Neural Networks : From Python To FPGA

Hugo BABIN-RIBY[1]

[1] ISAE Supaero & DUNCHA France, SoC &
FPGA Engineering

Sep 2024

Course objectives

Objective

At the end of this course, you will be able to train and deploy a Quantized Neural Network on FPGA.

You will also develop key skills including, but not limited to :

- Train & quantize a Neural Network & provide relevant analysis.
- Manipulate your Neural Network for the end use case.
- Use tools to generate HDL and HLS code depending on the requirements.
- Deploy the end solution to **any** Xilinx FPGA.

Course overview

- Lecture 1 : Basic concepts (1 hour)
  - Neural Networks Overview & Examples
  - Quantization Overview & Examples
  - Hardware overview
- Lecture 2 : Advanced overview (2 hours)
  - Workflow & Framework (Presentation + Examples)
  - FINN Presentation
  - Hardware advanced overview
- Labs
  1. Model training & Quantization (2 hours)
  2. Conversion to HW Layer (2 hours)
  3. Manual Zynq inference. (2 hours)
- Course GitHub Here (click)

Python to FPGA : Accelerating Neural Networks

Context

In 2024, the AI industry reached a US$184.00bn market cap. This growth creates numerous research and professional opportunities. However, efficient NN training and inference on hardware requires skilled engineers & researchers.

Stakes

As an engineer, understanding how to create custom hardware end-to-end AI solutions and mastering efficient tools is becoming increasingly essential.

This course aims to bridge the gap between high-level Python programming and FPGA implementation for neural networks and help you stand out using efficient tools alongside an acute understanding of NN for FPGA concepts.

Lecture Summary

1. Basic Reminders
   - Neural Networks : Basics + Classifier Example
   - Hardware implementation introduction
   - Quantization : The best trade-of for FPGAs
   - Hardware overview

2. Project Concepts

Neural Networks Vocabulary

Our use case

Neural Networks (NNs) are used for **classification**.

Architecture

They are made of **layers** with *n* inputs and *m* outputs.

Layers ?

Layers takes as an input the output of the previous layers. Each output is called the **activation** of a ”neuron”. They consist in a basic Matrix-Vector multiplication.

Neural Networks basics

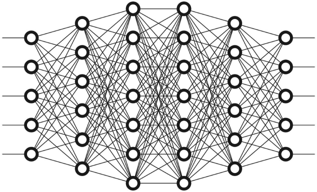


Figure – Neural Network : Nodes and Weights

Definition

Middle layers are also called *”hidden”* as their I/O has nothing to do with the initial data or our classification classes. (Works as a black box)

Neural Networks basics

Layers types

There are multiple type of layers, involving different types of computations.

- Fully connected (or linear)
- Convolution layer
- Max pooling
- dropout
- Batch norm
- ...

Neural Networks basics

Architectures types

There are many architectures type, involving **different layer types**.

The goal is to find the best network for your use case that yields the best :

- Accuracy
- Number of parameters
- Computing efficiency

The (widely known) Example of MNIST

- Goal of MNIST : Classify handwritten number form 0 to 9.
- Data type : 28x28 pixels (784), each UINT8 (0 to 255), FP32 (0 to 1), ...

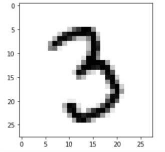


Figure – MNIST Number

Example of architecture ?

IN ? OUT ? Hidden Layers ?

The (widely known) Example of MNIST

- Goal of MNIST : Classify handwritten number form 0 to 9.
- Data type : 28x28 pixels (784), each UINT8 (0 to 255), FP32 (0 to 1), ...

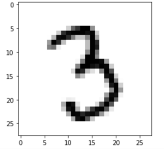


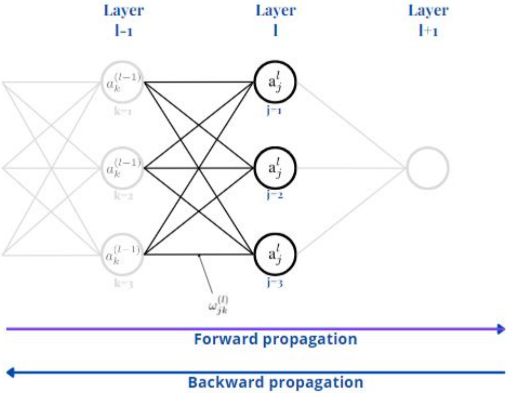
Figure – MNIST Number

Example

Let's review a quick example in Python using Pytorch...

Intro to the Hardware Acceleration

Here is a basic fully connected network :



Intro to the Hardware Acceleration

The math behind a single node from the previous slide [deeplizard 2018] :

$$a_j^l = \sigma\left(\sum_{k=1}^{n} w_{jk}^{(l-1)} a_k^{(l-1)} + b_k\right) \tag{1}$$

Or in a simpler tensor notation :

$$y = Ax + B \tag{2}$$

First analysis

This is a very simple Multiply Accumulate Operation (MAC), easily accelerated via hardware.

Intro to the Hardware Acceleration

Implementation on hardware

First we train the model to get the weights, Then we run inference on FPGA using custom hardware.

Example

Let’s try to sketch a basic node in hardware... (refer to the next slide)

Basic Hardware Node

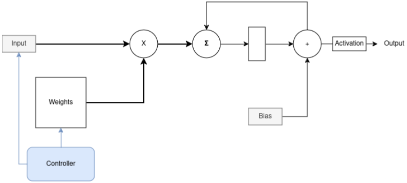


Figure – A Basic Node Implementation

Simple node

We have to organize in layers using shared memory for the weights.

Concerns

Number of clock cycles ? *(Folding)* Data type of *Input* ? *Weights* ? *Output* ? How to handle overflow ? etc ...

Hardware Nodes Conclusion

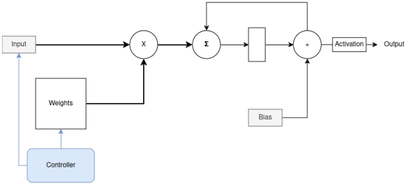


Figure – A Basic Node Implementation

Conclusion...

Easy on paper but requires careful thinking to implement and is not an ”agile” workflow (good for industry work).

⇒ The next course will list tools to automate this workflow.

Many Concerns...

We have concerns... How are those problems solved ?

Data types

Typically, weights are floating point data type, resulting in Floating Point Operations (FOPs) requiring specialized hardware.

Workflow

This ”handmade” workflow is far from agile and is likely to introduce bugs.

Solution ?

In the next lecture, we will explore the tools available to make our lives easier.

Quantization [Jamil 2023]

Quantization

Changing the data type (e.g. from *FP* to *INT*) and reducing precision (from 32*b* to 1*b*).

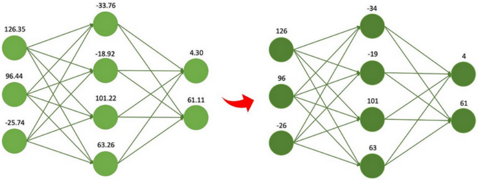


Figure – Quantization

Vector quantization

Asymmetric Quantization

Compute q, the quantized vector, with $z \neq 0$ and $\alpha \neq -\beta$ i.e. range = $[\beta, \alpha]$.

$$q = \left\lfloor \frac{x}{s} + z \right\rfloor \tag{3}$$

$$s = \frac{\alpha - \beta}{2^n - 1} \tag{4}$$

$$z = \left\lfloor -1 * \frac{\beta}{s} \right\rfloor \tag{5}$$

Dequant :

$$x = s \cdot (q - z) \tag{6}$$

With α and β max and min values of the source vector : x.

Vector quantization

Symmetric Quantization

Compute q, the quantized vector, with $z = 0$ and $\alpha = -\beta$ i.e. range = $[-\alpha, \alpha]$.

$$q = \left\lfloor \frac{x}{s} + z \right\rfloor \tag{7}$$

$$s = \frac{|\alpha|}{2^n - 1} \tag{8}$$

Dequant :

$$x = s \cdot q \tag{9}$$

With α and β max and min values of the source vector : x.

Quantization facts

- Symmetric or not, the choice is ours to make !
- Both introduces error.
- Asymmetric
  - Better for $\beta \neq \alpha$
  - Maps $[\beta, \alpha] \rightarrow [0, 2^n - 1]$
- Symmetric
  - Better for $\beta \simeq \alpha$
  - Maps $[-\alpha, \alpha] \rightarrow [-2^{n-1} - 1, 2^{n-1} - 1]$

Values

Quantization works as a ”translation”, values **does** change ! if you want to recover actual values, you have to de-quantize the data (e.g. $1_{FP32} \neq 255_{UINT8}$)

Quantization : Hands on example

Quantization

Quantization is not limited to AI.

We can use quantization to represent any vector in a lower bit resolution !

Example

Let's see a quick example !

Quantization for NNs (already trained)

Post Training Quantization (PTQ)

You already trained an AI model, you have all the weights and would like to quantize it ?

1. Add observers to you model
2. NN Inference using regular data
3. observers will pick up data for quantization
4. You can now quantize using torch built-in API calls

Example

Let's practice NN Post Training Quantization ourselves in PyTorch !

Quantization for NNs

Quantization Aware Training (QAT)

You want to train a brand new quantized model ? Let’s train it so it becomes more robust to quantization !

1. Same principles : observers
2. We introduce ”fake” quant error
3. The model adapts
4. The model become more robust to quant error

How ?

Example in the next lecture (Brevitas chapter).

Hardware overview

The following slides will go over the following hardware concepts :

- The System On a Chip (SoC) we'll use for FPGA inference.
- An overview of the future strategies for resources usage.

A Word on ZYNQ [Crockett 2014]

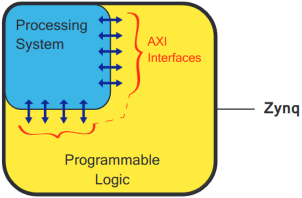


Figure – Zynq simplified (image for Zynq Book)

How will we use it ?

The Zynq's FPGA will be programmed with our NN converted into a HDL IP.
We will use the ARM CPU to move data in an out of the model.

(More on the communication protocols in the next course.)

Hardware used throughout the course

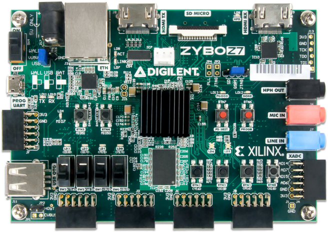


Figure – Zybo Z7-20 Board

Concerns

Small board : Resources usage ?

Loop Unrolling

Unrolling

Choosing the number of operations / elements to process in parallel, wrt. specs. (e.g. LUT availability in small boards).

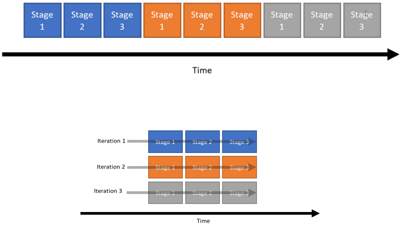
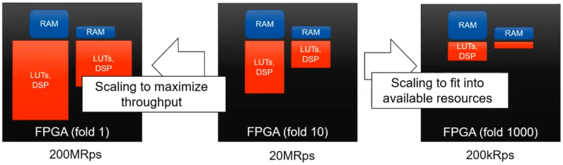


Figure – Rolled VS Unrolled Loop (image from Xilinx)

Loop Unrolling & Folding

Unrolling

Unrolling is a compromise between throughput and resources utilization.



Scaling to maximize throughput

Scaling to fit into available resources

Later...

More on this later in LAB 2 when using the FINN Compiler.

Folding parameters effects

Here are the effect of folding parameters on Binary Neural Networks (BNNs). [Umuroglu 2017]

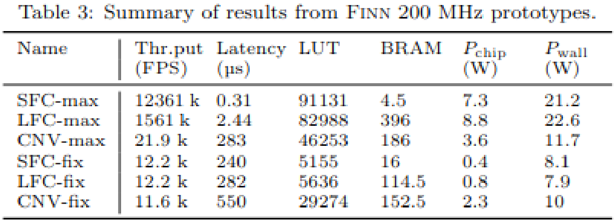


Table 3: Summary of results from FINN 200 MHz prototypes.

Name	Thr.put (FPS)	Latency (μs)	LUT	BRAM	P_{chip} (W)	P_{wall} (W)
SFC-max	12361 k	0.31	91131	4.5	7.3	21.2
LFC-max	1561 k	2.44	82988	396	8.8	22.6
CNV-max	21.9 k	283	46253	186	3.6	11.7
SFC-fix	12.2 k	240	5155	16	0.4	8.1
LFC-fix	12.2 k	282	5636	114.5	0.8	7.9
CNV-fix	11.6 k	550	29274	152.5	2.3	10

Lecture Summary

1. Basic Reminders
2. Project Concepts
    - General workflow for ”Python to FPGA”
    - PyTorch, Brevitas & ONNX
    - HLS
    - The FINN Framework
    - FINN Workflows & Example
    - AXI & DMA For inference
    - Useful informations

Workflow overview

The Workflow

We will go through several steps.
This lecture is about providing details on all of these steps and the corresponding tools.

1. Create and train a quantized model (PyTorch, Brevitas)
2. Prepare the model's layers for FINN (FINN, ONNX)
3. Conversion to Stitched IP
4. Prepare a Vivado project
5. Prepare the software in Vitis

PyTorch

Presentation

PyTorch is an open-source library for tensor computations and deep learning.
It excels in GPU-accelerated tensor operations and automatic differentiation.

Example

We already did an example in Lecture 1 (MNIST Classifier).

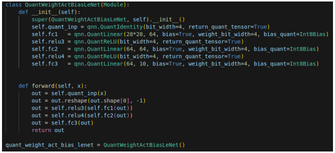
Brevitas [Pappalardo 2023]

Presentation

Brevitas (Xilinx Software) is an open-source library for quantization-aware training in PyTorch.
It allows creation of reduced-precision neural networks (good for FPGA).

- Integrates perfectly with other tools like FINN (described later).
- Handles all the quantization automatically for us.
- Handles QAT .

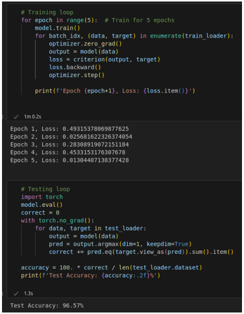
Brevitas Example



```python
class QuantWeightActBiasLeNet(Module):
    def __init__(self):
        super(QuantWeightActBiasLeNet, self).__init__()
        self.quant_inp = qnn.QuantIdentity(bit_width=4, return_quant_tensor=True)
        self.fc1   = qnn.QuantLinear(28*28, 64, bias=True, weight_bit_width=4, bias_quant=Int8Bias)
        self.relu3 = qnn.QuantReLU(bit_width=4, return_quant_tensor=True)
        self.fc2   = qnn.QuantLinear(64, 64, bias=True, weight_bit_width=4, bias_quant=Int8Bias)
        self.relu4 = qnn.QuantReLU(bit_width=4, return_quant_tensor=True)
        self.fc3   = qnn.QuantLinear(64, 10, bias=True, weight_bit_width=4, bias_quant=Int8Bias)


    def forward(self, x):
        out = self.quant_inp(x)
        out = out.reshape(out.shape[0], -1)
        out = self.relu3(self.fc1(out))
        out = self.relu4(self.fc2(out))
        out = self.fc3(out)
        return out

quant_weight_act_bias_lenet = QuantWeightActBiasLeNet()
```

Example

Let’s do it ourselves !



```python
# Training loop
for epoch in range(5):  # Train for 5 epochs
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

    print(f'Epoch {epoch+1}, Loss: {loss.item()}')
```

✓ 1m 0.2s

```
Epoch 1, Loss: 0.49315378069877625
Epoch 2, Loss: 0.025681622326374054
Epoch 3, Loss: 0.28308919072151184
Epoch 4, Loss: 0.4533153176307678
Epoch 5, Loss: 0.01304407138377428
```

```python
# Testing loop
import torch
model.eval()
correct = 0
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

accuracy = 100. * correct / len(test_loader.dataset)
print(f'Test Accuracy: {accuracy:.2f}%')
```

✓ 1.3s

```
Test Accuracy: 96.57%
```

ONNX

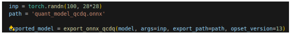
Presentation

ONNX (Open Neural Network Exchange) is an open-source format for AI models, enabling interoperability across different machine learning frameworks.

ONNX formats can be customized by software distributors for specific need. e.g : FINN-ONNX.

ONNX Example

Exporting to ONNX :



```
inp = torch.randn(100, 28*28)
path = 'quant_model_qcdq.onnx'

exported_model = export_onnx_qcdq(model, args=inp, export_path=path, opset_version=13)
```

Graph representation

We can then visualize the NN in Netron.

Example

onnx FINN notebook example [ONNX 2024].

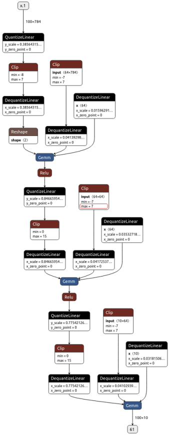


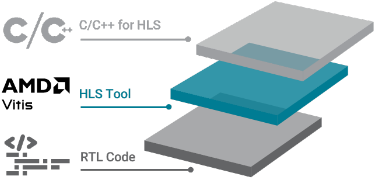
Figure – ONNX model in Netron

HLS : Principle

Presentation

HLS allows us to turn a C/C++ algorithms into RTL level HDL. This is especially helpful for our purpose to ease conversion.

Note that we will not use HLS directly.



HLS : Principle

HLS is a tool gaining traction in the industry because :

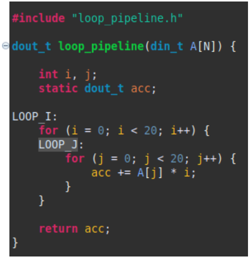
- It allows for faster end-to-end workflow.
- Allows for faster testing of different configurations.

Example

Let’s see a quick example to understand how it works !

HLS Example

In this example (from Xilinx), we have a simple loop...

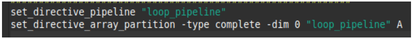


```
#include "loop_pipeline.h"

dout_t loop_pipeline(din_t A[N]) {

    int i, j;
    static dout_t acc;

LOOP_I:
    for (i = 0; i < 20; i++) {
    LOOP_J:
        for (j = 0; j < 20; j++) {
            acc += A[j] * i;
        }
    }

    return acc;
}
```

objective ?

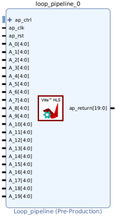
Pipeline it and turn it into an IP !

We add directives...



```
set_directive_pipeline "loop_pipeline"
set_directive_array_partition -type complete -dim 0 "loop_pipeline" A
```

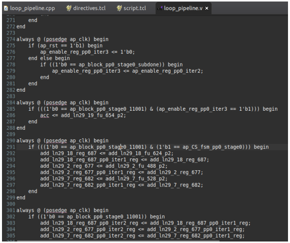
Verify.. and export as an IP in Vivado... Without ever writing HDL code !



HLS : RTL Visualization

RTL Access

Note that we have full access to the resulting HDL/RTL code (Verilog or VHDL).



```
    end
end

always @ (posedge ap_clk) begin
    if (ap_rst == 1'b1) begin
        ap_enable_reg_pp0_iter3 <= 1'b0;
    end else begin
        if ((1'b0 == ap_block_pp0_stage0_subdone)) begin
            ap_enable_reg_pp0_iter3 <= ap_enable_reg_pp0_iter2;
        end
    end
end

always @ (posedge ap_clk) begin
    if (((1'b0 == ap_block_pp0_stage0_11001) & (ap_enable_reg_pp0_iter3 == 1'b1))) begin
        acc <= add_ln29_19_fu_654_p2;
    end
end

always @ (posedge ap_clk) begin
    if (((1'b0 == ap_block_pp0_stage0_11001) & (1'b1 == ap_CS_fsm_pp0_stage0))) begin
        add_ln29_18_reg_687 <= add_ln29_18_fu_624_p2;
        add_ln29_18_reg_687_pp0_iter1_reg <= add_ln29_18_reg_687;
        add_ln29_2_reg_677 <= add_ln29_2_fu_488_p2;
        add_ln29_2_reg_677_pp0_iter1_reg <= add_ln29_2_reg_677;
        add_ln29_7_reg_682 <= add_ln29_7_fu_528_p2;
        add_ln29_7_reg_682_pp0_iter1_reg <= add_ln29_7_reg_682;
    end
end

always @ (posedge ap_clk) begin
    if ((1'b0 == ap_block_pp0_stage0_11001)) begin
        add_ln29_18_reg_687_pp0_iter2_reg <= add_ln29_18_reg_687_pp0_iter1_reg;
        add_ln29_2_reg_677_pp0_iter2_reg <= add_ln29_2_reg_677_pp0_iter1_reg;
        add_ln29_7_reg_682_pp0_iter2_reg <= add_ln29_7_reg_682_pp0_iter1_reg;
```

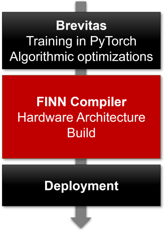
Figure – Resulting RTL code

Even if the auto-generated coding style may be hard to read...

FINN Presentation [Umuroglu 2017] [Blott 2018]

Presentation

FINN is an experimental Xilinx framework for accelerating quantized neural networks on FPGAs.



Brevitas
Training in PyTorch
Algorithmic optimizations

FINN Compiler
Hardware Architecture
Build

Deployment

FINN leverages Brevitas, ONNX Normalization & HLS to create a hardware version of our network logic.

FINN Workflow [Xinlinx 2021]

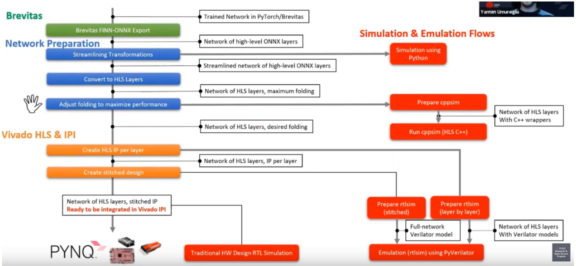
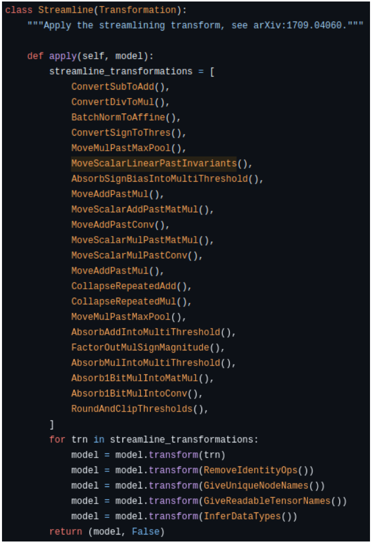

Yaman Umuroglu
Brevitas
Trained Network in PyTorch/Brevitas
Brevitas FINN-ONNX Export
Network Preparation
Network of high-level ONNX layers
Streamlining Transformations
Streamlined network of high-level ONNX layers
Convert to HLS Layers
Network of HLS layers, maximum folding
Adjust folding to maximize performance
Network of HLS layers, desired folding
Vivado HLS & IPI
Create HLS IP per layer
Network of HLS layers, IP per layer
Create stitched design
Network of HLS layers, stitched IP
Ready to be integrated in Vivado IPI
PYNQ
Traditional HW Design RTL Simulation
Simulation & Emulation Flows
Simulation using Python
Prepare cppsim
Network of HLS layers With C++ wrappers
Run cppsim (HLS C++)
Prepare rtlsim (stitched)
Prepare rtlsim (layer by layer)
Full-network Verilator model
Network of HLS layers With Verilator models
Emulation (rtlsim) using PyVerilator

Figure – FINN Workflow overview from [Xinlinx 2021]

Working with ZYNQ

Note that we will modify the end workflow as we will work with ZYNQ (Not using an official PYNQ supported board).

FINN Streamlining [Umuroglu 2018]



```python
class Streamline(Transformation):
    """Apply the streamlining transform, see arXiv:1709.04060."""

    def apply(self, model):
        streamline_transformations = [
            ConvertSubToAdd(),
            ConvertDivToMul(),
            BatchNormToAffine(),
            ConvertSignToThres(),
            MoveMulPastMaxPool(),
            MoveScalarLinearPastInvariants(),
            AbsorbSignBiasIntoMultiThreshold(),
            MoveAddPastMul(),
            MoveScalarAddPastMatMul(),
            MoveAddPastConv(),
            MoveScalarMulPastMatMul(),
            MoveScalarMulPastConv(),
            MoveAddPastMul(),
            CollapseRepeatedAdd(),
            CollapseRepeatedMul(),
            MoveMulPastMaxPool(),
            AbsorbAddIntoMultiThreshold(),
            FactorOutMulSignMagnitude(),
            AbsorbMulIntoMultiThreshold(),
            Absorb1BitMulIntoMatMul(),
            Absorb1BitMulIntoConv(),
            RoundAndClipThresholds(),
        ]
        for trn in streamline_transformations:
            model = model.transform(trn)
            model = model.transform(RemoveIdentityOps())
            model = model.transform(GiveUniqueNodeNames())
            model = model.transform(GiveReadableTensorNames())
            model = model.transform(InferDataTypes())
        return (model, False)
```

Streamlining

This process moves linear transformations and uses multi-Thresholds to represent quantization. The end goals : having a streamlined model.

Why Streamlining ?

Using streamlining, we can now operate the model using integers only and now have a ”standardized” structure that FINN can easily process into hardware layers.

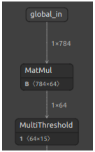
FINN ”To Hardawre”

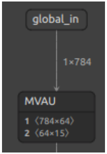
To Hardware

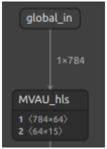
Once streamlined, FINN can easily process the model into Matrix Vector Activation Units (MVAU) (and Sliding Window Units for convolution layer).

Specialize Layers

Description : ”Specialize all layers to either HLS or RTL variants”. It will produce HLS/RTL Layers, ready for IP generation.







FINN : Under the hood

Actual Hardware Logic

The underlying logic of the generated hardware is somewhat similar to the reflexion we had during the first lecture (similarity with PEs).

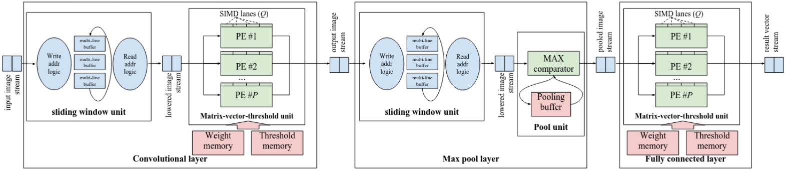


Figure – Model's Hardware Logic Example [Blott 2018]

FINN : Design verification

Verification

When dealing with hardware, verification is extremely important, FINN include an API to ease the verification at different step.

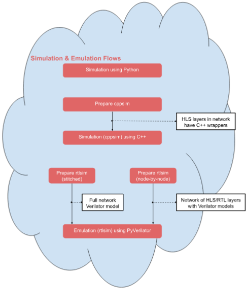
OXE Runtime

OXE : A runtime included in FINN to execute ONNX graphs using different modes

Simulation scenarios :

- Python model
- Streamlined model
- HLS C++ Code Simulation
- RTL Inference

FINN : Design verification



Simulation & Emulation Flows

Simulation using Python

Prepare cppsim

HLS layers in network have C++ wrappers

Simulation (cppsim) using C++

Prepare rtlsim (stitched)

Prepare rtlsim (node-by-node)

Full network Verilator model

Network of HLS/RTL layers with Verilator models

Emulation (rtlsim) using PyVerilator

Note : RTL Tesbench can also be generated through a custom workflow by specifying it as an output in the Hardware Builder. The small FINN example covers a use case.

Example

LAB 2 will cover the whole verification process.

Lab Workflow Overview

ZYNQ Specific Workflow

We will use FINN to generate an IP and use Regular Vivado/Vitis workflow to integrate it into a design.

1. Create network in Brevitas
2. Export an IP & Verify
3. Create a coherent design in Vivado
4. Use Vitis to create the software

A word on FINN & Zynq workflow

FINN Original workflow

FINN was meant to work with PYNQ using only python APIs.

Our custom workflow will :

- Allow for better control.
- Still leverage FINN optimized HLS & IP generation.
- Allow us to target any Zynq board.
- Allow for a better teaching and understanding experience.

FINN Simple Example / Overview

Worflows

FINN Offers multiple workflows depending on the projects. We can use the Finn docker environment to run a FPGA flow example. This workflow is by far the simplest but also the most rigid.

- Uses simple MNIST model
- Generates IP
- we can then integrate the IP to a Vivado project

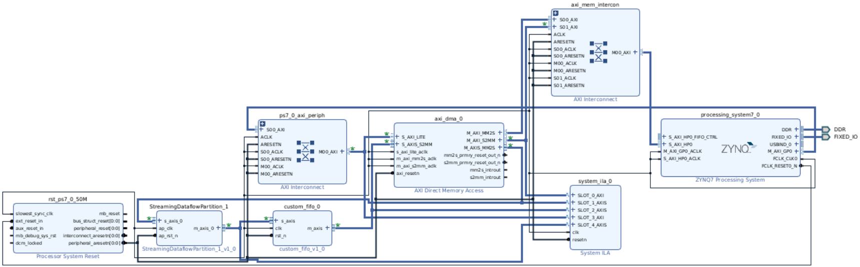
Example

Details about the ”simple” FPGA flow. Use output to compute potential throughput.

Project including the Stitched IP example

LAB 3 Preview

Here is a project including the final stitched IP from LAB 3.



Alternative Workflows for FINN

Workflow Alternatives

If you wish, you can use other workflows now that you have the tools to understand different workflows.

- Use FINN a PYNQ Board (better time to market, less control).
- Use FINN a regular FPGA (Need a softcore CPU).
- Use FINN RTL generator alternative listed before.
- Use Python AI Model to C/Cpp Converter (listed before) and port manually to HLS.

AXI Presentation [Wiki 2024]

AXI is a way to interconnect component within an SoC.
We'll use AXI to interconnect the FINN IP and our ZynQ CPU.

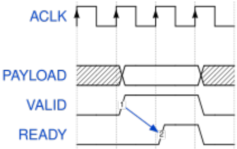


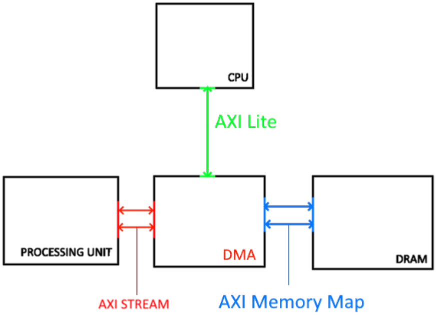
Figure – Basic principle

Note

This principle, where the payload (data, destination address, ...) is raised with a valid flag until the destination asserts "ready" is called a "handshake".

DMA and AXI [BRH 2024]

Alongside DMA, AXI will be used in many forms to connect PS, Memory (via DMA) and the custom IP.



Other useful tools

Tools Alternatives

If you wish, you can use other tools now that you have the tools to understand different possibilities. *Note that FINN is one of the best option when doing this course.*

- **hls4ml** : Converts neural networks to HLS code for FPGAs
- **Keras2c** : Converts Keras models to C
- **TensorFlow Lite** : Generates C++ code for embedded devices
- **ONNX-to-C** : Converts ONNX models to C
- **TVM** : Compiles deep learning models to various hardware targets
- **LeFlow** : Converts TensorFlow models to Verilog RTL

Current industry state

Insights

We saw that solutions exists for practical NN inference on FPGA, but how does it apply to the real world ?

- FPGA Are very suitable for edge/embedded solutions.
- As FPGA are niche, some companies actively use them for production line products, space embedded systems and has a strong IOT potential !
- ASICs and CPU can also be used for embedded use cases but in these fields, FPGA are the way to go for both flexibility and efficiency.
- The steep learning curve, niche applications and lack of community are a break to FPGA AI Inference expansion.

References I

Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser et Kees Vissers.
FINN-R : An end-to-end deep-learning framework for fast exploration of quantized neural networks.
ACM Transactions on Reconfigurable Technology and Systems (TRETS), vol. 11, no. 3, pages 1–23, 2018.

BRH.
How GPUs access memory without using CPU — DMA Zynq FPGA tutorial, 7 2024.

Crockett.
Embedded processing with the arm cortex-a9 on the xilinx zynq-7000 all programmable soc.
Strathclyde Academic Media, Glasgow, UK, 2014.

References II

deeplizard.
Backpropagation explained — Part 2 - The mathematical notation, 2 2018.

Umar Jamil.
Quantization explained with PyTorch - Post-Training Quantization, Quantization-Aware Training, 12 2023.

ONNX.
ONNX Notebook.
https://github.com/Xilinx/finn/blob/main/notebooks/basics/0_how_to_work_with_onnx.ipynb, 2024.
[Accessed 12-09-2024].

References III

Alessandro Pappalardo.
Xilinx/brevitas, 2023.

Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre et Kees Vissers.
FINN : A Framework for Fast, Scalable Binarized Neural Network Inference.
In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17, pages 65–74. ACM, 2017.

Yaman Umuroglu et Magnus Jahre.
Streamlined Deployment for Quantized Neural Networks, 2018.

References IV

- Wiki.
  Advanced eXtensible Interface, 8 2024.

- Xinlinx.
  YT : Neural Network Accelerator Co-Design with FINN.
  https://youtu.be/zw2aG4PhzmA?si=cnGQ1wZXThlQ8F4s&t=2276, 2021.
  [Accessed 15-08-2024].