1 Introduction

Edge computing devices are becoming increasingly popular for running deep learning models in real-time, especially in applications like object detection, voice recognition, and recommendation systems. However, deploying high-precision models to edge devices can be computationally expensive and power-inefficient. To address this, dedicated hardware accelerators like the AMD Ryzen AI Neural Processing Unit (NPU) offer an efficient alternative.

The assignment provides practical exposure to machine learning deployment on specialized hardware and emphasizes the importance of optimization techniques like quantization to achieve power-efficient yet accurate inference at the edge. We use the Ryzen AI NPU to deploy and run inference of a pre-trained CNN model for the Fashion-MNIST dataset.

2 Theory

2.1 Neural Network Quantization

Quantization is the process of reducing the precision of the weights and activations in a neural network from floating-point (FP32) to lower-bit representations like int8. While this reduces computational load and memory footprint, it often comes at a cost of accuracy degradation.

Two major types of quantization:

- **Post-Training Quantization (PTQ):** Quantizes a fully trained model without re-training.
- **Quantization-Aware Training (QAT):** Simulates quantization effects during training to reduce accuracy loss.

In this assignment, PTQ is used with calibration data to map the range of activations.

2.2 AMD Ryzen AI NPU

The AMD Ryzen AI NPU is an integrated AI engine designed to accelerate AI workloads directly on Ryzen processors. It enables efficient inference of deep learning models while offloading the CPU and GPU, thereby improving system performance and reducing power consumption.

Key benefits:

- High throughput and low latency
- Reduced power consumption
- Seamless integration with AMD's software ecosystem (ONNX, Vitis AI EP)

2.3 ONNX Runtime and Vitis AI Execution Provider

ONNX (Open Neural Network Exchange) provides an open format to represent deep learning models. It allows interoperability between frameworks like PyTorch and TensorFlow. ONNX Runtime, combined with the Vitis AI Execution Provider, enables deployment of ONNX models on AMD hardware accelerators like the Ryzen AI NPU.

2.4 Ryzen AI Software Platform Architecture

The below diagram illustrates the software stack for deploying AI models on the Ryzen AI NPU.

- At the top, frameworks such as **TensorFlow**, **PyTorch**, and **ONNX** are used to develop and train models.
- In the next layer, **ONNX conversion, optimization, and quantization** tools like **Microsoft Olive** and **AMD Quark Quantizer** prepare the models for efficient inference.
- The execution provider layer contains runtime options:

  - **CPU EP**: CPU execution
  - **DirectML EP**: GPU-based execution
  - **AMD Vitis AI EP**: NPU acceleration

- Finally, models are executed on hardware: CPU, integrated GPU (iGPU), or NPU (Neural Processing Unit), making full use of Ryzen AI-enabled processors.

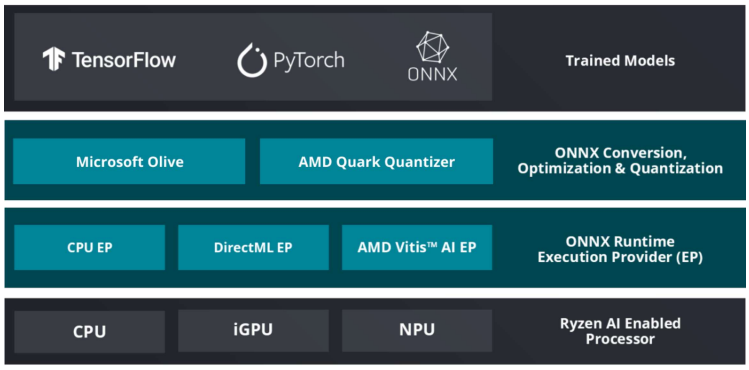

Figure 1: Ryzen AI Software Platform Architecture

This layered architecture ensures models move efficiently from training to hardware-accelerated deployment with minimal developer effort.

2.5 Assignment Workflow

The end-to-end workflow for this assignment consists of the following stages:

1. **Data Loading and Preparation:** Load the Fashion-MNIST dataset and prepare calibration data.
2. **Model Definition:** Use a pre-trained CNN architecture suitable for image classification.
3. **Model Export to ONNX:** Convert the PyTorch model to ONNX format.
4. **Quantization:** Apply post-training quantization to reduce precision from FP32 to int8.
5. **Deployment:** Run inference on the quantized model using the Ryzen AI NPU.
6. **Evaluation:** Measure accuracy and performance of the deployed model.

3 Implementation Details

In this assignment, several key modifications were made by inserting code into the provided "ADD CODE HERE" blocks. Below is a step-by-step outline of how each component is organized:

3.1 Understanding the Data

Fashion-MNIST is a dataset of 70,000 grayscale images (28x28), each belonging to one of 10 clothing categories (T-shirt/Top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle Boot). The official split is 60,000 training images and 10,000 test images.

Dataset Splitting

It is done to reserve a subset of the full training data for calibration during quantization.

Figure 2 shows the code block where the full training dataset. The calibration set is essential for calculating the activation ranges needed for effective quantization.

- The total number of training samples is reduced by 5,000 to form the calibration set.
- The `random_split` function ensures that both training and calibration sets are disjoint.

Split the full training set into a smaller 'train_set' and a 'calibration_set' ((e.g., 5,000 samples for calibration)

```
## == ADD CODE HERE ==

# split full training set into train_set and calibration_set
calib_sz = 5000  # calibration size
train_sz = len(full_train_set) - calib_sz  # training size

train_set, calib_set = random_split(full_train_set, [train_sz, calib_sz])  # split the dataset

print("training set size is", len(train_set))
print("calibration set size is", len(calib_set))

## ===================
```

```
training set size is 55000
calibration set size is 5000
```

Figure 2: Code block for splitting the dataset into training and calibration sets

Visualization:

Figure 3 shows a visualization of Fashion-MNIST classes, giving us an initial glimpse of the dataset diversity.

Figure 3: Sample visualization of Fashion-MNIST images

3.2 Building the CNN Model

`FashionCNN` class inherits from PyTorch's `nn.Module`. The network consists of:

- Two sequential convolutional layers, each followed by batch normalization, ReLU activation, and max pooling.

- A flattening step that converts the 2D feature maps into a 1D vector.
- Three fully connected (linear) layers. The first two are separated by a dropout layer (probability 0.25), and the final output layer has 10 units (one per class).

Hyperparameters and Loss Function:

- Learning rate: 0.001
- Optimizer: Adam
- Error Function: `CrossEntropyLoss`, which measures the difference between the predicted probability distribution and the true distribution.

3.3 Training (Read-Only)

The training code provided in the notebook is not meant to be run for this assignment. A pretrained model file (`onnx/fashion-mnist.pt`) is already available. Normally, it could be trained for several epochs, adjusting parameters such as the learning rate and monitoring validation accuracy to achieve the best results.

3.4 ONNX Model Export

We convert the pretrained PyTorch model into an ONNX(Open Neural Network Exchange) format, which allows us to perform quantization and later deploy the model on the AMD Ryzen AI NPU.

```
## == ADD CODE HERE ==

# export model to onnx format

# dummy input with proper shape (batch size 1, channel 1, 28x28 image)
dummy_input = torch.randn(1, 1, 28, 28)

# export model to onnx file
torch.onnx.export(
    model,                          # trained model in eval mode
    dummy_input,                    # input for tracing
    'onnx/fashion-mnist.onnx',      # onnx file path
    export_params=True,             # store trained weights in model file
    opset_version=13,               # set opset version
    input_names=['input'],          # input tensor nm
    output_names=['output'],        # output tensor nm
    dynamic_axes={'input': {0: 'batch_size'}, 'output': {0: 'batch_size'}}  # dynamic batch size
)

print("onnx model exported successfully!")

## ==================
```

```
onnx model exported successfully!
```

Figure 4: Exporting the PyTorch model to ONNX format

- **Dummy Input Creation:** A tensor of shape (1, 1, 28, 28) is created to mimic one sample from the Fashion-MNIST dataset. This dummy input ensures that the exported ONNX model correctly captures the model architecture and operations.
- **Model Tracing:** The model is executed with the dummy input to record its computation path. This step collects all the layers and operations of the network, which is necessary to generate the ONNX graph.
- **Export Settings:** The export call defines where to save the ONNX model (at `onnx/fashion-mnist.onnx`) and sets the opset version. Enabling dynamic axes allows the model to handle different batch sizes during inference.

3.5 Calibration Data Reader

Since we're performing post-training quantization (PTQ), we need a representative set of images to calculate activation ranges. `CalibrationDataReader` supplies the quantizer with batches from our 5,000-image calibration set.

Initialization: In the `__init__` method, a DataLoader is created using the calibration set (a subset of the full training data). The DataLoader is configured with:

- A specified batch size (default set to 1, but configurable),
- `shuffle` set to `False` to maintain a consistent data order,
- `drop_last` set to `True` to ensure only full batches are used.

Then an iterator created from this DataLoader serves as the data source for calibration during quantization.

```python
class FmnistCalibrationDataReader(CalibrationDataReader):
    def __init__(self, batch_size: int = 1):
        super().__init__()

        ## == ADD CODE HERE ==

        self.iterator = iter(torch.utils.data.DataLoader(
        calib_set,
        batch_size=batch_size,
        shuffle=False,
        drop_last=True
    )
)  # Add code here

        ## ==================

    def get_next(self) -> dict:
        try:
            images, labels = next(self.iterator)
            return {"input": images.numpy()}
        except Exception:
            return None

def fmnist_calibration_reader():
    return FmnistCalibrationDataReader()
```

Figure 5: Quantizing the ONNX model to int8 using AMD Quark

- **Data Feeding:** The class implements a `get_next` method that retrieves the next batch of data from the iterator. This method converts the batch (specifically, the images) into a NumPy array and wraps it in a dictionary with the key `"input"`. This format is required by the quantization tool.
- **Factory Function:** A helper function (`fmnist_calibration_reader`) instantiates the calibration data reader, making it convenient to supply calibration data to the quantizer.

3.6 Quantization Process

We converts the FP32 ONNX model into an INT8 quantized model using AMD Quark. Quantization reduces computation and memory usage while aiming to maintain acceptable accuracy.

```python
# Run the following cell to quantize and save the model:

## == ADD CODE HERE ==
# set input and output model paths
input_mdl_path = "onnx/fashion-mnist.onnx"
output_mdl_path = "onnx/fashion-mnist.qdq.U8S8.onnx"

# get default quantization config
quant_config = get_default_config("XINT8")
config = Config(global_quant_config=quant_config)

# create model quantizer
quantizer = ModelQuantizer(config)

# run quantization using calibration reader
quantizer.quantize_model(input_mdl_path, output_mdl_path, fmnist_calibration_reader())

print("quantized model saved at:", output_mdl_path)

## ==================
```

Op Type	Float Model
Conv	2
Relu	2
MaxPool	2
Shape	1
Constant	3
Gather	1
Unsqueeze	1
Concat	1
Reshape	1
Gemm	3
Quantized model path	onnx/fashion-mnist.qdq.U8S8.onnx

The quantized information for all operation types is shown in the table below.
The discrepancy between the operation types in the quantized model and the float model is du

Op Type	Activation	Weights	Bias
Conv	UINT8(2)	INT8(2)	INT8(2)
MaxPool	UINT8(2)		
Reshape	UINT8(1)		
Gemm	UINT8(3)	INT8(3)	INT8(3)

Figure 6: Quantization code block

- **Setting Input and Output Paths:** We define two paths, one for the original model file (“`onnx/fashion-mnist.onnx`”) and another for the new quantized file (“`onnx/fashion-mnist.qdq.U8S8.onnx`”).
- **Obtaining the Default Configuration:** A call to `get_default_config("XINT8")` provides a baseline quantization setup for 8-bit integer representation. This baseline is known to work reliably on a variety of models.
- **Customization of Quantization Parameters:** Certain parameters (e.g., `per_channel`, `reduce_range`, `calibrate_method`) were adjusted to try and improve accuracy. However, hardware constraints (enable_dpu) prevent enabling per-channel quantization and other changes didn't have any impact on acuuracy.
- **ModelQuantizer and Calibration:** After creating a `Config` object from these parameters, a `ModelQuantizer` instance is initialized. This tells Quark how to gather activation ranges if calibration data is required.

These configurations and the calibration process produce an INT8 model that is compatible with the AMD Ryzen AI NPU, enabling more efficient inference than the original FP32 version.

3.7 Deployment on NPU

Quantized model is loaded and ran on the AMD Ryzen AI NPU. By using the Vitis AI Execution Provider, the model can run on dedicated AI hardware rather than relying solely on the CPU or GPU.

```
## == ADD CODE HERE ==

# set environment variables for NPU setup
os.environ['XLNX_ENABLE_CACHE'] = '0'     # disable cache
print("XLNX_VART_FIRMWARE:", os.environ.get('XLNX_VART_FIRMWARE'))


quantized_model_path1 = r'onnx/fashion-mnist.qdq.U8S8.onnx'      # quantized onnx file path
model = onnx.load(quantized_model_path1)          # load onnx model

# providers and provider options for vitis ai ep
providers = ['VitisAIExecutionProvider']         # providers list
cache_dir = os.path.join(os.getcwd(), "onnx")    # cache directory
provider_options = [{
    'config_file': 'onnx/vaip_config.json',
    'cacheDir': str(cache_dir),
    'cacheKey': 'modelcachekey_lab4'
}]       # provider options

# onnx runtime inference session
session = ort.InferenceSession(quantized_model_path1, providers=providers, provider_options=provider_options)

print("model deployed on npu successfully!")


## ==================


XLNX_VART_FIRMWARE: C:\Program Files\RyzenAI\1.3.1\voe-4.0-win_amd64\xclbins\strix\AMD_AIE2P_Nx4_Overlay.xclbin
model deployed on npu successfully!
```

Figure 7: Deployment code block for running the quantized model on NPU

- **Environment Settings:** Two environment variables are set. `XLNX_ENABLE_CACHE`, is assigned a value of `0` to disable caching if needed. `XLNX_VART_FIRMWARE`, ensures the NPU uses the correct firmware binary (here it's simply referenced to confirm it is set).
- **Quantized Model Path:** A path to the quantized ONNX file (`onnx/fashion-mnist.qdq.U8S8.onnx`) is specified. This file contains the int8 weights and graph structure needed for hardware-accelerated inference.
- **Provider Configuration:** The `providers` list includes `VitisAIExecutionProvider` to indicate the NPU target. The `provider_options` dictionary references a `config_file` (`vaip_config.json`) and sets up a `cacheDir` and `cacheKey`. These options help manage how the model is compiled and cached for the NPU.
- **Inference Session Creation:** An `InferenceSession` is created with the quantized model path and the provider options. When the session is established, it compiles or optimizes the model for the NPU if necessary.

3.8 Inference on the Test Set:

- We feed images from the 10,000-image Fashion-MNIST test set into the deployed model.
- For each image, we retrieve the predicted class index, convert it to a class label (e.g., "Sandal," "Trouser," etc.), and compare it with the true label.

4 Results and Observations

4.1 Inference Results

Predictions on a Sample of 20 Images: Figure 8 presents the results of running inference on 20 test images. Each image is annotated with its predicted label, demonstrating the model's ability to classify various clothing items correctly.

Figure 8: Predictions on 20 test images from the Fashion-MNIST dataset

Misclassified Images: Figure 9 displays the images that were incorrectly classified. This visualization helps in identifying potential areas for improvement in the model, although the overall performance is within expected ranges for a post-training quantized model.

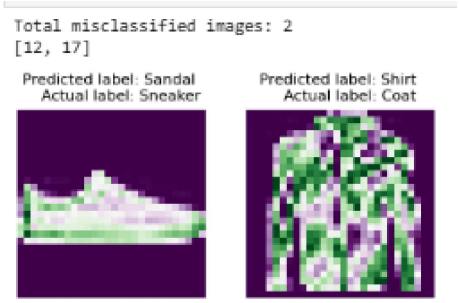

Figure 9: Visualization of misclassified test images

4.2 Accuracy Calculation and Evaluation

The quantized model was evaluated on the entire test set of 10,000 images. Accuracy is calculated using the formula:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \times 100\%$$

In our experiments, 8883 images were correctly classified and 1117 images were misclassified. Thus, the overall accuracy is:

$$\text{Accuracy} = \frac{8883}{10000} \times 100\% \approx 88.83\%$$

The overall accuracy of 88.83% is within the expected range for post-training quantization on a small dataset like Fashion-MNIST.

4.3 Confusion Matrix Analysis

Confusion matrix provides a deeper understanding of the model's classification behavior, indicating specific classes that may require further refinement or data augmentation to reduce errors. The horizontal axis indicates the predicted classes, while the vertical axis corresponds to the actual classes.

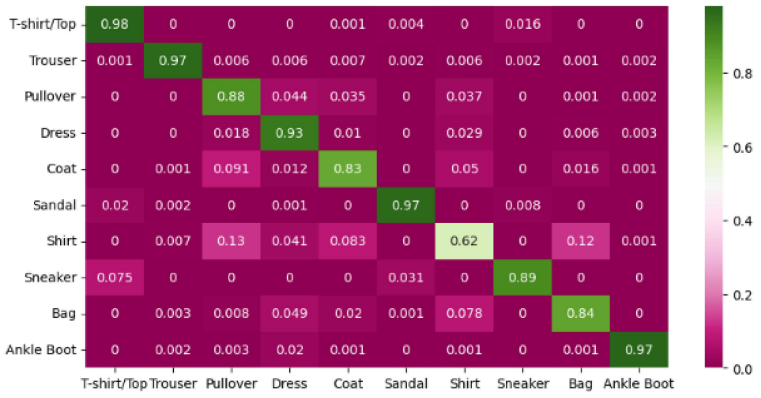

Figure 10: Confusion matrix for quantized model's predictions on Fashion-MNIST test set

- **True Positives (Diagonal):** Each diagonal cell shows how many images of a given class were correctly identified as that class.
- **Off-Diagonal Errors:** Misclassifications appear off the diagonal. For example, if the T-shirt/Top class has a 15% misclassification rate into the Pullover class, the corresponding cell's value will be 0.15.
- **Class-wise Insights:** Observing which rows have higher off-diagonal values reveals which classes the model struggles to distinguish. This can guide future enhancements, such as additional training epochs or more advanced quantization techniques.

4.4 Vitis AI EP Report Analysis

After quantizing and deploying the model on the AMD Ryzen AI NPU, the `vitisai_ep_report.json` file was examined to determine how the computational graph was partitioned between the NPU and the CPU.

- **Operator Partitioning:** The report shows that 36 nodes (e.g., Conv, Gemm, MaxPool) were successfully allocated to the NPU, while a few nodes (such as DequantizeLinear) remained on the CPU. This partitioning strategy shifts the most computationally intensive operations to specialized hardware.
- **Performance Insight:** Assigning the heaviest operations to the NPU improves inference speed, while simpler or unsupported operations are managed by the CPU, thereby reducing overall processing load.

- **Debug and Optimization:** The report provides valuable insights into whether all critical nodes were mapped to the NPU. If key operations remain on the CPU, further optimization or modifications to the model structure might be necessary for full acceleration.

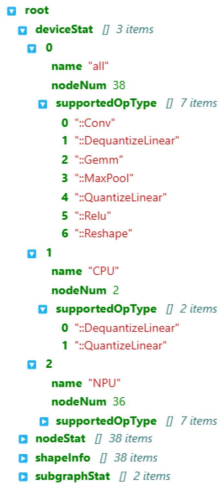

Figure 11: JSON report showing the partitioning of model nodes between the NPU and the CPU

Here, this balanced partitioning leverages the strengths of both hardware components for efficient execution.

5 Conclusion

This assignment demonstrates a complete process for deploying a quantized CNN model on the AMD Ryzen AI NPU. The workflow included preparing a calibration set from the Fashion-MNIST training data, exporting a pretrained PyTorch model to ONNX, and performing post-training quantization using AMD Quark. The quantized model was then deployed on the NPU using ONNX Runtime with the Vitis AI Execution Provider, achieving an accuracy of 88.83% on the test set.

The approach maintained the model's structure and allowed dynamic batch sizing during export. Due to hardware constraints, only per-tensor quantization was used, so advanced techniques like per-channel quantization or quantization-aware training were not applied. The Vitis AI report confirmed that 36 nodes were executed on the NPU, which helped speed up inference.

So, this work shows an effective and practical way to deploy deep learning models on specialized hardware. The achieved results are in line with expectations for post-training quantization on Fashion-MNIST, providing a strong foundation for future improvements in edge AI applications.

6 Future Work

Future work could focus on several improvements to enhance model performance. For example, using per-channel quantization may help preserve more detail in the convolution layers. Exploring alternative calibration methods like entropy and applying quantization-aware training could also reduce the drop in accuracy. Additionally, increasing the diversity of the calibration data and extending the training epochs may further boost the model's robustness.

7 References

[1] AMD Quark Quantizer Documentation (`https://quark.docs.amd.com/latest/`)

[2] ONNX Runtime Documentation (`https://onnxruntime.ai/`)

[3] Fashion-MNIST Dataset (`https://github.com/zalandoresearch/fashion-mnist`)

[4] Vitis AI Execution Provider (https://www.xilinx.com/products/design-tools/vitis.html)

[5] AMD Ryzen AI Model Run Documentation (https://ryzenai.docs.amd.com/en/latest/modelrun.html)

[6] 1_pytorch_onnx_inference-NPU.ipynb (provided in the assignment repository)

[7] 2_pytorch_onnx_re-train.ipynb (provided in the assignment repository)