FPGA NN Accelerator

AI from a Hardware Perspective

- Long training times
- High compute load
- Highly Parallelizable

- Typically accelerated on a GPU
  - Using Floating Point arithmetic
  - High power consumption / temperature

- Need for high speed inference and training in all applications
- Need for energy efficient operation in mobile/embedded applications

What are our options?

GPU/ASIC Acceleration

Pros

- Highly Parallel
- High Clock speeds (~5 GHz)
- Much faster than CPU
- Well documented software (easily programmed)
- Plentiful Memory

Cons

- Energy Inefficient
- Static Architecture (optimized for FP)
- Software Stack / Instruction Overhead

FPGA Acceleration

Pros

- Potentially faster than GPUs*
- Energy Efficient (compared to GPUs)
- Much higher parallelism potential
- Reconfigurable Architecture

Cons

- Requires HDL programming
- FPGA Fabric Overhead
- Lower Clock speeds (50MHz)
- Poor at FP arithmetic
- Less onboard memory

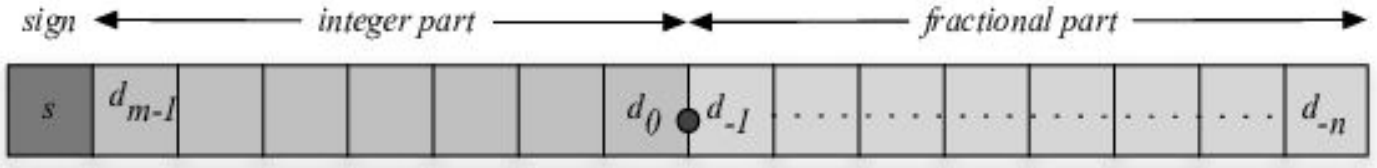
So why FPGAs?

Reconfigurability.

Highly optimized hardware for a particular application

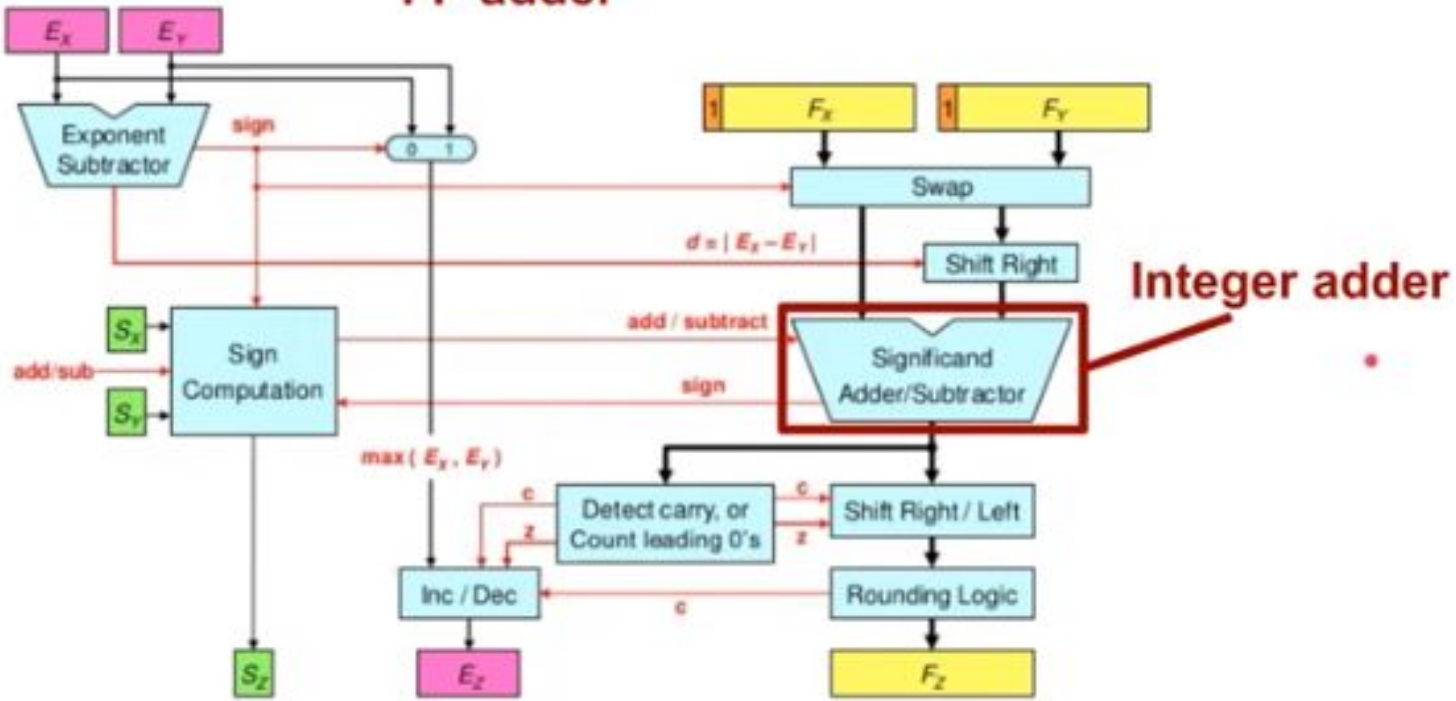
A Short Aside: Floating Point vs Fixed Point (Q8.8)

Floating-Point Format



Fixed-Point Format

FP adder


E_X
E_Y
Exponent Subtractor
sign
0 1
1 F_X
1 F_Y
Swap
$d = |E_X - E_Y|$
Shift Right
Integer adder
S_X
add/sub
S_Y
Sign Computation
add / subtract
sign
Significand Adder/Subtractor
max (E_X , E_Y)
c
z
Detect carry, or Count leading 0's
c
z
Shift Right / Left
Inc / Dec
c
Rounding Logic
S_Z
E_Z
F_Z

GPU Overhead

Half-precision Fused Multiply-Add

4-way dot-product

Operation (GPU)	Energy**	Overhead*
HFMA	1.5pJ	2000%
HDP4A	6.0pJ	500%

Source: Bill Dally "Hardware for Deep Learning" SysML 2018

Sources of Overhead During Inference

- Moving data from RAM to registers
- Memory Fragmentation
- Unutilized Compute Cores
- Synchronization and Scheduling
- Accounting for FP overflow, mantissas
- And last, but not least, API/Libraries

GPU Precision

FPGA Precision

Block Floating Point [FP11, block = 8]

Literature Review

Guo, Kaiyuan, et al. [DL] a Survey of FPGA-Based Neural Network Inference Accelerators. 2017

- Binarization (1 bit quantization), Pruning
- LUTs and Nonlinear LUT Quantization

Liang, Shuang, et al. “FP-BNN: Binarized Neural Network on FPGA.” 2018

- Quantization by layer
- Pipelining, Caching, and Loop Unrolling

```
for (i = 0;  i < 128;  i ++)
{
        sum1 + = const[i] * input[128 - i];
}
```

a.

```
for (i = 0;  i < 32;  i ++)
{
        sum1 += const[i] * input[128 - i];
        sum2 += const[2*i] * input[128 - (2*i)];
        sum3 += const[3*i] * input[128 - (3*i)];
        sum4 += const[4*i] * input[128 - (4*i)];
}
```

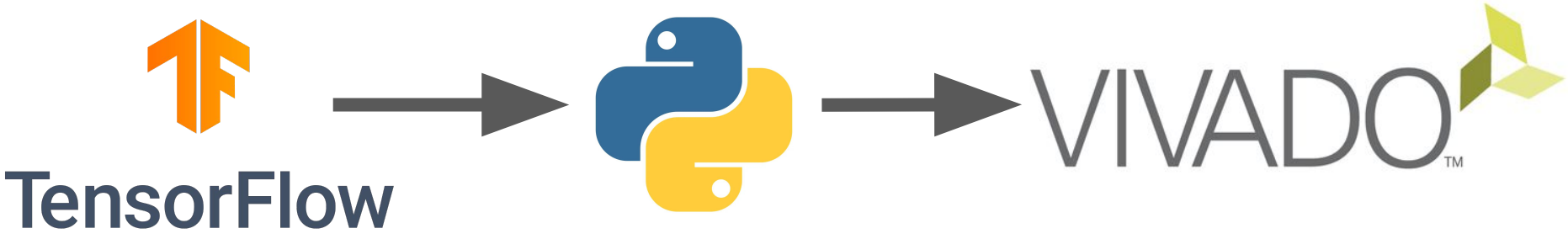
b.

Loop Unrolling (rolled and unrolled examples)

My Implementation

- **Goal: fast, energy efficient, and lightweight inference of small models**
- Work with existing training software to create small models to prove concepts
- Create extendable Hardware Descriptions for basic NN operations
  - A neuron consisting of X inputs, any fixed point bit width, and LUT based activation functions
  - A framework for describing MLP networks
- Create a set of scripts that will automate the process of training and exporting model information to an FPGA

TensorFlow



- Describe and Train Model

- Try varying levels of fixed point quantization, verifying model meets performance target

- Export Weights, Biases, Activation functions into LUTs

- Describe Network Architecture for HDL

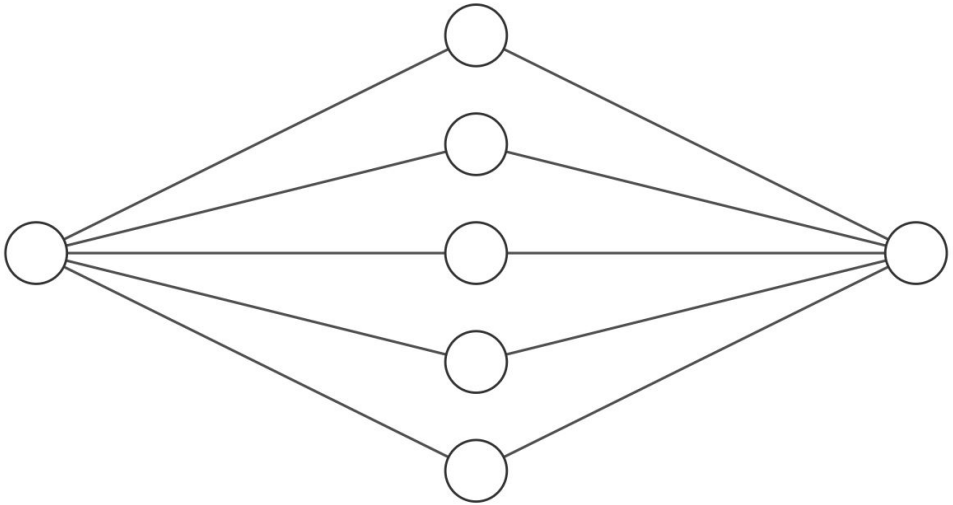
- Synthesize hardware design and verify inference performance

- Export test simulations to txt files, visualize with matplotlib

Trial Network:

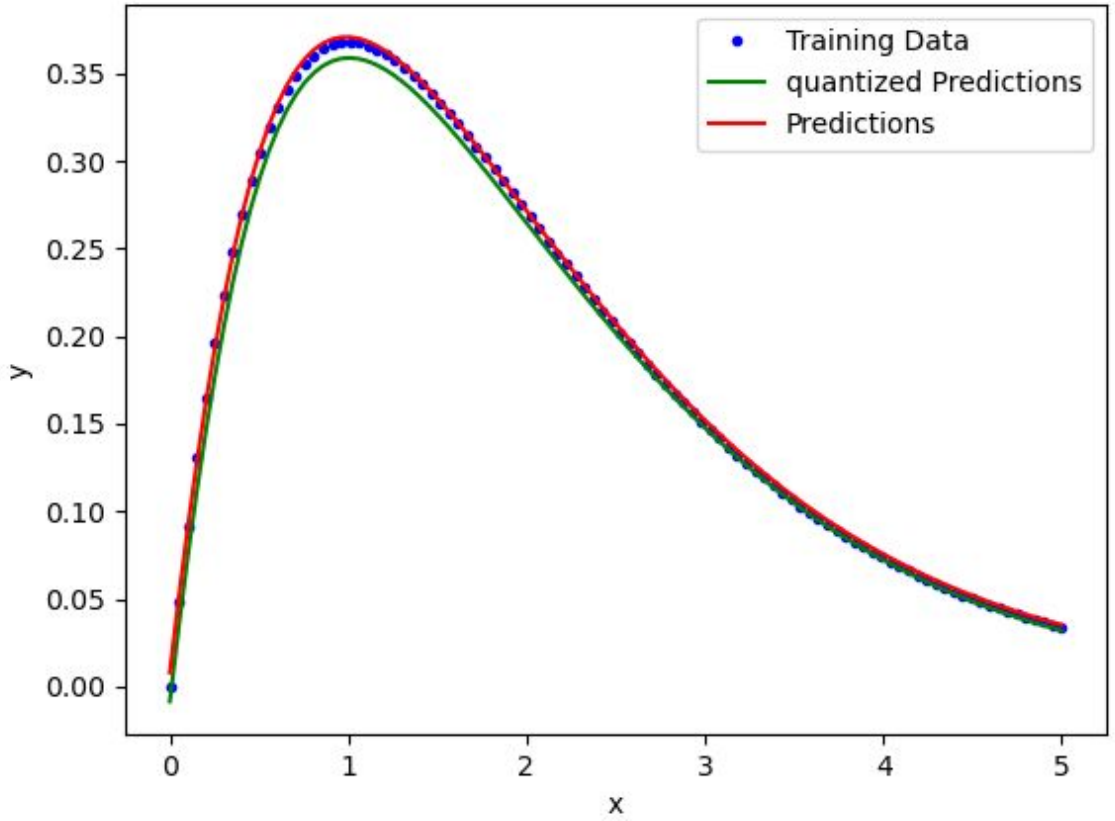
- Single Hidden Layer
- 5 hidden neurons
- Modeling a simple nonlinear function, xe^{-x}

- Trained in Floating Point
- Weights, Biases, and Sigmoid Function quantized to Q4.6

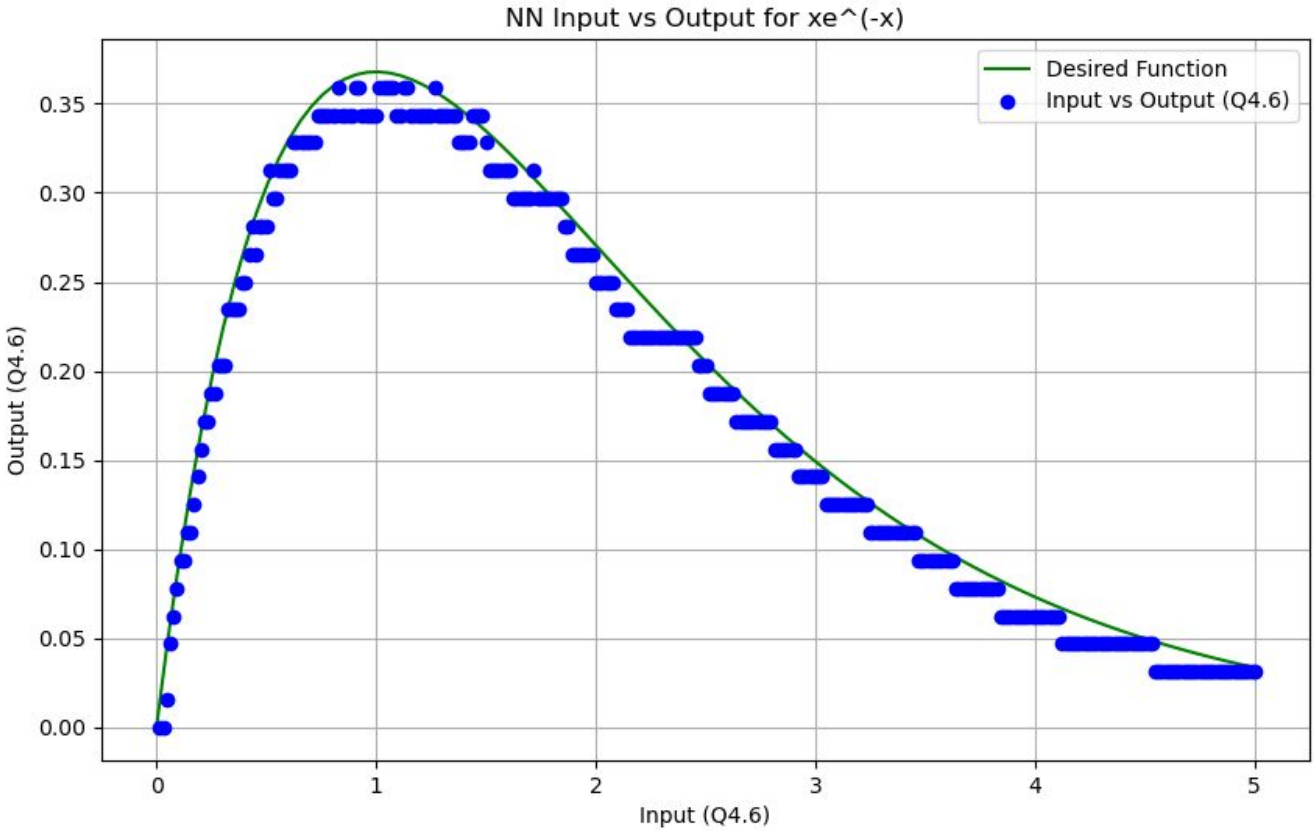


Input Layer $\in \mathbb{R}^1$ Hidden Layer $\in \mathbb{R}^5$ Output Layer $\in \mathbb{R}^1$

Tensorflow Quantization



FPGA Implementation



FPGA Utilization

- 30 ns inference time
- Small models (minimum 800 Q4.6 neurons) are able to fit onto single entry level FPGA
- By sharing Sigmoid LUTs, this could be bumped up to (~4,000 Q4.6 neurons!)
- Some floating point layers can be implemented for maximum precision (with a large performance and area compromise)

Resource	Utilization	Available	Utilization %
LUT	111	20800	0.53
FF	10	41600	0.02
IO	41	106	38.68

Next Steps

- Integrate training and validation with FPGA simulations
- Convolutional Layers
- Pipelining and Caching to allow larger models to fit on smaller FPGAs
- Nonlinear LUT Quantization
- Training during deployment

Sometime In the not-so-far Future…

FPGAs with Dedicated Accelerators

- Offloading the stuff that ASICs are good at to an on-chip accelerator