Prioritizing Platform Portability for Scientific Libraries

Hartwig Anzt and Terry Cojean

What Makes Software Sustainable?

When thinking about what makes software sustainable, the first associations might be the existence of a continuous integration (CI) framework; the existence of a testing framework composed of unit tests, integration tests, and end-to-end tests; and also the existence software documentation. Following those closely, we then think of platform portability and efficiency. However, when asking what is a common deathblow for a scientific software product, it is often the lack of platform portability.

When surveying the research software landscape, we can identify some products that have a lifetime that far exceeds the typical 3–5 years that a specific supercomputer is operational, and some of those software products have been around for several decades (Arndt et al. (2019); The Trilinos Project Team (2020)). On the other hand, some libraries are successful for a period of time and then fade out. When investigating the source of decline for some products, it is often that the jump from one hardware architecture to the next was too big, and the product failed to keep up with the development of other software and hardware ecosystems.* In that sense, the lack of software portability and the lack of flexibility to embrace future hardware designs is a time bomb that limits the lifetime of software.

The Importance of Platform Portability Grows

The lack of platform portability is becoming a critical weakness as we see an explosion of diversity in hardware architectures employed in supercomputers. In the last century, the hardware development was mostly incremental, as it was driven by the clock frequency increase of the processors (Schaller (1997)). During that time, the software developers usually succeeded in transferring to newer chip technologies by applying minor modifications or by simply leveraging the "free lunch" (Sutter (2005)) that came with higher operating frequency. That said, the move from single-core processors to multi-core processors in the early 21st century was incremental enough to be mastered by many software products that did not embrace platform portability as central design principle. This is partly because using pragmas and the OpenMP language allowed for a smooth transition. In addition, only the performance—not the functionality—of software was endangered when ignoring multi-threaded or multi-core hardware capacity. In fact, single-threaded software remains functional and can still achieve acceptable performance. However, at least since the rise of many-core accelerators (e.g., GPUs) and the adoption of special function units and lightweight ARM processors for supercomputing, software libraries can no longer ignore the hardware changes. As a consequence, the lack of platform portability for emerging and future hardware technology is among the main threats for the sustainability of a given software product. Unfortunately, we already have a wide range of hardware architectures deployed in supercomputers, and many of these chips come with their own programming language and intrinsic routines.

The Levels of Portability

There are multiple levels to portability. Depending on the use case, platform targets, and objectives, some applications may find it sufficient to restrict themselves to a specific portability level. The first distinct level is *no portability*, where the code compiles and runs for only one type of high-performance computing (HPC) system. The same sort of hardware and compute capabilities are expected. Another option is to support *partial software portability*. An application using such a model will be dependent on some platform model abstraction. For example, the model could expect any CPU type combined with one or more accelerators, either from AMD or NVIDIA. In such a case, a hybrid programming approach featuring a CPU programming model like OpenMP is combined with an accelerator programming model like HIP to ensure portability (and possibly good performance) on the machine. As a more advanced case, one might consider *full software portability*, where the application is able to execute and run on any type of platform, including hypothetical future machines that might feature field-programmable gate arrays (FPGAs). In this case, a practical example is the SYCL programming model, which features compiler backends that support some FPGAs, all mainstream HPC accelerators, and ARM-based hardware. Finally, and especially important for HPC applications, there is the level of *performance portability*, which means that the code will not only compile and run on target platforms, but it will also achieve high efficiency by providing performance close to the machine's total capabilities. To achieve performance portability, one needs good software design practices (e.g., code portability) *and* full command and understanding of the problems inherent in computing unit granularity vs. problem

*Another reason is often the lack of resources for sustained software development, but here we refrain from addressing the topic of under-funding the field of research software engineering.

granularity. The latter requires using specific programming techniques to fully express an application's parallelism and scheduling to spread the workload, dynamically, depending on the machine hardware's computing units.

Designing for Platform Portability

Ignoring efforts that are likely doomed to fail, as they permanently redesign to reflect the changes in hardware architecture, one can identify two different approaches to enable cross-platform portability and readiness for future hardware architectures. One approach is the adoption of a hardware portability layer that is devoted to supporting hardware through an abstraction. Popular examples are the Kokkos (Carter Edwards et al. (2014)) abstraction layer and the Raja (Beckingsale et al. (2019)) abstraction layer; both are extremely successful in supporting new hardware technologies and providing the users with a unique interface that allows applications to run the same parallel code on very different hardware architectures. The software products that rely on these abstractions have virtually no porting effort, and are guaranteed to run on current and future hardware supported by the abstraction.

The second approach is to decouple the library-core functionality from hardware-specific kernels and support the backends for different hardware (e.g., the Ginkgo software package). From a high-level, one could argue that the second approach takes the first approach and combines the high-level algorithms, the hardware abstraction layer, and the hardware-specific kernel support into a single software product. However, unlike the first approach, this supports hand-written optimized kernels for each hardware architecture. In addition, the abstraction and kernel development being focused on a single product allows for a more consumer-specific kernel design and performance optimization.

Relying on a Portability Layer

Relying on a portability layer removes the burden of platform portability from the library developers and allows them to focus exclusively on the development of sophisticated algorithms. This convenience comes at the price of a strong dependency on the portability layer, and moving to another programming model or portability layer is usually extremely difficult or even impossible. Furthermore, relying on a portability layer naturally implies that the performance of algorithms and applications is determined by the quality and hardware-specific optimization of the portability layer. This performance penalty may not always be insignificant, as portability layers usually have a wide user base, and dramatic changes to the interface, logic, or kernel design of the portability layer would likely result in the failure of some applications that rely on the portability layer. Hence, performance portability layers should avoid modifying the design or hardware coverage, which can limit the opportunities to heavily optimize kernels for a new hardware architecture.

Natively Supporting Various Hardware Backends

Libraries that decouple the core algorithms from the hardware-specific kernels and supporting various hardware backends can apply much more aggressive hardware-specific optimization and often achieve higher performance. One reason is that the set of kernels is usually much smaller than what portability layers provide as the hardware-specific backends, because only the kernels required by the library's core algorithms are included. A second reason is that a library has more freedom to phase out support for a specific hardware architecture. This can usually be justified because the dependency on a library is generally much looser than the dependency on a portability layer, and applications "just" need to find a new library that provides the same functionality, while the much deeper dependency on a portability layer virtually prohibits moving to an alternative portability layer. To use this model, a library must be designed with modularity and extensibility in mind. Only a library design that relies on separation of concerns between the parallel algorithm and the different hardware backends can allow such a feature. The different backends need to be managed and interacted with thanks to a specific interface layer between algorithms and kernels. However, the price for the higher performance potential is high: the library developers have to synchronize several hardware backends, monitor and react to changes in compilers, tools, and build systems, and adopt new hardware backends and programming models. The effort of maintaining multiple hardware backends and keeping them synchronized usually results in a significant workload that can easily exceed the developers' resources.

Decisions in Platform Portability

We are unable to provide general advice whether relying on a portability layer or natively supporting various backends is the better choice. This decision depends on the scope of the library, the tradeoff between performance and platform coverage, and the available resources. However, we are convinced that only software packages that provide platform portability and are able to jump to new hardware architectures are sustainable and can play a role in the very diverse world of HPC. We therefore believe strongly that platform portability is a central design principle in the creation and development of research software libraries.

References

Arndt D, Bangerth W, Clevenger TC, Davydov D, Fehling M, Garcia-Sanchez D, Harper G, Heister T, Heltai L, Kronbichler M, Kynch RM, Maier M, Pelteret JP, Turcksin B and Wells D (2019) The `deal.II` library, version 9.1. *Journal of Numerical Mathematics* 27(4): 203–213. DOI:10.1515/jnma-2019-0064. URL `https://dealii.org/deal91-preprint.pdf`.

Beckingsale D, Hornung R, Scogland T and Vargas A (2019) Performance portable c++ programming with raja. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19. New York, NY, USA: Association for Computing Machinery. ISBN 9781450362252,

p. 455–456. DOI:10.1145/3293883.3302577. URL https://doi.org/10.1145/3293883.3302577.

Carter Edwards H, Trott CR and Sunderland D (2014) Kokkos. *J. Parallel Distrib. Comput.* 74(12): 3202–3216. DOI:10.1016/j.jpdc.2014.07.003. URL https://doi.org/10.1016/j.jpdc.2014.07.003.

Schaller RR (1997) Moore's law: Past, present, and future. *IEEE Spectr.* 34(6): 52–59. DOI:10.1109/6.591665. URL https://doi.org/10.1109/6.591665.

Sutter H (2005) The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal* 30(3): 202–210. URL http://www.gotw.ca/publications/concurrency-ddj.htm.

The Trilinos Project Team (2020) The Trilinos Project Website URL https://trilinos.github.io.