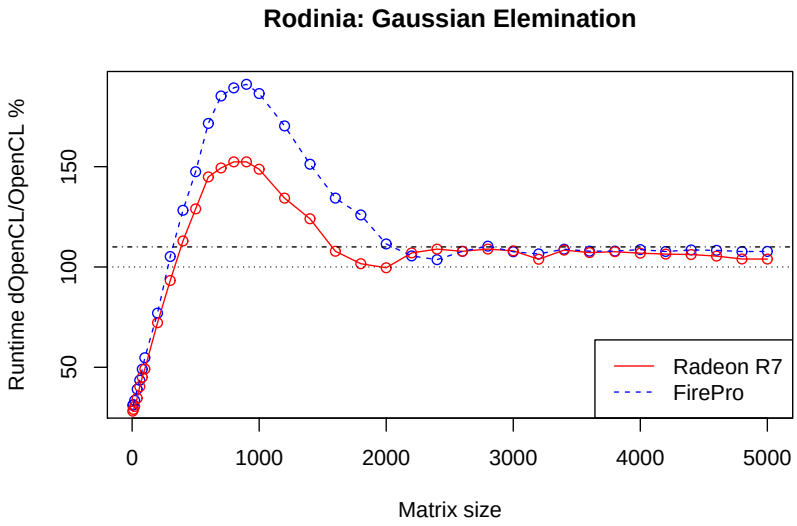


Figure 7: Average quotients of dOpenCL and OpenCL execution times. For very small data sets, dOpenCL is even faster than direct OpenCL calls, probably due to caching of OpenCL contexts in the dOpenCL daemon. For larger data sets, dOpenCL's overhead remains lower than 10% (dash-dot line).

of compute devices to applications. Compared to other solutions, compute devices are not linked to specific VMs for the lifetime of the VM.

Compared to plain OpenCL, dOpenCL introduces an asymptotic runtime overhead of less than 10%, which makes it efficiently usable in productive environments. In the worst case, for a limited range of task sizes, dOpenCL resulted in a roughly doubled runtime. When using dOpenCL, care must be taken to omit this effect for data set sizes and task runtimes that are employed. When spawning a large amount of quickly finishing OpenCL tasks, dOpenCL can be beneficial when used as back-end. Its daemon internally caches OpenCL states, so that the platform and device setup can significantly accelerate a shortly running kernels.

In further studies, the impact of the OpenCL task runtime and size of corresponding data sets could be evaluated separately. Furthermore, in our experiments we noticed unexpectedly high runtimes for specific parameter sets, both when using plain OpenCL and dOpenCL. These effects should be evaluated by analyzing the implementations of dOpenCL and Rodinia more thoroughly.

Acknowledgement

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866.

Disclaimer

This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] R. Aoki, S. Oikawa, R. Tsuchiyama, and T. Nakamura. "Hybrid OpenCL: Connecting Different OpenCL Implementations over Network". In: *Computer and Information Technology, International Conference on* (2010), pages 2729–2735. DOI: http://doi.ieeecomputersociety.org/10.1109/CIT.2010.457.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. "Rodinia: A benchmark suite for heterogeneous computing". In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Oct. 2009, pages 44–54. DOI: 10.1109/IISWC.2009.5306797.

[3] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads". In: *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*. IISWC '10. Washington, DC, USA: IEEE Computer Society, 2010, pages 1–11. ISBN: 978-1-4244-9297-8. DOI: 10.1109/IISWC.2010.5650274.

[4] J. Duato, F. D. Igual, R. Mayo, A. J. Peña, E. S. Quintana-Ortı, and F. Silla. "An Efficient Implementation of GPU Virtualization in High Performance Clusters". In: *Proceedings of the 2009 International Conference on Parallel Processing*. Euro-Par'09. Delft, The Netherlands: Springer-Verlag, 2010, pages 385–394. ISBN: 3-642-14121-8, 978-3-642-14121-8.

[5] J. Duato, A. J. Peña, F. Silla, J. C. Fernández, R. Mayo, and E. S. Quintana-Ortı. "Enabling CUDA acceleration within virtual machines using rCUDA". In: *18th International Conference on High Performance Computing, HiPC 2011, Bengaluru, India, December 18-21, 2011*. 2011, pages 1–10. DOI: 10.1109/HiPC.2011.6152718.

[6] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. "A GPGPU Transparent Virtualization Component for High Performance Computing Clouds". In: *Euro-Par 2010 - Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part I*. Edited by P. D'Ambra, M. Guarracino, and D. Talia. Berlin, Heidelberg: Springer Berlin Heidelberg,

2010, pages 379–391. ISBN: 978-3-642-15277-1. DOI: 10.1007/978-3-642-15277-1_37.

[7] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. "GViM: GPU-accelerated Virtual Machines". In: *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*. HPCVirt '09. Nuremburg, Germany: ACM, 2009, pages 17–24. ISBN: 978-1-60558-465-2. DOI: 10.1145/1519138.1519141.

[8] P. Kegel, M. Steuwer, and S. Gorlatch. *dOpenCL*. 2016. URL: http://dopencl-uni-muenster.de/.

[9] P. Kegel, M. Steuwer, and S. Gorlatch. "dOpenCL: Towards uniform programming of distributed heterogeneous multi-/many-core systems". In: *Journal of Parallel and Distributed Computing* 73.12 (2013). Heterogeneity in Parallel and Distributed Computing, pages 1639–1648. ISSN: 0743-7315. DOI: http://dx.doi.org/10.1016/j.jpdc.2013.07.021.

[10] Khronos Group. *The OpenCL Specification. Version: 1.2*. Nov. 14, 2012.

[11] 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*. ICS '12. San Servolo Island, Venice, Italy: ACM, 2012, pages 341–352. ISBN: 978-1-4503-1316-2. DOI: 10.1145/2304576.2304623.

[12] C. M. Kohlhoff. *Boost.Asio - 1.61.0*. 2016. URL: http://www.boost.org/doc/libs/1_61_0/doc/html/boost_asio.html.

[13] L. Shi, H. Chen, J. Sun, and K. Li. "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines". In: *IEEE Trans. Comput.* 61.6 (June 2012), pages 804–816. ISSN: 0018-9340. DOI: 10.1109/TC.2011.112.

[14] J. Song, Z. Lv, and K. Tian. *KVMGT: a Full GPU Virtualization Solution*. Oct. 2014.

[15] A. P. u. V. S. University of Muenster. *The Real-Time-Framework*. 2016. URL: http://www.uni-muenster.de/PVS/en/research/rtf/index.html.