SEAI_2024_R12

Giulio Capecchi, Jacopo Niccolai

December 2024

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 (Field Programmable Gate Array). To achieve this, the network parameters were first obtained using Python and the *PyTorch* library; these were subsequently hardcoded into `C` code, enabling the hardware synthesis process.

Figure 1: Xilinx Vitis HLS

Vitis Unified Software Platform is a comprehensive suite designed to accelerate the development of applications on FPGAs, Adaptive SoCs, and ACAPs (Adaptive Compute Acceleration Platforms). By combining high-level software programming techniques with hardware-optimized implementations, Vitis enables developers to write applications in *C*, *C++*, or *OpenCL* while leveraging hardware-specific optimizations for enhanced performance. In this project, Vitis was used to synthesize neural network architectures - MLP, ConvNet, and Transformer - onto an FPGA. Its **High-Level Synthesis (HLS)** tools allow for rapid prototyping and optimization of the *C* code, ensuring efficient resource utilization, parallelism and low-latency execution. The platform's ability to integrate high-level design, simulation, and hardware synthesis simplifies the workflow, bridging the gap between software and hardware development.

2 Project Description

2.1 Workflow Overview

The project follows a structured workflow. For each neural network architecture, a Jupyter Notebook is provided in the `PyTorch` folder. As the name of the folder suggests, these 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 designed to be compatible with FPGA synthesis tools, such as Vitis HLS/Vivado, and can be found inside the `HLS-Implementation` folder.

2.2 Jupyter Notebooks

For each network architecture, a Jupyter Notebook was created to train the model and export the parameters. These notebooks are similar to each other since they contain almost the same steps for each neural network. Their main sections are:

- **Dataset Preparation**: Loading the chosen dataset to train the model.
- **Model Definition**: Defining the neural network architecture.
- **Model Training**: Training the model on the dataset, ensuring that it converges to a satisfactory accuracy.
- **Exporting Parameters**: Extracting the weights and biases for the forward pass to a text file.

Figure 2: Jupyter Notebooks logo

2.3 C Implementation

As for the Jupyter Notebooks, the `C` code is structured in a similar way for each architecture. There are always *three files* provided:

- `architecture.c`: contains the forward pass function and the definition of the architecture.
- `architecture.h`: contains the definition of the architecture and the prototype of the forward pass function.
- `testbench.c`: reads the dataset and contains the main function to test the forward pass function.

To optimize the C implementation for hardware synthesis, specific **HLS directives** were applied to critical portions of the code. These directives guide the High-Level Synthesis (HLS) tool to produce more efficient hardware designs by controlling resource allocation, loop unrolling, and pipeline creation. The two main directives used in this project are:

- `HLS INLINE`: This directive forces the complete insertion of the body of a function or loop directly at the point where it is called, eliminating the overhead associated with function calls or separate hardware resource allocation. By doing so, it reduces latency by eliminating function call delays. However, it may increase the usage of the hardware area, as the logic is replicated wherever the function is called. It is typically used for simple or frequently invoked functions to reduce latency.
- `HLS PIPELINE`: This directive breaks a loop or function into multiple stages (pipeline), allowing multiple iterations or operations to execute simultaneously, thereby increasing the design's throughput. It enables the processing of new iterations in every clock cycle (or at specific intervals called *initiation interval (II)*). The options for this directive include `II=N` (to specify the interval between iterations, such as 1 clock cycle) and `rewind` (to automatically restart the loop after completion). It is typically used in loops that process large amounts of data to maximize throughput in repetitive operations.
- `Other Directives Used:` Several other HLS directives are employed in the implementation to further optimize performance and resource utilization. They will be explained in detail at the points where they appear in the code.

It is important to note that the directives discussed in this section represent only a subset of the directives executed by the HLS compiler during the synthesis process. These directives were explicitly added to address specific warnings and improve the design's performance. They were placed on the exact lines of code where they were required to resolve issues or optimize critical sections of the implementation.

Hardcoding Network Parameters The network's weights and biases, extracted from the trained PyTorch model, were directly integrated into the *C* implementation as hardcoded values. This allows the FPGA hardware to access the pre-trained parameters directly during the forward pass, avoiding external memory accesses. The process of exporting these parameters is detailed in 3.2.3. After extraction, the parameters were organized in the `C` code as arrays within dedicated data structures. These structures and their integration will be explained in detail later on for each architecture.

2.4 Loading the project into the Vitis Unified IDE

The project was loaded into the Vitis Unified IDE to synthesize the `C` code and generate the corresponding hardware description. The IDE provides a comprehensive environment for developing, debugging, and deploying applications on Xilinx devices. The following steps has to be followed to load the project into the Vitis Unified IDE:

1. Open the Vitis Unified IDE and under `HLS Development` select `Create component...`.
2. Click on `Create Empty HLS Component`. A new window should appear, where you can specify the component name and location.
3. You will then be asked about the configuration file : you can either provide one or click on *next* and let the IDE create one for you.
4. In the following page, specify in the `Top Function` field the name of the function that you want to synthesize: this should be the one that you want to test on the FPGA. In our case, it is the forward pass function (the name is `forward` for all the networks we worked on).
5. Click on next and select a device: it should have enough resources to contain the design.
6. Click on `next` until you reach the `finish` button. Click on it to create the project.

Once done, you should see on the left a structure similar to the one provided in the image. Inside `Settings` you can find the configuration file for the project, but the simpler way to add the necessary files to the project is to right-click on `Sources` and then add the `C` and header files. Do the same for the `testbench`, where you should add the `txt` file containing the dataset and the `testbench.c` file, specific for the current component.
To "run" the project, you can then use the buttons on the bottom of this section (`Run, C Synthesis, C/RTL Cosimulation, Package, Implemenattion`). Their usage will be explained later on.

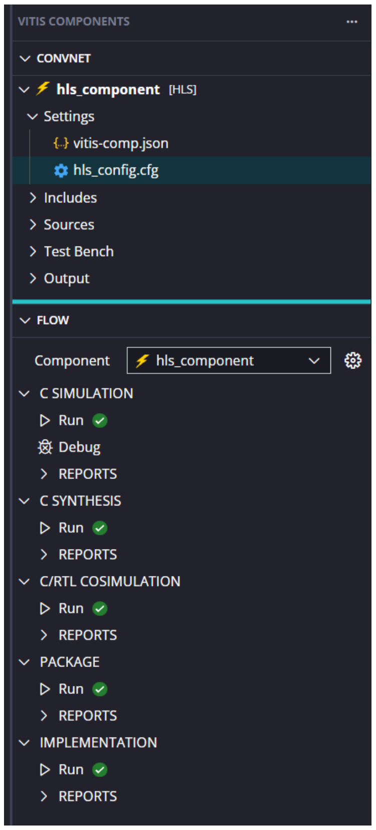


Figure 3: IDE Structure

3 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 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.

3.1 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*.

3.2 PyTorch Model

3.2.1 Model Architecture

The architecture of the MLP model consists of three fully connected 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, while the output layer has 3 neurons, corresponding to the 3 classes of the Iris dataset. The ReLU activation function is applied after each dense layer, except for the output layer.

The ReLU function is defined as:

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

The forward pass of the model applies the ReLU activation function after the first and second layers.

The model was implemented using the *PyTorch* library as follows, by defining a custom class MLP that inherits from nn.Module:

```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))  # Apply ReLU after
            first layer
        x = torch.relu(self.fc2(x))  # Apply ReLU after
            second layer
        x = self.fc3(x)  # Output layer (no activation)
        return x
```

3.2.2 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)
```

```python
            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.

3.2.3 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, *mlp_weights.txt*, as shown below:

```python
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}")

import numpy as np

with open('./mlp_weights.txt', 'w') as f:
    for name, weight in weights.items():
        f.write(f"// {name}, shape: {weight.shape}\n")
        if weight.ndim == 2:  # Fully connected layer
            weights
            for row in weight:
                f.write("{" + ", ".join(map(str, row)) + "
                    },\n")
        elif weight.ndim == 1:  # Biases or 1D weights
            f.write("{" + ", ".join(map(str, weight)) + "},\
                n")
```

3.3 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.

3.3.1 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.

3.3.2 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];

    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;
}
```

As already explained, 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 were inserted manually into the code.

A notable feature of this implementation is the use of the `#pragma HLS UNROLL` directive. This directive was necessary to address specific warnings related to loop dependencies and to enhance the throughput of the design by enabling parallel execution of loop iterations. Without this directive, the synthesis tool generated warnings indicating potential performance bottlenecks. By unrolling the loops, the forward pass achieves higher performance, making it suitable for FPGA-based acceleration.

3.3.3 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) {
```

```c
        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]);
    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*.

3.4 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. The used device for this section was from the Product family **zynq**, and the Target device used was the **xc7z007s-clg225-2**.

The selected target device belongs to the Zynq-7000 family and is designed to integrate ARM processing systems with programmable logic. It features the following specifications:

- **Logic Resources**: The device provides 14,400 Look-Up Tables (LUTs) for implementing combinatorial logic.
- **Flip-Flops (FFs)**: A total of 28,800 flip-flops are available, offering robust sequential logic capabilities.
- **DSP Blocks**: The device includes 66 DSP slices, making it suitable for high-performance signal processing tasks.
- **Block RAM (BRAM)**: 50 BRAMs are available, ensuring ample on-chip memory for intermediate data storage.

This combination of resources makes the `xc7z007s-clg225-2` well-suited for applications requiring both computation and flexibility, such as neural network inference on FPGA hardware.

The selected device has a **speed grade of** `-2`, which represents a medium performance level within the Zynq-7000 family. Speed grades for this family typically range from `-1` (lowest performance) to `-3` (highest performance). The `-2` speed grade offers a balance between performance and power efficiency, providing sufficient timing capabilities for the neural network inference tasks targeted in this design. Lower speed grade numbers correspond to higher achievable clock frequencies and reduced propagation delays, making the `-2` grade an optimal choice for this application.

For this implementation, we used the **default clock setting** provided by Vitis, which is configured to a period of *10 ns*. This corresponds to a clock frequency of *100 MHz*. The following results will demonstrate how this default clock period fits well with our solution, as we achieved efficient performance without the need to modify this default value.

3.4.1 Stages of Development in the Vitis Unified IDE

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 important 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 Cosimulation:** 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 other architectures aside of the MLP, so we will not repeat them in the following sections, but just present them.

3.4.2 Performance Metrics

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

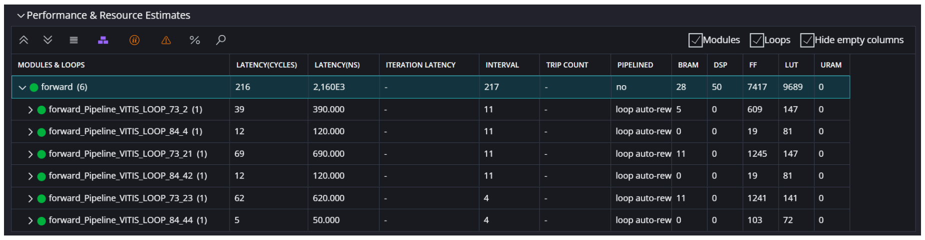
C-Simulation C-simulation provides preliminary performance metrics, focusing on the steady-state execution of the design. These estimates, including the Transaction Interval (TI), highlight potential bottlenecks and optimization areas but may be overly optimistic unless the code is made canonical. This stage serves as an initial evaluation, guiding further refinement and more accurate analysis during synthesis and co-simulation. Here, we could mainly observe the correctness of the code (given by the expected output in the terminal), but we also noticed that there are some dependencies in the code: indeed, we obtained the following guidance message `SIM 211-201A cyclic dependence prevents further acceleration of this process. This generally requires some algorithmic changes to improve`. However, we still have to remember that this is the pre-synthesis phase, so we can't expect the best performance metrics yet. As we will see, results will be good in the following stages.

C-Synthesis Here, we can see the results of the synthesis of the MLP forward pass on the FPGA. The table below shows the *Estimated Quality of results*, which is the first metric presented in the report:

TARGET	**ESTIMATED**	**UNCERTAINTY**
10.00 ns	6.329 ns	2.70 ns

Table 2: Estimated Quality of Results for MLP Forward Pass

As we can see from the table, the estimated latency is 6.329 ns, with an uncertainty of 2.70 ns. This metric provides an initial indication of the design's performance, with lower values indicating faster execution. The uncertainty value represents the range within which the actual latency is expected to fall, providing a margin of error for the estimation. This falls within the expected range for the MLP forward pass, indicating that the design should be good for efficient execution.



MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	ITERATION LATENCY	INTERVAL	TRIP COUNT	PIPELINED	BRAM	DSP	FF	LUT	URAM
forward (6)	216	2,160E3	-	217	-	no	28	50	7417	9689	0
forward_Pipeline_VITIS_LOOP_73_2 (1)	39	390.000	-	11	-	loop auto-rew	5	0	609	147	0
forward_Pipeline_VITIS_LOOP_84_4 (1)	12	120.000	-	11	-	loop auto-rew	0	0	19	81	0
forward_Pipeline_VITIS_LOOP_73_21 (1)	69	690.000	-	11	-	loop auto-rew	11	0	1245	147	0
forward_Pipeline_VITIS_LOOP_84_42 (1)	12	120.000	-	11	-	loop auto-rew	0	0	19	81	0
forward_Pipeline_VITIS_LOOP_73_23 (1)	62	620.000	-	4	-	loop auto-rew	11	0	1241	141	0
forward_Pipeline_VITIS_LOOP_84_44 (1)	5	50.000	-	4	-	loop auto-rew	0	0	103	72	0

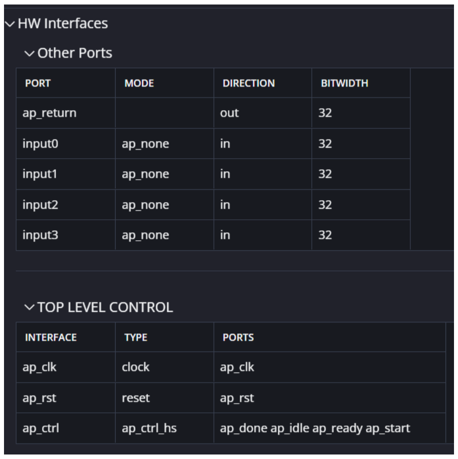
Figure 4: Perfomance and Resource Estimates in the C-Synthesis Report

From the above image, we observe that the `forward` module exhibits an overall estimated latency of 216 cycles, corresponding to an execution time of 2,160 ns under the target clock frequency of 100 MHz (10 ns per cycle). The initiation interval (II) for the main module is reported as 217 cycles, which indicates that the design could benefit from further pipelining to optimize parallel execution and reduce the interval. Examining the resource utilization, the design employs:

- **9,689 Look-Up Tables (LUTs)**, which corresponds to approximately 67.3% of the total 14,400 LUTs available on the `xc7z007s-clg225-2`.
- **7,417 Flip-Flops (FFs)**, utilizing 25.7% of the total 28,800 FFs available.
- **50 DSP slices**, accounting for 75.8% of the total 66 available DSPs.
- **No BRAM (BlockRam) or URAM(UltraRam)**, which indicates that the design relies solely on external or internal registers for storage.

The internal loops of the `forward` pass module demonstrate varied latencies, with some loops optimized using `#pragma HLS UNROLL` and `#pragma HLS PIPELINE`. Latencies range from 5 ns to 690 ns, with the primary loop exhibiting the highest latency (690 ns). This latency suggests potential bottlenecks in data dependencies or resource contention, which could be addressed by restructuring loops or leveraging more efficient parallelization strategies.

We can also check from the report that, as expected, the hardware interface corresponds to the input and output of the forward pass function, and that the utilized pragma syntax is correct and corresponds to the one we used in the code.


HW Interfaces

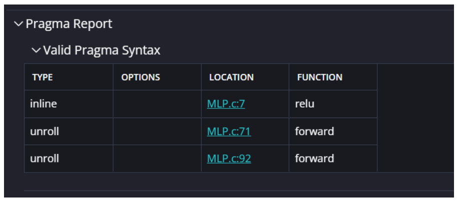
Other Ports

PORT	MODE	DIRECTION	BITWIDTH
ap_return		out	32
input0	ap_none	in	32
input1	ap_none	in	32
input2	ap_none	in	32
input3	ap_none	in	32

TOP LEVEL CONTROL

INTERFACE	TYPE	PORTS
ap_clk	clock	ap_clk
ap_rst	reset	ap_rst
ap_ctrl	ap_ctrl_hs	ap_done ap_idle ap_ready ap_start

Figure 5: Hardware interfaces


Pragma Report

Valid Pragma Syntax

TYPE	OPTIONS	LOCATION	FUNCTION
inline		MLP.c:7	relu
unroll		MLP.c:71	forward
unroll		MLP.c:92	forward

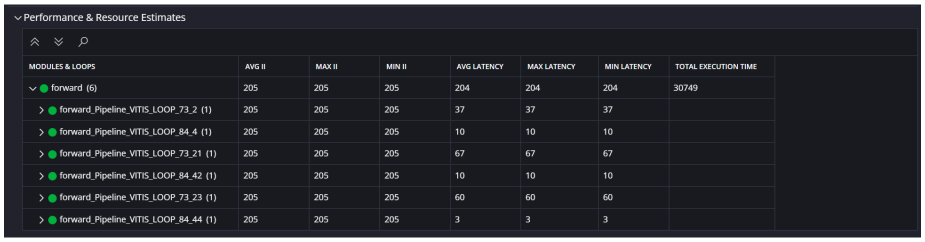
Figure 6: Pragma syntax

C/RTL Simulation C-RTL cosimulation is a verification process that ensures the functional equivalence between the high-level C/C++ design and the synthesized Register-Transfer Level (RTL) code. This step is critical as it confirms that the behavior of the RTL implementation matches the original C/C++ description after synthesis. The benefits are multiple:

- **Validation of Functional Correctness**: Verifies that the generated RTL implementation functions identically to the original high-level design for the same inputs.
- **Timing and Latency Estimates**: Provides insights into the actual timing behavior of the synthesized RTL.
- **Resource Utilization Check**: Highlights any discrepancies between resource usage reported during synthesis and actual utilization in hardware.

How It Works

1. **Input Stimuli**: A testbench written in C/C++ is used to provide input data to both the high-level C/C++ design and the RTL design.
2. **Output Comparison**: The outputs from the high-level simulation and the RTL simulation are compared.
3. **Reports**: Any mismatches or timing violations are reported for debugging purposes.


Performance & Resource Estimates

MODULES & LOOPS	AVG II	MAX II	MIN II	AVG LATENCY	MAX LATENCY	MIN LATENCY	TOTAL EXECUTION TIME
● forward (6)	205	205	205	204	204	204	30749
● forward_Pipeline_VITIS_LOOP_73_2 (1)	205	205	205	37	37	37	
● forward_Pipeline_VITIS_LOOP_84_4 (1)	205	205	205	10	10	10	
● forward_Pipeline_VITIS_LOOP_73_21 (1)	205	205	205	67	67	67	
● forward_Pipeline_VITIS_LOOP_84_42 (1)	205	205	205	10	10	10	
● forward_Pipeline_VITIS_LOOP_73_23 (1)	205	205	205	60	60	60	
● forward_Pipeline_VITIS_LOOP_84_44 (1)	205	205	205	3	3	3	

Figure 7: C/RTL Cosimulation Report - Performance and resource estimates

From an analysis of the results from the C/RTL co-simulation and synthesis, we can highlights several key observations and metrics:

- **Initiation Interval (II):** The initiation interval across all loops in the design remains constant at **205 cycles**. This uniform II suggests a consistent level of pipelining efficiency across the design. However, it also indicates that certain dependencies or resource constraints may limit further reduction of the II.
- **Loop Latencies:** The latencies for individual loops vary significantly:
  - The loop labeled `forward_Pipeline_VITIS_LOOP_73_2` exhibits an average latency of **37 cycles**, which aligns with expectations for its complexity.
  - Other loops, such as `forward_Pipeline_VITIS_LOOP_84_44`, achieve minimal latencies of just **3 cycles**, indicating highly efficient implementation.

The overall latency of the forward pass main module is **204 cycles**, which matches the expected values from the high-level design.

- **Total Execution Time:** The total execution time for the forward pass is reported as 30,749 ns. This result reflects the aggregated runtime of all components and their interactions.
- **Pipeline Observations:** While the loops in the forward pass are pipelined, the relatively high initiation interval (205 cycles) suggests potential bottlenecks. These could stem from data dependencies or limited resource availability, particularly in critical paths of the design.
- **Resource Utilization:**
  - The design effectively utilizes available DSPs, LUTs, and Flip-Flops, as previously described.
  - However, no usage of BRAM or URAM is reported. Leveraging these resources could reduce dependency on external memory and improve performance in memory-intensive operations.

Observations and Suggestions The results confirm that the design is functional and performs as expected. However, several areas for improvement are identified:

1. **Reducing II:** Efforts should be directed towards decreasing the initiation interval by addressing resource contention and loop dependencies. Techniques such as loop unrolling or splitting could be beneficial.
2. **Memory Utilization:** Introducing BRAM or URAM for intermediate data storage can minimize external memory accesses and improve throughput.
3. **Optimization of Critical Loops:** High-latency loops should be reviewed and restructured to enhance parallelism, potentially improving the overall execution time.

By implementing these optimizations, the design could achieve higher efficiency and better alignment with the hardware capabilities of the target FPGA device.

Packaging Regarding the *Package* section, there is not much to be said, since the Vitis IDE doesn't provide a report for this stage. However, we can still infer that the packaging process was successful.

Implementation Here in this section we can mainly analyze the *RTL synthesis* and the *Place and Route* stages: the first provides a detailed report on the synthesis of the design into Register-Transfer Level (RTL) code, while the latter focuses on the physical implementation of the design on the FPGA. Regarding the *RTL synthesis*, the report provides insights into the resource utilization, timing constraints, and design hierarchy. The metrics include the number of Look-Up Tables (LUTs), Flip-Flops (FFs), and Digital Signal Processors (DSPs) used, as well as the critical path delay and maximum frequency. These metrics are crucial for assessing the design's efficiency and performance, guiding further optimization efforts.

From the *Implementation Report* results, the following observations can be made:

- **Resource Utilization:** The design consumes a total of 4,438 LUTs, 5,332 FFs, 50 DSP blocks, 11 BRAMs, and 104 SRLs. Notably, no URAM, latches, or slices are utilized. This indicates an efficient use of FPGA resources without exceeding critical limits.
- **Timing Constraints:** The required clock period for the design is set to 10.000 ns, corresponding to a target clock frequency of 100 MHz. During the synthesis phase, the achieved clock period is reported as 6.872 ns, which exceeds the performance requirements and indicates that the synthesized design meets the desired constraints. However, after implementation, the achieved clock period is reported as 7.860 ns. While this is slightly higher than the synthesized value, it still satisfies the required 10.000 ns clock period.

  The increase in the clock period from synthesis to implementation can be attributed to additional routing delays and resource constraints introduced during placement and routing. These results confirm that the design meets the required timing constraints post-implementation while providing a buffer for further optimizations if needed.

3.4.3 Fail Fast Analysis

The Fail Fast analysis is a preliminary verification step designed to ensure that the design adheres to fundamental guidelines before proceeding to computationally intensive stages such as placement and routing. This stage evaluates key aspects of the design, including resource utilization and timing constraints, to identify potential bottlenecks early in the development process.
The following columns are analyzed:

- **Criteria:** Key aspects of the design, such as LUT usage, FD (Flip-Flop Density), and DFP (Dynamic Floating-Point operations), are monitored to ensure they fall within acceptable thresholds.
- **Guideline:** Defines a reference threshold for each criterion to guide the design toward optimal FPGA resource usage and performance.
- **Actual:** Displays the measured values of each criterion after the analysis of the current design.

- **State:** Indicates the compliance status of each criterion, with possible values:
  - **OK:** The criterion satisfies the guideline and requires no action.
  - **WARNING:** The criterion approaches the threshold, suggesting caution.
  - **FAIL:** The criterion exceeds the guideline, necessitating immediate attention.

Criteria Evaluated:

- **LUT Usage:** Monitors the utilization of Look-Up Tables, ensuring that logic mapping remains within device capacity.
- **Flip-Flop Density:** Assesses the distribution of flip-flops to avoid routing congestion.
- **DSP Allocation:** Evaluates the usage of DSP slices for arithmetic operations, critical for neural network implementations.
- **Timing Constraints:** Verifies whether the design meets the required clock period and ensures no timing violations.

Results: The analysis revealed that all criteria were marked as `OK`, indicating that the design complies with resource and timing guidelines. A summary of the key metrics is shown in Table 3.

Criteria	Threshold	Actual	Status
LUT Usage	14,400	3,738	OK
Flip-Flop Usage	28,800	5,364	OK
DSP Allocation	66	50	OK
BRAM Usage	50	11	OK

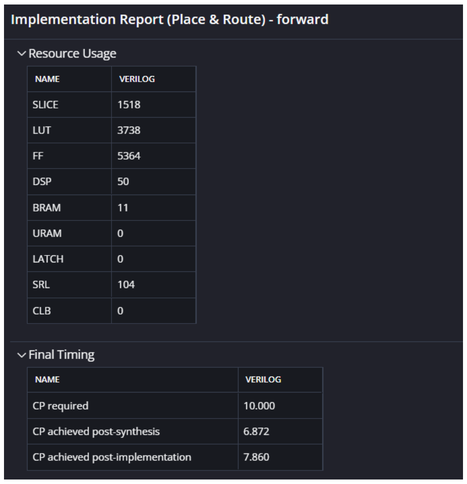
Table 3: Fail Fast Analysis Results

Observations and Recommendations:

1. **Resource Utilization:** While all resources are within acceptable thresholds, DSP allocation is relatively high at 75.8%. Future optimizations could focus on reducing DSP dependency by leveraging LUTs for simpler arithmetic operations.
2. **Timing Constraints:** The achieved clock period of 7.860 ns provides sufficient margin against the required 10.000 ns, demonstrating robust timing compliance.

3. **Critical Loops:** If further optimizations are required, consider reviewing high-latency loops to improve pipeline efficiency and reduce initiation intervals.

The combination of the *Implementation Report* and *Fail Fast* analysis highlights the robustness of the design, ensuring efficient resource utilization and adherence to timing requirements while avoiding potential pitfalls early in the development process.



Implementation Report (Place & Route) - forward

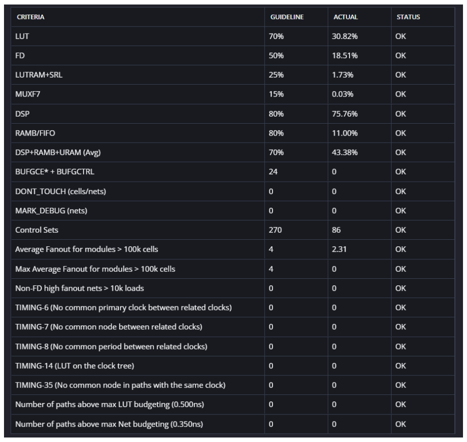
Resource Usage

NAME	VERILOG
SLICE	1518
LUT	3738
FF	5364
DSP	50
BRAM	11
URAM	0
LATCH	0
SRL	104
CLB	0

Final Timing

NAME	VERILOG
CP required	10.000
CP achieved post-synthesis	6.872
CP achieved post-implementation	7.860

Figure 8: Resource Usage and Final Timing Report in the RTL Synthesis



CRITERIA	GUIDELINE	ACTUAL	STATUS
LUT	70%	30.82%	OK
FD	50%	18.51%	OK
LUTRAM+SRL	25%	1.73%	OK
MUXF7	15%	0.03%	OK
DSP	80%	75.76%	OK
RAMB/FIFO	80%	11.00%	OK
DSP+RAMB+URAM (Avg)	70%	43.38%	OK
BUFGCE* + BUFGCTRL	24	0	OK
DONT_TOUCH (cells/nets)	0	0	OK
MARK_DEBUG (nets)	0	0	OK
Control Sets	270	86	OK
Average Fanout for modules > 100k cells	4	2.31	OK
Max Average Fanout for modules > 100k cells	4	0	OK
Non-FD high fanout nets > 10k loads	0	0	OK
TIMING-6 (No common primary clock between related clocks)	0	0	OK
TIMING-7 (No common node between related clocks)	0	0	OK
TIMING-8 (No common period between related clocks)	0	0	OK
TIMING-14 (LUT on the clock tree)	0	0	OK
TIMING-35 (No common node in paths with the same clock)	0	0	OK
Number of paths above max LUT budgeting (0.500ns)	0	0	OK
Number of paths above max Net budgeting (0.350ns)	0	0	OK

Figure 9: Fail Fast Report in the RTL Synthesis

4 Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed to process grid-like data, such as images. Their ability to automatically learn spatial hierarchies of features makes them highly effective for image classification tasks. In this implementation, a simple yet efficient CNN is designed and trained to classify handwritten digits from the famuos MNIST dataset, with the aim of achieving high accuracy while remaining compatible with hardware synthesis constraints. So the ConvNet forward pass was implemented in a similar manner to the MLP, with the main difference being the convolution and pooling operations.

4.1 Dataset

The CNN was trained on the MNIST dataset, a widely used benchmark for handwritten digit classification tasks. The dataset consists of 70,000 grayscale images of digits (0,1,2,3,4,5,6,7,8,9), each with a resolution of 28×28 pixels. The dataset was split into two subsets, with 70% used as the training set and 30% as the validation set, resulting in the following distribution:

- Training set: 42,000 samples.
- Validation set: 18,000 samples.
- Test set: 10,000 samples for evaluating the model's performance.

The dataset was preprocessed using normalization to scale pixel values to the range $[-1, 1]$, this normalization centers the pixel values around 0 and scales them to have a standard deviation of 1, which can often help with model training and convergence.

4.2 PyTorch Model

4.2.1 Model Architecture

The ConvNet is designed to be as simple as possible. The structure includes the following layers:

- **Convolutional Layer:** A single convolutional layer with 3 filters, each of size 3×3, stride 1, and padding 1. This layer increases feature representation by extracting local patterns from the input images.
- **ReLU Activation:** Applied after the convolutional layer to introduce non-linearity into the model.
- **Pooling Layer:** A max-pooling layer with a kernel size of 2×2 and a stride of 2, reducing the spatial dimensions of the feature map by half.

- **Fully Connected Layer:** It takes as input the flattened feature map from the convolutional and pooling layers. As output 10 neurons representing the 10 digit classes (0–9). This layer does not apply an activation function, it will be simply followed by `max(1)` to obtain the predicted class.

The total number of parameters in this configuration is **5880**, which is relatively low compared to more complex CNN architectures. The forward pass through the network is done as follows:

1. Apply the convolutional layer to the input image, followed by the ReLU activation function.
2. Perform max-pooling on the resulting feature map to reduce spatial dimensions.
3. Flatten the feature map into a one-dimensional vector.
4. Pass the flattened vector through the fully connected layer to produce class probabilities.

The simplicity of this architecture ensures compatibility with hardware synthesis while maintaining high accuracy for digit classification tasks.

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

```python
# Define the CNN model
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        # Input: x = [1, 28, 28]
        self.conv1 = nn.Conv2d(1, 3, kernel_size=3, stride
            =1, padding=1)
        # Convolutional Layer: Output: x = [3, 28, 28]
        # Formula: (W - F + 2P) / S + 1
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        # Pooling Layer: Output: x = [3, 14, 14]
        # Formula: (W - F) / S + 1
        self.fc1 = nn.Linear(3 * 14 * 14, 10)
        # Fully Connected Layer: Flatten the output to match
             10 classes

    def forward(self, x):
        x = torch.relu(self.conv1(x))  # Apply ReLU after
            convolution
        x = self.pool(x)               # Apply max pooling
        x = x.view(x.size(0), -1)      # Flatten the tensor
        x = self.fc1(x)                # Fully connected
            layer
        return x
```

4.2.2 Model Training

The training process for the Convolutional Neural Network (CNN) was performed using the *PyTorch* library. The objective was to minimize the cross-entropy loss, a suitable loss function for multi-class classification problems. The training configuration was as follows:

- **Loss Function:** CrossEntropyLoss.
- **Optimizer:** Adam optimizer with a learning rate of 0.001.
- **Batch Size:** 64.
- **Epochs:** 100.

```python
# initialize network, loss function and optimizer
model = ConvNet().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

After each epoch, the model was evaluated on the validation set to monitor the training progress and prevent overfitting. The training and validation results at selected epochs are shown in the table below:

Epoch	Train Loss	Train Accuracy (%)	Val Loss	Val Accuracy (%)
10	0.1194	96.49	0.1360	96.03
20	0.0907	97.30	0.1232	96.38
30	0.0735	97.74	0.1130	96.77
40	0.0636	98.03	0.1122	96.79
50	0.0571	98.27	0.1147	96.75
60	0.0521	98.34	0.1147	96.88
70	0.0491	98.39	0.1259	96.73
80	0.0459	98.53	0.1285	96.69
90	0.0439	98.55	0.1342	96.61
100	0.0411	98.71	0.1325	96.76

Table 4: Training and Validation Results at Selected Epochs

The final test performance was as follows:

- **Test Loss:** 0.1260
- **Test Accuracy:** 97.06%

Which can be considered good results, showing that the model was able to generalize well to unseen data even with only a ”few” parameters and EPOCHs of training.

The following Python code snippet shows the main training loop:

```python
# Training loop
for epoch in range(NUM_EPOCHS):
    model.train()
    train_loss = 0.0
    train_correct = 0
    train_total = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device
            )

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

        train_loss += loss.item()
        _, predicted = outputs.max(1)
        train_total += labels.size(0)
        train_correct += predicted.eq(labels).sum().item()

    train_accuracy = 100. * train_correct / train_total

    # Validation phase
    model.eval()
    val_loss = 0.0
    val_correct = 0
    val_total = 0

    with torch.no_grad():  # Disable gradient calculations
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(
                device)

            outputs = model(images)
            loss = criterion(outputs, labels)

            val_loss += loss.item()
            _, predicted = outputs.max(1)
            val_total += labels.size(0)
            val_correct += predicted.eq(labels).sum().item()

    val_accuracy = 100. * val_correct / val_total

    if (epoch + 1) % 10 == 0:
        print(f"Epoch [{epoch + 1}/{NUM_EPOCHS}], "
              f"Train Loss: {train_loss / len(train_loader):.4f}, Train Acc: {train_accuracy:.2f}%, "
              f"Val Loss: {val_loss / len(val_loader):.4f}, Val Acc: {val_accuracy:.2f}%")
```

4.2.3 Exporting Parameters

To proceed with hardware implementation, the trained parameters (weights and biases) of the CNN model were exported to a structured format so that it could be easily loaded into the C implementation. The following Python code was used to extract and store the parameters:

```python
with open('./convnet_weights.txt', 'w') as f:
    for name, weight in weights.items():
        f.write(f"// {name}, shape: {weight.shape}\n")

        if weight.ndim == 4:  # Convolutional weights: [out_channels, in_channels, kernel_height, kernel_width]
            for oc, w_slice in enumerate(weight):  # Iterate over output channels
                f.write(f"{{ // Output Channel {oc}\n")
                for row in w_slice:  # Iterate over rows (flattened kernels)
                    f.write("  {" + ", ".join(map(str, row.flatten())) + "},\n")
                f.write("},\n")

        elif weight.ndim == 2:  # Fully connected layer weights
            for row in weight:
                f.write("{" + ", ".join(map(str, row)) + "},\n")

        elif weight.ndim == 1:  # Biases or 1D weights
            f.write("{")
            f.write(", ".join(map(str, weight)) + "},\n")
```

This code snippet exports the weights and biases of the ConvNet model to the `convnet_weights.txt` file. From it, the weights and biases for each layer can be simply copy-pasted inside the `ConvNet` structure in the C implementation.

4.3 C Implementation

The implementation of the Convolutional Neural Network (CNN) model in C is divided into three main files:

- `ConvNet.h`: This header file defines the data structures and constants used in the implementation, including the convolutional and fully connected layers, and the overall network structure.
- `ConvNet.c`: This file contains the implementation of the CNN's functionality, including the forward pass and activation functions, along with predefined weights and biases.
- `testbench.c`: This file serves as a testbench to verify the network's functionality. It includes functions for reading input data, executing the forward pass, and evaluating the model's output against the expected results.

The details of each file are described in the following subsections.

4.3.1 CNN Structure

The Convolutional Neural Network (CNN) structure is implemented in C to match the architecture defined in the PyTorch model. Going into the details, the CNN consists of the following layers and operations:

- **Convolutional Layer:** This layer performs the convolution operation by *sliding* a set of filters (or **kernels**) over the input image to extract spatial features. The implementation uses nested loops to apply the filters, taking into account padding and stride.
- **ReLU Activation:** The Rectified Linear Unit (ReLU) activation function is applied after the convolutional operation to introduce non-linearity. The ReLU function replaces negative values in the feature map with zeros, its formulas was already provided in the MLP section.
- **Pooling Layer:** A max-pooling operation reduces the spatial dimensions of the feature map by selecting the maximum value within a kernel-sized region. This layer helps to downsample the feature map and make the model more robust to small spatial variations.
- **Fully Connected Layer:** The fully connected layer takes the flattened output from the pooling layer as input and maps it to the output classes. This is achieved by performing a linear transformation using the preloaded weights and biases.

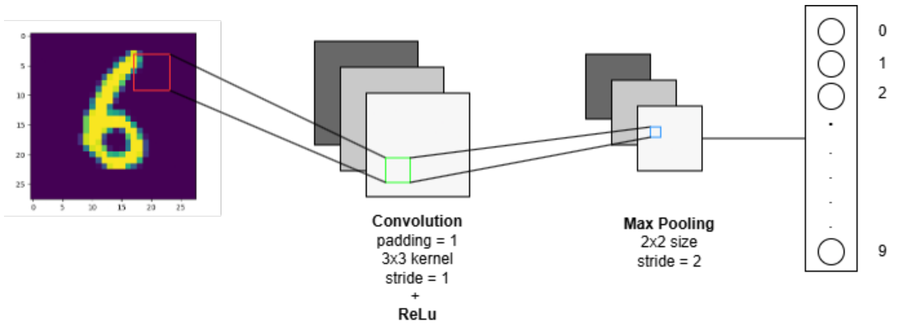


Figure 10: Convolutional Neural Network Architecture

The CNN structure is defined in the header file `ConvNet.h`, which outlines the data structures and constants used for the network. Here the ConvNet structure:

```c
// Structure to represent a convolutional layer
typedef struct {
    float weights[CONV1_OUTPUT_CHANNELS][INPUT_CHANNELS][3][3]; // Filters of the convolutional layer
    float biases[CONV1_OUTPUT_CHANNELS]; // Biases for the filters
} ConvLayer;

// Structure to represent a fully connected layer
typedef struct {
    float weights[NUM_CLASSES][FC1_INPUT_SIZE]; // Weights of the fully connected layer
    float biases[NUM_CLASSES]; // Biases of the fully connected layer
} FullyConnectedLayer;

// General structure of the network
typedef struct {
    ConvLayer conv1; // First convolutional layer
    FullyConnectedLayer fc1; // Fully connected layer
} ConvNet;
```

As for the MLP implementation, all weights and biases in the CNN are hard-coded and directly integrated into the `ConvNet.c` file, where they can be found at the top.

4.3.2 Forward Pass

The forward pass function propagates an input image through the Convolutional Neural Network (CNN) to generate class probabilities as the output. It sequentially applies operations such as convolution, activation, pooling, and fully connected layers.

The following C code implements the forward pass:

```c
int forward(float input[INPUT_HEIGHT][INPUT_WIDTH][INPUT_CHANNELS], float output[NUM_CLASSES]) {

    // Convolutional layer output buffer
    float conv_output[CONV1_OUTPUT_CHANNELS][INPUT_HEIGHT][INPUT_WIDTH];

    // Array partitioning pragmas for optimization in hardware (HLS-specific)
    #pragma HLS ARRAY_PARTITION variable=input complete dim=3
    #pragma HLS ARRAY_PARTITION variable=convnet.conv1.weights complete dim=1
    #pragma HLS ARRAY_PARTITION variable=conv_output complete dim=1

    // Convolutional layer operation
    // Loop over output channels
    for (int oc = 0; oc < CONV1_OUTPUT_CHANNELS; oc++) {
        // Loop over input height
        for (int h = 0; h < INPUT_HEIGHT; h++) {

            // Temporary buffer for input data (with padding)
            // Kernel size is 3x3, so a 5-row buffer is needed
            float temp_input[5][INPUT_WIDTH+2][INPUT_CHANNELS];
            #pragma HLS ARRAY_PARTITION variable=temp_input complete dim=3

            // Load the input data into the buffer with padding
            // Load 5 rows of input data
            for (int i = 0; i < 5; i++) {
                // Adjust row index for kernel centering (-2 for padding)
                int curr_h = h + i - 2;
                // Input width with padding
                for (int w = 0; w < INPUT_WIDTH + 2; w++) {
                    #pragma HLS PIPELINE II=1
                    // Loop over input channels
                    for (int c = 0; c < INPUT_CHANNELS; c++) {
                        // Apply zero-padding for boundary conditions
                        if (curr_h >= 0 && curr_h <
                            INPUT_HEIGHT && w-1 >= 0 && w-1 < INPUT_WIDTH) {
                            temp_input[i][w][c] = input[curr_h][w-1][c];
                        } else {
                            temp_input[i][w][c] = 0.0f;
                        }
                    }
                }
            }

            // Perform convolution for the current row of the output
            for (int w = 0; w < INPUT_WIDTH; w++) {
                #pragma HLS PIPELINE II=1
                // Initialize with bias value
                float sum = convnet.conv1.biases[oc];

                // Convolve the kernel with the input buffer
                // Loop over input channels
                for (int ic = 0; ic < INPUT_CHANNELS; ic++) {
                    // Kernel height
                    for (int kh = 0; kh < 3; kh++) {
                        // Kernel width
                        for (int kw = 0; kw < 3; kw++) {
                            sum += temp_input[kh + 1][w + kw][ic] * convnet.conv1.weights[oc][ic][kh][kw];
                        }
                    }
                }
                // Apply ReLU activation function
                conv_output[oc][h][w] = reLu(sum);
            }
        }
    }

    // MaxPooling layer output buffer
    float pool_output[CONV1_OUTPUT_CHANNELS][INPUT_HEIGHT / POOL_SIZE][INPUT_WIDTH / POOL_SIZE];

    // MaxPooling operation
    // Loop over output channels
    for (int oc = 0; oc < CONV1_OUTPUT_CHANNELS; oc++) {
        // Loop over pooled height
        for (int h = 0; h < INPUT_HEIGHT / POOL_SIZE; h++) {
            // Loop over pooled width
            for (int w = 0; w < INPUT_WIDTH / POOL_SIZE; w++) {
```

```c
                // Initialize with a very small value
                float max_val = -1e9;
                // Pooling window height
                for (int ph = 0; ph < POOL_SIZE; ph++) {
                    // Pooling window width
                    for (int pw = 0; pw < POOL_SIZE; pw++) {
                        // Calculate input height index
                        int ih = h * POOL_SIZE + ph;
                        // Calculate input width index
                        int iw = w * POOL_SIZE + pw;
                        // Ensure within bounds
                        if (ih < INPUT_HEIGHT && iw < INPUT_WIDTH) {
                            // Find max value in window
                            if (conv_output[oc][ih][iw] > max_val) {
                                max_val = conv_output[oc][ih][iw];
                            }
                        }
                    }
                }
                // Store max value in pooled output
                pool_output[oc][h][w] = max_val;
            }
        }
    }

    // Fully connected layer input buffer
    float fc_input[FC1_INPUT_SIZE];

    // Flatten pooling output into a 1D array
    // Index for flattened array
    int idx = 0;
    // Loop over output channels
    for (int oc = 0; oc < CONV1_OUTPUT_CHANNELS; oc++) {
        // Loop over pooled height
        for (int h = 0; h < INPUT_HEIGHT / POOL_SIZE; h++) {
            // Loop over pooled width
            for (int w = 0; w < INPUT_WIDTH / POOL_SIZE; w++) {
                fc_input[idx++] = pool_output[oc][h][w];
            }
        }
    }

    // Fully connected layer computation
    // Loop over output classes
    for (int o = 0; o < NUM_CLASSES; o++) {
        // Initialize with bias value
        float sum = convnet.fc1.biases[o];
        #pragma HLS PIPELINE II=1
        // Loop over flattened input
        for (int i = 0; i < FC1_INPUT_SIZE; i++) {
            // Weighted sum of inputs
            sum += fc_input[i] * convnet.fc1.weights[o][i];
        }
        // Store the computed class score
        output[o] = sum;
    }
    return 0; // Success
}
```

4.3.3 Considerations on Pragmas and Forward-pass Code

1. **Array Partitioning (`#pragma HLS ARRAY_PARTITION`)**: The use of the `ARRAY_PARTITION` pragma is important for improving parallelism, especially in operations that require concurrent access to multiple data elements. Partitioning arrays into independent variables, as in the case of the input array, convolution weights, and convolution output, allows the design to access different data in parallel without conflicts, improving efficiency. Completing the partition for each dimension of the array means that each element of the array is mapped to a separate hardware resource, enabling simultaneous access to the data.

2. **Pipeline (`#pragma HLS PIPELINE`)**: The `PIPELINE` pragma is used to insert pipelining commands, which means operations can begin executing in parallel. The directive `II=1` reduces the initiation interval to 1 clock cycle, speeding up loop execution and reducing latency.

   This is particularly useful in loops where the operations are independent, such as in the convolution and max-pooling calculations.

3. **Efficiency of Convolution and Max-Pooling Operations**: Convolution and pooling operations are performed in separate, parallelized loops, utilizing temporary buffers like `temp_input` to manage padding and data access. This approach reduces latency and enhances the overall throughput of the design.

4. **Memory Efficiency and Buffer Management**: The use of temporary buffers to store intermediate results, such as `temp_input` and `conv_output`, is a common strategy to improve the efficiency of operations. However, it's essential to properly manage the buffer sizes to avoid memory conflicts and resource overloads.

4.3.4 Testbench

The testbench reads input data from a file, processes it through the CNN using the `forward` function, and verifies the output against the expected label.

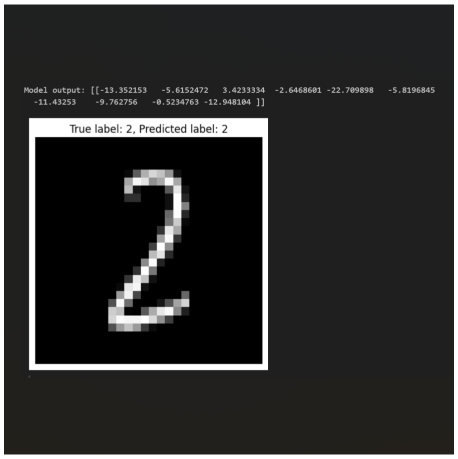
This ensures that the implementation behaves as expected and aligns with the PyTorch model. The testbench performs the following tasks:

- **Input Loading**: Reads the input image and its corresponding label from a text file (`input_image.txt`), produced by the Jupyter Notebook. This has to be loaded inside the *testbench* section of the Vitis IDE.
- **Forward Pass Execution**: Propagates the input through the CNN to compute class probabilities.
- **Result Validation**: Compares the predicted label with the true label and outputs the result.

The following code showcases the implementation of the testbench:

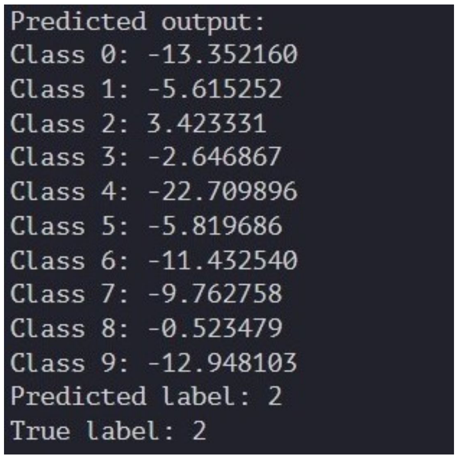
```c
int main() {
    float input[INPUT_HEIGHT][INPUT_WIDTH][INPUT_CHANNELS];
    float output[NUM_CLASSES];
    int label;

    read_input_image(INPUT_FILE_PATH, input, &label);

    int results = forward(input, output);

    if (results != 0) {
        printf("Error during forward pass\n");
        return 1;
    }

    printf("Predicted output:\n");
    for (int i = 0; i < NUM_CLASSES; i++) {
        printf("Class %d: %f\n", i, output[i]);
    }

    float max_prob = output[0];
    int predicted_label = 0;

    for (int i = 1; i < NUM_CLASSES; i++) {
        if (output[i] > max_prob) {
            max_prob = output[i];
            predicted_label = i;
        }
    }
    printf("Predicted label: %d\n", predicted_label);
    printf("True label: %d\n", label);

    return 0;
}
```

Apart from the `read_input_image` (not reported here for brevity) we can see that the code follows the workflow explained above: reads an input image, passes it to the forward function, and then compares the predicted label with the true label. With PyTorch, the model prediction was the following:



```
Model output: [[-13.352153   -5.6152472   3.4233334  -2.6468601 -22.709898   -5.8196845
  -11.43253    -9.762756   -0.5234763 -12.948104 ]]
```

Figure 11: PyTorch Model Prediction



```
Predicted output:
Class 0: -13.352160
Class 1: -5.615252
Class 2: 3.423331
Class 3: -2.646867
Class 4: -22.709896
Class 5: -5.819686
Class 6: -11.432540
Class 7: -9.762758
Class 8: -0.523479
Class 9: -12.948103
Predicted label: 2
True label: 2
```

Figure 12: Prediction inside Vitis IDE

As we can see, results are equal to the third decimal digit, which is a good sign that the C implementation is working as expected.

4.4 Results

////To be implemented after further clarifications by professor