# Quant_distilBERT_forward-pass

March 2, 2025

[1]:
```
from pynq import Overlay
overlay = Overlay("/home/ubuntu/workspace/pynq_bitfiles/2-28/MatMul_SA10.bit")
accel_ip = overlay.mmult_accel_0
```

## 1 Core Function

- `call_fpga()`: Handles memory management and parameter configuration for the hardware accelerator

[2]:
```
def call_fpga(A_buf, B_buf, C_buf, accel_ip, N, K, M, update_A):
    """
    Runs a 2D matrix multiplication on the FPGA accelerator:
      (N, K) x (K, M) => (N, M)

    A_buf, B_buf, C_buf are PYNQ buffers allocated with shape=(N,K), (K,M),␣
 ↪(N,M).
    update_A: 1 to load A into BRAM (new input), 0 to reuse persistent A.
    """
    # print("calling fpga, update_A =", update_A)

    # Flush input buffers to ensure data consistency.
    # Only flush A_buf if we intend to update A (update_A==1).
    if update_A:
        A_buf.flush()
    B_buf.flush()

    # Configure the accelerator registers
    accel_ip.register_map.A_1 = A_buf.physical_address & 0xFFFFFFFF
    accel_ip.register_map.A_2 = (A_buf.physical_address >> 32) & 0xFFFFFFFF
    accel_ip.register_map.B_1 = B_buf.physical_address & 0xFFFFFFFF
    accel_ip.register_map.B_2 = (B_buf.physical_address >> 32) & 0xFFFFFFFF
    accel_ip.register_map.C_1 = C_buf.physical_address & 0xFFFFFFFF
    accel_ip.register_map.C_2 = (C_buf.physical_address >> 32) & 0xFFFFFFFF
    accel_ip.register_map.N = N
    accel_ip.register_map.K = K
    accel_ip.register_map.M = M
    # Pass the update_A flag to the accelerator
```

```
    accel_ip.register_map.update_A = update_A

    # Start the accelerator
    accel_ip.register_map.CTRL.AP_START = 1

    # Wait for finish
    while accel_ip.register_map.CTRL.AP_DONE == 0:
        pass

    # Invalidate output buffer so the CPU sees the updated data from DDR
    C_buf.invalidate()
```

## 2 Helper Functions

This block contains utility functions for FPGA-based acceleration.
- **call_fpga()**: Sends matrix multiplication tasks to the FPGA and retrieves results. - **pynq_buffer_from_numpy()**: Converts NumPy arrays to PYNQ-compatible buffers. - **requantize()**: Converts int32 arrays to int8 using a scaling factor and zero point. - **display_model_confidence()**: Converts logits to human-readable class confidence.

[21]:
```
import numpy as np
from pynq import allocate

def pynq_buffer_from_numpy(np_array):
    """
    Allocates a PYNQ buffer with the same shape and dtype as np_array,
    then copies the data into the buffer.
    """
    buf = allocate(np_array.shape, dtype=np_array.dtype)
    np.copyto(buf, np_array)
    return buf

def requantize(int32_array, scale, zero_point=0):
    """
    Requantizes an int32 numpy array to int8 using the provided scale and␣
 ↪zero_point.
    Operation: int8_val = clip(round(int32_val * scale + zero_point), -128, 127)
    """
    scaled = np.round(int32_array * scale + zero_point)
    int8_array = np.clip(scaled, -128, 127).astype(np.int8)
    return int8_array

def display_model_confidence(logits, device_name="Model"):
    """
    Converts logits to probabilities and prints a user-friendly confidence␣
 ↪message.
```

```
    Parameters:
    logits (torch.Tensor): The raw model output (logits).
    device_name (str): Name of the device (e.g., "CPU", "FPGA") for comparison.
    """
    # Convert logits to probabilities
    probs = torch.softmax(logits, dim=1)

    # Get predicted class and confidence
    predicted_class = torch.argmax(probs, dim=1).item()
    confidence = probs[0, predicted_class].item() * 100

    # Print result
    print(f"{device_name}: The model is {confidence:.2f}% confident in␣
↪predicting class {predicted_class}.")
```

## 3 Custom Module for FPGA Offload

**FPGA-Optimized Linear Layer for Q, K, V Projections** This block defines **FPGAQuantizedLinear**, a custom PyTorch module that replaces
the standard linear layers with FPGA-accelerated equivalents. It: - **Quantizes activations** before computation. - **Uses PYNQ buffers** to store inputs and weights. - **Invokes the FPGA accelerator** for matrix multiplications. - **Dequantizes the result** back to floating point. This module is later integrated into DistilBERT for hardware acceleration.

[4]:
```
import torch
import numpy as np

class FPGAQuantizedLinear(torch.nn.Module):
    def __init__(self, quantized_linear, act_scale, accel_ip, hidden_size=768,␣
↪update_A=True):
        """
        Parameters:
          quantized_linear : an instance of DynamicQuantizedLinear from the␣
↪quantized model.
          act_scale        : scaling factor for quantizing input activations.
          accel_ip         : the FPGA accelerator IP handle.
          hidden_size      : hidden dimension size (typically 768).
          update_A         : flag indicating whether to update A in persistent␣
↪BRAM (True for Q, False for K/V).
        """
        total_fpga_compute_time = 0.0
        call_count = 0

        super(FPGAQuantizedLinear, self).__init__()
        self.accel_ip = accel_ip
        self.hidden_size = hidden_size
```

```
        self.act_scale = act_scale
        self.update_A = update_A  # Store the update flag

        # Extract quantized weight and its parameters.
        self.weight_int8_tensor = quantized_linear.weight().int_repr()
        self.weight_scale = quantized_linear.weight().q_scale()
        self.weight_zero_point = quantized_linear.weight().q_zero_point()
        # Transpose so that the weight shape becomes (in_features, out_features)
        self.weight_int8 = self.weight_int8_tensor.cpu().numpy().T  # shape:␣
↪(hidden_size, hidden_size)

        # Effective scale: multiplication of activation scale and weight scale.
        self.effective_scale = self.act_scale * self.weight_scale

        # Check for bias. Note that in DynamicQuantizedLinear, bias remains in␣
↪FP32.
        bias_val = quantized_linear.bias()  # This calls the bound method.
        if bias_val is not None:
            # Save bias as a NumPy array (shape: (hidden_size,))
            self.bias = bias_val.detach().cpu().numpy().astype(np.float32)
        else:
            self.bias = None

    def forward(self, x):
        """
        Forward pass for FPGA offload.
        Accepts input x which may be 2D (N, D) or 3D (B, S, D). In case of 3D␣
↪input,
        the tensor is reshaped to 2D for matrix multiplication and then reshaped␣
↪back.
        The input is quantized to int8 using self.act_scale. After the FPGA␣
↪multiplication,
        the int32 result is dequantized to FP32 and the bias is added (if␣
↪available).
        """
        # Save the original shape.
        orig_shape = x.shape
        if x.dim() == 3:
            B, S, D = x.shape
            x_flat = x.reshape(B * S, D)
        else:
            x_flat = x

        # Determine the number of rows for the FPGA call.
        N = x_flat.shape[0]
```

```
        # Quantize the input if it is in float32.
        if x_flat.dtype == torch.float32:
            x_int8 = torch.clamp(torch.round(x_flat / self.act_scale), -128,␣
↪127).to(torch.int8)
        else:
            x_int8 = x_flat

        # Convert to a NumPy int8 array.
        x_np = x_int8.cpu().numpy().astype(np.int8)

        # Convert input activation and weight to PYNQ buffers.
        A_buf = pynq_buffer_from_numpy(x_np)
        W_buf = pynq_buffer_from_numpy(self.weight_int8)
        # Allocate an output buffer for the int32 result (shape: (N,␣
↪hidden_size))
        C_buf = allocate((N, self.hidden_size), dtype=np.int32)

        # Call the FPGA accelerator:
        # Instead of hardcoding update_A=1, we now use self.update_A:
        # Time just the FPGA computation
        start_fpga = time.time()
        call_fpga(A_buf, W_buf, C_buf, self.accel_ip, N, self.hidden_size, self.
↪hidden_size, update_A=int(self.update_A))
        fpga_duration = time.time() - start_fpga

        FPGAQuantizedLinear.total_fpga_compute_time += fpga_duration
        FPGAQuantizedLinear.call_count += 1

        # Retrieve the int32 result.
        C_int32 = np.array(C_buf)
        # Dequantize: convert int32 accumulator to FP32 using the effective␣
↪scale.
        out_fp32 = C_int32.astype(np.float32) * self.effective_scale

        # If a bias is present, add it (broadcast along axis 0).
        if self.bias is not None:
            # Ensure bias is added to each row.
            out_fp32 = out_fp32 + self.bias

        # Convert back to a torch tensor.
        out_tensor = torch.tensor(out_fp32, dtype=torch.float32)

        # If the original input was 3D, reshape back to (B, S, hidden_size).
        if x.dim() == 3:
            out_tensor = out_tensor.reshape(B, S, self.hidden_size)
        return out_tensor
```

## 4 Replacing Q, K, V Layers with FPGA Versions

This function walks through all transformer layers in the quantized DistilBERT model and replaces the **Q, K, and V projection layers** with the custom **FPGAQuantizedLinear** module. - Ensures **Q projection updates A in BRAM** (update_A=True). - **K and V projections reuse A** for efficiency. - Enables model acceleration while preserving transformer layer structure.

[5]:
```
def integrate_fpga_offload(model_quant, act_scale, accel_ip, hidden_size=768):
    """
    Replaces the Q, K, V projection layers in each transformer layer with the␣
↪FPGA-accelerated custom module.

    Parameters:
      model_quant  : Quantized DistilBertForSequenceClassification model.
      act_scale    : Scaling factor for quantizing activations (assumed same for␣
↪demo).
      accel_ip     : Configured FPGA accelerator IP handle.
      hidden_size  : Hidden dimension (typically 768).
    """
    for layer in model_quant.distilbert.transformer.layer:
        # For the Q projection, set update_A to True so that the persistent A is␣
↪updated.
        layer.attention.q_lin = FPGAQuantizedLinear(layer.attention.q_lin,␣
↪act_scale, accel_ip, hidden_size, update_A=True)
        # For K and V projections, set update_A to False to reuse A from BRAM.
        layer.attention.k_lin = FPGAQuantizedLinear(layer.attention.k_lin,␣
↪act_scale, accel_ip, hidden_size, update_A=False)
        layer.attention.v_lin = FPGAQuantizedLinear(layer.attention.v_lin,␣
↪act_scale, accel_ip, hidden_size, update_A=False)
```

[6]:
```
import numpy as np

def compute_activation_scale(activation_list, percentile=99.9, use_demo=0):
    """
    Computes a global activation scale from a calibration set of activations.

    Parameters:
      activation_list: List of NumPy arrays representing activations
                       (for example, from the embedding layer).
      percentile:      The percentile to use for robust scale computation (if␣
↪use_demo=0).
      use_demo:        If set to 1, uses the demo method (scale = max_abs_value/
↪127.0);
                       otherwise, uses the robust method (scale =␣
↪percentile_value/127.0).
```

```
    Returns:
      A scaling factor computed as:
         - Demo method: scale = (max(|activations|)) / 127.0
         - Robust method: scale = (percentile(|activations|)) / 127.0
    """
    # Concatenate all activations from the calibration samples into one array.
    all_activations = np.concatenate([act.flatten() for act in activation_list])

    if use_demo:
        # Demo method: use the maximum absolute value.
        act_abs_max = np.max(np.abs(all_activations))
        scale = act_abs_max / 127.0 if act_abs_max != 0 else 1.0
    else:
        # Robust method: use the specified percentile.
        act_abs_percentile = np.percentile(np.abs(all_activations), percentile)
        scale = act_abs_percentile / 127.0 if act_abs_percentile != 0 else 1.0

    return scale
```

## 5 Example Usage – Custom Forward Pass Integration

This block demonstrates how to: 1. **Load and quantize a DistilBERT model**. 2. **Extract activations** from the embedding layer. 3. **Integrate FPGA acceleration** into transformer layers. 4. **Run a forward pass** through the modified model. Only the **Q, K, and V projections** are offloaded to FPGA; the remaining layers run on CPU/GPU.

[24]:
```
import torch
from transformers import DistilBertTokenizer, DistilBertForSequenceClassification

# Assume call_fpga() is already defined and accel_ip is configured on your KV260.
# For example:
# accel_ip = get_accel_ip_handle()   # <-- user-specific setup

# 1. Load and Quantize the Model
model_name = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = DistilBertTokenizer.from_pretrained(model_name)
model = DistilBertForSequenceClassification.from_pretrained(model_name)
model.eval()

# Apply dynamic quantization to convert Linear layers to int8.
model_int8 = torch.quantization.quantize_dynamic(model, {torch.nn.Linear},␣
 ↪dtype=torch.qint8)
model_int8.eval()

# 2. Gather a Calibration Set of Activations to Compute a Global Activation Scale
calib_sentences = [
```

```
    "The moonlight shimmered over the ocean as waves gently kissed the sandy␣
 ↪shore, while distant lanterns flickered in the cool evening breeze. A lone␣
 ↪traveler wandered along the coastline, footsteps sinking softly into the damp␣
 ↪sand, lost in thought. The rhythmic sound of the water mixed with the rustling␣
 ↪palms, creating a nice"
]
calib_activations = []
for sentence in calib_sentences:
    inputs = tokenizer(sentence, return_tensors="pt")
    with torch.no_grad():
        # Get the embedding output; shape: (B, L, 768). Here B=1.
        emb = model.distilbert.embeddings(inputs.input_ids)  # shape: (1, L, 768)
        # Remove the batch dimension.
        emb = emb.squeeze(0)  # shape: (L, 768)
        calib_activations.append(emb.cpu().numpy())

# Compute the activation scale using the robust method (percentile-based):
# global_act_scale_robust = compute_activation_scale(calib_activations,␣
 ↪percentile=99.9, use_demo=0)
# print("Global Activation Scale (Robust):", global_act_scale_robust)

# # Compute the activation scale using the demo method (max-based):
global_act_scale_demo = compute_activation_scale(calib_activations, use_demo=1)
print("Global Activation Scale (Demo):", global_act_scale_demo)

test_sentence = calib_sentences[0]
print(f"input = '{test_sentence}'")
```

```
Global Activation Scale (Demo): 0.06596306177574819
input = 'The moonlight shimmered over the ocean as waves gently kissed the sandy
shore, while distant lanterns flickered in the cool evening breeze. A lone
traveler wandered along the coastline, footsteps sinking softly into the damp
sand, lost in thought. The rhythmic sound of the water mixed with the rustling
palms, creating a nice'
```

# 6 FPGA vs. CPU Inference Benchmarking

### 6.0.1 Objective

This block measures and compares the inference performance of **CPU-only vs. FPGA-accelerated execution**
using a **quantized DistilBERT model** for text classification.

### 6.0.2 Steps & Key Operations

1. **Run Inference on CPU**
   - Tokenizes the input sentence and processes it on the CPU.
   - Captures inference time for PyTorch execution (`cpu_time`).

- Extracts and stores CPU-based logits (`logits_cpu`).

2. **Enable FPGA Offloading**
   - Replaces **Q, K, V projections** with FPGA-accelerated versions.
   - Resets FPGA timing counters before inference.
   - Runs inference on the FPGA-accelerated model (`model_int8`).
   - Captures **total FPGA computation time** and **average FPGA call duration**.
3. **Accuracy & Performance Comparison**
   - **Computes absolute differences** between CPU and FPGA logits.
   - Applies **softmax** to logits to determine class probabilities.
   - Extracts **predicted class & confidence scores** for both CPU and FPGA.
   - Displays model confidence in an **easy-to-read format**.

#### 6.0.3 Performance Metrics Reported

**CPU Inference Time** (Baseline execution time)
**FPGA Compute Time** (Total and per-call breakdown)
**Speedup Analysis** (CPU vs. FPGA execution time)
**Accuracy Check** (Max and Mean difference between CPU and FPGA logits)
**Predicted Class & Confidence Scores** (to validate inference consistency)

This block provides a **detailed comparison** of **inference speed, accuracy, and prediction confidence**
between **CPU and FPGA-accelerated execution**.

[25]:
```
import time

# CPU-only Inference
inputs = tokenizer(test_sentence, return_tensors="pt")

start_time = time.time()
with torch.no_grad():
    outputs_cpu = model(inputs.input_ids)
    logits_cpu = outputs_cpu.logits
cpu_time = time.time() - start_time

print(f"CPU Inference Time: {cpu_time:.6f} seconds")
# print("CPU Logits:", logits_cpu)

# FPGA-Offloaded Inference
integrate_fpga_offload(model_int8, global_act_scale_demo, accel_ip,␣
 ↪hidden_size=768)

# Reset the timing counters before inference
FPGAQuantizedLinear.total_fpga_compute_time = 0.0
FPGAQuantizedLinear.call_count = 0

# Run inference normally with your existing code
with torch.no_grad():
```

```
    outputs_fpga = model_int8(inputs.input_ids)
    logits_fpga = outputs_fpga.logits

# After inference, report the detailed timing
print(f"FPGA calls: {FPGAQuantizedLinear.call_count}")
print(f"Total time in FPGA compute: {FPGAQuantizedLinear.total_fpga_compute_time:
 ↪.6f} seconds")
print(f"Average time per FPGA call: {FPGAQuantizedLinear.total_fpga_compute_time/
 ↪FPGAQuantizedLinear.call_count:.6f} seconds")
# print(f"Adjusted speedup (FPGA compute only): {cpu_time / FPGAQuantizedLinear.
 ↪total_fpga_compute_time:.2f}x")

# Compute differences
diff = logits_cpu - logits_fpga
max_diff = diff.abs().max().item()
mean_diff = diff.abs().mean().item()

# Compute softmax probabilities
probs_cpu = torch.softmax(logits_cpu, dim=1)
probs_fpga = torch.softmax(logits_fpga, dim=1)

# Get predicted class and confidence
predicted_class_cpu = torch.argmax(probs_cpu, dim=1).item()
confidence_cpu = probs_cpu[0, predicted_class_cpu].item() * 100

predicted_class_fpga = torch.argmax(probs_fpga, dim=1).item()
confidence_fpga = probs_fpga[0, predicted_class_fpga].item() * 100

# Print results
# print(f"Max Logits Difference: {max_diff:.6f}")
# print(f"Mean Logits Difference: {mean_diff:.6f}")

display_model_confidence(logits_cpu, device_name="CPU")
display_model_confidence(logits_fpga, device_name="FPGA")
```

```
CPU Inference Time: 1.141709 seconds
FPGA calls: 18
Total time in FPGA compute: 0.430758 seconds
Average time per FPGA call: 0.023931 seconds
CPU: The model is 99.95% confident in predicting class 1.
FPGA: The model is 99.80% confident in predicting class 1.
```

## 6.1 Final Reflections: FPGA Offloading vs. PyTorch Inference

### 6.1.1 Key Takeaways

1. **FPGA Delivers Significant Computational Speedup, But System Bottlenecks Persist**

- Our **benchmark shows a 7.01× speedup over PyTorch and a 214.24× speedup over NumPy** for a single **(64, 768) × (768, 3072) MatMul** operation.

- Despite this raw speedup, **end-to-end inference time reduction remains limited** due to system-level inefficiencies such as data transfer overhead and synchronization delays.

2. **Challenges We Encountered**
   - **Data Transfer Overhead Dominates Latency**
     - Moving activations and weights **between CPU and FPGA introduces significant latency**, partially negating the speedup from FPGA computation.

   - **CPU-FPGA Synchronization Delays Reduce Efficiency**
     - The CPU often **idles while waiting for FPGA execution**, rather than executing operations in parallel.
   - **QKV Projection Alone Does Not Account for a Large Portion of Total Compute**
     - While FPGA significantly accelerates MatMul, **QKV projection alone does not contribute enough to total inference time to yield major speedup**.
   - **FPGA Start-Up & Control Overhead Adds Unavoidable Delays**
     - Register setup, memory flushing, and synchronization introduce additional latency before computation even starts.
   - **PyTorch's Highly Optimized GEMM Kernels Reduce the Acceleration Gap**
     - PyTorch's **int32 GEMM (MKL-DNN/TensorRT)** is already highly efficient, making it harder to achieve dramatic acceleration unless we offload a larger portion of the workload.

---

#### 6.1.2 Lessons Learned and Future Considerations

**1. Reducing Data Transfer Overhead is Crucial**

- To unlock **FPGA's full potential**, future work should minimize data movement **between CPU and FPGA**.
- Using more **persistent FPGA memory** for activations (like what we did for input embedding) and implementing **DMA (if available)** could further reduce transfer latency.

**2. Improving CPU-FPGA Execution Pipelining**

- **Overlapping CPU and FPGA execution** is essential—while the FPGA computes QKV projection, the CPU should process **attention softmax or FFN layers** in parallel.
- **Double buffering techniques** can ensure that **data is transferred while computation is ongoing**, reducing idle time, but it requrire careful timing design otherwise more latency will be introduced.

**3. Expanding FPGA Workload Beyond QKV**

- The **Feed-Forward Network (FFN) layer**, which consists of **(64, 3072) × (3072, 768) MatMul**, is computationally expensive and an ideal candidate for FPGA acceleration.

- **Self-attention softmax computation** could also be offloaded to FPGA to further reduce CPU workload and improve overall efficiency.

**4. Optimizing FPGA Utilization and Scaling**

- **Parallelizing Q, K, and V projections** on separate systolic arrays could further improve efficiency.
- Keeping the **FPGA accelerator active between layers**, rather than resetting registers and flushing memory for each computation, would reduce unnecessary overhead.

---

### 6.1.3 Final Thoughts

This project has **successfully validated the power of FPGA acceleration for deep learning workloads**, demonstrating a **7× speedup over PyTorch CPU execution** at the **MatMul level**. However, we have also seen firsthand that **raw computation speed is only part of the equation—data movement, synchronization, and system integration play an equally critical role** in achieving real-world performance gains.

For future FPGA acceleration research, a **holistic approach** that integrates **both compute and system-level optimizations** will be necessary to fully harness the hardware's potential. Our findings serve as a **guiding reference for future efforts**, emphasizing that true performance gains require a **balance of raw compute acceleration and efficient system-level execution**.

While this marks the completion of our current project, we hope that the insights gained here will pave the way for more **efficient and scalable deep learning inference on FPGA**.