Experimenting with PyTorch on RISC-V

Iacopo Colonnelli*, Robert Birke and Marco Aldinucci

University of Torino, Computer Science Dept., Corso Svizzera 185, 10149, Torino, Italy

Abstract

RISC-V is an emerging instruction set architecture. Its modular and extensible open-source royalty-free design is increasingly attracting interest from both research and industry. Nowadays, different RISC-V-based boards can be bought off the shelf. However, software availability is equivalently vital in guaranteeing the RISC-V ecosystem's success. Here we contribute with the first publicly available port of PyTorch. PyTorch is one of the most popular Deep Learning libraries available today. As such, it is a crucial enabler in running state-of-the-art AI applications on RISC-V-based systems and a first step towards a fully democratic end-to-end codesign process.

Introduction

Open ISA is a crucial ingredient of a fully democratic end-to-end codesign process, moving vendors' market competition from compatibility to performance. However, software availability is equally vital for the success of the RISC-V ecosystem, as it ensures a broad and long-lasting adoption from the community. In this work, we describe how we ported the PyTorch Deep Learning [1] framework to RISC-V, hoping it will serve as a practical guide for other similar initiatives in the future.

Methodologies

Porting PyTorch to RISC-V

We started the porting effort back in early 2022, using PyTorch v1.11. At that time, at least three internal dependencies were incompatible with the RISC-V ISA: the Chromium Breakpad library[1], the SLEEF Vectorized Math Library[2], and the PyTorch CPU INFOrmation library (cpuinfo)[3]. Since then, we have compiled several versions of PyTorch up to the latest v2.0 version without further modifying the codebase. The list of pull requests we opened during the porting and all the wheel binaries targeting RISC-V architecture are available in our repository[4].

Porting Breakpad The Chromium Breakpad library implements a client-server approach to distributed crash reporting. A *client* library is included in the user application. When it crashes, the client writes a minidump file capturing system information like threads' state (processor registries and stack memory) or loaded executables and libraries. The minidump is then read by the Breakpad *processor*, which produces a human-readable C/C++ stack trace. The main effort for porting Breakpad to RISC-V was devoted to properly treating the state of the processor registries. Internally, Breakpad reimplements the POSIX `getcontext()` function in assembly to retrieve a snapshot of the processor register set. Then, architecture-specific data structures load, store, print, and manipulate the processor context across the various Breakpad modules. All these components have been extended to handle `riscv` and `riscv64` architectures.

Porting SLEEF The SIMD Library for Evaluating Elementary Functions (SLEEF) [2] implements a vectorised version of all C99 real floating-point math functions with two different levels of accuracy (1 ULP and 3.5 ULP). Plus, a third implementation guarantees bit-wise consistent results across all hardware architectures. The RISC-V V (RVV) extension augments the base RISC-V architecture with 32 vector registers and a set of vector instructions. However, the hardware platform we used to develop and compile did not implement this extension. Therefore, we just ported the scalar implementation of math functions based on the Fused Multiply-Add (FMA) optimised instruction. More recently, SiFive® forked the SLEEF library to introduce support for RVV intrinsics[5].

Porting cpuinfo The PyTorch CPU INFOrmation library (cpuinfo) provides a uniform cross-platform layer to access information about the host CPU. PyTorch relies on cpuinfo for performance optimisation when running on a CPU, e.g., to detect support for SIMD instructions or to pin threads to cores in NUMA architectures. Porting cpuinfo to RISC-V on a Linux OS means extracting information on the processor and the cache hierarchy from a set of sources (e.g., `/proc/cpuinfo`, `hwcap`, and the manufacturer manual) and exposing them in a coherent way to the user. The intrinsically dynamic nature of the RISC-V ISA and the somehow loose coupling between the ISA itself and the available hardware make it quite complex to

*Corresponding author: `iacopo.colonnelli@unito.it`

[1] `https://chromium.googlesource.com/breakpad`

[2] `https://sleef.org`

[3] `https://github.com/pytorch/cpuinfo`

[4] `https://gitlab.di.unito.it/alpha/riscv/torch`

[5] `https://github.com/sifive/sifive-sleef`

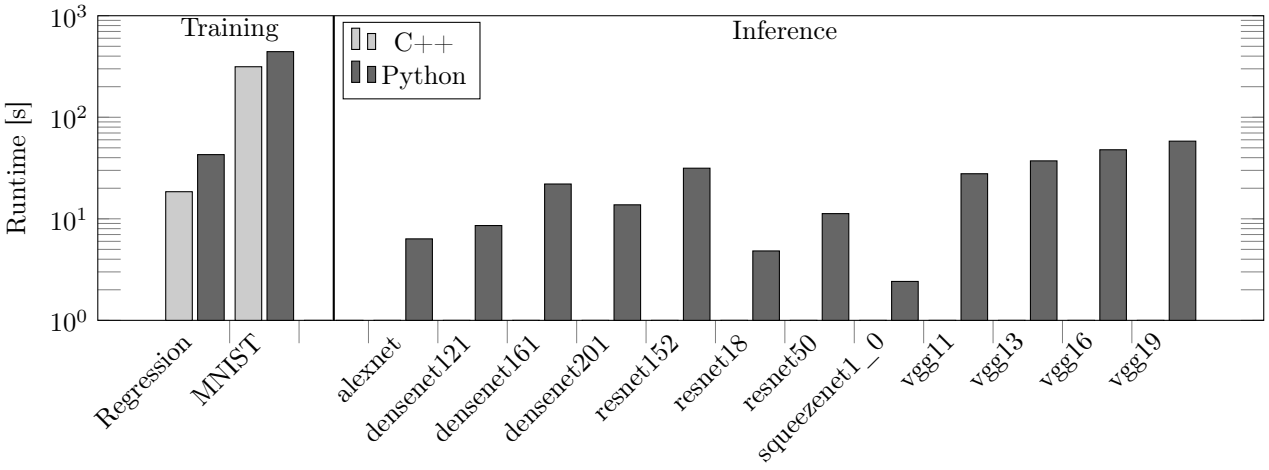

Figure 1: *Timing results of selected* `pytorch/examples`[4] *benchmarks: mean across 5 runs/images.* `regression` *and* `mnist`*: training one epoch using C++/Python API.* `imagenet`*: single image inference for different popular DNN architectures.*

fully port this library. Therefore, we started porting a minimum set of features to support PyTorch multi-threading functions, leaving full support for extensions as future work.

Evaluation

Testbed We compile and evaluate our RISC-V port on the SiFive® Freedom U740 SoC (4 U74 RV64GCB 1.2 GHz cores, 16GB DDR4 RAM, 1 TB node-local NVMe storage). The node runs Linux Ubuntu 20.04 with kernel v5.15 for `riscv64`. As a build environment, we use Python v3.9, GNU C++ compiler v10.3.0, and cmake v3.26.0 to compile PyTorch v2.0.

Benchmarks The PyTorch project provides a repository of curated, short, and high-quality examples[4] that we use to test and benchmark our RISC-V port. We run various programs covering training and inference tasks using Python and C++ PyTorch APIs. For training, we consider `regression` and `mnist`. The `regression` code fits a fourth-degree polynomial using a single fully-connected layer. We modified the code to train over a fixed number of 32K samples (instead of using the fitting error as a stopping condition). The `mnist` code trains a simple Convolutional Neural Network (CNN) on the MNIST dataset. The C++ and Python versions differ slightly in the used architecture. We uniform both to the C++ version, i.e. a CNN of two convolutional layers followed by one dropout layer and two fully connected layers. Here we evaluate the runtime to train one epoch comprising 60K images. For inference, we use `imagenet` using different image classification models provided by the PyTorch Vision module v0.2. Here we evaluate the inference time for a single image. Fig. 1 reports the results. The C++ API runs faster, while the Python API offers the flexibility to modify the DNN architecture without recompiling. All tests ran without any issues.

Conclusion

The current PyTorch porting to RISC-V is mature enough for research and development, and it has already been used to assess RISC-V hardware readiness for decentralised Machine Learning [3]. Plus, efforts to provide a fully-optimised RISC-V implementation are ongoing in the context of the EUPilot project[5].

Acknowledgements

This work has been partially supported by the Spoke "FutureHPC & BigData" of the ICSC – Centro Nazionale di Ricerca in "High Performance Computing, Big Data and Quantum Computing" (NextGenerationEU), and the European PILOT project (EuroHPC-JU, G.A. n. 101034126).

References

[1] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. 2019, pp. 8024–8035.

[2] Naoki Shibata and Francesco Petrogalli. "SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions". In: *IEEE Trans. Parallel Distributed Syst.* 31.6 (2020), pp. 1316–1327. DOI: 10.1109/TPDS.2019.2960333.

[3] Gianluca Mittone et al. "Experimenting with Emerging ARM and RISC-V Systems for Decentralised Machine Learning". In: *CoRR* abs/2302.07946 (2023). DOI: 10.48550/arXiv.2302.07946. arXiv: 2302.07946.

[4] `https://github.com/pytorch/examples`
[5] `https://eupilot.eu`