1-23 personal research

author: Sylvex Hung/洪祐鈞

Blue color text contains link.

What's the point of posits adoptions?

What is posit?

- Wikipedia Unum Shows where the posit come from.
- Posit official
    - projects
        - There are some project related to deep learning, however most of them are archived and may not be maintained.
- Posits, a New Kind of Number, Improves the Math of AI
    - Good and short article that mentioned how posit works, the potential adoption in machine learning, and hardware aspect.
- Posits: the good, the bad and the ugly
    - Very good paper analyzing the pros and cons of posit
    - From the conclusion, posit is good especially when the numerical computation can be controlled in certain range.
    - Hence machine learning is indeed a use case.

Personal comment:

The main question I want to ask is: "Do we really need that much of precision in terms of deep learning?" My intuition of this question is as follows:

- Normally Deep Learning has 2 steps, training and inferencing. The inferencing normally didn't require a lot of numerical precision. The training step require more numerical precision. Simply replace float to same sized posits at inferencing may not help that much. Also we have multiple floating points format to choose from:
    - Training vs Inference – Numerical Precision
        - The main takeaway is that usually training and inference is default fp32. (IEEE single precision format)
        - When deployment, inference doesn't care that much of precision, hence the model would get compress or quantize, and the final format might be int8 or format smaller than fp32.
        - There are other improved format like BF16 and TF32 aims optimized training or inference.
- However, I think the potential of the posit is if we can use less bit to represent normal IEEE floating points. There are few research focus on using 8-bit posit to save the bit used (may use several hacks) to achieve similar result with fp32.
    - Rethinking floating point for deep learning, used 8-bit posit and modified posit to achieve similar result with fp32at inference phase.

    - Evaluations on Deep Neural NetworksTraining Using Posit Number System also use 8-bits deep learning training getting similar result to fp32.
    - Lot of the paper here is related to hardware design so my point might that be valid.

How to integrate posit to existing infrastructure?

I think we can refer to how people integrate various kinds of floating point format to LLVM, MLIR, deep learning frameworks and MLIR ecosystem. The following project or code commits give me few ideas to add posit data type to existing infrastructure.

- llvm-xposit(github project)
    - Support posit arithmetic in llvm, and can be integrate to RISC-V.
- Deep PeNSieveSupport posit format in deep learning in tensorflow.
- Add APFloat and MLIR type support for fp8 (e5m2).
    - Nvidia, ARM, Intel proposed fp8 formatfor machine learning. Mainly speed up training and inferencing.
    - It feel like adding a first class support to llvm?
    - The RFC of implementation of fp8
        - There's also interesting discussion of various of floating point format developed by many big tech company.
- Adding fp16 support to torch-mlir
    - Follow the commit may help how to add floating point type to torch-mlir.
- TPU MLIR lowering
    - For the conversion between types, I guess this would help?
    - (I review the code still not able to figure out currently.)

However, I don't quite understand how does they work currently by simply review the code

- Even viewing the git diff for implementing fp8, I know it is changed in llvm's APFloat type and mlir type system, but don't know what exactly happening under the hood.
- I should implement small prototype and have more knowledge in the ecosystem in order to actually get stuff working now.

What I have currently done?

- How to build pytorch-mlir correctly
    - reference link
    - Before build, you need to setup python environment.
        - You need to deactivate your python environment manager e.g. conda, before you create python env.
            - I have trolled myself few time because of it.

```
python -m venv mlir_venv # Create environment
source mlir_venv/bin/activate
```

    - The successful cmake config

```
cmake -GNinja -Bbuild \
  -DCMAKE_C_COMPILER=clang \
  -DCMAKE_CXX_COMPILER=clang++ \
  -DCMAKE_BUILD_TYPE=Release \
  -DPython3_FIND_VIRTUALENV=ONLY \
  -DLLVM_ENABLE_PROJECTS=mlir \
  -DLLVM_EXTERNAL_PROJECTS="torch-mlir" \
  -DLLVM_EXTERNAL_TORCH_MLIR_SOURCE_DIR="$PWD" \
  -DMLIR_ENABLE_BINDINGS_PYTHON=ON \
  -DLLVM_TARGETS_TO_BUILD=host \
  -DCMAKE_C_COMPILER_LAUNCHER=ccache \
  -DCMAKE_CXX_COMPILER_LAUNCHER=ccache \
  -DCMAKE_EXE_LINKER_FLAGS_INIT="--ld-path=ld.lld" \
  -DCMAKE_MODULE_LINKER_FLAGS_INIT="--ld-path=ld.lld" \
  -DCMAKE_SHARED_LINKER_FLAGS_INIT="--ld-path=ld.lld" \
  -DTORCH_MLIR_ENABLE_LTC=ON \
  externals/llvm-project/llvm
```

    - I have enabled clang compiler, ccache, and linker lld which improve the compile time drastically. And you should install them beforehand.
        - clang is the main compiler for llvm, usually compiles faster.
        - ccache is good for rebuild the project
        - lld is meant to resolve the many minutes link time of ld in to seconds.
        - From the github issue, lazy tensor core (ltc) need to be enable in order to run the ltc
        - I ran into the problem which clang can not compile any program error, Specifically cannot find libstdc++ header. Strangely, install `g++-12` resolve the problem. (ref)
    - Compile: `cmake --build build --target tools/torch-mlir/all`
- Running the torch-mlir example:
    - `torchscript_resnet18_all_output_types.py`
        - Basically torchscript compile a trained model to dialect for inferencing.
        - This would output torch, lingalg, tosa dialect.
        - e.g.

```
TORCH OutputType
 module attributes {torch.debug_module_name = "ResNet"} {
  func.func @forward(%arg0: !torch.vtensor<[1,3,224,224],f32>) ->
!torch.vtensor<[1,1000],f32> {
    %float1.000000e00 = torch.constant.float 1.000000e+00
    %int-1 = torch.constant.int -1
    %int7 = torch.constant.int 7
...

LINALG_ON_TENSORS OutputType
 #map = affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>
#map1 = affine_map<(d0, d1, d2, d3) -> (d1)>
#map2 = affine_map<(d0, d1, d2, d3) -> (0, d1, d2, d3)>
#map3 = affine_map<(d0, d1) -> (d0, d1)>
#map4 = affine_map<(d0, d1) -> (d1, d0)>
#map5 = affine_map<(d0, d1) -> (0, d1)>
```

```
#map6 = affine_map<(d0, d1) -> (d1)>
module attributes {torch.debug_module_name = "ResNet"} {
  ml_program.global private mutable @global_seed(dense<0> :
tensor<i64>) : tensor<i64>
  func.func @forward(%arg0: tensor<1x3x224x224xf32>) ->
tensor<1x1000xf32>
...

TOSA OutputType
 module attributes {torch.debug_module_name = "ResNet"} {
  func.func @forward(%arg0: tensor<1x3x224x224xf32>) ->
tensor<1x1000xf32> {
    %0 = "tosa.const"() <{value = dense_resource<__elided__> :
tensor<1x512x1x1xf32>}> : () -> tensor<1x512x1x1xf32>
    %1 = "tosa.const"() <{value = dense_resource<__elided__> :
tensor<1x256x1x1xf32>}> : () -> tensor<1x256x1x1xf32>
    %2 = "tosa.const"() <{value = dense_resource<__elided__> :
tensor<1x128x1x1xf32>}> : () -> tensor<1x128x1x1xf32>
    %3 = "tosa.const"() <{value = dense_resource<__elided__> :
tensor<1000x512xf32>}> : () -> tensor<1000x512xf32>
    %4 = "tosa.const"() <{value = dense_resource<__elided__> :
tensor<512x512x3x3xf32>}> : () -> tensor<512x512x3x3xf32>
```

- ltc_backend_mnist.py
  - Basically lazy tensor core backend would try to capture the training graph, and optimize it.
  - It would output captured graph and it's torch dialect.
  - e.g.

```
JIT Graph:
graph(%p0 : Long(),
      %p1 : Float(10, 5),
      %p2 : Float(1, 5),
      %p3 : Float(10),
      %p4 : Long(1),
      %p5 : Long(),
      %p6 : Double(),
      %p7 : Double(),
      %p8 : Float(10, 5)):
  %2 : Float(10, 5) = aten::detach_copy(%p1)
  %3 : Float(5, 10) = aten::t_copy(%2)
  %59 : int = aten::IntImplicit(%p0)
...

MLIR:
module {
  func.func @graph(%arg0: !torch.vtensor<[],si64>, %arg1:
!torch.vtensor<[10,5],f32>, %arg2: !torch.vtensor<[1,5],f32>,
%arg3: !torch.vtensor<[10],f32>, %arg4: !torch.vtensor<[1],si64>,
%arg5: !torch.vtensor<[],si64>, %arg6: !torch.vtensor<[],f64>,
%arg7: !torch.vtensor<[],f64>, %arg8: !torch.vtensor<[10,5],f32>)
```

```
-> (!torch.vtensor<[1,10],f32>, !torch.vtensor<[],f32>,
!torch.vtensor<[5,10],f32>, !torch.vtensor<[1,10],f32>,
!torch.vtensor<[10],f32>, !torch.vtensor<[10,5],f32>,
!torch.vtensor<[10,5],f32>,
...
```

What to do next?

- Walk through the toy tutorial of MLIR, getting more familiar with LLVM.
  - I might need to pause the research on torch-mlir and focus on how to implement a prototype.
- Knowing how to design a dialect and how they are get optimized(e.g. vectorize)
  - MLIR vector dialect and patterns
  - MLIR Linalg dialect and patterns
  - Linalg anatomy
- Having more knowledge about ecosystem around AI compiler
  - ZOMI, this guy has a lot of video on this topic.