Productivity, Portability, and Performance: Pick any two in Uncertain Hardware Landscape?

Piotr Luszczek

June 29, 2020

1 INTRODUCTION

Focusing only on the second stage of the US Department of Energy's (DOE's) CORAL hardware deployments, exascale machines, and their ever-increasing complexity, becomes even more exacerbated in a wider scope of compute-intensive systems. Some of the post-petascale and pre-exascale systems already exhibited this increased complexity trend. This highlights the need to to utilize the effort of scientific programmers on the functionality and scalability of numerical libraries, the need for *performance portability* without sacrificing *productivity* is greater than ever. Often enough, these three P's—PERFORMANCE, PORTABILITY, and PRODUCTIVITY—constitute conflicting requirements at the exascale regime. Our goal is to redirect the developer productivity away from the platform-specific performance engineering to writing portable and performance-conscious code that is seamlessly deployed with automated adaptation to the exascale hardware platforms by using the performance constraints of the target machine.

Transitioning the existing library codes from custom GPU acceleration to full GPU residency, and from plain GPU-offload to advanced CPU reverse-offload, will continue to be disruptive. The tacit assumption about optimal library performance across the spectrum of deployment targets will slowly come undone. This is a result of the combinatorial growth of the number of hardware platforms and their possible software configurations. Consider a single library routine running across all available CPU sockets and GPU accelerators for a common input data. This is the showcase scenario that is optimized by the library developers to feature the best performance gains for the library and the hardware. However, any departure from the ideal configuration will result in sub-optimal efficiency, as this represents less-than-desirable conditions for the library to achieve close-to-peak performance (whatever the peak may be for the library, function, and hardware combination). Exascale Computing Project (ECP) applications integrate multiple libraries as the cross-library integration effort have been expressly stated goal from the very inception of the program. With all the software pieces coming together in a single application run while the constituent libraries that were optimized in separation. Worst yet, the hardware, OS, and middleware can be configured optimally for only a single purpose and will likely not fit well with most of the software components that the application uses. Runtime scheduling, by OpenMP, Kokkos, RAJA, etc., further complicates the situation with runtime-only effects that cannot be repeated let alone reliably reproduced on the developer machine due to the plethora of factors such as system configuration, software versioning, and a variety of hardware constraints such as limited availability of components outside of specific NDA and contractual agreements.

2 RUNTIME MIDDLEWARE: SCHEDULING AND COMMUNICATION

Runtime scheduling with adaptive load balancing has been used in scalable applications, and ECP is no exception. It can alleviate many of the issues often plaguing HPC codes. However, the limiting factor is the library performance that the runtime can only schedule around, but it is not optimized. In other words, the library kernels are black boxes invoked by the runtime as tasks. The kernel performance and cooperation across nodes inside communication tasks must be optimized by the library developer and can only be minimally improved by the runtime. For example, unblocked matrix-multiply is inherently inefficient and so are global collectives that use heavily unbalanced trees. No scheduling runtime can affect these fundamental flaws of implementation. Therefore, the library development team is responsible for the optimization and must take into account the plethora of execution scenarios that the runtime is likely to use during the application run. Due to the complexity of multi-GPU distributed memory development process, such comprehensive optimization accommodations are rarely done let alone possible and we aim to tackle this issue. By exposing performance configurations, we made them amenable to statistical modeling and thus remove the productivity burden of the shoulders of core development team and off-load it to automate integration pipelines and data collection systems for automated optimization provisioning.

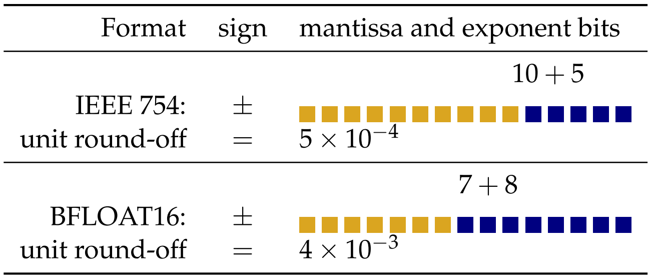

Figure 1: Half precision representations in industry standards and modern hardware. Yellow squares 🟨 represent mantissa bit and navy blue squares 🟦 represent exponent bit.

Table 1: Summary of floating point standards.

Acronym	Name	mantissa	exponent
fp128	quad	122	15
fp80	extended	64	15
fp64	double	52	11
fp32	single	23	8
fp16	half	10	5
tf32	tensor float	10	8
bf16	brain float	7	8

3 REPRODUCIBILITY: DRAG ON PRODUCTIVITY BUT CRUCIAL FOR COMPARING PERFORMANCE

One of the goals is also to correlate the performance configuration of the numerical libraries with the experimental testing data on ECP hardware. This would enable us to increase the reproducibility of the runs and their results. These correlations can be compared statistically between users' machines outside of the development environment. They can also be compared between the users' and developers' machines to discover performance regressions or to reproduce performance issues. Owing to the natural variables of modern hardware execution, Machine Learning (ML) metrics will be used to establish robust similarity (e.g., in the presence of multi-modal latency on dynamically routed interconnects or size-dependent performance variance).

4 FLOATING-POINT PRECISION STANDARD THAT ISN'T

In the time between the two updates of the IEEE 754 standard for floating-point, 2008 and 2019, the industry, and the Deep Learning (DL) hardware in particular, was fully engaged in a war of floating-point representations for ML workloads. The main two contenders are shown in Fig. 1: the half precision fp16 and bf16. Espousing the merits or criticising the flaws of either of them is beyond the scope here as the year 2020 brought a new one in the form of TF32 shown in Table 1 in context of all the other prevailing floating-point format types. Choosing, adapting, and implementing software for any of these formats becomes the new normal for library development as only some of these formats are portable, some – productive for the applications, and few perform at the peak of the platform.

It is hardly a coincidence that FP16, BF16, and TF32 form a triad that exchanges exponent and mantissa bits to achieve a different goal: either be range-limited or accuracy-limited, or sacrifice performance. Neither of them can eschew all three limitations at the same time. Some of these considerations are highlighted in Table 1 but listing these three formats in adjacent rows at the bottom for easy comparison.

5 STANDARDS

Choosing an ISO/ANSI, de-facto, or an industry standard is now an issue that touches all three components of the productivity, portability, and performance triad. The humble beginnings were marked by MPI-X paradigm and are now subsumed by more of X+Y+Z zoo of quickly maturing and scalable solutions. To name a few, OpenSHMEM and nvSHMEM are global shared-memory interfaces supported by community and a vendor, respectively. When driven primarily by performance they might be able to be a solution of choice in contexts. And with node performance slowly entering hundred Tera-FLOP range, many existing application may be able to successfully use some of these alternatives.

What remains now platform stack support (compilers vary widely in their language standard compatibility) and exposure to versioning changes. The emergence of C++ as the replacement language for many scientific libraries has now exposed them to the fast-paced development and deprecation cycle and the need for supporting a number of vastly different vendor compilers. The universal adoption of the LLVM tool chain by the vendors vastly improved the situation whereby the applications do not have to require exclusive use of GCC as many have done in the past. Our own efforts in standardizing BLAS in the C++23 library will make this case even stronger.

The minimalist nature of OpenMP still remains an attractive alternative to low-level threading. Even though, in some contexts, it might have lost some of its past appeal after the addition of the accelerator off-load portion of the standard. However, useful functionality is still available with a small subset of the standard. And the available implementations that perform well and integrate seamlessly with the other members of the scientific software stack. Therefore, OpenMP remains the solution of choice across our library offerings.

6 CONCLUSIONS

Our libraries use a mix of techniques in order to deliver the promise of reconciling three the productivity, portability, and performance triad. Delegation, standardization, and autotuning give us opportunities to tame the complexity of contemporary hardware trends and be able hide it effectively and efficiently behind useful abstractions for the scientific applications and correctly answer the question raised in the title and productively take advantage of the available performance in a portable manner.