Performance portability with alpaka

J Stephan[1,2,4], S Bastrakov[2,5], A Di Pilato[1,2,6], S Ehrig[1,2], B M Gruber[1,2,3,4], J Vyskočil[1,2], R Widera[2] and M Bussmann[1,2]

[1] Center for Advanced Systems Understanding, Untermarkt 20, 02826 Görlitz, DE
[2] Helmholtz-Zentrum Dresden-Rossendorf, Bautzner Landstraße 400, 01328 Dresden, DE
[3] CERN, Esplanade des Particules 1, 1211 Meyrin, CH
[4] Technische Universität Dresden, 01062 Dresden, DE

E-mail: `j.stephan@hzdr.de`

Abstract. The alpaka library is a header-only C++17 abstraction library for development across hardware accelerators (CPUs, GPUs, FPGAs). Its aim is to provide performance portability across accelerators through the abstraction of the underlying levels of parallelism. In this paper we will show the concepts behind alpaka and how they are mapped to the various underlying hardware models. In addition, we will also present the software ecosystem surrounding alpaka.

1. Introduction

Over the last two decades, the hardware landscape in the field of high-performance computing (HPC) has changed drastically. After making their first appearance in the TOP500 list in 2008 with the TSUBAME supercomputer [1] graphics processing units (GPUs) are now available on many different HPC systems, including seven of the top ten systems in the current TOP500 list [2]. Field-programmable gate arrays (FPGAs) with high-bandwidth memory (HBM) are available in data centers [3], while various companies are developing specialized accelerators for artificial intelligence (AI) [4]. In addition, new designs for central processing units (CPUs) have appeared, such as the Fujitsu A64FX processor with 32 GiB of HBM memory hardwired to the 48 CPU cores [5].

These increasingly heterogeneous hardware setups pose a challenge for programmers and scientists since they have to adapt to the different processor architectures and accelerator types. On the one hand, different vendors often offer incompatible programming models for their hardware. On the other hand, performance optimizations require detailed knowledge about a specific target hardware, but this knowledge usually does not apply to different hardware types or may even be counterproductive.

These aspects have a severe impact on a program's portability and maintainability, potentially leading to a great amount of technical debt (see E. Allman's introduction to the management of Technical Debt [6]). Therefore we propose to use abstractions which mitigate this impact [7]. However, those come with a set of obstacles: How is a parallel problem expressed in an abstract and user-friendly fashion? How are these expressions transformed into hardware-aware

[5] Present address: Northern Data AG, An der Welle 3, 60322 Frankfurt am Main, DE
[6] Present address: Section de Physique, Université de Genève, 24 Rue du Général-Dufour, 1211 Genève 4, CH

algorithms and data structures? How can one achieve maximum performance without sacrificing portability, i.e. achieve *performance portability*?

To tackle these challenges we have developed the *Caravan* ecosystem with the kernel abstraction library *alpaka* at its core.

2. Related Work

Originally, alpaka started as a Master's Thesis project [8] and has been in continuous development ever since[7] (see Zenker et al. [9] and Matthes et al. [10] for more recent alpaka-related publications). Libraries like RAJA [11] and Kokkos [12] share alpaka's goal of performance-portable C++ programming but follow different programming philosophies to achieve it.

Apart from the library-based solutions listed above there are also two industry standards available: OpenCL [13] and SYCL [14]. They are different from alpaka in a few aspects: Both are API specifications provided by an industry consortium. Industry players (hardware vendors and third parties) need to provide an implementation suitable for their specific set of hardware. This results in varying support across vendors. Sometimes vendors do not (yet) support a newer revision of the specification; in other cases they rely on custom extensions to maximize performance on their hardware.

In addition, OpenCL is a split-source language (in contrast to the other solutions which are single-source C++ APIs): The host program is coded in C, C++ or another language while the device code is written in the OpenCL C(++) dialect and requires separate compilation at some point.

3. Mapping to hardware models

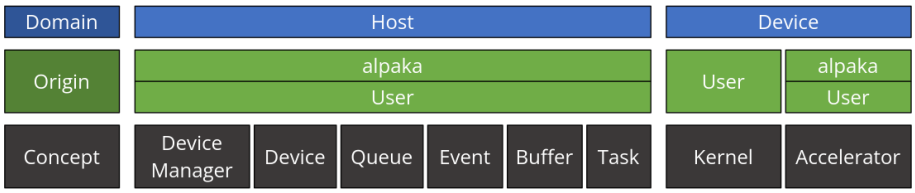

Figure 1. Overview of alpaka's concepts

Conceptually, alpaka is split into two domains: *host* and *device* (see Figure 1). The *host* controls the overall program flow which includes device management, memory management and synchronization with the device(s). Attached to the host are one or more *devices* which are used for the execution of computational tasks (called kernels). A *device* can be any kind of accelerator, like a GPU or a multi-core CPU.

Kernels are implemented using standard C++. However, alpaka is influenced by CUDA's kernel language and uses similar concepts for kernel development, for example *grids*, *blocks* and *threads* for work division. In addition, it supports special device-side functionality, such as mathematical functions which correspond to those from the C++ standard library (including complex numbers), atomics or on-device synchronization. All kernels require a generic C++ template parameter which is propagated through an alpaka program and used for back-end specialization. A target platform is selected as an in-program data type at compile time. Kernels are mapped to specific target hardware architectures during compilation via template instantiation.

[7] alpaka is available on GitHub: https://github.com/alpaka-group/alpaka

alpaka's host-side abstractions (like memory buffers or device queues) are resolved in a similar way to kernels. Because alpaka internally makes heavy use of C++ template metaprogramming techniques the abstraction layers are removed at compile time and therefore incur no runtime overhead.

Additionally, alpaka is designed to be user-extensible. For this purpose alpaka's API comes with a set of requirements, called *concepts*, which are inspired by C++20 concepts. As long as a user-defined abstraction (for example a custom device queue) fulfills the corresponding concept it can simply be used in place of the standard alpaka abstraction.

alpaka currently supports a variety of different platforms that can be targeted using the appropriate back-end. These platforms include (among others) CUDA for NVIDIA GPUs, ROCm for AMD GPUs, (experimentally) SYCL for oneAPI hardware (including FPGAs) and OpenMP or TBB for multi-core CPUs.

4. The Caravan ecosystem

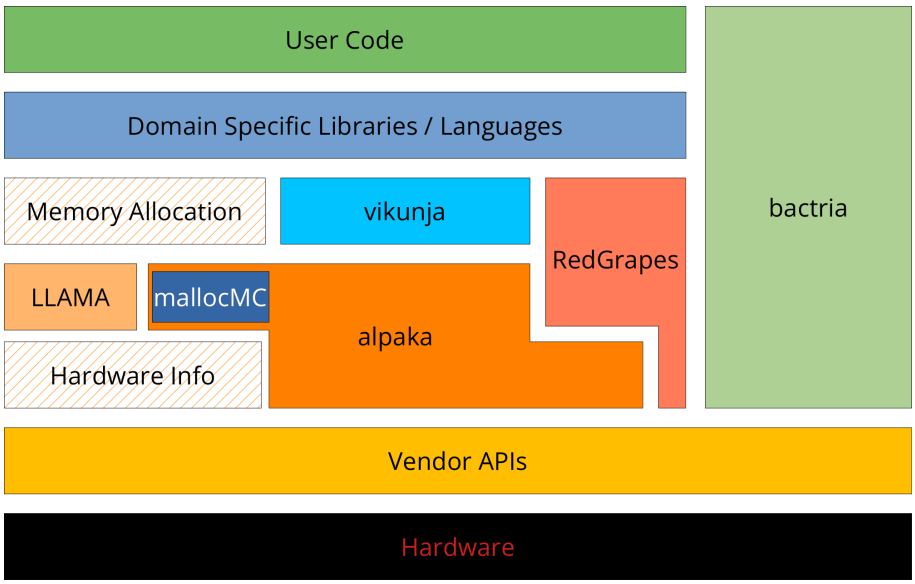

Figure 2. Overview of the Caravan ecosystem. The shaded libraries do not yet exist.

Over the past few years we have developed an ecosystem of libraries related to alpaka called *Caravan* (see Figure 2. With the exception of Vikunja (presented in subsection 4.3) and mallocMC (presentend in subsection 4.4) all projects are orthogonal to alpaka: They can be used as standalone products or in a codebase that integrates both alpaka and the individual libraries.

4.1. bactria

Fine-grained performance analysis is an important part of software engineering in an HPC environment. The programming platforms for the various hardware accelerators also offer functionality for user-defined code instrumentation, such as the NVIDIA Tools Extensions (NVTX) or AMD's ROCTX. However, like their general purpose API counterparts they are not portable to other vendors' hardware. The *Broadly Applicable C++ Tracing and Instrumentation API* (bactria)[8] is a work-in-progress library which offers a unified C++17 API for user-defined

[8] bactria is available on GitHub: `https://github.com/alpaka-group/bactria`

code instrumentation. At runtime, performance analysis can be achieved by dynamically loading one of bactria's plugins. The plugin will map bactria's user-facing API onto the hardware platform's instrumentation API. The output of a program instrumented with bactria can then be analyzed with the tools accompanying the native instrumentation APIs, like Nsight Systems for CUDA or `rocprof` + Perfetto UI for ROCm.

4.2. LLAMA

Choosing the best memory layout for each hardware architecture is increasingly important as more and more programs become memory bound. The *low-level abstraction of memory access* (LLAMA)[9] is a standard C++ library that provides a zero-runtime-overhead abstraction layer, underneath which memory mappings can be freely exchanged to customize data layouts, memory access, access instrumentation and memory allocation. After its scientific debut [15], several improvements and extensions have been added to LLAMA. These latest additions are discussed in our most recent update on LLAMA [16], and in a case study where we demonstrate LLAMA's memory instrumentation capabilities to optimize a particle transport simulation [17].

4.3. Vikunja

Many algorithms can be expressed as variations of software patterns commonly used in parallel programming. However, implementing these common patterns by hand in a portable and performant way is no trivial task. Vikunja[10] is a performance portable C++ algorithm library that defines functions operating on ranges of elements for a variety of purposes. It is implemented using alpaka and therefore supports program execution on various accelerators. The user interface is similar to the C++ standard library which makes the usage of common parallel algorithms (like reductions and transformations) easy. In addition, Vikunja provides full interoperability with alpaka. This allows for mixed execution of the various Vikunja-defined algorithms and user-defined alpaka kernels on the same memory buffers.

4.4. mallocMC

Many algorithms require dynamic memory allocations in device code, for example to create linked data structures. However, the programming platforms for accelerator devices either do not offer such functionality or it entails high latencies. Our C++17 library mallocMC[11] allows for on-device, dynamic, chunked allocations by utilizing a memory pool which was pre-allocated from the host side. It allows to exchange the allocation algorithm without introducing runtime overhead or the necessity to refactor the application code. This is again achieved via compile time abstractions using C++ templates. As one possible allocation method, mallocMC ships our implementation of the ScatterAlloc algorithm [18], modified from Steinberger et al. [19], which shows a performance increase over the native CUDA on-device allocation method.

4.5. RedGrapes

Modern compute nodes can concurrently perform computational tasks over various resources, such as memory pools, cores and accelerator devices. In order to achieve high scalability in such a system, communication and computation tasks need to be overlapped extensively. However, the manual management of data and execution dependencies is a tedious and error-prone task. Real-world applications often use global shared states or vary the workload at runtime. In addition, asynchronous communication models complicate the program flow even further. The

[9] LLAMA is available on GitHub: `https://github.com/alpaka-group/llama`
[10] Vikunja is available on GitHub: `https://github.com/alpaka-group/vikunja`
[11] mallocMC is available on GitHub: `https://github.com/alpaka-group/mallocMC`

RedGrapes[12] library is an attempt to tackle the aforementioned challenges. It provides a light-weight, application-level C++ programming framework for the creation of task-graphs. These are directed acyclic graphs (DAGs) whose vertices represent computational or communication tasks and whose edges denote the execution order.

5. Performance

The approach to hardware abstraction presented in the previous section results in machine code that is similar to what is generated by using the native programming models. alpaka's performance has been evaluated recently by Markomanolis et al. [20] and Valassi et al. [21]. Both groups demonstrated that code written in alpaka's kernel language achieves performance which is close to code written in the native counterparts. In the case of Markomanolis et al., the alpaka-based benchmarks consistently achieve more than 97% of their native counterparts (with the maximum at 99.99%) across GPUs from different vendors.

6. Summary

We have presented the kernel abstraction library *alpaka* and its surrounding software ecosystem *Caravan* as solutions to performance portability of scientific applications as well as the avoidance of technical debt. Through abstraction using C++ template metaprogramming techniques this collection of libraries enables the development of portable code bases and provides performance close to vendor's native programming models.

Acknowledgments

This work was partially funded by the Center of Advanced Systems Understanding (CASUS) which is financed by Germany's Federal Ministry of Education and Research (BMBF) and by the Saxon Ministry for Science, Culture and Tourism (SMWK) with tax funds on the basis of the budget approved by the Saxon State Parliament.

This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant no. 13E18CHA).

References

[1] Meuer H W, Strohmeier E, Dongarra J and Simon H 2008 TOP500 List – November 2008 URL `https://www.top500.org/lists/top500/list/2008/11/`

[2] Meuer M, Strohmeier E, Dongarra J and Simon H 2022 TOP500 List – November 2022 URL `https://www.top500.org/lists/top500/list/2022/11/`

[3] Bobda C *et al.* 2022 *ACM Transactions on Reconfigurable Technology and Systems* **15** ISSN 1936-7406

[4] Varghese B *et al.* 2019 *Computer* **52** 68–77 ISSN 0018-9162

[5] Odajima T, Kodama Y, Tsuji M, Matsuda M, Maruyama Y and Sato M 2020 *Proceedings of the 2020 IEEE International Conference on Cluster Computing (CLUSTER 2020)* ed Kondo M and Cappello F (Los Alamitos, CA, USA: IEEE Computer Society) pp 523–530 ISBN 978-1-7281-6677-3

[6] Allman E 2012 *Communications of the ACM* **55** 50–55 ISSN 0001-0782

[7] Sbalzarini I F 2012 *Technology Integration Advancements in Distributed Systems and Computing* ed Bessis N (Hershey, PA, USA: IGI Global) chap 10, pp 161–178 1st ed ISBN 978-1-4666-0906-8

[8] Worpitz B 2015 *Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures* Master's thesis Technische Universität Dresden Dresden, SN, Germany

[9] Zenker E, Worpitz B, Widera R, Hübl A, Juckeland G, Knüpfer A, Nagel W and Bussmann M 2016 *2016 IEEE 30th International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2016)* ed Hollingsworth J (Los Alamitos, CA, USA: IEEE Computer Society) pp 631–640 ISBN 978-1-5090-2140-6

[10] Matthes A, Widera R, Zenker E, Worpitz B, Hübl A and Bussmann M 2017 *High Performance Computing. ISC High Performance 2017* (*Lecture Notes in Computer Science* vol 10524) ed Kunkel J M, Yokota R, Taufer M and Shalf J (Cham, ZG, Switzerland: Springer) pp 496–514 ISBN 978-3-319-67629-6

[12] RedGrapes is available on GitHub: `https://github.com/ComputationalRadiationPhysics/redGrapes`

[11] Beckingsale D A, Burmark J, Hornung R, Jones H, Killian W, Kunen A J, Pearce O, Robinson P, Ryujin B S and Scogland T R 2019 *Proceedings of P3HPC 2019: International Workshop on Performance, Portability and Productivity in HPC* ed Doerfler D, Neely R, Pennycook J and Newburn C J (Los Alamitos, CA, USA: IEEE Computer Society) pp 71–81 ISBN 978-1-7281-6003-0
[12] Trott C R *et al.* 2022 *IEEE Transactions on Parallel and Distributed Systems* **33** 805–817
[13] The Khronos® OpenCL Working Group 2022 *The OpenCL™ Specification* The Khronos Group, Inc. Beaverton, OR
[14] The Khronos® SYCL™ Working Group 2022 *SYCL™ 2020 Specification (revision 6)* The Khronos Group, Inc. Beaverton, OR
[15] Gruber B M, Amadio G, Blomer J, Matthes A, Widera R and Bussmann M 2022 *Journal of: Software: Practice and Experience* **53** 115–141
[16] Gruber B M 2023 Updates on the Low-Level Abstraction of Memory Access 21st International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT 2022)
[17] Gruber B M, Amadio G and Hageböck S 2023 Challenges and opportunities integrating LLAMA into AdePT 21st International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT 2022)
[18] Eckert C 2014 *Enhancements of the massively parallel memory allocator ScatterAlloc and its adaption to the general interface mallocMC* Research project Technische Universität Dresden Dresden, SN, Germany
[19] Steinberger M, Kenzel M, Kainz B and Schmalstieg D 2013 *2012 Innovative Parallel Computing (InPar 2012)* (New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE)) pp 65–74 ISBN 978-1-467-32632-2
[20] Markomanolis G S *et al.* 2022 *Supercomputing Frontiers. 7th Asian Conference, SCFA 2022* (*Lecture Notes in Computer Science* vol 13214) ed Panda D K and Sullivan M (Cham, ZG, Switzerland: Springer) pp 79–101 ISBN 978-3-031-10418-3
[21] Valassi A, Childers T, Field L, Hageböck S, Hopkins W, Mattelaer O, Nichols N, Roiser S and Smith D 2022 *Proceedings of 41st International Conference on High Energy physics* arXiv. Preprint. URL `https://arxiv.org/abs/2210.11122`