The OpenCL Library Ecosystem: Current Status and Future Perspectives

Karl Rupp
Freelance Computational Scientist
Linke Bahnzeile 7/6
A-2486 Landegg, Austria
me@karlrupp.net

ABSTRACT

OpenCL as an open standard for parallel programming of heterogeneous systems seems to be an attractive choice for software library implementations. Indeed, iwocl.org[1] lists 83 OpenCL-enabled libraries as of February 12, 2016, suggesting a healthy library ecosystem. On closer inspection, however, a significant share of these libraries are either OpenCL bindings for other languages, libraries with OpenCL features in experimental state at best, or orphaned. Clearly, there is room for improvement; but what is required to improve the state of the OpenCL-enabled library ecosystem? Which future extensions to OpenCL can make library development easier? This talk aims to stimulate discussion by sharing lessons learnt in the area of high performance computing through the development of ViennaCL[2].

CCS Concepts

•**Computing methodologies** → *Parallel programming languages;* •**Software and its engineering** → **Software libraries and repositories;** *Just-in-time compilers; Reusability;*

Keywords

OpenCL; SPIR; SyCL; ViennaCL

1. INTRODUCTION

A rich software library ecosystem is important for building multi-layered software stacks pushing the limits of what can be done with available hardware. OpenCL can play an important role in such a software stack, as it allows for targeting different hardware architectures with a single kernel language and allows for running implementations on the best available hardware on the respective target machine. At the same time, the requirements on a software library in such as stack differ substantially from those of an application directly tailored to the needs of an end-user. A non-exhaustive list of key differences of a software library stack in comparison to an application for end-users is as follows:

1. **High number of kernels.** If several OpenCL-enabled libraries are supposed to interact, each will contribute its own set of compute kernels. These kernels need to be compiled at some point, which is usually achieved by the just-in-time compiler. Consider a family of kernels parameterized by indices (i, j), where the i-th entry of a buffer is set to j. Each of these kernels requires merely a single line of code in the body and is thus expected to impose small just-in-time compilation overhead. As Figure 1 shows, the compilation of 64 such kernels takes on the order of seconds, depending the number of OpenCL programs the kernels are distributed over. In many cases, a just-in-time overhead on the order of seconds is too high.

2. **Complex interaction of kernels.** Often a code skeleton is provided by a library, to which a library user provides customization through user-defined routines. Prominent example are custom operators for reductions or sorting to avoid unnecessary pre- and post-processing of data. The conventional approach in host languages such as C and C++ is to provide a callback routine. In OpenCL, the just-in-time compilation of kernels requires the user to either provide OpenCL kernel code, which is then just-in-time compiled with the full OpenCL program, or to rely on compiler-based approaches such as SyCL. Neither of these two options is fully satisfactory.

3. **Heterogeneous OpenCL support.** The OpenCL ecosystem evolves quickly, with a new standard being released every 18-24 months. Library implementers, however, often have to rely on the smallest common denominator among mainstream implementations (which to-date is OpenCL 1.1 or 1.2), because the resources for supporting different code branches for different OpenCL versions are not available. The problem is amplified by the average life-time of machines on the order of three to five years, where the software stack may not be up-to-date with the latest features.

Performance portability of OpenCL is not further considered in the following, because it is not a concern specific to a software stack rather than an application. Moreover, strategies for ensuring performance portability with OpenCL have already been developed [1, 3, 4, 5].

[1] http://www.iwocl.org/
[2] http://viennacl.sourceforge.net/

IWOCL '16 April 19-21, 2016, Vienna, Austria
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4338-1/16/04.
DOI: http://dx.doi.org/10.1145/2909437.2909452

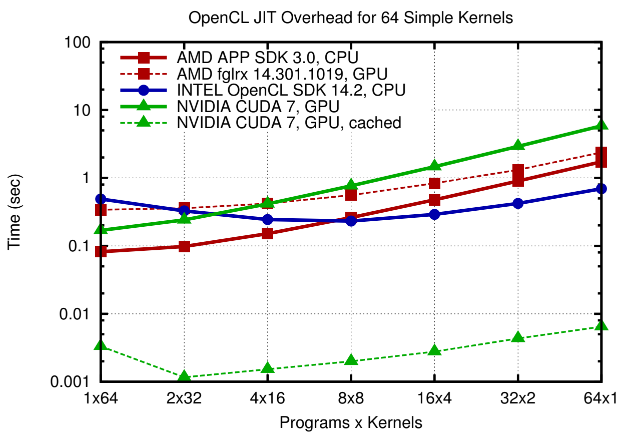

Figure 1: Overhead of the just-in-time compilation of 64 simple OpenCL kernels with different OpenCL SDKs [2]. The kernels are equally spread over a varying number of OpenCL programs. A kernel buffering mechanism can reduce the significant (possibly unacceptable) overhead on the order of seconds to the acceptable range on the order of microseconds. Unfortunately, such a caching mechanism as implemented in the NVIDIA CUDA 7 SDK is not provided by all SDKs.

2. POSSIBLE PATHS FORWARD

The three points discussed in the introduction underline that OpenCL needs to improve its attractiveness for software libraries. The availability of OpenCL as a shared library with a standardized C binary interface circumvents many problems of compiler-based approaches for programming massively parallel hardware. On the other hand, the problems outlined above need to be addressed. To do so, the following improvements are proposed for discussion in the community:

1. **Reducing overhead of just-in-time compilation**. A kernel cache can significantly reduce just-in-time compilation overheads, but most OpenCL SDKs do not provide such a cache (cf. Figure 1). While a library may implement such a cache by itself, the complex interaction with peculiarities of the filesystem is error-prone. Moreover, N different kernel caching mechanisms by N interacting OpenCL-enabled libraries is a maintenance nightmare for a complex software stack. Consequently, requiring the availability of (optional) kernel caching in an OpenCL SDK would benefit the whole library ecosystem.

   Another possible remedy for the overheads of just-in-time compilation is the standard portable intermediate representation (SPIR). Library maintainers can leverage SPIR to circumvent the kernel language front-end in the just-in-time compiler and to directly feed the machine code generators in the compiler backend. In addition to reduced just-in-time compilation overhead, the more low-level nature of SPIR may also offer better optimization opportunities. These enhancements to the OpenCL library ecosystem, however, require widespread availability of a stable SPIR, which is (as of early 2016) not the case.

2. **Better support for user-provided function pointers.** If OpenCL is used for execution on the host CPU, a user should be given the option to pass host-based function pointers (external C linkage) to OpenCL kernels. For example, an OpenCL work item should have explicit means to call thread-safe routines in external (static or shared) libraries. This would significantly improve composability of libraries (even if only on the CPU), allowing a mix-and-match of libraries with and without OpenCL support. In particular, OpenCL kernels would no longer be restricted to only calling other OpenCL routines.

3. **Fusion of OpenCL and Vulkan?** There may be incentives of different nature for certain vendors to only support an older OpenCL version, or to not support OpenCL at all. As such, once a single vendor possesses a certain market share, the vendor gets into the position of quasi-vetoing against OpenCL standards by not providing support. However, OpenCL with its cross-vendor nature can only be successful if it is available for all major hardware architectures on the market. In the context of graphics processing units, an integration of OpenCL into Vulkan is an option to resolve an eventual gridlock if a vendor decided to only support Vulkan, but not OpenCL.

3. CONCLUSIONS

The OpenCL ecosystem with a strict separation of host code and device code imposes additional challenges for software library development. This results in a chicken-and-egg problem: Only high demand will push vendors to provide OpenCL SDKs of highest quality. However, without high-quality OpenCL SDKs, it is unnecessarily difficult to build a rich set of software libraries to encourage users to demand optimized OpenCL SDKs. A reduction of just-in-time compilation overhead, better support for user-provided function pointers on CPUs and, to the extent possible, GPUs, and a merge of OpenCL and Vulkan to encourage better vendor implementations are the possible paths outlined here to solve this chicken-and-egg problem.

4. REFERENCES

[1] P. Jääskeläinen, C. Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming*, 43(5):752–785, 2014.

[2] K. Rupp. OpenCL Just-In-Time (JIT) Compilation Benchmarks, 2016. https://www.karlrupp.net/2016/01/opencl-just-in-time-jit-compilation-benchmarks/.

[3] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, T. Grasser, and A. Jüngel. Performance Portability Study of Linear Algebra Kernels in OpenCL. In *Proceedings of the International Workshop on OpenCL (IWOCL 2013-2014)*, pages 8:1–8:11. ACM, 2014.

[4] P. Tillet, K. Rupp, S. Selberherr, and C.-T. Lin. Towards Performance-Portable, Scalable, and Convenient Linear Algebra. In *5th USENIX Workshop on Hot Topics in Parallelism (HotPar'13)*, 2013.

[5] Y. Zhang, M. Sinclair, and A. Chien. *Proceedings of the 28th International Supercomputing Conference (ISC 2013)*, chapter Improving Performance Portability in OpenCL Programs, pages 136–150. Springer, 2013.