Technologies of Computing Systems

LAB Demo - Pytorch emulation of MX-compatible formats

Authors: Ricardo Nobre, Leonel Sousa

1. Introduction to use of low-precision computations and MX-formats

Deep Neural Networks (DNNs) are nowadays widely used in the context of a number of domains (e.g. self-driving cars, chat bots). The training of those networks and/or their use in production can often rely on lower precision formats in order to achieve the required performance at a given cost or power level, and/or to reduce energy-consumption. Furthermore, using lower precision can also be used as a means to comply with stringent memory requirements or to tackle extremely large DNNs with billions or even trillions of parameters (e.g. GPT-4 has ~1.7 trillion parameters).

Half precision (i.e. 16-bit) floating point operations using the IEEE 754 16-bit format or the later introduced “brain” float format (bfloat16) have been used for some years as a means to accelerate DNN processing. Narrow integer formats (e.g. INT4 / INT8) have also been proposed, but those are typically only used in the trained models after quantization. More recently, the Microscaling (MX) specification was developed by an industry consortium (AMD, Arm, Intel, Meta, Microsoft, NVIDIA, and Qualcomm) [1] as an effort to standardize a set of interoperable data formats that combine a per-block scaling factor with floating-point or integer types for private individual elements [2]. MX-compatible formats enable unprecedented levels of specialization for the particular codes under study in regard to numerical precision, with support for types employing widely different amounts of bits, as well as support for different configurations in what concerns the number of bits used for the mantissa and the exponent parts when relying on floating-point numbers for the individual elements.

This lab is focused on evaluating the impact of the use of MX-compatible floating-point formats within the Pytorch machine learning framework. For that purpose we rely on the open-source `microxcaling` library. In addition to enabling the exploration of the use of MX-compatible formats in the context of matrix multiplication operations (`torch.matmul`, `torch.linear`, `torch.bmm`), it also supports narrow precision floating-point quantization (e.g. bfloat16, IEEE 754 16-bit) when performing elementwise operations (`GELU`, `softmax`, `layernorm`). Notice that this library relies on the IEEE 32-bit floating point format to emulate the narrower numerical formats, being values restricted to the representable ranges.

2. Installing and using the library

To install the library, the first step is to clone it from its GitHub code repository (or click "download ZIP"):
`$ git clone https://github.com/microsoft/microxcaling (for Windows` "download ZIP" from git).

Then it is needed to enable access to the library from within Python. This can be achieved by adding to the `PYTHONPATH` environment variable the directory (`PATH_TO_LIBRARY`) to where it has been installed.
`$ export PYTHONPATH=$PYTHONPATH:PATH_TO_LIBRARY`
`(for Windows` set PYTHONPATH=%PYTHONPATH%;C:\My_python_lib).

The formats one wants to use are passed as input to the operators, functions, and layers provided by this library (all take `mx_specs` dictionary as an argument). The process of modifying a given Pytorch DNN model construction source code is explained in the "MX_Integration_Guide.pdf" document that is on the GitHub repository of the library. The provided example explains the steps for the creation of the `ffn_mx_manual.py` code (code that uses the library) from the `fnn.py` original code implementing a Transformer MLP. Both the original and manually modified codes are in the GitHub repository of the library, as well as a version that uses uses a recent feature of the library that enables automatic injection of MX modules and functions (`ffn_mx_auto.py`).

One can create the mx_specs dict in one of the two following ways. It can be constructed by calling `mx.MxSpecs()`, and then setting the dict entries in the Python code (e.g. `mx_specs['w_elem_format'] = 'fp6_e3m2'`). Or from input to the command line calling the Python script. To achieve that call `add_mx_args(parser)`, where parser is the return of `argparse.ArgumentParser()`. Then, construct `mx_specs` dict by calling `get_mx_specs(args)`, where `args` is the return of `parser.parse_args()`.

3. Evaluating MX-compatible formats

For evaluating the use of MX-compatible formats we will use the Fashion-MNIST dataset. It is a more challenging drop-in replacement for the MNIST dataset with the same amount of training samples (60,000) and test samples (10,000). However, instead of representing 10 handwritten digits (0-9) it represents 10 classes of articles in the same 28x28 grayscale image format used in the MNIST dataset.

1. Download scripts provided on the Webpage of the course, which were prepared to train a DNN model with the Fashion-MNIST dataset. One of the scripts does not use the `microxcaling` library (`pytorch_code.py`). Using the library (`pytorch_code_mx.py`) is achieved passing `mx_specs` as an argument to the DNN model, and replacing the Pytorch functions by the equivalent ones from `microxcaling`. For example, `torch.nn.Conv2d` is replaced by `mx.Conv2d`, `nn.Linear` by `mx.Linear` and `torch.nn.functional.relu` by `mx.relu`. Library functions receive `mx_specs` as an argument.

2. Assess the impact of using some of the MX formats (e.g. `int8`, `int2`, `fp6_e2m3`, `fp4_e2m1`) for matrix multiplication operations, as well as `bfloat` and `fp16` for elementwise operations. Produce a chart displaying the accuracy after at least 10 epochs for the settings evaluated.

3. **(Optional 1)** Adapt another DNN model (an existing one or created by you) to use the library and evaluate the impact of precision tuning. Special points if impact on accuracy is higher than for the code provided. Please submit both the original and adapted codes as part of the lab resolution.

4. **(Optional 2)** Explore other features of the `microxcaling` library, such as the capability of performing elementwise operations with data types smaller than 16 bits with the `fp` option. Note that, as explained in the documentation, 5 bits is always used for the exponent. As a result, setting the `fp` entry of `mx_specs` to 7 (minimum value supported), results in a mantissa of 2 bits.

IMPORTANT: The Python script given as part of this lab to process Fashion-MNIST has been set to use the CPU by default. However, if you have a compatible GPU, you can reduce your processing time by passing `--use-cuda` in the command executing the Python script. Notice however, that if you use CUDA acceleration, you should also set `mx_specs['custom_cuda'] = True`. This reportedly makes the executions more numerically accurate than if using CUDA acceleration without the custom CUDA code. You should also be able to leverage accelerated training on Mac through Pytorch's Metal Performance Shaders (MPS) backend (`--use-mps`), but that has not been tested.

[1] OCP Microscaling Formats (MX) Specification Version 1.0, Open Compute Project, 2023.
[2] Rouhani et al., Microscaling Data Formats for Deep Learning, ArXiv, 2023.