SEAI_2024_R12

Giulio Capecchi, Jacopo Niccolai

December 2024

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 (Field Programmable Gate Array). 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

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 plays a pivotal role in synthesizing the neural network architectures—MLP, ConvNet, and Transformer—onto the 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 streamlines the workflow, bridging the gap between software and hardware development.

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.

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 hardware area usage since 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 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.

These directives were applied strategically to balance trade-offs between latency, resource utilization, and overall performance. For example, `HLS INLINE` was used in frequently called small functions to minimize latency, while `HLS PIPELINE` was applied to loops processing neural network layers to maximize parallelism and throughput. The appropriate use of these directives is crucial for achieving efficient FPGA-based implementations.

Hardcoding Network Parameters
The network parameters (weights and biases) exported from the trained PyTorch model were directly integrated into the *C* implementation in a hardcoded manner. This process ensures that the FPGA hardware has direct access to pretrained values during the forward pass, avoiding the need for external memory accesses and thereby reducing latency.
The specific details of how the weights and biases were exported, including the Python code used to save them into a text file, can be found in Section 4.2.3

TODO(check the correct section number). These parameters were then structured in the `C` code as arrays within the MLP structure, as defined in Section 4.3.1 TODO(check the correct section number). This approach, while limiting the flexibility of re-training or updating the model without recompilation, was chosen to optimize execution time and reduce complexity in hardware synthesis.

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:

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

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

```
# 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.

```
model = MLP()

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

```
6
7  criterion = nn.CrossEntropyLoss()
8  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.

```
1  # Training loop
2  NUM_EPOCHS = 100
3  for epoch in range(NUM_EPOCHS):
4      model.train()
5      running_loss = 0.0
6      for inputs, labels in train_loader:
7          inputs, labels = inputs.to(device), labels.to(device
               )
8
9          optimizer.zero_grad()
10         outputs = model(inputs)
11         loss = criterion(outputs, labels)
12         loss.backward()
13         optimizer.step()
14
15         running_loss += loss.item()
16
17     # Evaluate the model after each epoch
18     model.eval()
19     correct = 0
20     total = 0
21     with torch.no_grad():
22         for inputs, labels in test_loader:
23             inputs, labels = inputs.to(device), labels.to(
                   device)
24             outputs = model(inputs)
25             _, predicted = torch.max(outputs.data, 1) # Get
                   the class index with the highest probability
26             total += labels.size(0)
27             correct += (predicted == labels).sum().item()
28
29     accuracy = correct / total
30
31     if (epoch + 1) % 10 == 0:
32         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, *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}")

with open('./mlp_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`):

```
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:

```
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;
}
```

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:

```
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]);
    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. 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 key 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*, ensuring compatibility with the Zynq-7000 device and facilitating synthesis and implementation with minimal adjustments. Using the default clock setting allowed us to focus on optimizing other aspects of the design, such as resource utilization and latency, while maintaining a reliable and stable timing configuration.
The following results will demonstrate how this default clock period fits well with our solution. Notably, we achieved efficient performance without the need to modify the default clock setting, highlighting the suitability of the 10 ns period for our design goals and hardware constraints.

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.

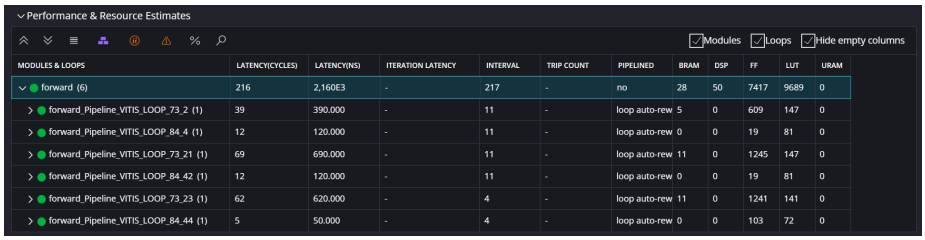
C-Simulation C-simulation provides preliminary performance metrics, focusing on the steady-state execution of the design. These estimates, including the Transaction Interval (TI), highlight potential bottlenecks and optimization areas but may be overly optimistic unless the code is made canonical. This 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 2: 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 or URAM**, which indicates that the design relies solely on external or internal registers for data 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.
To further enhance performance, improvements such as increasing the use of internal memory resources (e.g., BRAM/URAM) and reducing loop dependencies are recommended. These adjustments would help lower the initiation interval and improve overall throughput, making the design more efficient for hardware acceleration.

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.

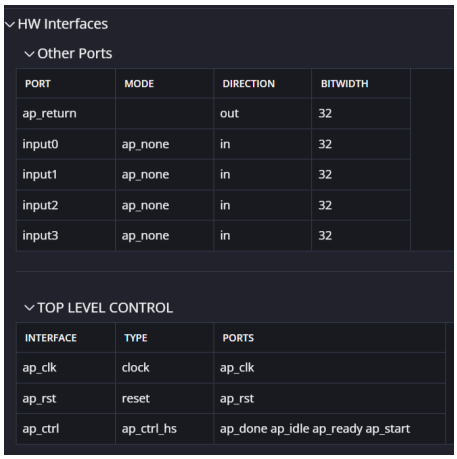


Figure 3: Hardware interfaces

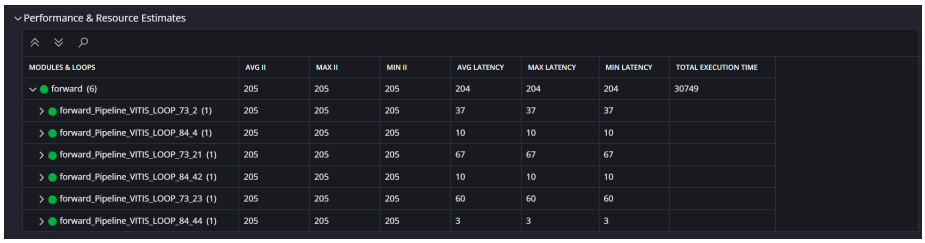
Figure 4: Pragma syntax

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.



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 5: C/RTL Cosimulation Report - Performance and resource estimates

4.4.3 Analysis of Results

The analysis of the results from the C/RTL co-simulation and synthesis reports 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, as the design was ready for the final implementation stage.

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.

4.4.4 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 6: 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 7: Fail Fast Report in the RTL Synthesis

5 Convolutional Neural Network (ConvNet)

The ConvNet forward pass was implemented in a similar manner to the MLP, with the main difference being the convolution and pooling operations. The ConvNet was trained on the famous *MNIST* dataset, which contains 70.000 32x32 black and white images in 10 classes (representing hand-written digits).