Reproducibility and Performance: Why Choose?

Ludovic Courtès

Reproducibility and Performance: Why Choose?

Ludovic Courtès
Inria

***Abstract*—Research processes often rely on high-performance computing (HPC), but HPC is often seen as antithetical to "reproducibility": one would have to choose between software that achieves high performance, and software that can be deployed in a reproducible fashion. However, by giving up on reproducibility we would give up on verifiability, a foundation of the scientific process. How can we conciliate performance and reproducibility? This article looks at two performance-critical aspects in HPC: message passing (MPI) and CPU micro-architecture tuning. Engineering work that has gone into performance portability has already proved fruitful, but some areas remain unaddressed when it comes to CPU tuning. We propose package multi-versioning, a technique developed for GNU Guix, a tool for reproducible software deployment, and show that it allows us to implement CPU tuning without compromising on reproducibility and provenance tracking.**

INTRODUCTION. It should come as no surprise that the execution speed of programs is a primary concern in high-performance computing (HPC). Many HPC practitioners would tell you that, among their top concerns, is the performance of high-speed networks used by the Message Passing Interface (MPI) and use of the latest vectorization extensions of modern CPUs.

This article focuses on the latter: tuning code for specific CPU micro-architectures, to reap the benefits of modern CPUs. This question is particularly acute in the context of GNU Guix, a software deployment tool with strong support for *reproducible deployment*. We like to present Guix as a key element of the reproducible research toolbox: as more research output is produced by software, the ability to *verify and validate* research results depends on the ability to *re-deploy and re-run* the software. We present a recently-introduced CPU-tuning option for Guix, the design choices we made, and how this affects reproducibility.

But let us first consider this central question in the HPC and scientific community: can "reproducibility" be achieved *without* sacrificing performance? Our answer is a resounding "yes", but that deserves clarifications.

Reproducibility & High Performance

The author remembers advice heard at the beginning of their career in HPC—advice still given today—: that to get optimal MPI performance, you would have to use the vendor-provided MPI library; that to get your code to perform well on this new cluster, you would have to recompile the complete software stack locally; that using generic, pre-built binaries from a GNU/Linux distribution will not give you good performance.

From a software engineering viewpoint, this looks like a sad situation and an inefficient approach, dismissing the benefits of automated software deployment as pioneered by Debian, Red Hat, and others in the 90's or, more recently, as popularized with container images. It also means doing away with reproducibility, where "reproducibility" is to be understood in two dif-

Published by the IEEE Computer Society

ferent ways: first as the ability to re-deploy the same software stack on another machine or at a different point in time, and second as the ability to *verify* that binaries being run match the source code—the latter is what reproducible builds are concerned with [10].

But does it really have to be this way? Engineering efforts to support *performance portability* suggest otherwise. A mature MPI implementation like Open MPI, today, does achieve performance portability: it takes advantage of high-speed networking hardware by determining, at run-time, which drivers to use to obtain optimal performance for the network at hand—no recompilation is needed [4].

Likewise, generic, pre-built binaries can and indeed often do take advantage of modern CPUs by selecting at run-time the most efficient implementation of performance-sensitive routines for the host CPU [3]. There are cases, though, where this is *not* the case; these are those we will focus on in the remainder of this article.

The Jungle of SIMD Extensions

While major CPU architectures such as x86_64, AArch64, and POWER9 were defined years ago, CPU vendors regularly extend them. Extensions that matter most in HPC are vector extensions: single instruction/multiple data (SIMD) instructions and registers. In this area, a *lot* has happened on x86_64 CPUs since the baseline instruction set architecture (ISA) was defined. As shown in Figure 1, Intel and AMD have been tacking ever more powerful SIMD extensions to their CPUs over the years, from SSE3 to AVX-512, leading to a wealth of CPU "micro-architectures". This gives a high-level view, but just looking at generations of Intel processors by their code name—from "Nehalem" to "Skylake" *via* "Ivybridge"—shows an already more complicated story.

Linear algebra routines that scientific software relies on greatly benefit from SIMD extensions. For example, on a modest Intel CORE i7 processor (of the Skylake generation), the AVX2-optimized version of the dense matrix multiplication routines of Eigen (*https://eigen.tuxfamily.org*), built with GCC 10.3, peaks at about 40 Gflops/s, compared to 11 Gflops/s for its baseline x86_64 version—four

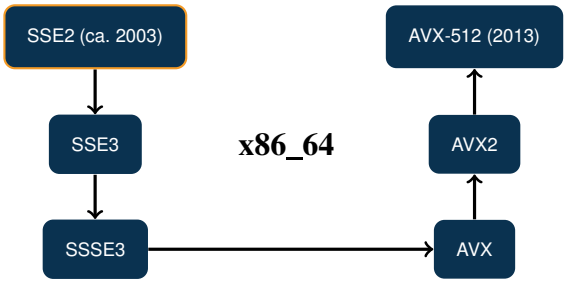

Figure 1. Timeline of x86_64 SIMD extensions

times faster!

Portable Performance Through Function Multi-Versioning

How to create binaries that are portable, yet are able to get the most out of the CPU on which they are executed? This has been an important question for distributors of binaries. Distributions such as Debian and CentOS provide the convenience of fast automated deployment, thanks to pre-built binaries; asking users to either recompile part of their software stack or give up on performance is not a reasonable alternative.

To address this and achieve performance portability, developers have largely adopted *function multi-versioning* (FMV): the implementation provides multiple versions of "hot" routines, one for each relevant CPU micro-architecture, and picks the best one for the host CPU at run time. Many pieces of performance-critical software already use this technique: the C standard library (libc) contains multiple versions of its string handling and math routines, the GMP library for multi-precision arithmetic uses FMV, and so do software packages ranging from cryptography libraries (Libgcrypt, Nettle) to linear algebra (OpenBLAS, FFTW).

To make it easier for developers to adopt FMV, the GNU compilation tool chain (GCC, the Binary Utilities, and the C Library), which is widely used in HPC, provides helpers at different levels. Developers can annotate relevant functions with the `target_clone` attribute to instruct the compiler to generate optimized versions of the function for each selected architecture. GCC not only generates these versions, but also generates code to choose the right function version for the host CPU at load time, with support from the

dynamic linker, `ld.so`. That relieves developers from the need to implement their own ad-hoc machinery. From that perspective, it would seem that performance portability, *via* FMV, is a solved problem.

There is at least one common pattern though where FMV is not applicable, or at least is not applied: C++ header-only libraries. These are libraries that provide generic template code in header files; that code is specialized *at build time* in software that uses them. There is no shortage of C++ header-only math libraries providing efficient, optimized SIMD versions of their routines: Eigen, MIPP, xsimd and xtensor, SIMD Everywhere (SIMDe), Highway, and many more. All these, except Highway, have in common that they do *not* support FMV. Since they "just" provide headers, it is up to *each* package using them to figure out what to do in terms of performance portability.

In practice though, software using these C++ header-only libraries rarely makes provisions for performance portability. Thus, when compiling those packages for the baseline ISA, one misses out on all the vectorized implementations that libraries like Eigen provide. This is a known issue in search of a solution—see *https://gitlab.com/libeigen/eigen/-/issues/2344*. It can have a very concrete impact on performance since many scientific packages—the ARPACK-NG library for solving eigenvalue problems, the Ceres solver for optimization problems, the FEniCSx platform for solving differential equations, to name a few—depend on Eigen.

Reproducible Deployment

Distributions such as Debian and Fedora that provide pre-built binaries miss out on SIMD optimizations of C++ header-only libraries like Eigen because they provide binaries targeting the baseline CPU architecture so that those binaries run on any CPU. The Spack [7] and EasyBuild [8] package managers address that by *rebuilding* software on the target computer, which allows them to instruct the compiler to optimize for the host CPU.

Unfortunately, EasyBuild and Spack both have limited support for reproducible deployment—they do not, in general, guarantee that you can redeploy the same software environment on different machines, or at different points in time. This is because they build upon software provided by the host system—the compiler tool chain, "system" libraries, etc.—and that foundation differs from one system to another—e.g., CentOS might provide some version of GCC, and Ubuntu might provide another.

To avoid that, Guix builds software in *isolated environments*, as pioneered by Nix [1, 6], and its package collection is *self-contained*—it does not rely on external software packages. This is what makes Guix builds reproducible bit-for-bit—or in other words, *verifiable* [10]. Given binaries and provenance data, anyone can independently verify the binary/source-code correspondence.

Guix provides a command-line interface similar to that of other package managers: `guix install python`, for instance, installs the Python interpreter. Package management is per-user rather than system-wide and does not require system administrator privileges, which makes it suitable for multi-user HPC clusters [2]. To offer the level of flexibility that HPC users expect, Guix lets users customize packages *via package transformation options* on the command line—for instance to swap two packages in the dependency graph—or through programming interfaces [2].

Quite uniquely, Guix supports *"time traveling"*: with `guix time-machine`, users can run a specific revision of Guix and use it to deploy packages as they were defined in that revision. The typical use case is redeploying software that was used to produce computational results for a scientific publication [5, 9, 11]. The command below deploys Python, NumPy, and their dependencies as they were defined in a Guix revision from October 2021:

```
guix time-machine --commit=b0735c79b0d1d341 -- \
  shell python python-numpy
```

Whether you run it today or two years from now, it will deploy the *exact same binaries*, bit-for-bit, down to the C library.

Package Multi-Versioning

With our packaging hammer, one could envision a solution to these CPU tuning problems: if we cannot do function multi-versioning, what about implementing *package* multi-versioning? Guix makes it easy to define package variants, so we can define package variants optimized for a

specific CPU—compiled with `-march=skylake`, for instance. What we need is to define those variants "on the fly".

The recently-introduced `--tune` package transformation option works along those lines. Users can pass `--tune` to any of the command-line tools (`guix install`, `guix shell`, etc.) and that causes "tunable" packages to be optimized for the host CPU. For example, here is how you would run Eigen's matrix multiplication benchmark from the `eigen-benchmarks` package with micro-architecture tuning:

```
$ guix shell --tune eigen-benchmarks -- \
    benchBlasGemm 240 240 240
guix shell: tuning for CPU skylake
240 x 240 x 240
cblas: 0.208547 (15.908 GFlops/s)
eigen : 0.0720303 (46.06 GFlops/s)
l1: 32768
l2: 262144
```

`--tune` determines the name of the host CPU as recognized by GCC's (and Clang's) `-march` option. Users can override auto-detection by passing a CPU name—e.g., `--tune=skylake--avx512`. As mentioned earlier, we made the conscious choice of letting `--tune` affect solely software that packagers explicitly marked as "tunable". This ensures Guix does not end up rebuilding packages that could not possibly benefit from micro-architecture-specific optimizations, which would be a waste of resources.

This implementation of package multi-versioning does not sacrifice reproducibility. When `--tune` is used, from Guix's viewpoint, it is just an alternate, but well-defined dependency graph that gets built. Guix records package transformation options that were used so it can "replay" them. For example, one can export a *manifest* representing packages that have been deployed:

```
$ guix shell eigen-benchmarks --tune
guix shell: tuning for CPU skylake
[env]$ guix package --export-manifest \
             -p $GUIX_ENVIRONMENT
(use-modules (guix transformations))

(define transform1
  (options->transformation
    '((tune . "skylake"))))

(packages->manifest
  (list (transform1
          (specification->package
            "eigen-benchmarks"))))
```

The manifest above is a code snippet that can be passed to `guix shell` or `guix package` to redeploy the package with the same tuning parameters. Like other transformation options, `--tune` is accepted by all the commands; for example, here is how you would build a Docker image tuned for a particular CPU:

```
guix pack -f docker -S /bin=bin
  eigen-benchmarks --tune=skylake
```

Conclusion and Outlook

We implemented what we call "package multi-versioning" for C/C++ software that lacks function multi-versioning and run-time dispatch, a notable example of which is optimized C++ header-only libraries. It is another way to ensure that users do not have to trade reproducibility for performance.

The scientific programming landscape has been evolving over the last few years. It is encouraging to see that Julia offers function multi-versioning for its "system image", and that, similarly, Rust supports it with annotations similar to GCC's `target_clones`. Hopefully these new development environments will support performance portability well enough that users and packagers will not need to worry about it.

But first and foremost, it is up to us, research software engineers and scientists, to dispel the myth that performance is a valid excuse for non-reproducible computational workflows.

References

[1] L. Courtès. Functional Package Management with Guix. In *European Lisp Symposium*, June 2013.

[2] L. Courtès, R. Wurmus. Reproducible and User-Controlled Software Environments in HPC with Guix. In *Euro-Par 2015: Parallel Processing Workshops*, Lecture Notes in Computer Science, pp. 579–591, August 2015.

[3] L. Courtès. Pre-Built Binaries vs. Performance. January 2018. *https://hpc.guix.info/blog/2018/01/pre-built-binaries-vs-performance/*.

[4] L. Courtès. Optimized and Portable Open MPI Packaging. December 2019. *https://hpc.guix.info/blog/2019/12/optimized-*

and-portable-open-mpi-packaging/.

[5] L. Courtès. [Re] Storage Tradeoffs in a Collaborative Backup Service for Mobile Devices. In *ReScience C*, 6(1) , June 2020, .

[6] E. Dolstra, M. d. Jonge, E. Visser. Nix: A Safe and Policy-Free System for Software Deployment. In *Proceedings of the 18th Large Installation System Administration Conference (LISA '04)*, pp. 79–92, USENIX, November 2004.

[7] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. d. Supinski, S. Futral. The Spack Package Manager: Bringing Order to HPC Software Chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, Association for Computing Machinery, 2015.

[8] M. Geimer, K. Hoste, R. McLay. Modern Scientific Software Management Using EasyBuild and Lmod. In *Proceedings of the First Workshop on HPC User Support Tools (HUST'14)*, pp. 41–51, IEEE Press, 2014.

[9] K. Hinsen. Staged Computation: The Technique You Did Not Know You Were Using. In *Computing in Science Engineering*, 22(4) , 2020, pp. 99–103.

[10] C. Lamb, S. Zacchiroli. Reproducible Builds: Increasing the Integrity of Software Supply Chains. In *IEEE Software*, 39(2) , March 2022, pp. 62–70.

[11] J. M. Perkel. Challenge to Scientists: Does Your Ten-Year-Old Code Still Run?. In *Nature*, 584, August 2020, pp. 656–658.

Ludovic Courtès is a research software engineer at Inria, France. He has been contributing to the development of GNU Guix since its inception in 2012 and works on its use in support of reproducible research workflows. He holds a PhD in computer science from LAAS-CNRS. You can reach him at *ludovic.courtes@inria.fr*.