SEAI_2024_R12

Giulio Capecchi, Jacopo Niccolai

December 2024

IN SUPREMÆ DIGNITATIS · 1343 ·

UNIVERSITÀ DI PISA

Contents

1 Introduction

This project focuses on the synthesis of the forward pass for three types of neural network architectures: a **Multilayer Perceptron (MLP)**, a **Convolutional Neural Network (ConvNet)**, and a **Transformer**, implemented on an FPGA. To achieve this, the network parameters were first obtained using Python and the *PyTorch* library. These parameters were subsequently hardcoded into `C` code, enabling the hardware synthesis process.

Figure 1: Xilinx Vitis HLS

2 Project Description

2.1 Workflow Overview

The neural networks were constructed and trained using the *PyTorch* library. Once trained, the weights and biases were exported to be hardcoded into the corresponding `C` implementation. The `C` code was specifically designed to be compatible with FPGA synthesis tools, such as Vitis HLS, ensuring efficient hardware synthesis.

2.2 C Implementation

The `C` code developed includes the forward pass for:

- **MLP**: implementation of propagation through dense layers.
- **ConvNet**: handling of convolution and pooling operations.
- **Transformer**: managing complex operations like attention.

#TODO : parlare delle direttive HLS /* HLS INLINE forza l'inserimento completo del corpo di una funzione o di un ciclo direttamente nel punto in cui è chiamato, eliminando l'overhead associato alla chiamata di funzione o all'allocazione di risorse hardware separate. Riduce la latenza perché elimina il ritardo di chiamata della funzione. Può aumentare l'area hardware utilizzata, perché la logica

della funzione viene replicata ovunque venga chiamata. Tipicamente per funzioni semplici o molto usate, per ridurre la latenza.
HLS PIPELINE La direttiva HLS PIPELINE suddivide un ciclo o una funzione in più fasi (pipeline), permettendo l'esecuzione di più iterazioni o operazioni simultaneamente, aumentando il throughput del design. Permette l'elaborazione di nuove iterazioni a ogni ciclo di clock (o a intervalli specificati, chiamati Initiation Interval - II). Aumenta la velocità del sistema a scapito del consumo di risorse hardware. Opzioni: II=N (specifica l'intervallo di avvio delle iterazioni, ad esempio, 1 ciclo di clock tra una e l'altra). rewind (riavvia automaticamente il ciclo alla fine). Uso tipico Nei loop che processano grandi quantità di dati. Per aumentare il throughput in operazioni ripetitive. */
The network parameters (weights and biases) were directly integrated into the code in a hardcoded manner.

3 Code Architecture

3.1 C File Structure

The forward pass is implemented using a sequence of functions for each layer type:

- Activation functions (`relu`, `softmax`, etc.).
- Functions for convolution and pooling operations.
- Functions for attention mechanisms in Transformers.

3.2 Hardware Synthesis

The code was designed to be compatible with tools such as Vitis HLS, leveraging specific pragmas to optimize the implementation.

4 Multi-Layer Perceptron

Let's analyze the implementation of the forward pass for a Multi-Layer Perceptron. The forward pass for an MLP consists of propagating the input through a series of dense layers, each followed by an activation function. The code for it can be found inside the `PyTorch` folder, that contains the notebooks used to train the models and export their parameters.

4.1 Dataset

The MLP was trained using the well-known *Iris* dataset, which contains 150 samples of iris flowers, each with four features and a class label (the last value of each row). There is a total of three classes: *setosa*, *versicolor*, and *virginica*.

The dataset was split into training and test sets, with 80% of the samples used for training and 20% for testing.
An example of the dataset is shown below:

```
sepal_length,sepal_width,petal_length,petal_width,species
5.1,3.5,1.4,0.2,setosa
5.7,2.8,4.5,1.3,versicolor
6.1,2.6,5.6,1.4,virginica
...
```

The dataset's labels were encoded as integers using the Scikit-learn `LabelEncoder` class, which maps each class to a unique integer value. To facilitate its further usage inside the `C` implementation, this encoded version of the dataset was saved to a txt file, *iris_dataset_encoded.txt*.

4.2 PyTorch Model

4.2.1 Model Architecture

The architecture of the MLP model consists of three fully connected (dense) layers. The input layer has 4 neurons corresponding to the 4 features of the Iris dataset. The first and second hidden layers each have 10 neurons, and the output layer has 3 neurons corresponding to the 3 classes of the Iris dataset. The chosen activation function is the ReLU function, which is applied after each dense layer except the output layer. The forward pass of the model involves applying the ReLU activation function after the first and second layers.
ReLU is defined as:

$$\mathrm{ReLU}(x) = \max(0, x)$$

The model was defined as follows, using the *PyTorch* library:

```python
# Define the MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(4, 10)
        self.fc2 = nn.Linear(10, 10)
        self.fc3 = nn.Linear(10, 3)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

4.2.2 Model Training

For the training phase, we first defined the model, loss function, and optimizer. We utilized the *CrossEntropyLoss* loss function, which is commonly used for multi-class classification problems, and the *Adam* optimizer, which is an adaptive learning rate optimization algorithm.

```python
model = MLP()

# Check if GPU is available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

The following code snippet demonstrates the trainign process using *PyTorch*. The model is trained for 100 epochs.

```python
# Training loop
NUM_EPOCHS = 100
for epoch in range(NUM_EPOCHS):
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    # Evaluate the model after each epoch
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1) # Get the class index with the highest probability
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

```

```
    accuracy = correct / total

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{NUM_EPOCHS}], Loss: {
            running_loss/len(train_loader):.4f}, Accuracy: {
            accuracy * 100:.2f}%')
```

The results of the training process are reported in the table below:

Epoch	Loss	Train Accuracy	Test Accuracy
10	0.3258	90.48%	88.89%
20	0.1060	97.14%	97.78%
30	0.1312	95.24%	97.78%
40	0.0853	97.14%	100.00%
50	0.0675	98.10%	100.00%
60	0.0971	96.19%	97.78%
70	0.0952	96.19%	97.78%
80	0.1020	96.19%	97.78%
90	0.0929	96.19%	97.78%
100	0.0788	94.29%	97.78%

Table 1: Training loss, train accuracy, and test accuracy of the MLP over 100 epochs.

The training process shows that the model is able to achieve a high level of accuracy on both the training and test sets: this is to be expected given the simplicity of the Iris dataset and the effectiveness of the MLP architecture in solving such problems. Given the relatively small dataset and the fact that the model achieves near-perfect accuracy on the test set, we can conclude that the model is generalizing well.

4.2.3 Exporting Parameters

The trained model's parameters were exported to be hardcoded into the `C` implementation. The weights and biases of each layer were extracted and saved in a txt file, *weights.txt*, as shown below:

```
import numpy as np

weights = {}
for name, param in model.named_parameters():
    weights[name] = param.detach().numpy()

# Print their shapes to verify the network architecture
for name, weight in weights.items():
    print(f"{name}: {weight.shape}")

with open('./weights.txt', 'w') as f:
```

```
for name, weight in weights.items():
    f.write(f"{name}\n")
    np.savetxt(f, weight, fmt='%f')
```

4.3 C Implementation

The implementation for Vitis HLS of the MLP was produced with three files:

- `mlp.c`, which contains the forward pass function, the activation function, and the definition of the MLP structure.
- `mlp.h`, which contains the definition of the MLP structure and the forward pass function prototype.
- `testbench.c`, which reads the *iris_dataset_encoded.txt* and contains the main function to test the forward pass function.

4.3.1 MLP Structure

The MLP structure was defined as follows (inside `mlp.h`):

```c
typedef struct {
    float weights[MAX_NEURONS][MAX_NEURONS]; // matrix
    float biases[MAX_NEURONS];    // biases of the layer
    float output[MAX_NEURONS];    // output of the layer
} Layer;

typedef struct {
    int num_layers;               // number of layers
    Layer layers[MAX_LAYERS];     // layers of the MLP (array of layers)
} MLP;
```

`MAX_NEURONS` and `MAX_LAYERS` are defined as 100 and 3, respectively.

4.3.2 Forward Pass

The forward pass for the MLP is implemented in `mlp.c` as a sequence of operations applied to each layer of the network. The function processes four input features through three layers, each defined by its respective weights and biases, to produce the predicted class index. It is defined as follows:

```c
int forward(float input0, float input1, float input2, float input3) {
    const int input_sizes[4] = {4, 10, 10, 3};
    const int num_layers = 3;

    float current_input[MAX_NEURONS];
    float next_input[MAX_NEURONS];

```

```c
    current_input[0] = input0;
    current_input[1] = input1;
    current_input[2] = input2;
    current_input[3] = input3;

    for (int i = 0; i < num_layers; i++) {
        #pragma HLS UNROLL
        Layer *layer = &mlp.layers[i];
        for (int j = 0; j < input_sizes[i + 1]; j++) {
            float sum = layer->biases[j];

            for (int k = 0; k < input_sizes[i]; k++) {
                sum += layer->weights[j][k] * current_input[k];
            }
            next_input[j] = reLu(sum);
        }

        for (int j = 0; j < input_sizes[i + 1]; j++) {
            current_input[j] = next_input[j];
        }
    }

    int max_index = 0;
    float max = current_input[0];
    for (int i = 1; i < NUM_CLASSES; i++) {
        #pragma HLS UNROLL
        if (current_input[i] > max) {
            max = current_input[i];
            max_index = i;
        }
    }
    return max_index;
}
```

In this implementation, all weights and biases are hardcoded and directly integrated into the MLP definition at the top of `mlp.c`. These values are exported from the PyTorch model and inserted manually during the setup phase.
A notable feature of this implementation is the use of the `#pragma HLS UNROLL` directive. This instructs the High-Level Synthesis (HLS) tool to unroll loops, which replicates the loop body multiple times. This mechanism significantly enhances the throughput of the design by enabling parallel execution of loop iterations: as a result, the forward pass should achieve higher performances, making it suitable for FPGA-based acceleration.

4.3.3 Testbench

The testbench function reads the encoded Iris dataset file and applies the forward pass function to each sample. The file is read line-by-line by using the fscanf function, and the four features are passed to the forward pass function. The predicted class index is then compared with the actual class index to calculate the accuracy of the model.

Below the code for the testbench function:

```c
int read_data_from_file(const char *path, int num_features,
    int label_size, float input_data[MAX_SAMPLES][
    MAX_FEATURES], float true_value[MAX_SAMPLES]) {
    FILE *file = fopen(path, "r");
    if (!file) {
        perror("Failed to open file");
        return -1;
    }

    int sample_count = 0;
    while (fscanf(file, "%f", &input_data[sample_count][0])
        != EOF) {
        for (int i = 1; i < num_features; i++) {
            fscanf(file, "%f", &input_data[sample_count][i])
                ;
        }
        for (int j = 0; j < label_size; j++) {
            fscanf(file, "%f", &true_value[sample_count]);
        }
        sample_count++;
        if (sample_count >= MAX_SAMPLES) {
            break;
        }
    }

    fclose(file);
    return sample_count;
}

int main() {
float input_data[MAX_SAMPLES][MAX_FEATURES];
float true_value[MAX_SAMPLES];

// Read data from file
const char *path = "./datasets/iris_dataset/
    iris_dataset_encoded.txt";
int sample_count = read_data_from_file(path, MAX_FEATURES,
    1, input_data, true_value);

// call the forward function and calculate the accuracy
int correct_predictions = 0;
for (int i = 0; i < sample_count; i++) {
    int prediction = forward(input_data[i][0], input_data[i
        ][1], input_data[i][2], input_data[i][3]);
```

```c
    if (prediction == true_value[i]) {
        correct_predictions++;
    }else{
        printf("Prediction: %d, True value: %f for input: %f
            %f %f %f\n", prediction, true_value[i],
            input_data[i][0], input_data[i][1], input_data[i
            ][2], input_data[i][3]);
    }
}
float accuracy = (float)correct_predictions / sample_count *
    100.0;
printf("Accuracy: %.2f%%\n", accuracy);
}
```

The testbench is used to test that everything is working correctly and to evaluate the accuracy of the model on the dataset. The accuracy obtained should be similar to the one achieved during the training phase in *PyTorch*, confirming that the forward pass function is correctly implemented in `C`. We always obtained an accuracy of 98% with it, consistent with the results obtained with *PyTorch*.

4.4 Results

The results obtained using the Vitis Unified IDE confirm the successful synthesis and implementation of the MLP forward pass on the FPGA. The development process in Vitis offers several stages where detailed reports are generated, providing valuable insights into the design's functionality and performance.

4.4.1 Stages of Development

These stages include:

- **C Simulation:** This initial step ensures the functional correctness of the high-level C implementation. During this phase, the input data is processed entirely in software, and the generated reports confirm that the output matches the expected results, validating the logic before hardware synthesis.

- **C Synthesis:** In this phase, the high-level C code is converted into a hardware description optimized for the target FPGA. The synthesis report provides crucial details, such as estimated resource utilization (LUTs, DSPs, BRAMs), latency, and initiation intervals. These metrics help identify potential bottlenecks and guide optimization efforts.

- **C/RTL Simulation:** This step bridges the gap between high-level and low-level design by validating the synthesized hardware description against the functional requirements. This stage is particularly important as it ensures consistency between the high-level model and the Register Transfer

Level (RTL) implementation. The reports include timing diagrams, functional waveforms, and a comparison of C simulation outputs with RTL simulation outputs to confirm correctness.

- **Packaging:** After verifying the synthesized hardware, the design is packaged into an IP (Intellectual Property) core. The packaging reports detail the generated IP core's properties, ensuring that it adheres to the FPGA's integration requirements and is ready for system-level implementation.
- **Implementation:** In the final stage, the IP core is placed and routed on the FPGA. Implementation reports include metrics such as timing analysis, power estimates, and resource utilization on the physical FPGA fabric. These reports confirm that the design meets the FPGA's constraints, such as timing closure and power consumption.

By consulting these reports, the development process is highly transparent: each step ensures the correctness, performance, and compliance with the FPGA's requirements, resulting in an efficient and reliable implementation of the top-function, in this case the forward pass.

These steps are valid also for the further architectures, so we will not repeat them in the following sections, but just present them.

4.4.2 Performance Metrics

Let's go into detail about the performance metrics obtained during the synthesis of the MLP forward pass on the FPGA.

C-Simulation The C-simulation stage in the Vitis Unified IDE serves as the initial validation step for the functional correctness of the high-level C implementation. At this stage, all metrics are presented as pre-synthesis estimates, offering insights into the steady-state performance of the design assuming repeated kernel execution with ready interfaces. These metrics, while useful, are optimistic until the code is made canonical, as they do not account for potential canonicalization issues.
Key metrics analyzed include the **Transaction Interval (TI)**, which represents the minimal delay between two process executions. Optimizing processes with the largest TIs is critical to improving overall region performance. Channels are characterized by metrics such as **Volume**, indicating the amount of data transmitted per region execution, and **Throughput**, which reflects the data transfer rate and is influenced by the TI and channel volume. Additionally, **Width**, defined by the size of data elements, plays a crucial role in enhancing throughput by widening communication channels.
Through these metrics, C-simulation provides a foundational understanding of performance bottlenecks and potential optimization opportunities, setting the stage for more accurate evaluations in subsequent synthesis and co-simulation phases.

5 Convolutional Neural Network (ConvNet)