Technologies of Computing Systems

Pytorch emulation of MX-compatible formats, with the purpose of designing AI accelerators

Authors: Ricardo Nobre, Martim Bento, Leonel Sousa

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

Deep Neural Networks (DNNs) are nowadays architectures for Artificial Intelligence (AI) that are 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 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 designing AI accelerators 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, and is also available on the course webpage. The provided example explains the steps for the creation of the `ffn_mx.py` code (code that uses the library) from the `fnn.py` original code implementing a Transformer MLP. Both the original and modified codes are in the GitHub repository of the library.

The mx_specs dict can be created 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()`.

IMPORTANT:

Performing the installation locally is highly recommended, for pedagogical reasons. However, students unable to do so may instead use the SCDEEC machines, by connecting via **ssh:**
ssh <USER>@<CUDA NAME>.scdeec.tecnico.ulisboa.pt
Where <CUDA NAME> is either ‘cuda1’, ‘cuda2’ or ‘cuda3’, and the <USER> is specific to each group (and will be made available later).

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.

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.

1. Inspect the scripts provided in the course page. Identify the layers of the neural network, as well as the relevant instructions for evaluating the MX-compatible formats. It is also recommended to read the specifications regarding the MX formats and data types, presented in this document.

2. Assess the impact of using some of the MX formats for matrix multiplication operations, considering different elementwise operations (`bfloat` and `fpX`). Note that, when performing elementwise operations with the `fp` option, the exponent always features a width of 5 bits, as explained in the documentation. As a result, setting the `fp` entry of `mx_specs` to 7 (i.e. the minimum supported value), results in a mantissa of 1 bit. Acquire the accuracy after 5 and 10 epochs for the evaluated settings. These results should be displayed in the tables presented in the last page of this assignment. Comment on your findings.

3. Consider the possibility of implementing a hardware accelerator for the presented neural network. Assuming that it should feature an accuracy of **90%**, select the data formats that allow you to minimize the size of the accelerator. What accuracy did you obtain? Note: It might be useful to check which data formats are supported by the `microxcaling` library.

4. **(Optional)** Select the data formats in order to obtain the worst possible accuracy for the considered network. How much did you obtain? Explain your methodology and comment on any notable behaviour that you find during training.

5. **(Optional)** Explore other features of the `microxcaling` library (e.g. backward pass quantization formats). Comment on your findings, and present the assessments in the corresponding table.

6. **(Optional)** 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.

IMPORTANT:

1. **Any** changes made to the code should be submitted, regardless of applying to any optional assignment or only to the main tasks.

2. 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.

1. Assert the impact of different data formats for matrix multiplication operations on accuracy and convergence, considering `bfloat` for elementwise operations.

Format	`int2`	`int8`	`fp8_e4m3`	`fp8_e5m2`	`bfloat16`
Accuracy after 5 epochs (%)					
Accuracy after 10 epochs (%)					

2. Assert the impact of different `fpX` formats for elementwise operations on accuracy and convergence, considering `fp8_e4m3` for matrix multiplications.

Format	`fp16`	`fp12`	`fp10`	`fp8`
Accuracy after 5 epochs (%)				
Accuracy after 10 epochs (%)				

3. Considering that the presented classifier network should have an accuracy of at least 90%, which data formats would you select? What accuracy did you obtain?

4. Additional Assessments (for the optional assignments).

Comments on the obtained results (including optional assignments).
