Auto-generating Quantized DNN Kernels using TVM

Author: Nikola Nikolić

1. Get an instance of a ResNet50 model implemented in PyTorch. It’s available in the torchvision package.

In order to get an instance of ResNet50 mode, we can use the code on the right side

```
import torchvision
import torch

# Load a pretrained PyTorch model, model has been trained on ImageNet,
# a large dataset of images used for image classification
model_name = "resnet50"
model = getattr(torchvision.models, model_name)(pretrained=True)
model = model.eval()

# We grab the TorchScripted model via tracing
# 1 - one image at time, 3 - number of channels, in this case RGB, 224 - height, 224 - width
input_shape = [1, 3, 224, 224]
# Creates a tensor of the specified shape with random numbers drawn from a standard normal distribution
input_data = torch.randn(input_shape)
# Converting Pytorch model into a TorchScript model
scripted_model = torch.jit.trace(model, input_data).eval()
```

2. It's a good idea to try TVM on an un-quantized DNN first. Give TVM the network and a sample input to the network, and compile the network into a function object that can be called from Python side to produce DNN outputs.

- Take a look at resnet50-unquantized-model.py script
- Function object that can be called from Python is called “m” and we can find it within execute_graph_on_tvm function

Pay attention to the compilation target: which hardware (CPU? GPU?) the model is being compiled for, and understand how to specify it. Compile for GPU, if you have one, or CPU otherwise.

GPU

```
# set GPU as target device
target = tvm.target.Target("cuda", host="llvm")
dev = tvm.cuda(0)
```

CPU

```
# set CPU as target device
target = tvm.target.Target("llvm", host="llvm")
dev = tvm.cpu(0)
```

- We can choose between CPU or GPU as a target hardware device
- If we want to use GPU as a target hardware device CUDA and NVIDIA graphic card are required

3. Now, quantize the model down to int8 precision. TVM itself has utilities to quantize a DNN before compilation; you can find how-tos in the guides and forum. Again, you should get a function object that can be called from Python side.

- Take a look at resnet50-quantized-int8-model.py script
- Function object that can be called from Python is called “m” and we can find it within execute_graph_on_tvm function

- The code for quantization can be seen below

```
input_name = "input0"
shape_list = [(input_name, img.shape)]
# convert scripted model to relay model
mod, params = relay.frontend.from_pytorch(scripted_model, shape_list)
# quantization to int8
with relay.quantize.qconfig():
    quantized_mod = relay.quantize.quantize(mod, params)
```

4. Just for your own check -- how can you see the TVM code in the compiled module? Did the quantization actually happen, for example, did the datatypes in the code change?

- By adhering to the instructions on the following discussion, we can get file that contains human-readable LLVM IR code:
  - https://discuss.tvm.apache.org/t/is-there-a-way-compiling-llvm-ir-and-parameters-to-target-library/14979
- As an outcome of this process, I have generated two files:
  - one for quantized - output-compiled-module-quantized-int8.ll
  - and one for unquantized model - output-compiled-module-unquantized.ll

- It can be proven that quantization has been performed by monitoring the reduction in the number of unique data types, indicating a transition from floating-point to lower-precision integer representations

data type \ output file	output-compiled-module-quantized-int8.ll	output-compiled-module-unquantized.ll
float32	19	110
int32	3084	1616
int16	412	220
int8	971	440

Conclusion

- After the quantization process, there’s a notable decrease of over 80% in the float32 data types within the model, accompanied by an increase exceeding 100% in int data type occurrences when compared to the unquantized model
- This substantial shift in data type usage strongly indicates that quantization was successfully implemented

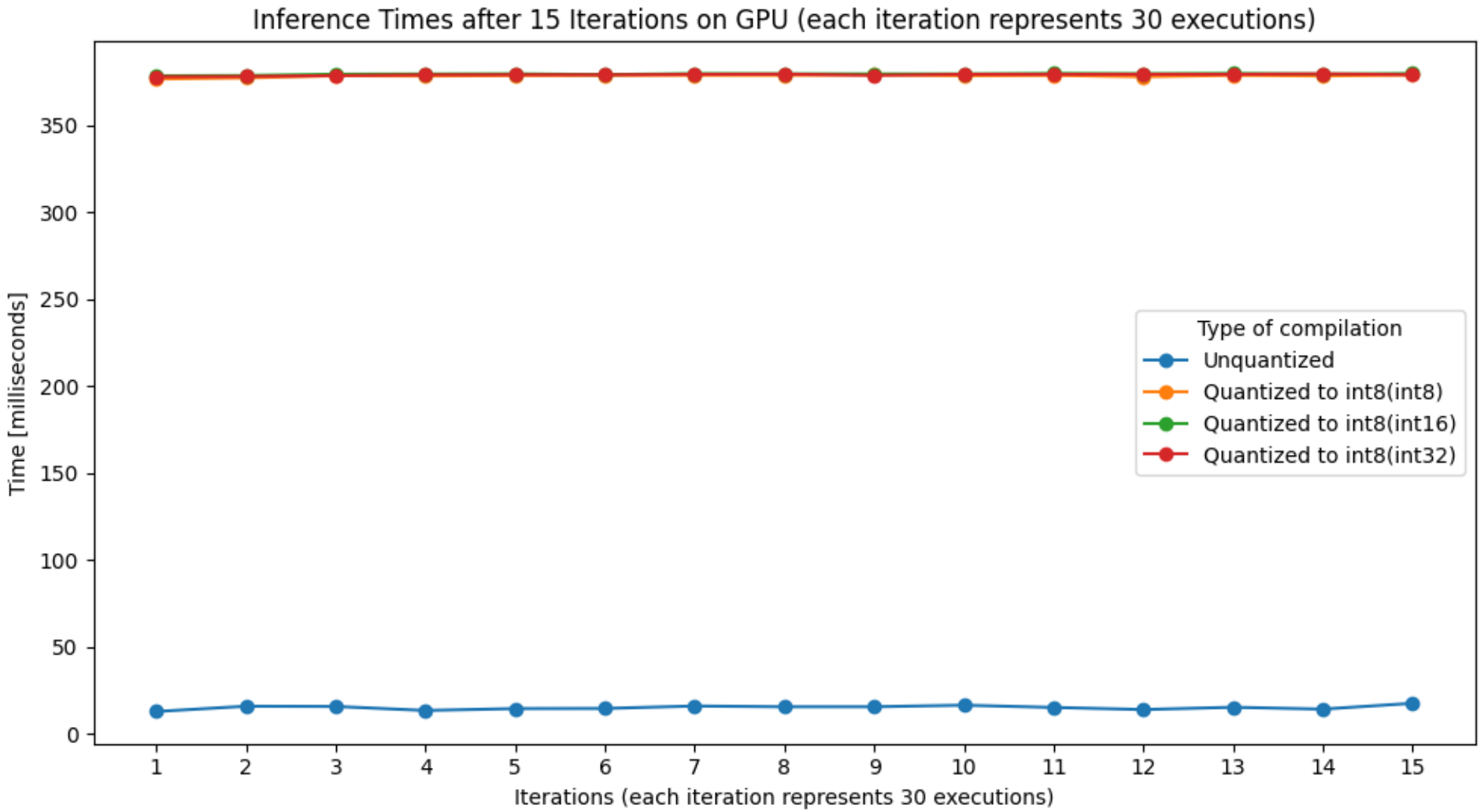
5. Use TVM's utility functions to benchmark the inference time of the quantized model vs. the un-quantized model

- When evaluating the ResNet50 model's accuracy using a cat image, we observe a noticeable decline in accuracy when the activation type is set to an 8-bit integer (int8). However, the model can accurately recognize a cat with 16-bit (int16) or 32-bit integer (int32) activation type
- The reduction in accuracy when using int8 is likely because it represents only 256 distinct values, a significantly narrower range compared to the extensive spectrum offered by float32, this is why I evaluated the inference times of models quantized with different activation types to monitor the effects of this reduced value range

Experiment

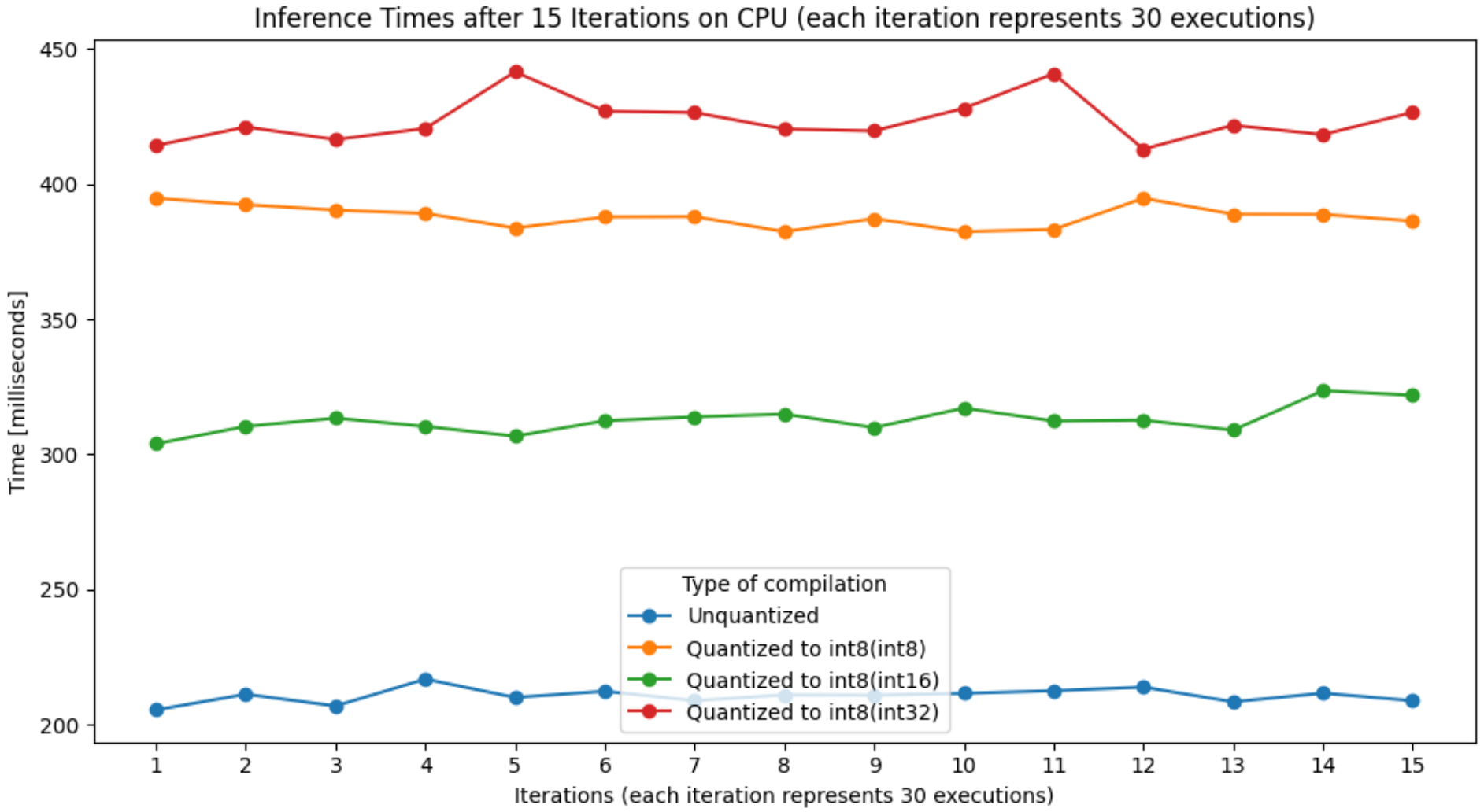
- I conducted various compilation tests on identical dummy inputs (randomly generated images) across 15 iterations
- At the beginning of each iteration, a new dummy input was generated
- Each iteration consisted of 30 executions
- For further details, refer to the resnet50-benchmarking-data.py and resnet50-plotting-data.py scripts

Results



Note: In the 'Type of Compilation' box, the values in brackets represent the 'activation_dtype' parameter.

Results



Note: In the 'Type of Compilation' box, the values in brackets represent the 'activation_dtype' parameter.

Results

Note: In the 'Type of Compilation', the values in brackets represent the 'activation_dtype' parameter.

Conclusions

- Firstly, it is evident that the unquantized model outperforms the quantized model in terms of execution time, regardless of the device being used (CPU or GPU)
- During the program execution, I encountered the following message:
  - One or more operators have not been tuned. Please tune your model for better performance. Use DEBUG logging level to see more details.
- Machine learning compilers, such as TVM, often require model operators to be fine-tuned to achieve optimal performance for a specific hardware

Conclusions

- Thus, the absence of graph-level and operator-level optimization in the code could result in diminished performance, as observed in this instance
- This issue can potentially be addressed by utilizing AutoTVM or AutoScheduler to optimize our model
- Following quantization, I anticipate a reduction in inference time, however I can not claim with certainty that it will outperform the unquantized model in terms of speed

Observations

- It is intriguing to note that the gap in inference time between GPU and CPU executions is not substantial for the quantized model, unlike its unquantized counterpart
- Furthermore, the inference time for the quantized model on the GPU remains consistent, irrespective of the chosen activation precision

Observations

- When executed on a CPU, the quantized model delivers optimal results with the activation set to int16, even though int8 is more aggressive
- Thus, we can optimize for int16 without concern for a negative impact on inference time, with the added benefit that the model’s accuracy is preserved

6. In your quantization setup, how did TVM know that you wanted to quantize to int8? Look into that, and vary the number of bits of quantization (the n in int-n). Searching in forum and peeking the source code of the quantizer class will both help

- We can either manually set configuration parameters by passing arguments to relay.quantize.qconfig() function, or let the function use its default parameters if no arguments are provided

Manual configuration

- The function is informed that we are targeting int8 quantization when we pass the following parameters:
  - dtype_input=“int8”, nbit_input=8
  - dtype_weight=“int8”, nbit_weight=8
  - dtype_activation=“int8”, nbit_activation=8
- The dtype_activation parameter could alternatively be set to int16 or int32, depending on the desired precision

Default configuration

- I visited the following thread:
- https://discuss.tvm.apache.org/t/question-about-qconfig-options-for-quantization/6602
- In this thread, I found a link to the original repository where the default values for quantization are displayed
- https://github.com/apache/tvm/blob/main/python/tvm/relay/quantize/quantize.py
- By visiting the provided GitHub page I reviewed the default settings for quantization which we can see on the right side:

```
_node_defaults = {
    "nbit_input": 8,
    "nbit_weight": 8,
    "nbit_activation": 32,
    "dtype_input": "int8",
    "dtype_weight": "int8",
    "dtype_activation": "int32",
    "calibrate_mode": "global_scale",
    "global_scale": 8.0,
    "weight_scale": "power2",
    "skip_dense_layer": True,
    "skip_conv_layers": [0],
    "do_simulation": False,
    "round_for_shift": True,
    "debug_enabled_ops": None,
    "rounding": "UPWARD",
    "calibrate_chunk_by": -1,
    "partition_conversions": "disabled",
}
```

Try out int8 -> int4 -> int2 -> int1; note which precisions work. When it doesn't work, note exactly which part is failing

- For this purpose, I used a model that can be found at the following link (the entire source code can be found at the bottom of the page):
- https://tvm.apache.org/docs/how_to/compile_models/from_pytorch.html#sphx-glr-how-to-compile-models-from-pytorch-py
- I added relay.quantize.qconfig() function in order to quantize the model

- The only precision that is working properly is int8. For others, the following error occurs:

Traceback (most recent call last):

File “path-to-tvm\tvm\src\relay\transforms\pattern_utils.h", line 304

TVMError: unknown data type int-n

- Here, n represents values from the list [4, 2, 1]
- Additionally, I attempted quantization to bool, which effectively corresponds to int1 quantization, and it executed without issues

Bool quantization parameters

```
with relay.quantize.qconfig(
        global_scale=8.0,
        nbit_input=1,
        nbit_weight=1,
        nbit_activation=1,
        dtype_input="bool",
        dtype_weight="bool",
        dtype_activation="bool"):
    quantized_mod = relay.quantize.quantize(mod, params)
```