Mapping High Level DL Models to RTL

B.Tech Project Report

Submitted by

Ayush Gangwar (B20CS008) and Gojiya Piyush (B20CS015)

Under the Supervision

Of

Dr. Binod Kumar

Department of Computer Science and Engineering

Indian Institute of Technology Jodhpur

January, 2024

Background

The rapid growth of deep learning applications has led to an increasing demand for the execution of complex models on resource-constrained devices that requires a shift from traditional software implementations to hardware-accelerated solutions. Presently, Deep learning models are often developed in high-level programming languages such as Python, and the transition to hardware execution on FPGAs is crucial for addressing constraints related to power efficiency and scalability . This project aims to explore the mapping of high-level DL models to Register Transfer Level (RTL), focusing on maintaining functionality while leveraging the benefits of FPGA architectures.

Objectives

1. **Language Transition:** Accurately convert Python-based deep learning models into efficient C++ representations compatible with hardware acceleration. Ensure that the translated models accurately preserve the mathematical operations, architecture, and overall functionality of the original Python models. The C++ codebase is to be designed in a modular and readable manner to facilitate future modifications and improvements.
2. **Quantization for Efficiency:** Implement weight quantization to enhance mathematical operations and optimize memory usage in deep learning models.
3. **Integration with FPGA:** Seamlessly integrate C++ code with FPGA architecture, utilizing platform-specific features for efficient resource utilization.
4. **Performance Evaluation:** Evaluate hardware-accelerated models against Python-based implementations, analyzing metrics like inference time, accuracies.

Possible methods to solve the problem

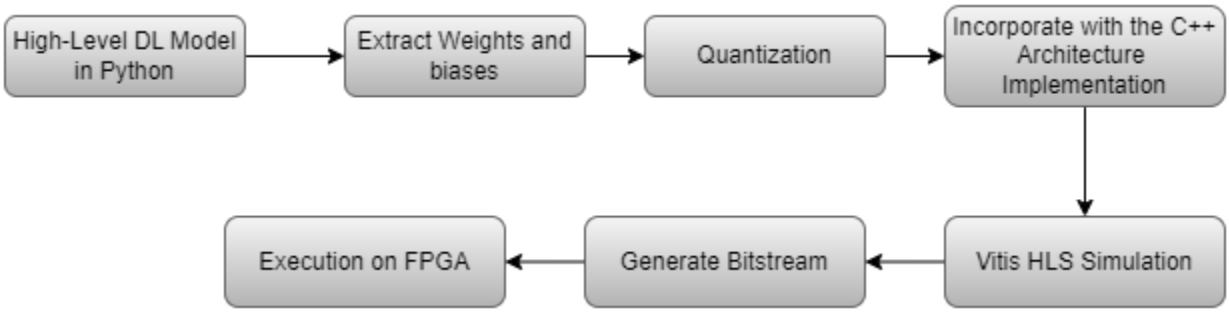
1. **Hardware Description Language (HDL) Translation**: Converting the high-level deep learning model description into a hardware description language (HDL) like Verilog or VHDL i.e defining hardware components such as neurons, activation functions, convolutional layers, and fully connected layers using HDL. This approach is effective for manual hardware design but may be time-consuming as well as quite complex to implement.

2. **High-Level Synthesis (HLS)**: Utilize high-level synthesis tools to automatically generate RTL code from high-level descriptions. Tools like Vivado HLS or Catapult C automatically translate high-level C/C++ code into RTL. This approach reduces manual effort and accelerates the hardware design process.

3. **Tensor Processing Units (TPUs) or AI Accelerators**: Leverage specialized hardware accelerators designed for deep learning workloads. Google's TPU, NVIDIA's GPU, or other AI-specific accelerators are designed to perform matrix multiplications efficiently. Integrating such accelerators can enhance the speed and efficiency of the hardware implementation but is expensive.

Approach

Block Diagram of Process Flow:



Detailed steps:

- Develop the high-level deep learning model in Python using a deep learning framework such as TensorFlow or PyTorch.
- Train and fine-tune the model on a dataset to achieve desired accuracy and later extract the trained weights and biases from the high-level model in Python.
- Quantize the weights to reduce memory requirements and enable more efficient hardware operations.
- Implement the architecture of the neural network in C++ taking care of hardware constraints, integrating the quantized weights to represent the hardware-friendly model.
- Utilize Xilinx Vitis HLS to simulate the hardware environment and validate the functionality of the C++ code of the high-level model.
- Generate the FPGA bitstream and load the generated bitstream onto the target FPGA using Vivado.
- Evaluate the performance of the hardware-accelerated neural network on the FPGA in terms of speed and accuracies.

Implementation

We have implemented a generic framework for mapping high-level deep learning models to Register Transfer Level (RTL). Different layers like convolution, max pooling, flatten, dropout, fully connected layers and activation functions like Relu, softmax, sigmoid have been implemented in C++ language. Convolution layer has functionalities of performing convolution operations on images with multiple channels(RGB) and other parameters such as stride, padding, number of kernels, etc. Weights and biases from pre-trained models are used to perform convolution operation and in fully connected layers to generate the inference from the model in an efficient manner.

Specifically models like MLP, CNN and VGG16 have been implemented using this framework but it provides a foundation for extending the implementation to other high level deep learning models beyond the mentioned examples. The generic nature of the design allows seamless integration with various deep learning architectures.

Results

MNIST :
Task: Handwritten Digit Recognition
Architecture: multi-layer perceptron comprising input layer, two hidden layers with 128 and 256 nodes respectively and finally an output layer with 10 outputs.

System	Inference Time (seconds)	Accuracy (%)
Software(python)	0.35	99
Vitis HLS	0.13	94

Performance on Vitis HLS

BRAM	DSP	FF	LUT
122	394	28584	25590

CIFAR-10 :
Task: Object Recognition in Natural Scenes
Architecture: fourteen layers, featuring pairs of convolutional and max-pooling layers (Conv2D and MaxPooling2D) leading to a flattening layer. The convolutional layers use varying filter

sizes. Dropout is applied before two dense layers, with the final layer having ten units for the output.

System	Inference Time (seconds)	Accuracy (%)
Software(python)	0.14	81
Vitis HLS	0.09	73.1

Performance on Vitis HLS

BRAM	DSP	FF	LUT
508	193	48674	24821

CT Scan Images :

Task: Medical Image Analysis

Architecture: six layers beginning with a Conv2D(50, 5x5), followed by a MaxPooling2D layer. The third layer is another Conv2D(30, 4x4), followed by a MaxPooling2D layer. Next is a Flatten layer, and the sixth layer is a Dense layer with 2 units for the output.

System	Inference Time (seconds)	Accuracy (%)
Software(python)	0.77	98
Vitis HLS	0.606	89

Performance on Vitis HLS

BRAM	DSP	FF	LUT
13216	187	58753	29594

Hand Gesture:

Task: Gesture Classification

Architecture: ten layers, starting with a Conv2D layer (8, 5x5), followed by MaxPooling2D and Dropout. Another Conv2D layer (16, 3x3) is followed by MaxPooling2D and Dropout. Next Flatten layer, followed by a Dense layer and a Dropout layer. Final is a Dense layer with 10 units for the output.

System	Inference Time (seconds)	Accuracy (%)
Software(python)	0.31	99
Vitis HLS	0.22	91.4

Performance on Vitis HLS

BRAM	DSP	FF	LUT
332	126	42798	20956

Fruit360 :

Task: Object Classification

Architecture: VGG16. Comprises 21 layers, starting with a block of convolutional and max-pooling layers. 2-3 convolution layers and a max pooling layer together form subsequent blocks. The model concludes with a flatten layer and a dense layer with 131 output units, having 2048 input units.

System	Inference Time (seconds)	Accuracy (%)
Software(python)	5.17	82.9
Vitis HLS		61.3

Performance on Vitis HLS

BRAM	DSP	FF	LUT
13216	187	58753	29594

Conclusion

The project successfully delivered a versatile C++ framework for high-level models, transferable to RTL for execution on diverse hardware. Emphasis on memory optimization through quantization maintains accuracy.

Future work

- Parallelization and Pipelining: Introduce data pipelining and instructions pipelining , Implement parallel computation for task distribution and improved efficiency.
- Pragma Integration: Integrate pragma directives to guide compiler optimizations for targeted performance improvements.
- C++ Inbuilt Libraries: Leverage existing C++ libraries (e.g., Eigen, MKL, Dlib) for optimized matrix calculations and other computations.

References

1. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls
2. http://dlib.net/
3. https://eigen.tuxfamily.org/index.php?title=Main_Page
4. https://ieeexplore.ieee.org/document/9286705