Chapter -1 “Vitis AI Overview”

Vitis AI

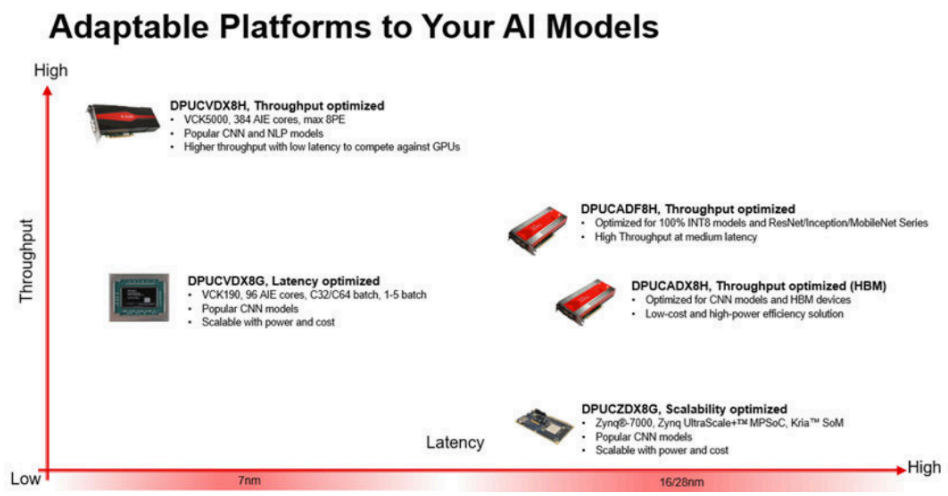
- accelerates AI inference on Xilinx® hardware platforms
- The Vitis AI development environment makes it easy for users without FPGA knowledge to develop deep-learning inference applications by abstracting the intricacies of the underlying programmable logic.

Features
Vitis AI includes the following features:
• Supports mainstream frameworks and the latest models capable of diverse deep learning tasks.
• Provides a comprehensive set of pre-optimized models that are ready to deploy on Xilinx devices.
• Provides a powerful quantizer that supports model quantization, calibration, and fine tuning. For advanced users, Xilinx also offers an optional AI optimizer that can prune a model by up to 90% with a tolerable accuracy loss.
• Provides layer-by-layer analysis to help with bottlenecks.
• Offers unified high-level C++ and Python APIs for maximum portability from Edge to Cloud.
• Customizes efficient and scalable IP cores to meet your needs for many different applications from a throughput, latency, and power perspective

Vitis AI tools overview
Deep-Learning Processor Unit

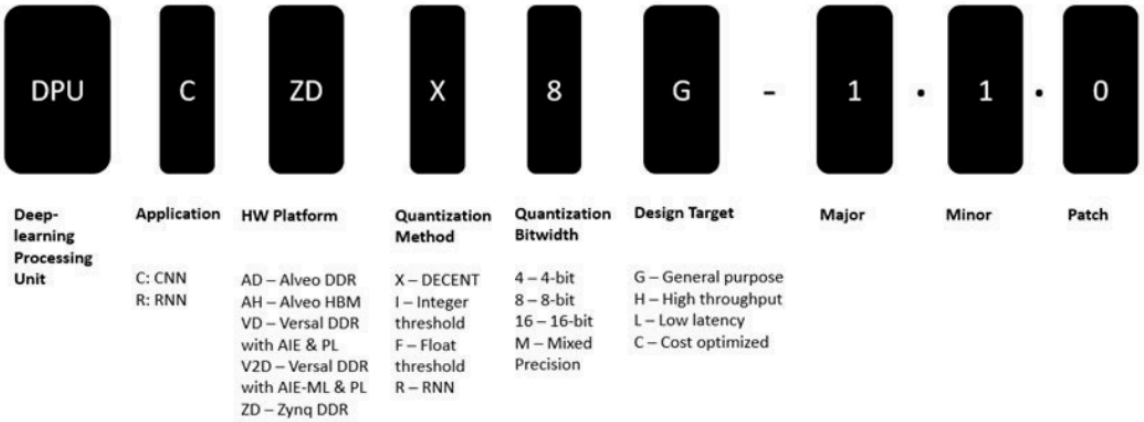
- a programmable engine optimized for deep neural networks. It is a group of parameterizable IP cores pre-implemented on the hardware with no place and route required.
- accelerate the computing workloads of deep learning inference algorithms widely adopted in various computer vision applications
- The DPU is released with the Vitis AI specialized instruction set, thus facilitating the efficient implementation of deep learning networks.

Figure 2: **DPU Options**



DPU Naming

Figure 3: **DPU Nomenclature**



Extra Stuff

- Semiconductor **Intellectual Property core**, commonly referred to as **IP Core or IP block**, is a reusable, circuit, block, or design that is the intellectual property of someone or a company. It can, however, be licensed for use by another party.

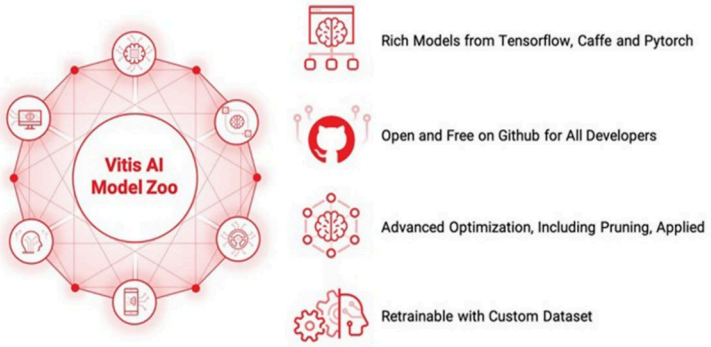
- IP Cores are very common in Application-Specific Integrated Circuits (ASICs) and System on Chip (SoC) designs since they can be used as building blocks to speed up the development time while reducing risk.

- **Programmable logic** refers to the ability to modify the internal logic and connections of a chip or a device that contains logic circuit components, such as logic gates and flip flops, using a software application or a pictorial language

Vitis AI Model Zoo

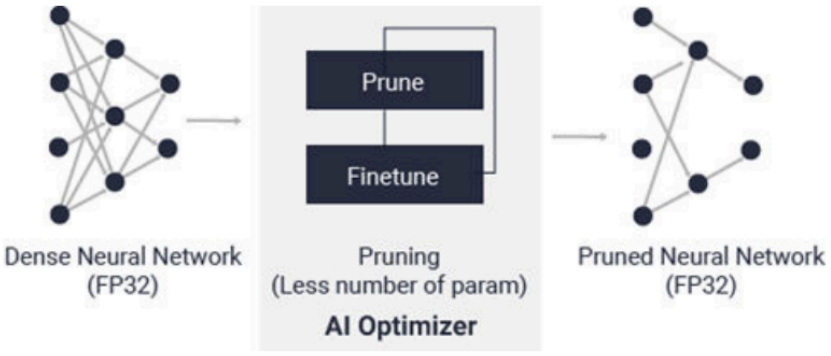
- optimized deep learning models to speed up the deployment of deep learning inference on Xilinx platforms.

Figure 7: **Vitis AI Model Zoo**



Vitis AI Optimizer

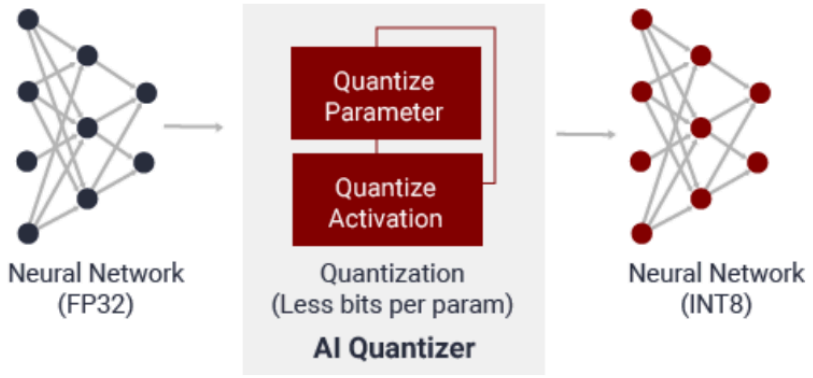
Figure 8: **Vitis AI Optimizer**



Vitis AI Quantizer

- The fixed point network model requires less memory bandwidth, thus providing faster speed and higher power efficiency than the floating-point model.

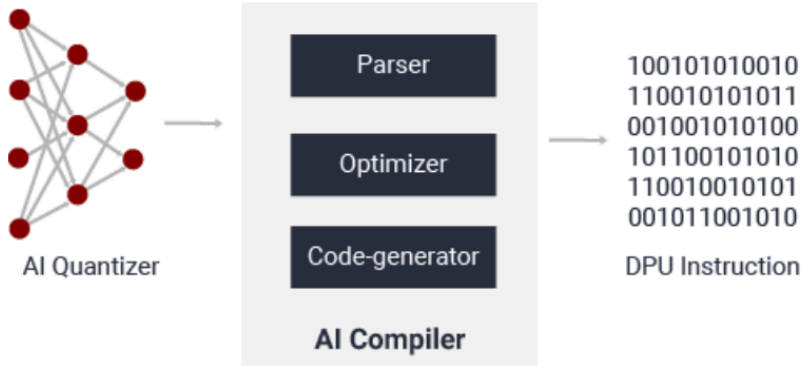
Figure 9: **Vitis AI Quantizer**



Vitis AI compiler

- maps the AI model to a highly-efficient instruction set and dataflow model.
- It also performs sophisticated optimizations such as layer fusion, instruction scheduling, and reuses on-chip memory as much as possible

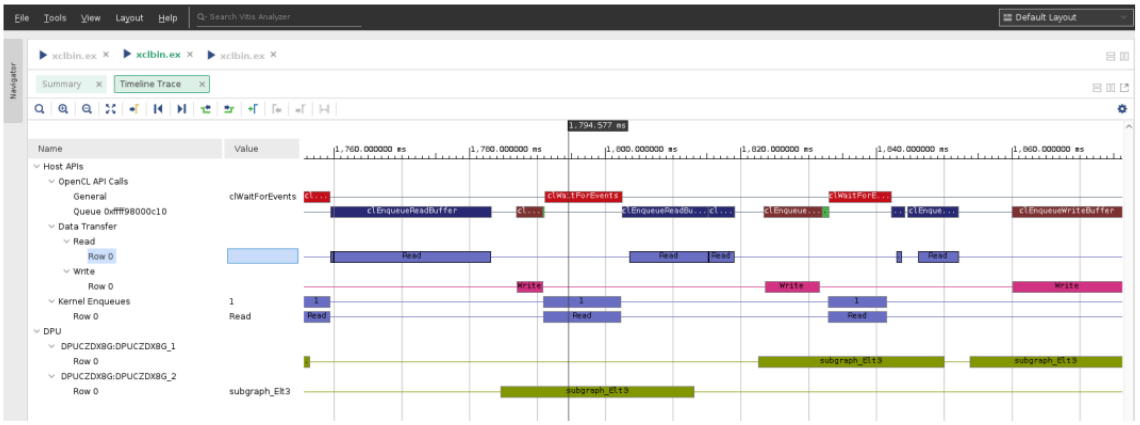
Figure 10: **Vitis AI Complier**



Vitis AI Profiler

- profiles and visualizes AI applications to find bottlenecks and allocates computing resources among different devices.

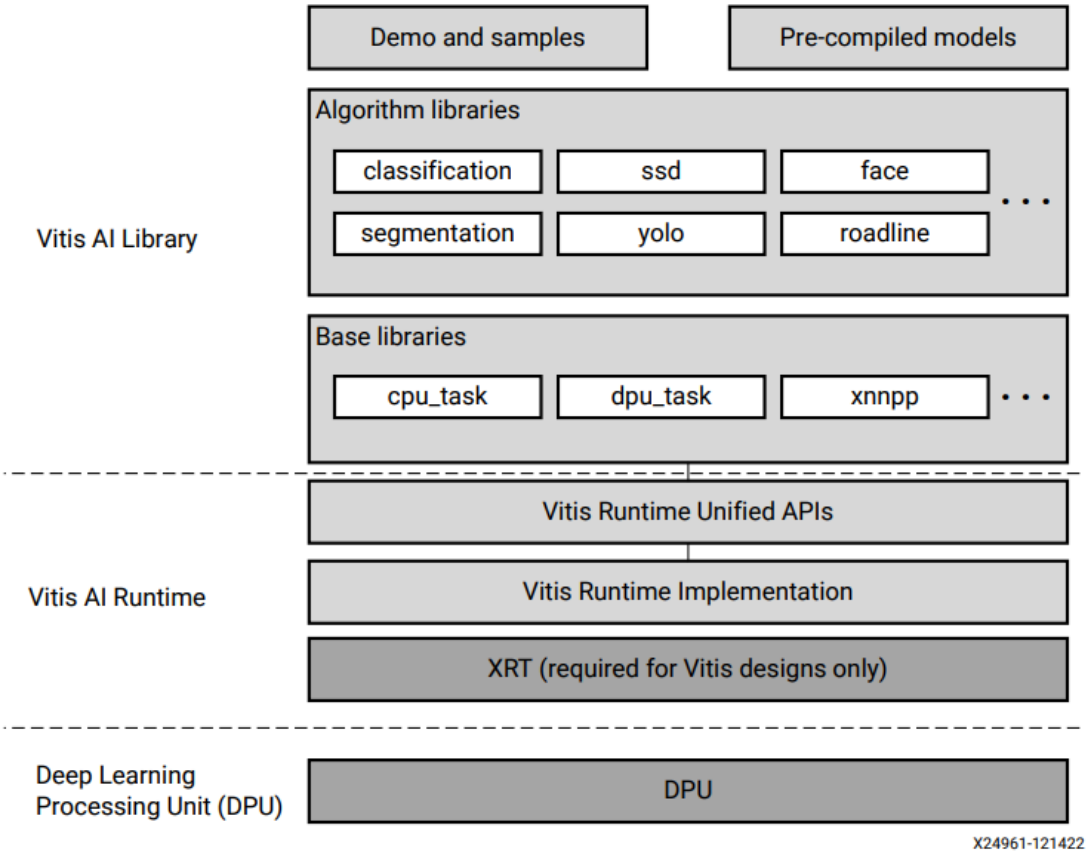
Figure 11: **Vitis AI Profiler**



Vitis AI Library

- set of high-level libraries and APIs
- built for efficient AI inference with DPUs.
- built on Vitis AI runtime with Vitis runtime unified APIs

Figure 12: **Vitis AI Library**



Vitis AI Runtime

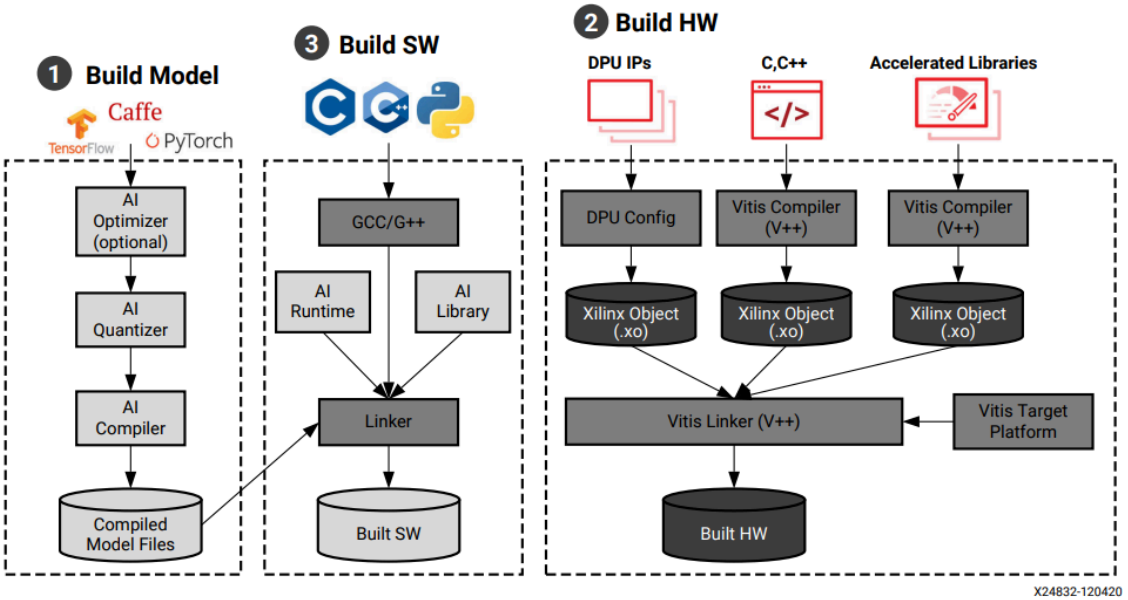
- enables applications to use the unified high-level runtime API for both Cloud and Edge making Cloud-to-Edge deployments seamless and efficient.

Following are the features for the AI runtime API:
• Asynchronous submission of jobs to the accelerator
• Asynchronous collection of jobs from the accelerator
• C++ and Python implementations
• Support for multi-threading and multi-process execution

Extra Stuff- In computer science, a **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The idea is to achieve parallelism by dividing a process into **multiple threads**. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc.

Development Flow Overview

Figure 14: **Vitis AI Flow**



Chapter 2 “Getting Started”

1. Setting up Vitis AI Development Environment
    - cd Vitis-AI
    - ./docker_run.sh xilinx/vitis-ai-pytorch-cpu:latest
2. List pre-trained models from model zoo
    - ls model_zoo/model-list
3. Select a model – gives description of the model and download link
    - cat model_zoo/model-list/<model_name>/model.yaml
4. Get the model
    - wget <model_download_link>
5. List the conda environment
    - conda-env list
6. Activate the environment
    - conda activate <environment_name>
7. Read model info
    - cat model_zoo/model-list/<model_name>/model_info.md

(vitis-ai-pytorch) Vitis-AI/workspace/pytorch_model_name > ls
Output: 1. dependency_installation.sh
2. quant.sh (for quantization)
3. compile.sh (for compilation)

Steps:

- pip install requirements.txt

Quantization:

- cat quant.sh → Output: Give python file for quantization and model with other parameters already set.
- ./quant.sh → To start the quantization, output: quantization_results which has quantized.h5 (model)

Compile:

- cat compile.sh → Output: Target, model name, arch.json (After DPU compilation)
- ./compile.sh → Start compilation, output: generates xmodel and compiles to DPU architecture. And compilation_results which has xmodel.

Design Tutorial

MNIST

Steps:

1. Setting up the workspace
2. Training
   (vitis-ai-pytorch) Vitis-AI /workspace > export BUILD=./build
   (vitis-ai-pytorch) Vitis-AI /workspace > export LOG=${BUILD}/logs
   (vitis-ai-pytorch) Vitis-AI /workspace > mkdir -p ${LOG}
   (vitis-ai-pytorch) Vitis-AI /workspace > python -u train.py -d ${BUILD} 2>&1 | tee ${LOG}/train.log

   The command you provided is a combination of several components in a shell command:

   1. `python -u train.py`: This executes the Python script `train.py` using the Python interpreter. The `-u` flag is used to enable unbuffered output, which means that the output will be immediately displayed rather than being buffered.

   2. `-d ${BUILD}`: This is a command-line argument passed to the `train.py` script. The value of `${BUILD}` is being passed as the value for the `-d` argument. The actual value of `${BUILD}` is not visible in the command you provided.

   3. `2>&1`: This is a shell redirection operator. `2` refers to the file descriptor for standard error (stderr), and `1` refers to the file descriptor for standard output (stdout). `2>&1` redirects stderr to the same location as stdout. In this case, both stderr and stdout are combined and sent to the next part of the command.

   4. `|`: This is the pipe operator, which takes the output from the previous command and sends it as input to the next command.

   5. `tee ${LOG}/train.log`: `tee` is a command that reads from standard input and writes to both standard output and files. `${LOG}/train.log` is the filename where the output will be written. The actual value of `${LOG}` is not visible in the command you provided.

   In summary, this command runs the `train.py` Python script with a specific argument, captures both stdout and stderr, and then pipes the combined output to the `tee` command, which writes the output to a file while also displaying it on the console. The specific filenames and values for `${BUILD}` and `${LOG}` are not visible in the command you provided.

3. Quantization
   (vitis-ai-pytorch) Vitis-AI /workspace > python -u quantize.py -d ${BUILD}
   --quant_mode calib 2>&1 | tee ${LOG}/quant_calib.log
   (vitis-ai-pytorch) Vitis-AI /workspace > python -u quantize.py -d ${BUILD}
   --quant_mode test 2>&1 | tee ${LOG}/quant_test.log

   The command you provided executes a Python script named `quantize.py` with specific command-line arguments and captures both the standard output (stdout) and standard error (stderr) streams. Here's a breakdown of the command:

   1. `python`: This is the command used to invoke the Python interpreter and run Python scripts.

   2. `-u`: This flag is used to enable unbuffered output. It ensures that the output is immediately displayed rather than being buffered.

   3. `quantize.py`: This is the name of the Python script that will be executed.

   4. `-d ${BUILD}`: This is a command-line argument passed to the `quantize.py` script. The value of `${BUILD}` is being passed as the value for the `-d` argument. The actual value of `${BUILD}` is not visible in the command you provided. It represents the build directory path.

   5. `--quant_mode calib`: This is another command-line argument passed to the `quantize.py` script. It specifies the quantization mode as "calib". The specific behavior of this mode depends on how it is implemented in the `quantize.py` script.

   6. `2>&1`: This is a shell redirection operator. It redirects the standard error (stderr) stream to the same location as the standard output (stdout). In other words, it combines both stderr and stdout into a single stream.

   7. `|`: This is the pipe operator, which takes the output from the previous command and sends it as input to the next command.

   8. `tee ${LOG}/quant_calib.log`: The `tee` command reads from standard input and writes to both standard output and files. `${LOG}/quant_calib.log` is the filename where the output will be written. The actual value of `${LOG}` is not visible in the command you provided. The output of the `quantize.py` script, which includes both stderr and stdout, will be displayed on the console and also written to the specified log file.

   In summary, this command runs the `quantize.py` Python script with the `-d ${BUILD}` and `--quant_mode calib` arguments, and captures the combined output (stderr and stdout) using the `tee` command. The output is displayed on the console and saved to the `quant_calib.log` file. The specific values for `${BUILD}` and `${LOG}` are not visible in the command you provided.

Extra Stuff:

- from pytorch_nndct.apis import torch_quantizer
- PyTorch supports INT8 quantization compared to typical FP32 models allowing for a 4x reduction in the model size and a 4x reduction in memory bandwidth requirements. Hardware support for INT8 computations is typically 2 to 4 times faster compared to FP32 compute. Quantization is primarily a technique to speed up inference and only the forward pass is supported for quantized operators.

- `torch_quantizer` and `dump_xmodel` are APIs (application programming interfaces) from the `pytorch_nndct` package. These APIs provide functionality related to quantization and model export. Here's an explanation of each API:

  1. `torch_quantizer`:
     - `torch_quantizer` is a function or class from the `pytorch_nndct` package that facilitates the quantization of PyTorch models.
     - It provides methods and utilities to perform various types of quantization, such as post-training quantization, quantization-aware training, or custom quantization schemes.
     - The exact usage and behavior of `torch_quantizer` depend on the specific implementation and documentation of the `pytorch_nndct` package you are using.
     - You can consult the package documentation or API reference to learn more about the available functionalities, method parameters, and usage examples.

  2. `dump_xmodel`:
     - `dump_xmodel` is a function or class from the `pytorch_nndct` package that is used to export a quantized model in the XModel format.
     - The XModel format is a specific model format that can be used for deployment on various platforms or inference engines.
     - The `dump_xmodel` function takes a quantized model object or related inputs and generates an XModel file.
     - The specific parameters and usage of `dump_xmodel` may vary depending on the implementation and documentation of the `pytorch_nndct` package you are using.
     - You can refer to the package documentation or API reference for detailed information on the available parameters and usage examples.

  In summary, `torch_quantizer` and `dump_xmodel` are APIs from the `pytorch_nndct` package that provide functionality related to quantization and exporting quantized models in the XModel format. The exact usage and behavior of these APIs depend on the specific implementation and documentation of the package you are using.

The command you provided executes the `quantize.py` Python script with specific command-line arguments, captures both the standard output (stdout) and standard error (stderr) streams, and redirects the combined output to both the console and a log file. Here's an explanation of the command:

1. `python`: This is the command used to invoke the Python interpreter and run Python scripts.

2. `-u`: This flag is used to enable unbuffered output. It ensures that the output is immediately displayed rather than being buffered.

3. `quantize.py`: This is the name of the Python script that will be executed.

4. `-d ${BUILD}`: This is a command-line argument passed to the `quantize.py` script. The value of `${BUILD}` is being passed as the value for the `-d` argument. The actual value of `${BUILD}` is not visible in the command you provided. It represents the build directory path.

5. `--quant_mode test`: This is another command-line argument passed to the `quantize.py` script. It specifies the quantization mode as "test". The specific behavior of this mode depends on how it is implemented in the `quantize.py` script.

6. `2>&1`: This is a shell redirection operator. It redirects the standard error (stderr) stream to the same location as the standard output (stdout). In other words, it combines both stderr and stdout into a single stream.

7. `|`: This is the pipe operator, which takes the output from the previous command and sends it as input to the next command.

8. `tee ${LOG}/quant_test.log`: The `tee` command reads from standard input and writes to both standard output and files. `${LOG}/quant_test.log` is the filename where the output will be written. The actual value of `${LOG}` is not visible in the command you provided. The output of the `quantize.py` script, which includes both stderr and stdout, will be displayed on the console and also written to the specified log file.

In summary, this command runs the `quantize.py` Python script with the `-d ${BUILD}` and `--quant_mode test` arguments, captures the combined output (stderr and stdout) using the `tee` command, and writes it to both the console and the `quant_test.log` file. The specific values for `${BUILD}` and `${LOG}` are not visible in the command you provided.

Extra Stuff:

- If the quant_mode is 'calib', it calls the export_quant_config() method of the quantizer object. This method is responsible for exporting the quantization configuration. The specifics of this method are not visible in the code snippet you provided, as it likely belongs to a custom class or module.

- If the quant_mode is 'test', it calls the export_xmodel() method of the quantizer object. This method is responsible for exporting an XModel file. The deploy_check argument is set to False, indicating that the XModel is not intended for deployment checking. The output_dir argument specifies the directory where the quantized model will be saved. The exact details of the export_xmodel() method are not visible in the provided code snippet.

4. Compiling for the Target
(vitis-ai-pytorch) Vitis-AI /workspace > source compile.sh zcu102 ${BUILD} ${LOG}

   The command `source compile.sh zcu102 ${BUILD} ${LOG}` is executing a shell script named `compile.sh` with three arguments: `zcu102`, `${BUILD}`, and `${LOG}`. Let's break down the command:

   1. `source`: This command is used to execute a shell script in the current shell environment. It ensures that any changes or environment variables set within the script are applied to the current shell session.

   2. `compile.sh`: This is the name of the shell script that will be executed.

   3. `zcu102`: This is the first argument passed to the `compile.sh` script. It appears to be a parameter or option specific to the script. The exact purpose or behavior of this argument depends on the implementation of the `compile.sh` script.

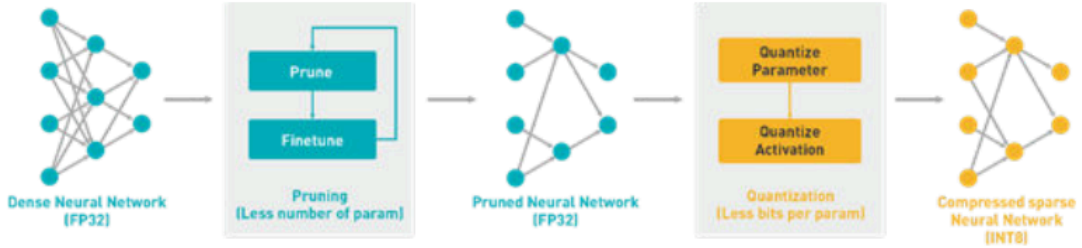
   4. `${BUILD}`: This is the second argument passed to the `compile.sh` script. The value of `${BUILD}` is being passed as an argument to the script. The actual value of `${BUILD}` is not visible in the command you provided. It likely represents a build directory path or some other configuration parameter.

   5. `${LOG}`: This is the third argument passed to the `compile.sh` script. The value of `${LOG}` is being passed as an argument to the script. The actual value of `${LOG}` is not visible in the command you provided. It likely represents a log directory path or another configuration parameter.

   In summary, this command executes the `compile.sh` shell script with the `zcu102` argument and passes the values of `${BUILD}` and `${LOG}` as additional arguments. The specific behavior and purpose of these arguments depend on the implementation of the `compile.sh` script.

Chapter-3 “Quantizing the Model”

Figure 18: **Pruning and Quantization Flow**



By converting the 32-bit floating-point weights and activations to 8-bit integer (INT8) format, the Vitis AI quantizer can reduce computing complexity without losing prediction accuracy. The fixed-point network model requires less memory bandwidth, thus providing faster speed and higher power efficiency than the floating-point model.

Step 1:
Quant mode = “calib” → To get Quant config file

[VAIQ_NOTE]: Quant config file is empty, use default quant configuration

[VAIQ_NOTE]: Quantization calibration process start up...

[VAIQ_NOTE]: =>Quant Module is in 'cpu'.

[VAIQ_NOTE]: =>Parsing CNN… (This is our model architecture)

[VAIQ_NOTE]: Start to trace and freeze model...

[VAIQ_NOTE]: The input model nndct_st_CNN_ed is torch.nn.Module.

[VAIQ_NOTE]: Finish tracing.

[VAIQ_NOTE]: Processing ops...

[VAIQ_NOTE]: =>Doing weights equalization...

[VAIQ_NOTE]: =>Quantizable module is generated.(./build/quant_model/CNN.py)

Here, it generates a quantized model using default quant configuration.

Then the model is tested:
Test set: Accuracy: 9892/10000 (98.92%)

[VAIQ_NOTE]: =>Exporting quant config.(./build/quant_model/quant_info.json)

Finally, we get quant config file.

Step 2:
Quant mode = “test” → To get xmodel

[VAIQ_NOTE]: Quant config file is empty, use default quant configuration

[VAIQ_NOTE]: Quantization test process start up...

[VAIQ_NOTE]: =>Quant Module is in 'cpu'.

[VAIQ_NOTE]: =>Parsing CNN…(This is the model architecture)

[VAIQ_NOTE]: Start to trace and freeze model...

[VAIQ_NOTE]: The input model nndct_st_CNN_ed is torch.nn.Module.

[VAIQ_NOTE]: Finish tracing.

[VAIQ_NOTE]: Processing ops...

[VAIQ_NOTE]: =>Doing weights equalization...

[VAIQ_NOTE]: =>Quantizable module is generated.(./build/quant_model/CNN.py)

Here, it generates a quantized model using quant configuration.

[VAIQ_NOTE]: =>Get module with quantization.

Then the model is tested:
Test set: Accuracy: 9893/10000 (98.93%)

[VAIQ_NOTE]: =>Converting to xmodel ...

[VAIQ_NOTE]: =>Successfully convert 'CNN' to
xmodel.(./build/quant_model/CNN_int.xmodel)

Finally, we get xmodel.

Table 4: **Vitis AI Quantizer Supported Frameworks and Features**

Model	Versions	Features			
		Post Training Quantization (PTQ)	Quantization Aware Training (QAT)	Fast Finetuning (Advanced Calibration)	Inspector
TensorFlow 1.x	Supports 1.15	Yes	Yes	No	No
TensorFlow 2.x	Supports 2.3 - 2.10	Yes	Yes	Yes	Yes
PyTorch	Supports 1.2 - 1.12	Yes	Yes	Yes	Yes

Post training quantization (PTQ) requires only a small set of unlabeled images to analyze the distribution of activations. The running time of quantize calibration varies from a few seconds to several minutes, depending on the size of the neural network. Generally, there is some drop in accuracy after quantization. However, for some networks such as Mobilenet, the accuracy loss might be large.

In this situation, quantization aware training (QAT) can be used to further improve the accuracy of the quantized models. QAT requires the original training dataset. Several epochs of finetuning are needed and the finetune time varies from several minutes to several hours. It is recommended to use small learning rates when performing QAT.

Here's an overview of the steps involved in quantization-aware training:

1. Training with simulated quantization: During quantization-aware training, the model is trained using a training dataset, but with simulated quantization applied to the weights and/or activations. This means that quantization is emulated by using lower precision representations during forward and backward passes, but the actual quantization is not applied to the model parameters.

2. Quantization-aware loss function: In quantization-aware training, a modified loss function is often used to account for the quantization effects. The loss function takes into consideration the discrepancy between the high-precision and low-precision representations. This encourages the model to learn representations that are more robust to quantization.

3. Fine-tuning with real quantization: After the initial training with simulated quantization, the model is typically fine-tuned using real quantization. This involves quantizing the model's weights and/or activations to the desired lower precision representation. The fine-tuning process aims to compensate for any discrepancies between the simulated quantization and the actual quantization, further improving the model's performance under quantization.

4. Validation and evaluation: The quantization-aware trained and fine-tuned model is evaluated on validation datasets to assess its accuracy and performance metrics, ensuring that it maintains acceptable performance after quantization.

For PTQ:

- Cross layer equalization algorithm
- AdaQuant algorithm

Vitis AI quantizer implements this algorithm and call it "fast finetuning" or "advanced calibration."

Inspector
Before quantizing the float model, there is an optional step called "inspector". It is used to inspect the model before quantizing it. Inspector will output the partition information, indicating which operators will run on which device (DPU/CPU).

VAI_Q_PYTORCH

1. Parse neural network computation graph from Pytorch framework to Intermediate Representation (IR).
2. Modify the computation graph based on IR. Pruning, quantization and graph optimization are handled in this stage.
3. Deploy the modified IR on different platform, such as DPU and Xilinx AI Engine.
4. Assemble the modified computation graph back to Pytorch. In that way abilities and tools in Pytorch such as pre-processing, post processing and distribution system can be used.

Steps:

1. cd Vitis-AI
2. git checkout 3.0 (checkout to the branch 3.0)
3. Start Vitis AI Development Environment

```
jinunyachhyon@VENOM:~/Vitis-AI$ ./docker_run.sh xilinx/vitis-ai-pytorch-cpu:latest
latest: Pulling from xilinx/vitis-ai-pytorch-cpu
Digest: sha256:baf43fd9d6e1ea5a2c12b727ac3aafd15ead936172bcd636d7af98b0cca49298
Status: Image is up to date for xilinx/vitis-ai-pytorch-cpu:latest
docker.io/xilinx/vitis-ai-pytorch-cpu:latest
Setting up jinunyachhyon 's environment in the Docker container...
usermod: no changes
Running as vitis-ai-user with ID 0 and group 0


==========================================

__      ___ _   _                 ____
\ \    / (_) | (_)          /\   |_  _|
 \ \  / / _| |_ _ ___ _____ /  \    | |
  \ \/ / | | __| / __|_____/ /\ \   | |
   \  /  | | |_| \__ \     / ____ \ _| |_
    \/   |_|\__|_|___/    /_/    \_\_____|

==========================================

Docker Image Version: ubuntu2004-3.5.0.306   (CPU)
Vitis AI Git Hash: 6a9757a
Build Date: 2023-06-26
WorkFlow: pytorch
```

4. Activate vitis-ai-pytorch

```
vitis-ai-user@docker-desktop:/workspace$ conda activate vitis-ai-pytorch
(vitis-ai-pytorch) vitis-ai-user@docker-desktop:/workspace$ ls
board_setup           CONTAINER-ROCM-LICENSE  docker         docs    dpu       LICENSE     README.md  third_party
CONTAINER-CPU-LICENSE  demos                  docker_run.sh  docsrc  examples  model_zoo  src
```

5. Inside ./src/vai_quantizer/vai_q_pytorch/example
   Create “build” directory: To store dataset and output quantized model
   Create “model” directory: To store model / pre-trained model downloaded

```
(vitis-ai-pytorch) vitis-ai-user@docker-desktop:/workspace/src/vai_quantizer/vai_q_pytorch/example$ ls
bfp                       jupyter_notebook             pytorch_quantize_config.json  resnet18_quant.py
build                     mobilenetv2_fast_finetune.py  resnet18_qat.py
inspector_tutorial.ipynb  model                        resnet18_quant_custom_op.py
```

6. Evaluate float model
   python resnet18_quant.py --quant_mode float
   By running the script with --quant_mode float, it is likely that the script will bypass the quantization process and execute the ResNet-18 model without any quantization applied. This can be useful for comparison or debugging purposes to observe the model's behavior in full precision before quantization.

```
(vitis-ai-pytorch) vitis-ai-user@docker-desktop:/workspace/src/vai_quantizer/vai_q_pytorch/example$ python resnet18_quan
t.py --quant_mode float
No CUDA runtime is found, using CUDA_HOME='/usr/local/cuda'

[VAIQ_NOTE]: Loading NNDCT kernels...
model/resnet18.pth
-------- Start resnet18 test
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./build/val/cifar-10-python.tar.gz
100%|                                                                                | 170498071/170498071 [00:23<00:00, 7148949.49it/s]
Extracting ./build/val/cifar-10-python.tar.gz to ./build/val
100%|                                                                                | 313/313 [12:57<00:00,  2.48s/it]
loss: 0.414608
top-1 / top-5 accuracy: 0.02 / 0.06
-------- End of resnet18 test
```

7.

Quantization Process
The Vitis AI transformation process of a trained model towards deployment goes through Optimization, Quantization and Compilation steps.

Quantization in particular can be achieved in three different ways.

1. Post Training Quantization: the simplest quantization method consists of two steps: "Calibration", where the quantization steps of tensors are determined, and the actual deployment of the quantized model.

2. Fast Finetuning Quantization: a more sophisticated method based on the AdaQuant algorithm, it not only calibrates the activations but it also finetunes the weights. This method can achieve better performance than Post Training Quantization. As with the previous method, it starts from an already trained model.

3. Quantization Aware Training (QAT): as the name suggests, the model is trained for best performance after quantization.

Fast Finetuning

```
# first step: calibration and finetuning
python -u fast_finetune_quant.py --quant_mode calib

# second step: test and model deployment
python -u fast_finetune_quant.py --quant_mode test  --deploy
```

Code:

```
 if finetune == True:
      ft_loader, _ = load_data(
          subset_len=20,
          train=False,
          batch_size=batch_size,
          sample_method='random',
          model_name=model_name)
      if quant_mode == 'calib':
        quantizer.fast_finetune(evaluate, (quant_model, ft_loader,
loss_fn))
      elif quant_mode == 'test':
        quantizer.load_ft_param()
```

- quantizer.fast_finetune() →
  The AdaQuant algorithm uses a small set of unlabeled data. It not only calibrates the activations but also finetunes the weights. The Vitis AI quantizer implements this algorithm and under the alias "fast finetuning". Though slightly slower, fast finetuning can achieve better performance than quantize calibration.

  Output:
  [VAIQ_NOTE]: OS and CPU information:
  system --- Linux
  node --- docker-desktop
  release --- 5.15.90.1-microsoft-standard-WSL2
  version --- #1 SMP Fri Jan 27 02:56:13 UTC 2023
  machine --- x86_64
  processor --- x86_64

  [VAIQ_NOTE]: Tools version information:
  GCC --- GCC 7.5.0
  python --- 3.8.6
  pytorch --- 1.13.1
  vai_q_pytorch --- 3.5.0+60df3f1+torch1.13.1

  [VAIQ_NOTE]: Quant config file is empty, use default quant configuration

  [VAIQ_NOTE]: Quantization calibration process start up...

  [VAIQ_NOTE]: =>Quant Module is in 'cpu'.

  [VAIQ_NOTE]: =>Parsing ResNet...

  [VAIQ_NOTE]: Start to trace and freeze model...

  [VAIQ_NOTE]: The input model nndct_st_ResNet_ed is torch.nn.Module.

[VAIQ_NOTE]: Finish tracing.

[VAIQ_NOTE]: Processing ops...

178/178 [00:00<00:00, 342.17it/s, OpInfo: name = return_0, type = Return]

[VAIQ_NOTE]: =>Doing weights equalization...

[VAIQ_NOTE]: =>Quantizable module is generated.(quantize_result/ResNet.py)

[VAIQ_NOTE]: =>Get module with quantization.

[VAIQ_NOTE]: =>Preparing data for fast finetuning module parameters ...

[VAIQ_NOTE]: Mem status(total mem: 3.76G, available mem: 1.50G).

[VAIQ_NOTE]: Memory cost by fast finetuning is 0.01 G.

[VAIQ_NOTE]: =>Find initial quantization steps for fast finetuning...
/opt/vitis_ai/conda/envs/vitis-ai-pytorch/lib/python3.8/site-packages/pytorch_nndct/quantization/torchquantizer.py:223: FutureWarning: Unlike other reduction functions (e.g. `skew`, `kurtosis`), the default behavior of `mode` typically preserves the axis it acts along. In SciPy 1.11.0, this behavior will change: the default value of `keepdims` will become False, the `axis` over which the statistic is taken will be eliminated, and the value None will no longer be accepted. Set `keepdims` to True or False to avoid this warning.
  bnfp[1] = stats.mode(data)[0][0]

[VAIQ_NOTE]: =>Fast finetuning module parameters for better quantization accuracy...
100% 
54/54 [1:00:02<00:00, 66.71s/it]

[VAIQ_NOTE]: =>Export fast finetuned parameters ...

[VAIQ_NOTE]: =>Exporting quant model parameters.(quantize_result/param.pth)
Loss: 4.4119, Accuracy: 10.0000

[VAIQ_NOTE]: =>Exporting quant config.(quantize_result/quant_info.json)
-------- End of resnet_model_OF test

- quantizer.load_ft_param() → Loads the fine finetuned model and test the fast finetuned model

  Output:
  [VAIQ_NOTE]: OS and CPU information:
  system --- Linux
  node --- docker-desktop
  release --- 5.15.90.1-microsoft-standard-WSL2
  version --- #1 SMP Fri Jan 27 02:56:13 UTC 2023
  machine --- x86_64
  processor --- x86_64

  [VAIQ_NOTE]: Tools version information:
  GCC --- GCC 7.5.0
  python --- 3.8.6
  pytorch --- 1.13.1
  vai_q_pytorch --- 3.5.0+60df3f1+torch1.13.1

  [VAIQ_NOTE]: Quant config file is empty, use default quant configuration

  [VAIQ_NOTE]: Quantization test process start up...

  [VAIQ_NOTE]: =>Quant Module is in 'cpu'.

  [VAIQ_NOTE]: =>Parsing ResNet...

  [VAIQ_NOTE]: Start to trace and freeze model...

  [VAIQ_NOTE]: The input model nndct_st_ResNet_ed is torch.nn.Module.

  [VAIQ_NOTE]: Finish tracing.

  [VAIQ_NOTE]: Processing ops...
  
  178/178 [00:00<00:00, 365.02it/s, OpInfo: name = return_0, type = Return]

  [VAIQ_NOTE]: =>Doing weights equalization...

  [VAIQ_NOTE]: =>Quantizable module is generated.(quantize_result/ResNet.py)

  [VAIQ_NOTE]: =>Get module with quantization.

  [VAIQ_NOTE]: =>Loading quant model parameters.(quantize_result/param.pth)
  Loss: 4.3783, Accuracy: 10.0000
  -------- End of resnet_model_OF test

Quantization Aware Training

The process is explained step by step below:

1) Set device to GPU and get a trainable model:

```
qat_processor = QatProcessor(model, rand_in, bitwidth=8,
device=torch.device('gpu'))
quantized_model = qat_processor.trainable_model()
train(quantized_model)
```

Note: the model and rand_in must be in the GPU, so when creating them be sure to set the variable device properly:

```
device = torch.device('cuda:0') # in case of multiple GPUs
model = CNN_QAT().to(device)
rand_in = torch.randn([batchsize, 1, 28, 28], dtype=torch.float32).to(device)
```

2 Save the trained model:

```
torch.save(quantized_model.state_dict(), './QAT_quantized_model.pth')
```

3) Set device to CPU and convert quantized model to deployable model:

```
qat_processor = QatProcessor(model, rand_in, bitwidth=8,
device=torch.device('cpu'))
quantized_model = qat_processor.trainable_model()
quantized_model.load_state_dict(torch.load('./QAT_quantized_model.pth'))

deployable_model = qat_processor.to_deployable(quantized_model, output_dir)
for images, _ in test_loader:
    deployable_model(images)
```

4) Finally, generate the xmodel starting from the deployable model

```
qat_processor.export_xmodel(output_dir)
```

Output:

```
No CUDA runtime is found, using CUDA_HOME='/usr/local/cuda'

[VAIQ_NOTE]: Loading NNDCT kernels...
Used arguments: Namespace(data_dir='build', display_freq=10, epochs=3, mode='train',
output_dir='qat_result', pretrained='resnet_model_OF.pth', quantizer_lr=0.01,
quantizer_lr_decay=0.5, quantizer_norm=True, save_dir='./qat_models',
train_batch_size=24, val_batch_size=10, val_freq=10, weight_decay=0.0001,
weight_lr=1e-05, weight_lr_decay=0.94, workers=4)
Files already downloaded and verified
Files already downloaded and verified

[VAIQ_NOTE]: Quant config file is empty, use default quant configuration

[VAIQ_NOTE]: Quantization calibration process start up...

[VAIQ_NOTE]: =>Quant Module is in 'cpu'.

[VAIQ_NOTE]: =>Parsing ResNet...

[VAIQ_NOTE]: Start to trace and freeze model...
```

[VAIQ_NOTE]: The input model nndct_st_ResNet_ed is torch.nn.Module.

[VAIQ_NOTE]: Finish tracing.

[VAIQ_NOTE]: Processing ops...

| 180/180 [00:00<00:00, 365.08it/s, OpInfo: name = return_0, type = Return]

[VAIQ_NOTE]: =>Quantizable module is generated.(.vai_qat/ResNet.py)

[VAIQ_NOTE]: =>Exporting quant config.(.vai_qat/quant_info.json)

[VAIQ_NOTE]: Start to trace and freeze model...

[VAIQ_NOTE]: The input model nndct_st_ResNet_ed is torch.nn.Module.

[VAIQ_NOTE]: Finish tracing.

[VAIQ_NOTE]: Processing ops...

| 180/180 [00:00<00:00, 374.02it/s, OpInfo: name = return_0, type = Return]
Adjust lr at epoch 0, step 0: group_name=quantizer, lr=0.01
Adjust lr at epoch 0, step 0: group_name=weight, lr=1e-05
Killed
(vitis-ai-pytorch)
vitis-ai-user@docker-desktop:/workspace/src/vai_quantizer/vai_q_pytorch/example/resnet$ python resnet_qat.py
No CUDA runtime is found, using CUDA_HOME='/usr/local/cuda'

[VAIQ_NOTE]: Loading NNDCT kernels...
Used arguments: Namespace(data_dir='build', display_freq=10, epochs=3, mode='train', output_dir='qat_result', pretrained='resnet_model_OF.pth', quantizer_lr=0.01, quantizer_lr_decay=0.5, quantizer_norm=True, save_dir='./qat_models', train_batch_size=1, val_batch_size=1, val_freq=10, weight_decay=0.0001, weight_lr=1e-05, weight_lr_decay=0.94, workers=4)
Files already downloaded and verified
Files already downloaded and verified

[VAIQ_NOTE]: Quant config file is empty, use default quant configuration

[VAIQ_NOTE]: Quantization calibration process start up...

[VAIQ_NOTE]: =>Quant Module is in 'cpu'.

[VAIQ_NOTE]: =>Parsing ResNet...

[VAIQ_NOTE]: Start to trace and freeze model...

[VAIQ_NOTE]: The input model nndct_st_ResNet_ed is torch.nn.Module.

[VAIQ_NOTE]: Finish tracing.

[VAIQ_NOTE]: Processing ops...

| 180/180 [00:00<00:00, 336.56it/s, OpInfo: name = return_0, type = Return]

[VAIQ_NOTE]: =>Quantizable module is generated.(.vai_qat/ResNet.py)

[VAIQ_NOTE]: =>Exporting quant config.(.vai_qat/quant_info.json)

[VAIQ_NOTE]: Start to trace and freeze model...

[VAIQ_NOTE]: The input model nndct_st_ResNet_ed is torch.nn.Module.

[VAIQ_NOTE]: Finish tracing.

[VAIQ_NOTE]: Processing ops...

| 180/180 [00:00<00:00, 367.69it/s, OpInfo: name = return_0, type = Return]
Adjust lr at epoch 0, step 0: group_name=quantizer, lr=0.01
Adjust lr at epoch 0, step 0: group_name=weight, lr=1e-05
Epoch[0], Step: [0/60] Time 13.206 (13.206) Data 0.002 (0.002) Loss 1.9782e+00 (1.9782e+00) Acc@1 0.00 (0.00) Acc@5 100.00 (100.00)
Test: [0/20] Time 1.866 (1.866) Loss 2.3719e+00 (2.3719e+00) Acc@1 0.00 (0.00) Acc@5 0.00 (0.00)
* Acc@1 10.000 Acc@5 55.000
Saving best ckpt to ./qat_models/model_best_10.000.pth, acc1: 10.0
Epoch[0], Step: [10/60] Time 3.790 (5.065) Data 0.000 (0.000) Loss 2.5787e+00 (2.4378e+00) Acc@1 0.00 (18.18) Acc@5 100.00 (45.45)
Test: [0/20] Time 2.047 (2.047) Loss 2.5780e+00 (2.5780e+00) Acc@1 0.00 (0.00) Acc@5 0.00 (0.00)
* Acc@1 10.000 Acc@5 50.000
Epoch[1], Step: [20/60] Time 17.291 (5.125) Data 0.000 (0.000) Loss 2.6173e+00 (2.4529e+00) Acc@1 0.00 (9.52) Acc@5 0.00 (42.86)
Test: [0/20] Time 2.601 (2.601) Loss 2.5049e+00 (2.5049e+00) Acc@1 0.00 (0.00) Acc@5 0.00 (0.00)
* Acc@1 0.000 Acc@5 50.000
Epoch[1], Step: [30/60] Time 3.879 (4.802) Data 0.000 (0.000) Loss 1.9720e+00 (2.4117e+00) Acc@1 0.00 (9.68) Acc@5 100.00 (45.16)
Test: [0/20] Time 1.310 (1.310) Loss 2.5032e+00 (2.5032e+00) Acc@1 0.00 (0.00) Acc@5 0.00 (0.00)
* Acc@1 10.000 Acc@5 50.000
Epoch[2], Step: [40/60] Time 6.136 (4.617) Data 0.000 (0.000) Loss 1.9370e+00 (2.4074e+00) Acc@1 0.00 (9.76) Acc@5 100.00 (43.90)

Test: [0/20] Time 3.606 (3.606) Loss 2.6346e+00 (2.6346e+00) Acc@1 0.00 (0.00) Acc@5 0.00 (0.00)
 * Acc@1 10.000 Acc@5 55.000
Epoch[2], Step: [50/60] Time 3.795 (4.460) Data 0.000 (0.000) Loss 2.4791e+00 (2.4131e+00) Acc@1 0.00 (7.84) Acc@5 0.00 (41.18)
Test: [0/20] Time 1.310 (1.310) Loss 2.6042e+00 (2.6042e+00) Acc@1 0.00 (0.00) Acc@5 0.00 (0.00)
 * Acc@1 10.000 Acc@5 55.000

[VAIQ_NOTE]: Quant config file is empty, use default quant configuration

[VAIQ_NOTE]: Quantization test process start up...

[VAIQ_NOTE]: =>Quant Module is in 'cpu'.

[VAIQ_NOTE]: =>Parsing ResNet...

[VAIQ_NOTE]: Start to trace and freeze model...

[VAIQ_NOTE]: The input model nndct_st_ResNet_ed is torch.nn.Module.

[VAIQ_NOTE]: Finish tracing.

[VAIQ_NOTE]: Processing ops...

| 180/180 [00:00<00:00, 245.94it/s, OpInfo: name = return_0, type = Return]

[VAIQ_NOTE]: =>Quantizable module is generated.(.vai_qat/ResNet.py)

[VAIQ_NOTE]: =>Exporting quant config.(qat_result/test/quant_info.json)

[VAIQ_NOTE]: =>Exporting quant config.(qat_result/quant_info.json)
I0717 04:59:16.760449 quant_aware_training.py] Saving deployable model to qat_result/deployable.pth, and you can get it by calling "deployable_model()"
Test: [0/20] Time 2.281 (2.281) Loss 2.3719e+00 (2.3719e+00) Acc@1 0.00 (0.00) Acc@5 0.00 (0.00)
 * Acc@1 20.000 Acc@5 50.000

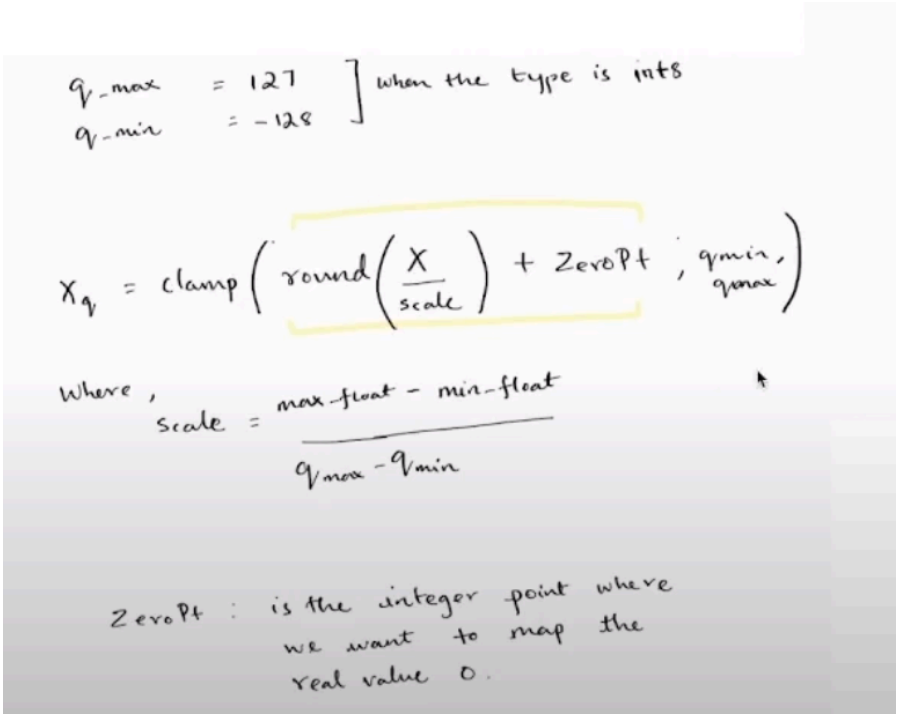
1. Quantization in NN | Affine and Symmetric Quantization

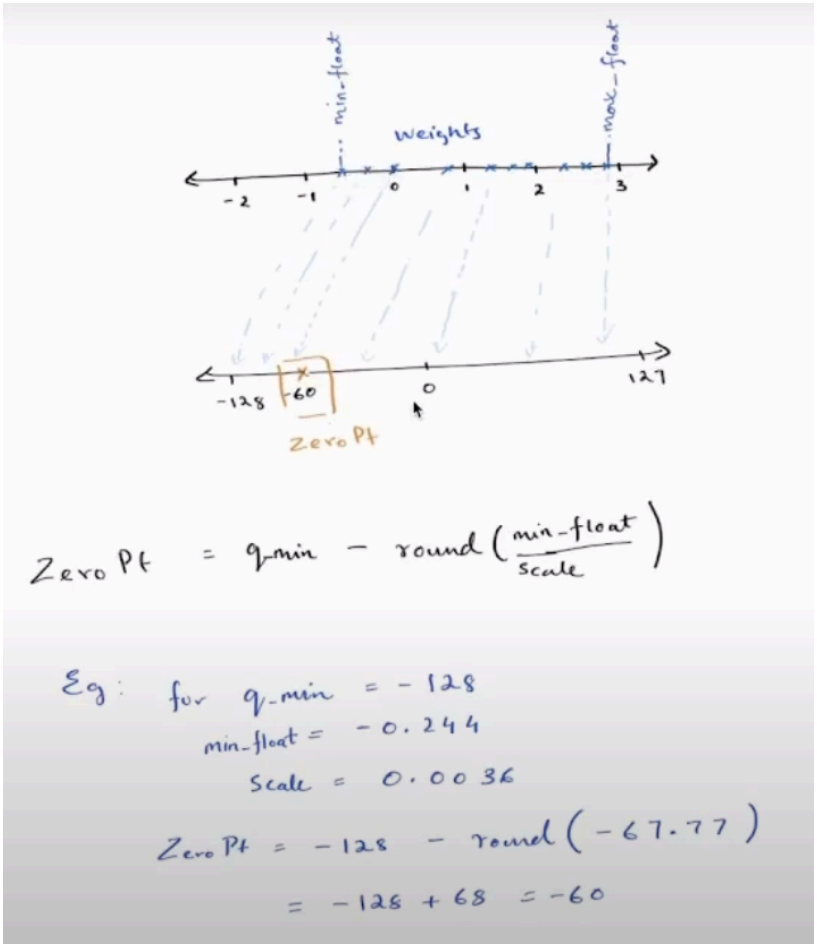
Float 32 → Quantization → Int8
The goal is to map the real values in the range [min_float, max_float] to the integers in [-2^b-1, (2^b-1) - 1] → [-128, 127] when b=8 for Int8.

How mapping is done?

- Uniform Affine Quantization
- Uniform Symmetric Quantization

Uniform Affine Quantization





$$\text{Zero Pt} = q\_min - \text{round}\left(\frac{min\_float}{scale}\right)$$

Eg: for $q\_min = -128$

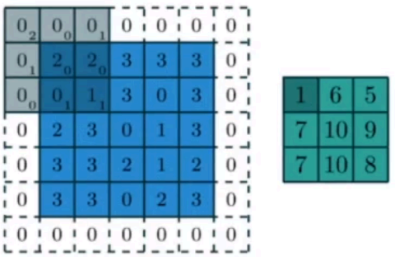
$min\_float = -0.244$

$scale = 0.0036$

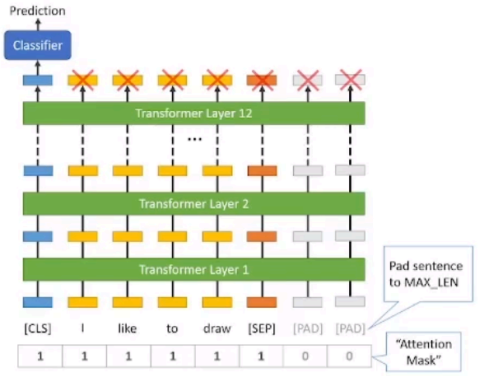
$$\text{Zero Pt} = -128 - \text{round}(-67.77)$$

$$= -128 + 68 = -60$$

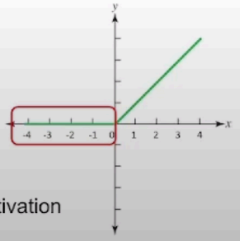
Why expressing zero correctly is important ?



Padding in CNNs

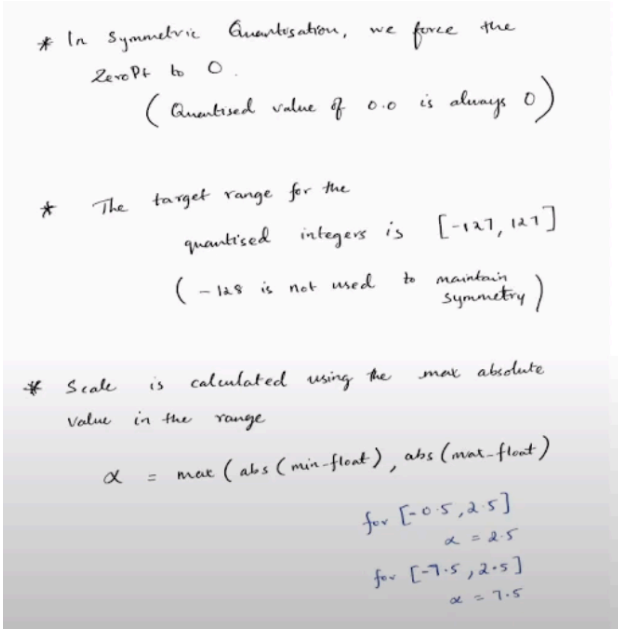


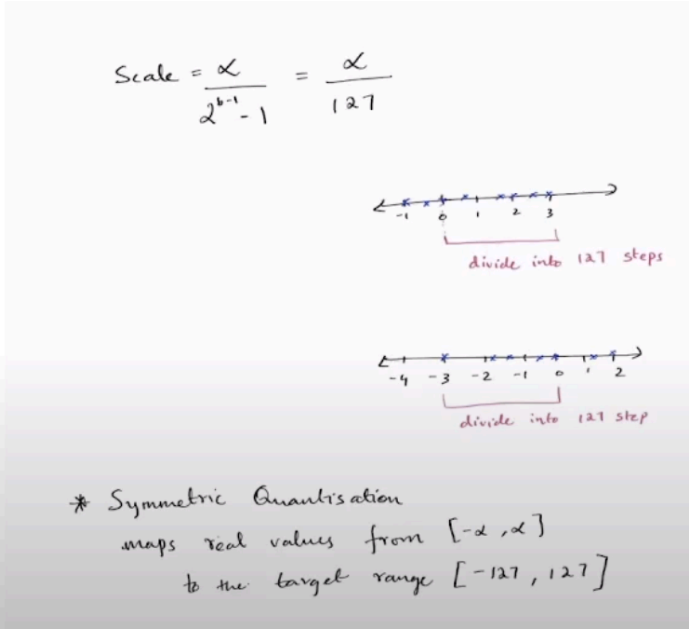
Attention mask in transformers



RELU activation

Uniform Symmetric Quantization





Quantization Supported in PyTorch

- Dynamic Quantization
- Post-Training Static Quantization
- Quantization Aware Training

Dynamic Quantization

- Weights converted to int8 offline.
- Activations are also converted to int8 dynamically just before the matrix computation.
- The computations will thus be performed using efficient int8 matrix multiplication and convolution implementations, resulting in faster compute.
- After the computation is done, the results are de-quantised back to floats

```
import torch
quantized_model = torch.quantization.quantize_dynamic(model, {torch.nn.Linear},
dtype=torch.qint8)
```

*By activations, it meant the inputs that get matrix multiplied with the weight matrix in current layer

Post-Training Static Quantization

- Weights converted to int8 offline.
- Static quantization performs the additional step of first feeding batches of data through the network and computing the resulting distributions of the different activations.
- This information is used to determine how specifically the different activations should be quantized at inference time

Quantization Aware Training

- this results in the highest accuracy of the three techniques.
- all weights and activations are “fake quantized” during both the forward and backward passes of training - which means float values are rounded to mimic int8 values, but all computations are still done with floating point numbers
- Thus, all the weight adjustments during training are made while “aware” of the fact that the model will ultimately be quantized;

2. Deep Dive on PyTorch Quantization - Chris Gottbrath

Motivation - While models are becoming more efficient, high accuracy still implies high complexity.

Quantized Tensors

To quantize:

- torch.quantize_per_tensor : Same scale factor for entire tensor
- torch.quantize_per_channel: Scale factors for each one of the channels

To operate on quantized tensors:

- torch.ops.quantized.* : Conv2d, Linear, etc

Higher Level operations:

- torch.nn.quantized.*
- torch.nn.quantized.dynamic.*

Contains sort of analogous of lot of routines that you're used to using in PyTorch to create your models.

- torch.quantization.Observer : For PTQ which extensively observes the activations
- torch.quantization.FakeQuantize : For QAT, to simulate the effect of quantization

Workflows

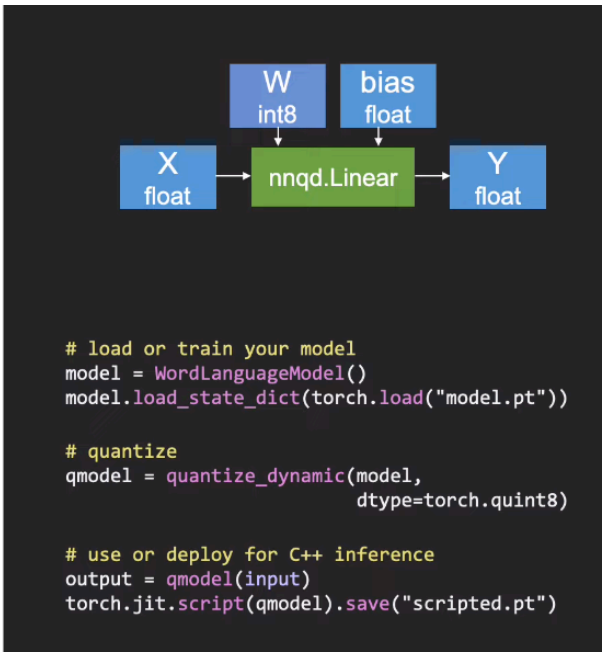
WORKFLOWS

	Quantization	Dataset Requirements	Works Best For	Accuracy	Notes
Dynamic Quantization	weights only (both fp16 and int8)	None	LSTMs, MLPs, Transformers	good	Suitable for dynamic models (LSTMs), Close to static post training quant when performance is compute bound or memory bound due to weights.
Static Post Training Quantization	weights and activations (8 bit)	calibration	CNNs	good	Suitable for static models, provides best perf
Static Quantization-Aware Training	weights and activations (8 bit)	fine-tuning	CNNs	best	Requires fine tuning of model, currently supported only for static quantization.

Workflow: Post Training Dynamic Quantization

WORKFLOW:
POST TRAINING DYNAMIC
QUANTIZATION

How: **tweak model**, one line API
What: quantize weights once, activations at runtime
Good for: LSTMs/Transformers and MLPs with small batch size
Savings: 2x faster compute, 4x less memory

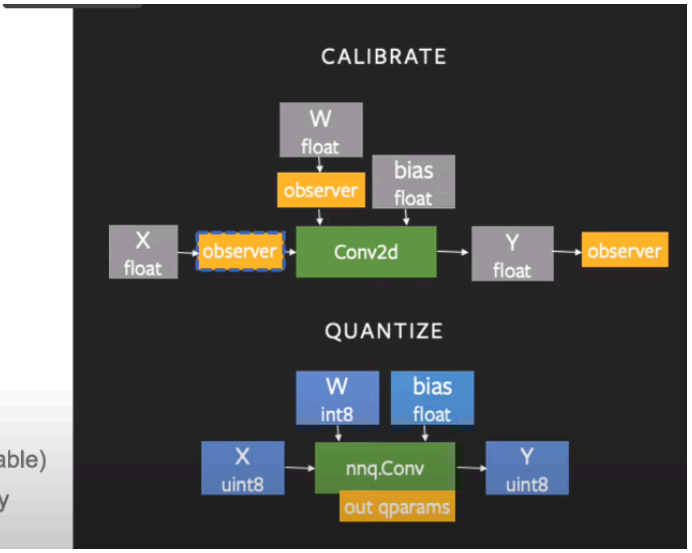



```
# load or train your model
model = WordLanguageModel()
model.load_state_dict(torch.load("model.pt"))

# quantize
qmodel = quantize_dynamic(model,
                          dtype=torch.quint8)

# use or deploy for C++ inference
output = qmodel(input)
torch.jit.script(qmodel).save("scripted.pt")
```

Workflow: Post Training Static Quantization

WORKFLOW:
POST TRAINING STATIC
QUANTIZATION

How: tweak model, calibrate on data, convert
What: quantize weight and activations for entire model or submodules
Good for: CNNs (if the accuracy drop is acceptable)
Savings: 1.5-2x faster compute, 4x less memory



1. Modify model

```
# load or train your model
model = ResNet50()
model.load_state_dict(torch.load("model.pt"))

# tweak model for best results
# change code directly or use manipulation APIs
model.eval()
model = quantization.fuse_modules(model,
  [["conv1", "bn1", "relu1"]])
```

*Fuse operations prior to quantization for performance and accuracy benefits

2. Prepare and calibrate

```
# specify which part to quantize and how
model.qconfig =
quantization.get_default_qconfig('fbgemm')
# configurable!

qmodel = quantization.prepare(model,
inplace=False)

# collect calibration statistics
qmodel.eval()
for batch, target in data_loader:
        model(batch)

print(model.conv1)

  ConvReLU2d(3, 64, kernel_size=(7, 7), ...
    (observer): MinMaxObserver(
      min_val=0.0, max_val=4.55)
  )
```

3. Convert

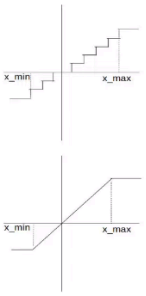
```
# get the quantized model
quantization.convert(qmodel)

print(qmodel.conv1)

  QuantizedConvReLU2d(3, 64,
    scale=0.035, zero_point=0,
    kernel_size=(7, 7), ...)
```

4. Deploy

```
# use or deploy for C++ inference
qmodel(input)
torch.jit.script(qmodel).save("quantized.pt")
```

Quantization Aware Training

- Emulate quantization by quantizing and de-quantizing in succession
  - Values are still in floating point, but with reduced precision
- $x\_out = FakeQuant(x)$
  $= DeQuant(Quant(x))$
  $= s * (Clamp(round(x/s) - z) + z)$



Fake Quantizer (top), showing the quantization of output values. Approximation for purposes of derivative calculation (bottom).

1. Modify model

```
# load or train your model
model = ResNet50()
model.load_state_dict(torch.load("model.pt"))

# tweak model for best results
# change code directly or use manipulation APIs
model.train()
model = quantization.fuse_modules(model,
  [["conv1", "bn1", "relu1"]])
```

2. Prepare and Train

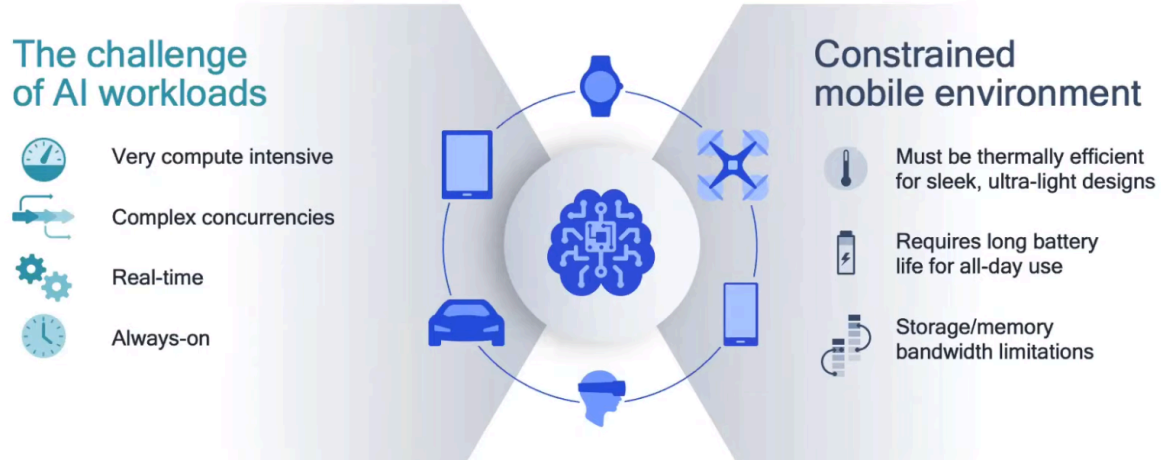
```
# specify which part to quantize and how
model.qconfig =
quantization.get_default_qat_qconfig('fbgemm')
qmodel = quantization.prepare_qat(model,
inplace=False)

# configurable
# fine tune model
train(qmodel, train_data)
```

```
(conv): ConvBnReLU2d(3, 3, kernel_size=(1, 1), stride=(1, 1), bias=False
    (activation_post_process): FakeQuantize(
      fake_quant_enabled=True, observer_enabled=True,
scale=tensor([0.0229]), zero_point=tensor([0])
      (activation_post_process): MovingAverageMinMaxObserver(min_val=0.0,
max_val=2.91)
    )
    (weight_fake_quant): FakeQuantize( fake_quant_enabled=True,
observer_enabled=True, scale=tensor([0.0029, 0.0022, 0.0027]),
zero_point=tensor([0, 0, 0])
      (activation_post_process):
MovingAveragePerChannelMinMaxObserver(min_val=tensor([-0.3693, -0.2504, -
0.2081]), max_val=tensor([-0.1531,  0.2852,  0.3477]))
    )
  )
  (bn): Identity()
```

*Rest of the workflow is identical to the post training quantization workflow

3. tinyML Talks: A Practical Guide to Neural Network Quantization

Introduction to NN Quantization

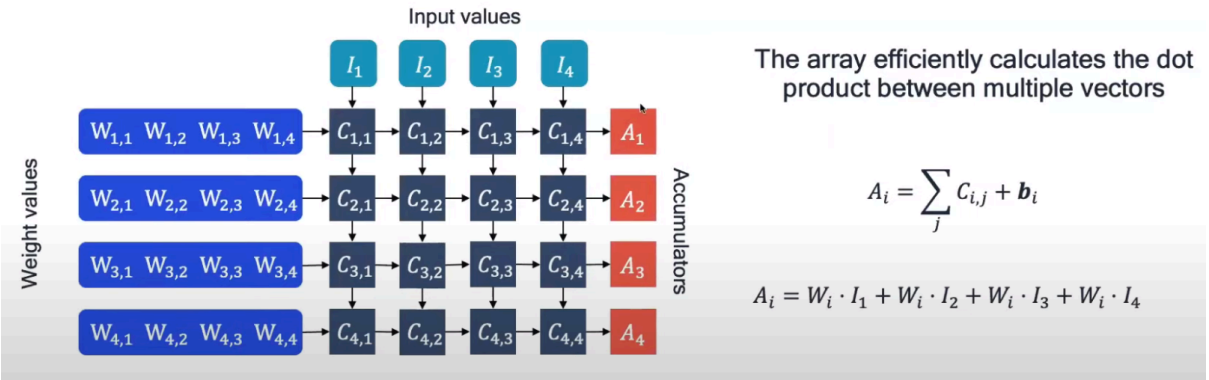


A White Paper on Neural Network Quantization:
2106.08295.pdf (arxiv.org)

Matrix operations are the backbone of neural networks

$$W = \begin{pmatrix} 0.97 & 0.64 & 0.74 & 1.00 \\ 0.58 & 0.84 & 0.84 & 0.81 \\ 0.00 & 0.18 & 0.90 & 0.28 \\ 0.57 & 0.96 & 0.80 & 0.81 \end{pmatrix} \quad X = \begin{pmatrix} 0.41 & 0.25 & 0.73 & 0.66 \\ 0.00 & 0.41 & 0.41 & 0.57 \\ 0.42 & 0.24 & 0.71 & 1.00 \\ 0.39 & 0.82 & 0.17 & 0.35 \end{pmatrix} \quad b = \begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \end{pmatrix}$$

How efficiently calculate WX + b ?



Quantization comes at a cost of lost precision

- We can approximate an FP tensor with an integer tensor multiplied by a scale-factor, s_X:

FP32 tensor → $\boldsymbol{X} \approx s_X \boldsymbol{X}_{\text{int}} = \widehat{\boldsymbol{X}}$ ← scaled quantized tensor

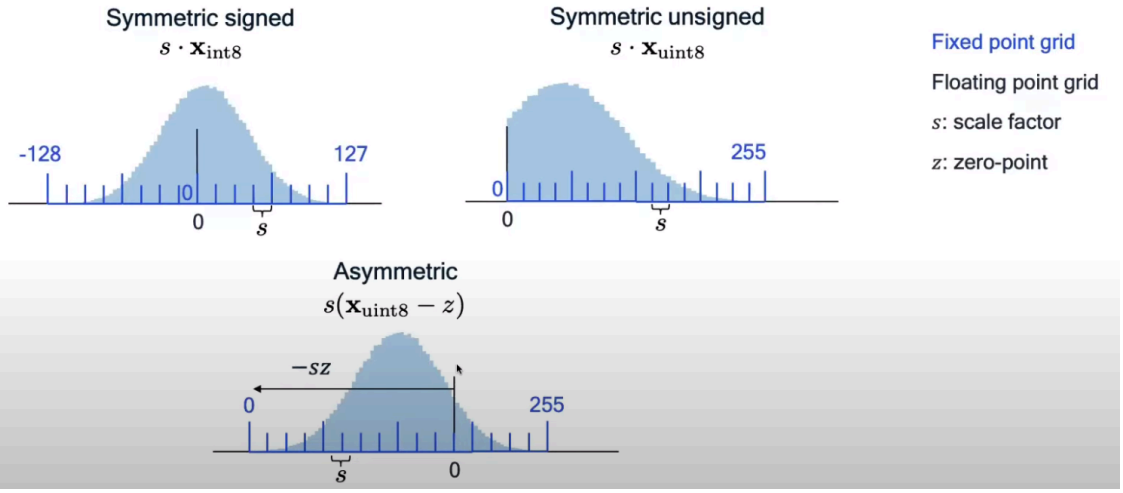
$$\boldsymbol{W} = \begin{pmatrix} 0.97 & 0.64 & 0.74 & 1.00 \\ 0.58 & 0.84 & 0.84 & 0.81 \\ 0.00 & 0.18 & 0.90 & 0.28 \\ 0.57 & 0.96 & 0.80 & 0.81 \end{pmatrix} \approx \frac{1}{255} \begin{pmatrix} 247 & 163 & 189 & 255 \\ 148 & 214 & 214 & 207 \\ 0 & 46 & 229 & 71 \\ 145 & 245 & 204 & 207 \end{pmatrix} = s_W \boldsymbol{W}_{\text{uint8}}$$

- Quantization is not free:

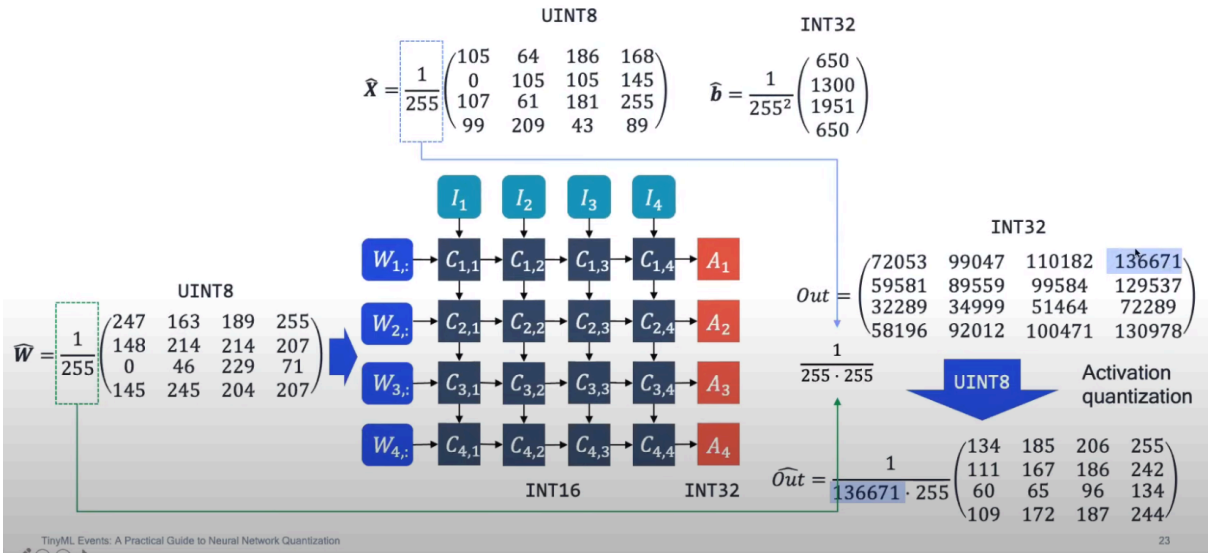
$$\boldsymbol{\epsilon} = \boldsymbol{W} - s_W \boldsymbol{W}_{\text{int}} = \frac{1}{255} \begin{pmatrix} 0.35 & 0.20 & -0.3 & 0 \\ -0.1 & 0.20 & 0.20 & -0.45 \\ 0.00 & -0.1 & -0.5 & 0.40 \\ 0.35 & -0.2 & 0 & -0.45 \end{pmatrix}$$

Different types of quantization have pros and cons

Symmetric, asymmetric, signed, and unsigned quantization



Quantized inference using symmetric quantization



What type of quantization should you use?

$\boldsymbol{W}$: weight matrix

$\boldsymbol{X}$: input of a layer

Symmetric quantization

$$\boldsymbol{WX} \approx s_W(\boldsymbol{W}_{\text{int}})\, s_X(\boldsymbol{X}_{\text{int}}) = s_W s_X(\boldsymbol{W}_{\text{int}}\, \boldsymbol{X}_{\text{int}})$$

Same calculation

Asymmetric quantization

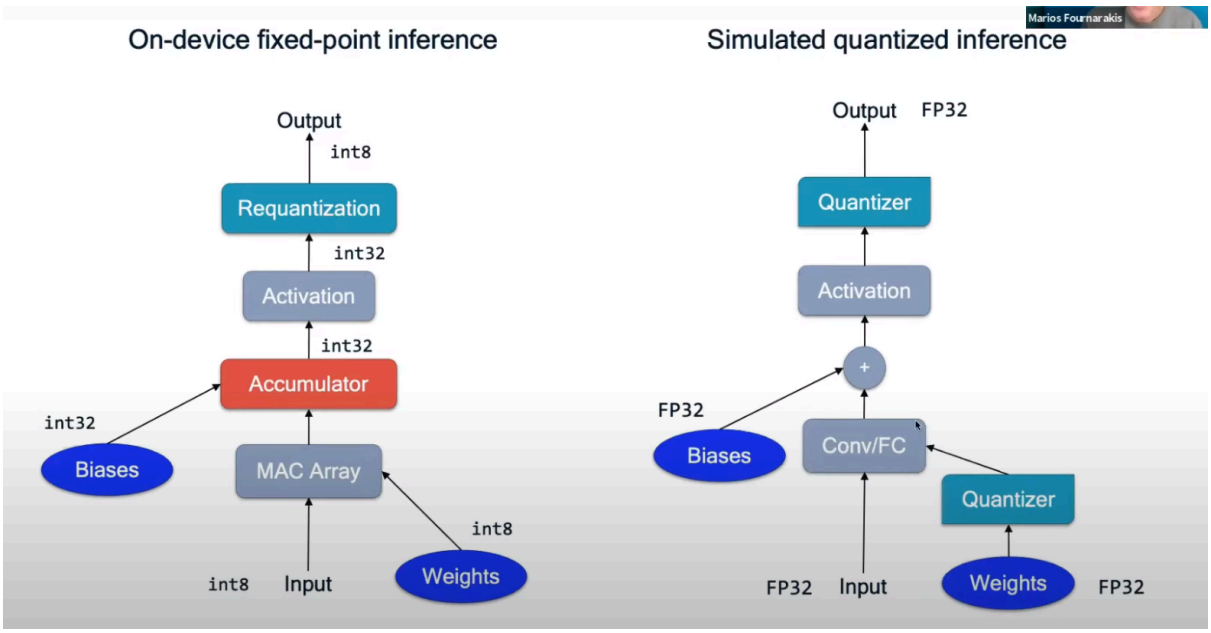
$$\boldsymbol{WX} \approx s_W(\boldsymbol{W}_{\text{int}} - z_W)\, s_X(X_{\text{int}} - z_X) = s_W s_X(\boldsymbol{W}_{\text{int}}\, \boldsymbol{X}_{\text{int}}) + s_W s_X z_X \boldsymbol{W}_{\text{int}} + s_W z_W s_X z_X + s_W s_X z_W \boldsymbol{X}_{\text{int}}$$

Precompute, add to layer bias

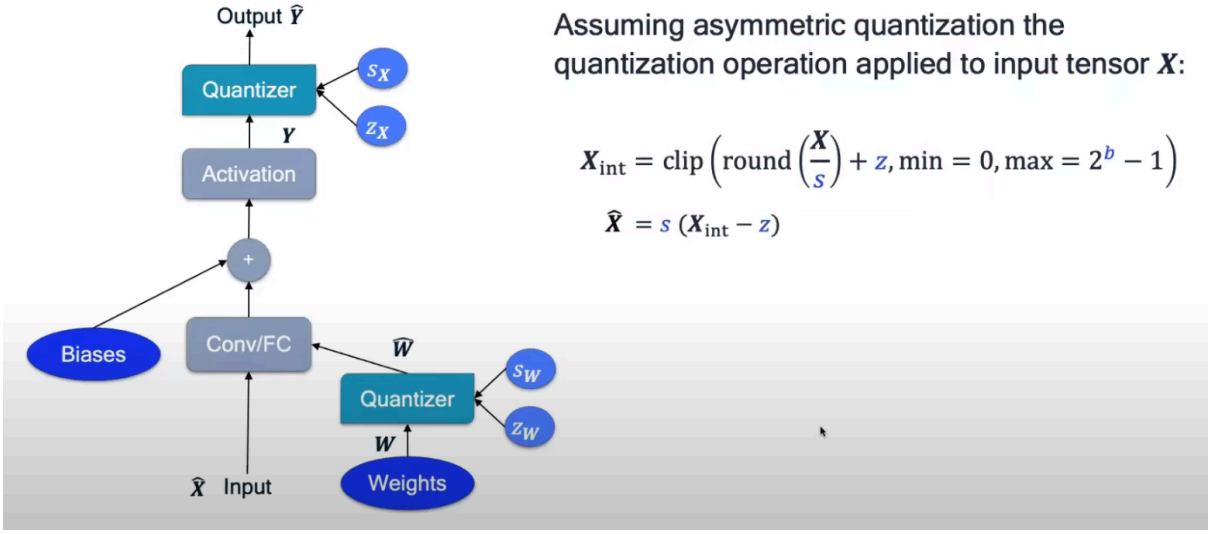
Data-dependent overhead

Simulating Quantization

- We simulate fixed-point operations with floating-point numbers using general purpose hardware (e.g. CPU, GPU)
- This simulation is achieved by introducing simulated quantization operations (quantizers) to the compute graph.
- Quantization simulation benefits:
  - Enables GPUs acceleration
  - No need for dedicated kernels
  - Test various quantization option and bit-widths



What operations do the quantizer perform?



Assuming asymmetric quantization the quantization operation applied to input tensor $\boldsymbol{X}$:

$$\boldsymbol{X}_{\text{int}} = \text{clip}\left(\text{round}\left(\frac{\boldsymbol{X}}{s}\right) + z, \text{min} = 0, \text{max} = 2^b - 1\right)$$

$$\hat{\boldsymbol{X}} = s\,(\boldsymbol{X}_{\text{int}} - z)$$

Example using $b = 4$:

$$X = \begin{pmatrix} 0.41 & 0.0 \\ 0.8 & -0.5 \end{pmatrix} \qquad s = \frac{1}{15} = 0.067$$

$$z = \text{round}\left(\frac{0.5}{0.067}\right) = 8$$

$$\text{round}\left(\frac{X}{s}\right) + z = \begin{pmatrix} 14 & 8 \\ 20 & 0 \end{pmatrix} \xrightarrow{\text{clip}} \begin{pmatrix} 14 & 8 \\ 15 & 0 \end{pmatrix}$$

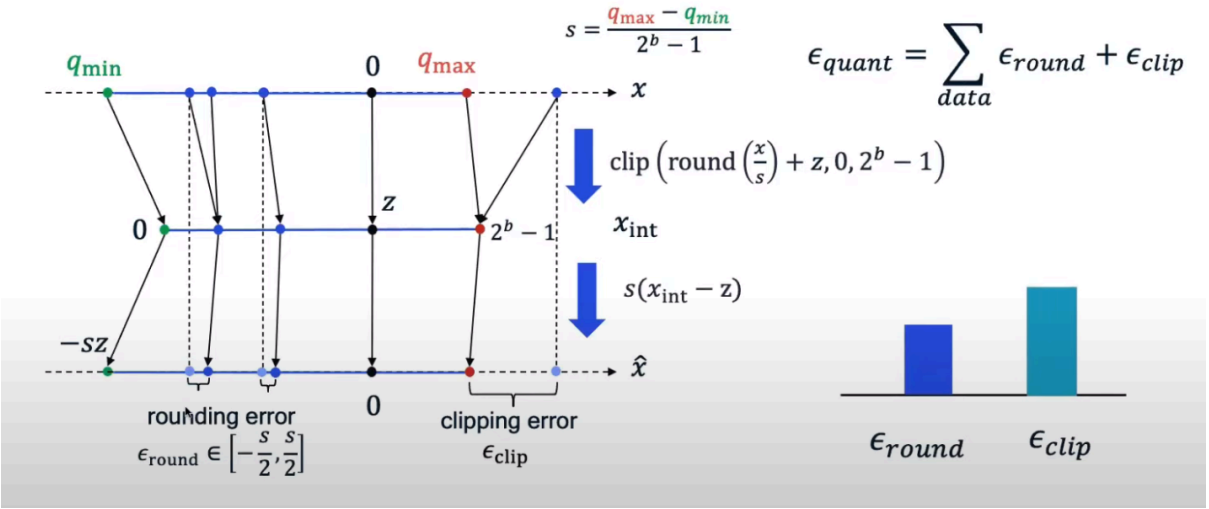
de-quantize

$$X = \begin{pmatrix} 0.41 & 0.0 \\ 0.8 & -0.5 \end{pmatrix} \qquad \hat{X} = \begin{pmatrix} 0.4 & 0.0 \\ 0.47 & -0.53 \end{pmatrix}$$

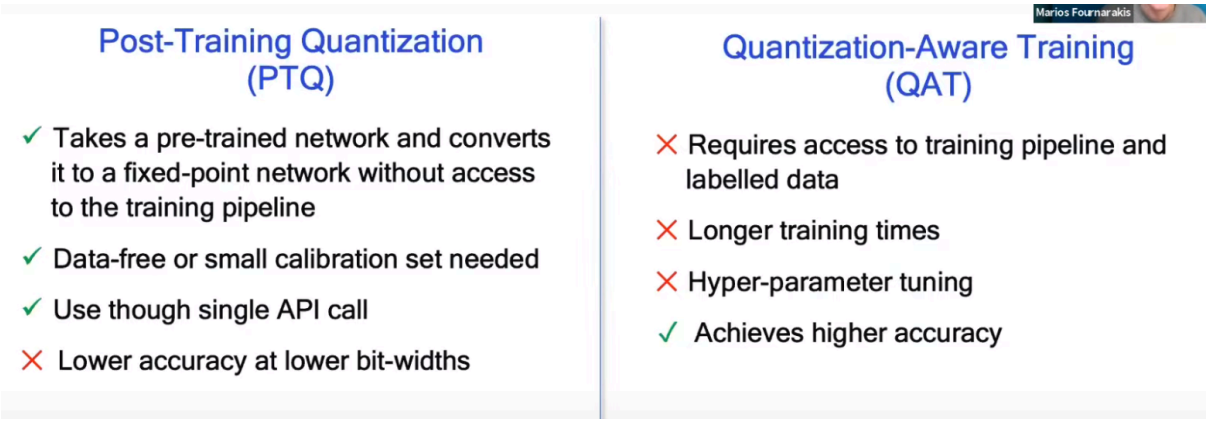
31

Choosing the quantization parameters

Sources of quantization error



PTQ vs. QAT



Chapter-4 “Compiling the Model”

compile.sh:

```
#!/bin/sh

# Copyright 2020 Xilinx Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#     http://www.apache.org/licenses/LICENSE-2.0
```

```
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# Author: Mark Harvey, Xilinx Inc

if [ $1 = zcu102 ]; then
      ARCH=/opt/vitis_ai/compiler/arch/DPUCZDX8G/ZCU102/arch.json
      TARGET=zcu102
      echo "-----------------------------------------"
      echo "COMPILING MODEL FOR ZCU102.."
      echo "-----------------------------------------"
elif [ $1 = zcu104 ]; then
      ARCH=/opt/vitis_ai/compiler/arch/DPUCZDX8G/ZCU104/arch.json
      TARGET=zcu104
      echo "-----------------------------------------"
      echo "COMPILING MODEL FOR ZCU104.."
      echo "-----------------------------------------"
elif [ $1 = vck190 ]; then
      ARCH=/opt/vitis_ai/compiler/arch/DPUCVDX8G/VCK190/arch.json
      TARGET=vck190
      echo "-----------------------------------------"
      echo "COMPILING MODEL FOR VCK190.."
      echo "-----------------------------------------"
elif [ $1 = u50 ]; then
      ARCH=/opt/vitis_ai/compiler/arch/DPUCAHX8H/U50/arch.json
      TARGET=u50
      echo "-----------------------------------------"
      echo "COMPILING MODEL FOR ALVEO U50.."
      echo "-----------------------------------------"
else
      echo  "Target not found. Valid choices are: zcu102, zcu104, vck190, u50 ..exiting"
      exit 1
fi
BUILD=$2
LOG=$3

compile() {
  vai_c_xir \
  --xmodel      ${BUILD}/quant_model/CNN_int.xmodel \
  --arch        $ARCH \
  --net_name    CNN_${TARGET} \
  --output_dir  ${BUILD}/compiled_model
}

compile 2>&1 | tee ${LOG}/compile_$TARGET.log
```

```
echo "----------------------------------------"
echo "MODEL COMPILED"
echo "----------------------------------------"
```

compile.sh explanation:
The `compile.sh` script provided is a shell script written in the Bourne shell (`/bin/sh`) language. It appears to be used for compiling a model targeting specific hardware platforms. Here's an explanation of the script:

1. License Information: The script begins with copyright and license information.

2. Conditional Statements: The script uses conditional statements to determine the target platform based on the first argument passed to the script (`$1`).

- If the first argument is `zcu102`, the script sets the `ARCH` variable to the corresponding architecture JSON file and the `TARGET` variable to `zcu102`. It then prints a message indicating that the model is being compiled for ZCU102.
- If the first argument is `zcu104`, the script sets the `ARCH` variable and the `TARGET` variable accordingly for ZCU104.
- If the first argument is `vck190`, the script sets the `ARCH` variable and the `TARGET` variable accordingly for VCK190.
- If the first argument is `u50`, the script sets the `ARCH` variable and the `TARGET` variable accordingly for Alveo U50.
- If the first argument is any other value, it prints an error message indicating that the target was not found and exits the script.

3. Variables:
- `BUILD`: The `BUILD` variable is set to the second argument passed to the script (`$2`), which represents the build directory path.
- `LOG`: The `LOG` variable is set to the third argument passed to the script (`$3`), which represents the log directory path.

4. Function: The script defines a function named `compile()` that contains the compilation logic.
- The `vai_c_xir` command is invoked with several parameters:
  - `--xmodel`: Specifies the path to the quantized model in XModel format.
  - `--arch`: Specifies the path to the architecture JSON file based on the target platform.
  - `--net_name`: Specifies the name of the compiled model.
  - `--output_dir`: Specifies the directory where the compiled model will be saved.

5. Compilation and Logging: The `compile()` function is called, and the output of the `compile` command is captured and redirected to both the console and a log file based on the `LOG` variable using the `tee` command.

6. Final Message: After the compilation is complete, a message is printed to indicate that the model has been compiled.

In summary, the `compile.sh` script determines the target platform based on the provided argument, sets the appropriate architecture and target variables, compiles a model using the specified target platform, and saves the compiled model in the designated directory. The script also logs the compilation process to a log file.

What does “vai_c_xir” command do?
The `vai_c_xir` command is a command-line tool provided by Xilinx Vitis AI (Artificial Intelligence) that is used for compiling and generating an executable representation of a quantized AI model. Here's a brief explanation of the command:

- `vai_c_xir`: This is the command-line tool for compiling an AI model represented in the Xilinx Intermediate Representation (XIR) format.

The XIR format is an intermediate representation used by Xilinx tools for AI model compilation and optimization. It represents the model in a hardware-agnostic format that can be targeted for various Xilinx platforms, such as FPGAs or AI Accelerator Cards.

The `vai_c_xir` command is typically used as part of the Xilinx Vitis AI workflow for compiling AI models. It takes several parameters to specify the input model, target platform, and output directory for the compiled model. The specific usage and available parameters of `vai_c_xir` depend on the version of Xilinx Vitis AI and the specific workflow being followed.

To get more detailed information about the usage and available options for the `vai_c_xir` command, you can refer to the official Xilinx Vitis AI documentation, user guides, or command-line tool reference manuals. These resources will provide comprehensive information on how to use the `vai_c_xir` command for compiling AI models in the XIR format.

Log:
[UNILOG][INFO] Compile mode: dpu
[UNILOG][INFO] Debug mode: null
[UNILOG][INFO] Target architecture: DPUCZDX8G_ISA1_B4096
[UNILOG][INFO] Graph name: CNN, with op num: 33
[UNILOG][INFO] Begin to compile...
[UNILOG][INFO] Total device subgraph number 3, DPU subgraph number 1
[UNILOG][INFO] Compile done.
[UNILOG][INFO] The meta json is saved to "/workspace/./build/compiled_model/meta.json"
[UNILOG][INFO] The compiled xmodel is saved to "/workspace/./build/compiled_model/CNN_zcu102.xmodel"
[UNILOG][INFO] The compiled xmodel's md5sum is c11489bf00498ab2f06400738b6bfb77, and has been saved to "/workspace/./build/compiled_model/md5sum.txt"
****
* VITIS_AI Compilation - Xilinx Inc.
****

Output:

- meta.json: information about the compiled model and the target

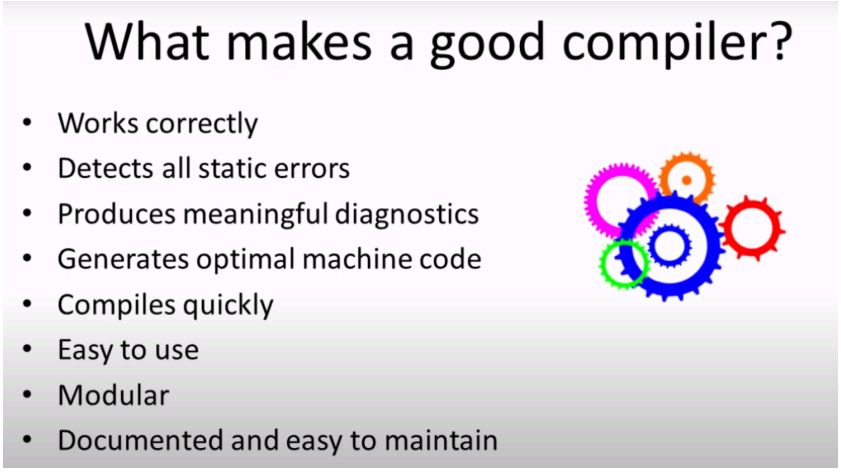
```
{
    "lib": "libvart-dpu-runner.so",
    "filename": "CNN_zcu102.xmodel",
    "kernel": [
        "subgraph_CNN__CNN_Sequential_network__Conv2d_0__ret_3"
    ],
    "target": "DPUCZDX8G_ISA1_B4096"
}
```

- Compiled xmodel: The model compiled for given hardware
- md5sum.txt: md5sum is used to verify the integrity of files, as virtually any change to a file will cause its MD5 hash to change. Most commonly, md5sum is used to verify that a file has not changed as a result of a faulty file transfer, a disk error or non-malicious meddling.

1. Compilation

Part One: Overview of the Stages of Compilation

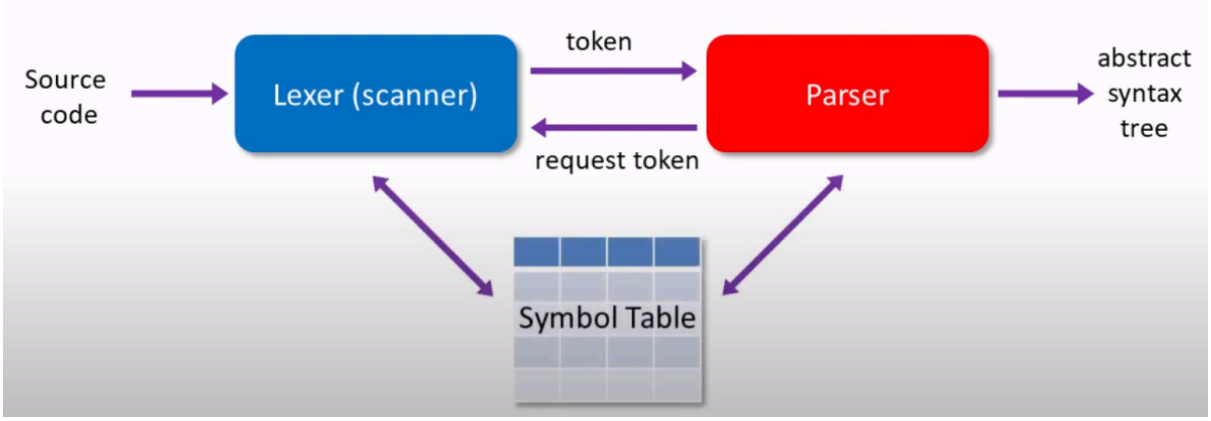
Introduction



Stages of Compilation

Lexical Analysis - Breaking a sentence into separate words and punctuation symbols
Syntax Analysis - Checking if the sentence has a meaning
Code Generation and Optimization - Translating into machine understandable format

Lexical Analysis



Abstract syntax tree - It is a dynamic data structure that represents the hierarchical nature of the source program. While it's being constructed the compiler uses it to check that the rules of the programming language have been followed.
*Convert Abstract Syntax ree → Low-level intermediate code → Machine code generation → Machine Code Optimization

Symbol Table - contains names a programmer has used in the source code eg. variables, functions

Part Two: Lexical Analysis (Scanner)

```
If Temperature <= 0 Then\n
  \tDescription = "Freezing"\n
End If
```

```
[Keyword:If]
[Identifier:Temperature]
[Relational_operator:<=]
[Number:0]
[Keyword:Then]
[Identifier:Description]
[Assignment_operator:=]
[Quote:"]
[String_literal:Freezing]
[Quote:"]
[Keyword:End If]
```

The job of **Lexical Analyzer** is to recognize and separate out the individual elements of the input string, breaking it up into the sub strings that are the individuals words of the program. The sub strings that the lexical analyzer is attempting to identify are known as **lexeme**.

- Very limited ability to detect the errors
- No knowledge of the order of source code
- No knowledge if anything is misspelled
- Just separate and create token and also create **symbol table**

Symbol Table

Lexeme	Token Class
If	Keyword
Temperature	Identifier
<=	Relational_operator
0	Number
Then	Keyword
Description	Identifier
=	Assignment_operator
"	Quote
Freezing	String_literal
"	Quote
End If	Keyword

- Lexical Analyzer can also create **String Table** such that:

```
If Temperature <= 0 Then\n
  \tDescription = "Freezing"\n
End If
```

```
[Keyword:1]
[Identifier:200]
[Relational_operator:<=]
[Number:0]
[Keyword:2]
[Identifier:201]
[Assignment_operator:=]
[Quote:"]
[String_literal:202]
[Quote:"]
[Keyword:4]
```

Strings Table

StringID	String
1	"If"
2	"Then"
3	"Else"
4	"End If"
5	"For"
6	"Next"
200	"Temperature"
201	"Description"
202	"Freezing"

Symbol Table

LexemeID	Token Class
1	Keyword
200	Identifier
<=	Relational_operator
0	Number
2	Keyword
201	Identifier
=	Assignment_operator
"	Quote
202	String_literal
"	Quote
4	Keyword

Part Three: Syntax Analysis (Parser)

Main Purpose → Checks Syntax Error or the rules of the language being followed correctly.

Syntax can be defined using → Context-Free Grammar

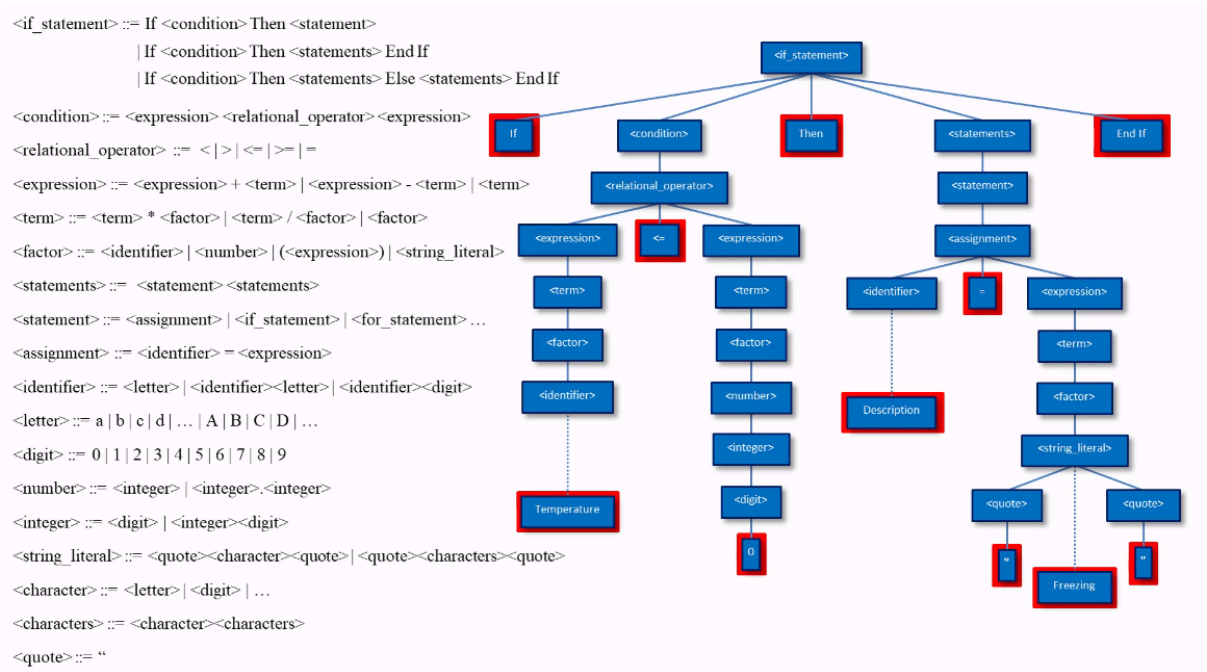
- **Backus–Naur form** or **Backus normal form (BNF)** is a metasyntax notation for **context-free grammars**, often used to describe the syntax of languages used in computing, such as computer programming languages

  For example:

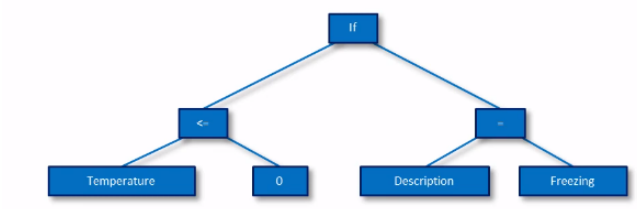
```
<if_statement> ::= If <condition> Then <statement>
                 | If <condition> Then <statements> End If
                 | If <condition> Then <statements> Else <statements> End If
```

These are the legal syntax of “if statements”. This “|” gives the productions valid for the “if statement”.

Syntax Analyzer - implement the relevant context-free grammar within the logic of its code as a set of recursive or hierarchically organized procedures.
Step 1: Create a Parse Tree
Step 2: Refine to create Abstarct Syntax tree

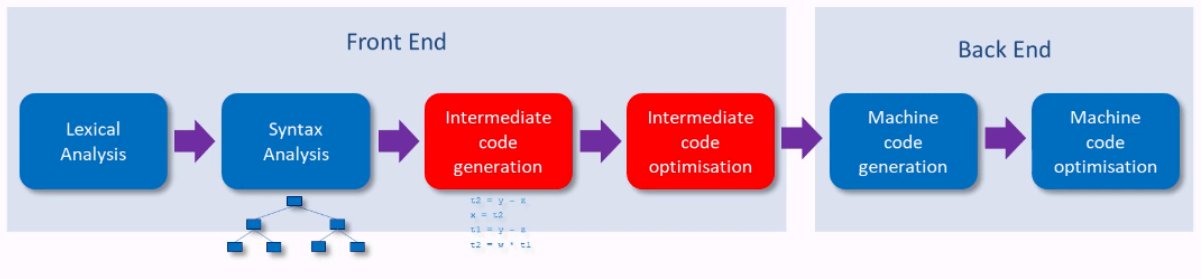


Output: Abstract Syntax Tree



- Abstract Syntax tree is traversed several times during semantic analysis.
- Symbol Table id frequently accessed and updated

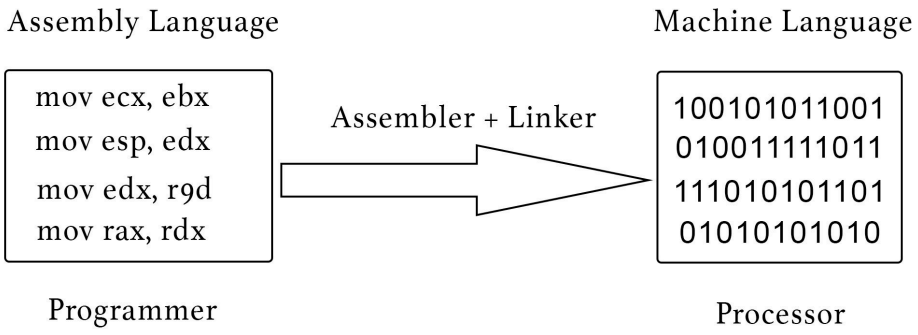
Part Four: Intermediate Code



Intermediate Code – Similar to assembly language, so brings compiler close to final goal i.e Machine code.

```
While a < b
     If c < d Then
          x =(y + z) * w
     Else
          x =(y - z) * w
     End If
End While
```

```
L1:  IfFalse a < b goto L3
     IfFalse c < d goto L2
     t1 = y + z
     t2 = w * t1
     x = t2
     goto L1
L2:  t2 = y - z
     x = t2
     t1 = y - z
     t2 = w * t1
     x = t2
     goto L1
L3:  ...
```



Prior to the final step of machine code generation, compiler might partition Three Address Code (Assembly Language) to Basic Blocks

Basic Blocks

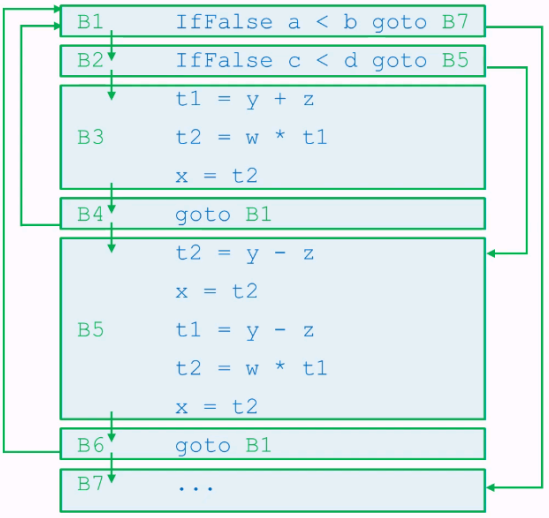
- Sequence of instructions that are always executed together
- Provide opportunities for machine independent optimisation
- Analyse and optimise flow control

```
While a < b
    If c < d Then
        x =(y + z) * w
    Else
        x =(y - z) * w
    End If
End While
```

```
L1:   IfFalse a < b goto L3
      IfFalse c < d goto L2
      t1 = y + z
      t2 = w * t1
      x = t2
      goto L1
L2:   t2 = y - z
      x = t2
      t1 = y - z
      t2 = w * t1
      x = t2
      goto L1
L3:   ...
```

Flow graph

```
While a < b
    If c < d Then
        x =(y + z) * w
    Else
        x =(y - z) * w
    End If
End While
```



Advantage - Faster Compilation - Code Optimization

Basic Block Optimization

- Changes the order of instructions to free up registers early
- Elimination of duplicate sub expressions
  Eg.

```
t = x + y
...
...
u = x + y
```

```
t = x + y
...
...
u = t
```

- Copy propagation and constant folding
- Elimination of dead code
- Algebraic transformations
  Eg.

```
t = x + 0      t = x
t = x - 0      t = x
t = x * 1      t = x
```

```
t = x²         t = x * x
t = x * 2      t = x + x
t = x/2        t = x * 0.5
t = x * 8      t = x << 3
```

- Flow of control between basic blocks

```
    goto L1                    goto L2
    ...                        ...
L1: goto L2                L1: goto L2

    If x < y goto L1           If x < y goto L2
    ...                        ...
L1: goto L2                L1: goto L2

    If d = 1 goto L1           If d <> 1 goto L2
    goto L2                    z = x + y
L1: z = x + y              L2: ...
L2: ...
```

Part Five: Object Code Generation

- Instruction selection
- Register allocation and assignment
- Instruction ordering
- Optimization

Indeed much of the work the compiler has to do depends very much on the processor architecture.

RISC
Reduced Instruction Set Computer

- Few instructions in the instruction set
- Three address instructions
- More instructions needed for a task
- One clock cycle per instruction
- Simpler chips are cheaper to produce
- Few addressing modes
- Many CPU registers

CISC
Complex Instruction Set Computer

- Many instructions in the instruction set
- Two address instructions
- Fewer instructions needed for a task
- Several clock cycles per instruction
- Complex chips cost more to produce
- Many addressing modes
- Few CPU registers

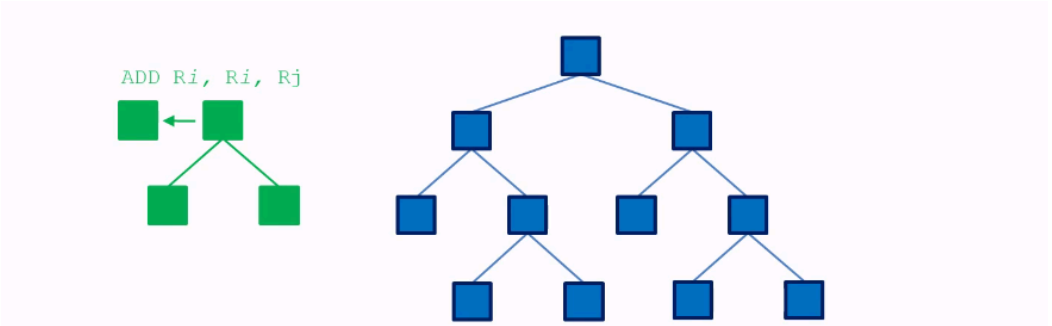
Instruction Selection

```
x = y + z
a = x + b
```

```
LD   R1, y
ADD  R1, R1, z
ST   x, R1
LD   R1, x
ADD  R1, R1, b
ST   a, R1
```

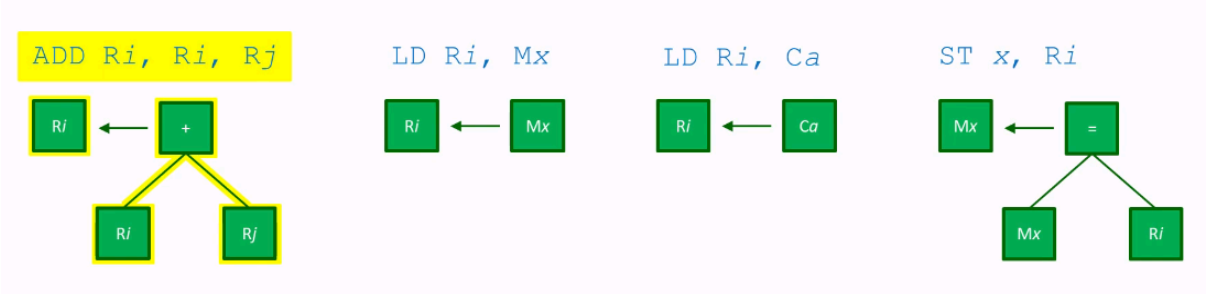
Remove such redundant instructions

Instruction Selection by Tree Rewriting



- Instruction set described by a set of tree rewriting rules
- Each rule is a simple tree template with a replacement node
- Each rule corresponds to a machine instruction
- Subtrees in the input tree are replaced according to the rules

Tree Translation Rules



Peephole Optimization

- Code examined through a sliding window
- Remove redundant identifiers
- Constant evaluation
- Remove unreachable sections of code
- Identify common sub-expressions
- Unfold loops
- Eliminating procedures

DIY

After calibration: quant_info.json


```
{
  "param":
  {
    "ResNet::conv1.weight":[[8,8]],
    "ResNet::conv1.bias":[[8,8]],
    "ResNet::layer1.0.conv1.weight":[[8,8]],
    "ResNet::layer1.0.conv1.bias":[[8,7]],
    "ResNet::layer1.0.conv2.weight":[[8,8]],
    "ResNet::layer1.0.conv2.bias":[[8,6]],
    "ResNet::layer1.0.conv3.weight":[[8,7]],
    "ResNet::layer1.0.conv3.bias":[[8,6]],
    "ResNet::layer1.0.identity_downsample.0.weight":[[8,7]],
    "ResNet::layer1.0.identity_downsample.0.bias":[[8,6]],
    "ResNet::layer1.1.conv1.weight":[[8,9]],
    "ResNet::layer1.1.conv1.bias":[[8,7]],
    "ResNet::layer1.1.conv2.weight":[[8,8]],
    "ResNet::layer1.1.conv2.bias":[[8,6]],
    "ResNet::layer1.1.conv3.weight":[[8,8]],
    "ResNet::layer1.1.conv3.bias":[[8,6]],
    "ResNet::layer1.2.conv1.weight":[[8,9]],
    "ResNet::layer1.2.conv1.bias":[[8,7]],
    "ResNet::layer1.2.conv2.weight":[[8,8]],
    "ResNet::layer1.2.conv2.bias":[[8,6]],
    "ResNet::layer1.2.conv3.weight":[[8,7]],
    "ResNet::layer1.2.conv3.bias":[[8,6]],
    "ResNet::layer2.0.conv1.weight":[[8,10]],
    "ResNet::layer2.0.conv1.bias":[[8,7]],
    "ResNet::layer2.0.conv2.weight":[[8,9]],
    "ResNet::layer2.0.conv2.bias":[[8,6]],
    "ResNet::layer2.0.conv3.weight":[[8,8]],
    "ResNet::layer2.0.conv3.bias":[[8,6]],
    "ResNet::layer2.0.identity_downsample.0.weight":[[8,10]],
    "ResNet::layer2.0.identity_downsample.0.bias":[[8,5]],
    "ResNet::layer2.1.conv1.weight":[[8,10]],
    "ResNet::layer2.1.conv1.bias":[[8,7]],
    "ResNet::layer2.1.conv2.weight":[[8,9]],
    "ResNet::layer2.1.conv2.bias":[[8,6]],
    "ResNet::layer2.1.conv3.weight":[[8,8]],
    "ResNet::layer2.1.conv3.bias":[[8,6]],
    "ResNet::layer2.2.conv1.weight":[[8,10]],
    "ResNet::layer2.2.conv1.bias":[[8,7]],
    "ResNet::layer2.2.conv2.weight":[[8,9]],
    "ResNet::layer2.2.conv2.bias":[[8,6]],
    "ResNet::layer2.2.conv3.weight":[[8,8]],
```

"ResNet::layer2.2.conv3.bias":[[8,6]],
"ResNet::layer2.3.conv1.weight":[[8,11]],
"ResNet::layer2.3.conv1.bias":[[8,7]],
"ResNet::layer2.3.conv2.weight":[[8,9]],
"ResNet::layer2.3.conv2.bias":[[8,7]],
"ResNet::layer2.3.conv3.weight":[[8,8]],
"ResNet::layer2.3.conv3.bias":[[8,6]],
"ResNet::layer3.0.conv1.weight":[[8,11]],
"ResNet::layer3.0.conv1.bias":[[8,7]],
"ResNet::layer3.0.conv2.weight":[[8,9]],
"ResNet::layer3.0.conv2.bias":[[8,7]],
"ResNet::layer3.0.conv3.weight":[[8,8]],
"ResNet::layer3.0.conv3.bias":[[8,6]],
"ResNet::layer3.0.identity_downsample.0.weight":[[8,10]],
"ResNet::layer3.0.identity_downsample.0.bias":[[8,6]],
"ResNet::layer3.1.conv1.weight":[[8,11]],
"ResNet::layer3.1.conv1.bias":[[8,7]],
"ResNet::layer3.1.conv2.weight":[[8,10]],
"ResNet::layer3.1.conv2.bias":[[8,7]],
"ResNet::layer3.1.conv3.weight":[[8,8]],
"ResNet::layer3.1.conv3.bias":[[8,6]],
"ResNet::layer3.2.conv1.weight":[[8,11]],
"ResNet::layer3.2.conv1.bias":[[8,7]],
"ResNet::layer3.2.conv2.weight":[[8,10]],
"ResNet::layer3.2.conv2.bias":[[8,7]],
"ResNet::layer3.2.conv3.weight":[[8,8]],
"ResNet::layer3.2.conv3.bias":[[8,6]],
"ResNet::layer3.3.conv1.weight":[[8,11]],
"ResNet::layer3.3.conv1.bias":[[8,6]],
"ResNet::layer3.3.conv2.weight":[[8,11]],
"ResNet::layer3.3.conv2.bias":[[8,6]],
"ResNet::layer3.3.conv3.weight":[[8,9]],
"ResNet::layer3.3.conv3.bias":[[8,6]],
"ResNet::layer3.4.conv1.weight":[[8,11]],
"ResNet::layer3.4.conv1.bias":[[8,6]],
"ResNet::layer3.4.conv2.weight":[[8,12]],
"ResNet::layer3.4.conv2.bias":[[8,7]],
"ResNet::layer3.4.conv3.weight":[[8,9]],
"ResNet::layer3.4.conv3.bias":[[8,6]],
"ResNet::layer3.5.conv1.weight":[[8,12]],
"ResNet::layer3.5.conv1.bias":[[8,6]],
"ResNet::layer3.5.conv2.weight":[[8,12]],
"ResNet::layer3.5.conv2.bias":[[8,7]],
"ResNet::layer3.5.conv3.weight":[[8,9]],
"ResNet::layer3.5.conv3.bias":[[8,6]],
"ResNet::layer4.0.conv1.weight":[[8,12]],
"ResNet::layer4.0.conv1.bias":[[8,6]],
"ResNet::layer4.0.conv2.weight":[[8,12]],

```
  "ResNet::layer4.0.conv2.bias":[[8,7]],
  "ResNet::layer4.0.conv3.weight":[[8,10]],
  "ResNet::layer4.0.conv3.bias":[[8,6]],
  "ResNet::layer4.0.identity_downsample.0.weight":[[8,12]],
  "ResNet::layer4.0.identity_downsample.0.bias":[[8,6]],
  "ResNet::layer4.1.conv1.weight":[[8,12]],
  "ResNet::layer4.1.conv1.bias":[[8,7]],
  "ResNet::layer4.1.conv2.weight":[[8,12]],
  "ResNet::layer4.1.conv2.bias":[[8,6]],
  "ResNet::layer4.1.conv3.weight":[[8,10]],
  "ResNet::layer4.1.conv3.bias":[[8,6]],
  "ResNet::layer4.2.conv1.weight":[[8,12]],
  "ResNet::layer4.2.conv1.bias":[[8,7]],
  "ResNet::layer4.2.conv2.weight":[[8,12]],
  "ResNet::layer4.2.conv2.bias":[[8,6]],
  "ResNet::layer4.2.conv3.weight":[[8,10]],
  "ResNet::layer4.2.conv3.bias":[[8,7]],
  "ResNet::fc.weight":[[8,12]],
  "ResNet::fc.bias":[[8,12]]
},
"output":
{
  "ResNet::nndct_input_0":[[8,6]],
  "ResNet::nndct_relu_2":[[8,5]],
  "ResNet::nndct_maxpool_3":[[8,5]],
  "ResNet::nndct_relu_5":[[8,5]],
  "ResNet::nndct_relu_7":[[8,4]],
  "ResNet::nndct_conv2d_8":[[8,4]],
  "ResNet::nndct_conv2d_9":[[8,5]],
  "ResNet::nndct_relu_11":[[8,4]],
  "ResNet::nndct_relu_13":[[8,4]],
  "ResNet::nndct_relu_15":[[8,3]],
  "ResNet::nndct_conv2d_16":[[8,3]],
  "ResNet::nndct_relu_18":[[8,3]],
  "ResNet::nndct_relu_20":[[8,3]],
  "ResNet::nndct_relu_22":[[8,2]],
  "ResNet::nndct_conv2d_23":[[8,3]],
  "ResNet::nndct_relu_25":[[8,2]],
  "ResNet::nndct_relu_27":[[8,3]],
  "ResNet::nndct_relu_29":[[8,2]],
  "ResNet::nndct_conv2d_30":[[8,3]],
  "ResNet::nndct_conv2d_31":[[8,3]],
  "ResNet::nndct_relu_33":[[8,2]],
  "ResNet::nndct_relu_35":[[8,3]],
  "ResNet::nndct_relu_37":[[8,2]],
  "ResNet::nndct_conv2d_38":[[8,3]],
  "ResNet::nndct_relu_40":[[8,2]],
  "ResNet::nndct_relu_42":[[8,3]],
```

```
  "ResNet::nndct_relu_44":[[8,3]],
  "ResNet::nndct_conv2d_45":[[8,3]],
  "ResNet::nndct_relu_47":[[8,2]],
  "ResNet::nndct_relu_49":[[8,3]],
  "ResNet::nndct_relu_51":[[8,2]],
  "ResNet::nndct_conv2d_52":[[8,3]],
  "ResNet::nndct_relu_54":[[8,2]],
  "ResNet::nndct_relu_56":[[8,3]],
  "ResNet::nndct_relu_58":[[8,2]],
  "ResNet::nndct_conv2d_59":[[8,3]],
  "ResNet::nndct_conv2d_60":[[8,3]],
  "ResNet::nndct_relu_62":[[8,2]],
  "ResNet::nndct_relu_64":[[8,3]],
  "ResNet::nndct_relu_66":[[8,2]],
  "ResNet::nndct_conv2d_67":[[8,2]],
  "ResNet::nndct_relu_69":[[8,2]],
  "ResNet::nndct_relu_71":[[8,2]],
  "ResNet::nndct_relu_73":[[8,2]],
  "ResNet::nndct_conv2d_74":[[8,2]],
  "ResNet::nndct_relu_76":[[8,1]],
  "ResNet::nndct_relu_78":[[8,2]],
  "ResNet::nndct_relu_80":[[8,1]],
  "ResNet::nndct_conv2d_81":[[8,1]],
  "ResNet::nndct_relu_83":[[8,0]],
  "ResNet::nndct_relu_85":[[8,1]],
  "ResNet::nndct_relu_87":[[8,1]],
  "ResNet::nndct_conv2d_88":[[8,0]],
  "ResNet::nndct_relu_90":[[8,0]],
  "ResNet::nndct_relu_92":[[8,0]],
  "ResNet::nndct_relu_94":[[8,0]],
  "ResNet::nndct_conv2d_95":[[8,0]],
  "ResNet::nndct_relu_97":[[8,-1]],
  "ResNet::nndct_relu_99":[[8,0]],
  "ResNet::nndct_relu_101":[[8,0]],
  "ResNet::nndct_conv2d_102":[[8,0]],
  "ResNet::nndct_conv2d_103":[[8,0]],
  "ResNet::nndct_relu_105":[[8,-1]],
  "ResNet::nndct_relu_107":[[8,0]],
  "ResNet::nndct_relu_109":[[8,-1]],
  "ResNet::nndct_conv2d_110":[[8,-1]],
  "ResNet::nndct_relu_112":[[8,-1]],
  "ResNet::nndct_relu_114":[[8,-1]],
  "ResNet::nndct_relu_116":[[8,-1]],
  "ResNet::nndct_conv2d_117":[[8,-1]],
  "ResNet::nndct_relu_119":[[8,-2]],
  "ResNet::nndct_adaptive_avg_pool2d_120":[[8,0]],
  "ResNet::nndct_dense_123":[[8,-2]]
},
```

```
  "input":
  {

  },
  "fast_finetuned":false,
  "bias_corrected":true,
  "version":"3.5.0+60df3f1+torch1.13.1",
  "graph_md5":"7429f15d324642a502fb93ab760dc1c2"
}
```


After test: quant_info.json