Performance Evaluation of Various Quantization Schemes for Fully Connected Layer of DNN

Tomasz Chadzynski
Department of Electrical Engineering
San Jose State University
Email: tomasz.chadzynski@sjsu.edu

***Abstract*—The growing popularity of solutions based on Deep Neural Networks creates demand for hardware implementations of the DNN solutions. The embedded systems inference engine implementations could not always use the high-precision floating-point leading to usage of alternative number representation systems. Using alternative number representations leads to the reduction in circuit size but also lowers the accuracy of computations. This research evaluates the accuracy and FPGA resources use of a fully connected DNN using fixed-point, half precision floating point and u-law number representations. Two variants on neural networks are evaluated, an architecture based on Sigmoid and an alternative version using ReLU activation function. To evaluate the behavior of DNN two versions of the inference engine model are developed. A bit-accurate high-level Python implementation and a synthesizable RTL implementation developed in SystemVerilog for the Xilinx Zynq platform. The simulation results for both models shown accuracy close to the reference implementation in Matlab for u-law, floating point, and fixed-point representation starting at 8-bits using the Sigmoid activation function. The ReLU version shown a noticeable drop in accuracy not crossing the 90% of correctly predicted results. The synthesis results showed steady growth of resources usage with a significant spike in the floating point implementation due to the complex implementation of the addition and multiplication logic. The use of Sigmoid function gives high accuracy but requires implementation as the look-up table — the evaluation of reduced index accuracy through removing the least significant bits shown minimal or no accuracy decrease. Reducing LUT memory requirement opens the possibility for memory optimizations in Sigmoid implementation.**

1. Introduction

With the growing popularity of deep neural networks, the AI spreads across a increasing variety of applications. As the reach of DNN applications grows, some systems impose specific design goals related to the physical implementation of neural networks. Not every application can utilize a fully featured computer system and needs to rely on embedded hardware implementations. In most cases, the hardware embedded implementations cannot afford to use full precision number representation in the calculations. The 64-bit size along with exponential growth of memory requirements with the increasing number of neurons can quickly make a design outgrow the area limits offered by most FPGA chipsets. To address the size growth the custom design uses alternative number representations that result in a smaller device but at the cost of accuracy.

The presented research investigates the effect of multiple quantization schemes on the size and accuracy of a neural network. The work focuses on using three number representations, fixed-point precision, floating point half precision and u-law encoding. A series of models are derived where a specific number representation is paired with the ReLU or Sigmoid activation function then both accuracy and resource usage is evaluated.

The scope of the research includes the development of inference engine based on the reference Matlab implementation. The developed engine exists in two variations, a high-level bit-accurate implementation that simulates the expected behavior of hardware implementation and a synthesizable RTL implementation.

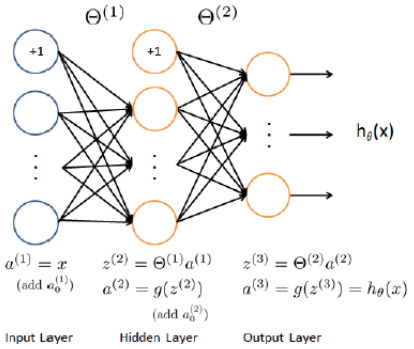
2. Background

The study uses a two-layer fully connected neural network as the reference model. The initial model is sourced from the ”Machine Learning” course by Andrew Ng [1], [2]. The network model is pre-trained in high-level Matlab model implementation. The neural network model is capable of classifying hand-written digits. The model implements a standard feed-forward inference engine featuring only the fully connected segment.

2.1. DNN Inference

The reference network is a fully connected DNN which consists of one input layer, one hidden layer an one output layer(Fig.1). The input data consists of 400 values, and each value corresponds to a single pixel in a 20x20 image. The input layer consists of 400-pixel values and one bias value summing to a total of 401 input features sent to the network. The hidden layer is composed of 25 neurons with each neuron receiving all 401 input arguments. The output layer contains ten neurons each neuron corresponds to a

Figure 1. Neural network model(source: coursera.org)



single detected digit. The output neuron of the highest value indicates the inferred digit.

The network implements standard feed-forward architecture with neuron calculations broken down into two phases. The first phase is the Multiply and Accumulate(MAC) where all inputs are multiplied by their corresponding weights and added together(1).

$$z^{(l)} = \sum_{i=0}^{N-1} (a_i^{(l)} * \theta_i^{(l)}) \tag{1}$$

In second phase the output from MAC module is fed to the activation function(Eq.2). The activation function adds non-linearity to the model allowing the network to fit the multi-variable data model accurately. This project models two types of activation functions the Sigmoid and ReLU activation functions. Both differ in achievable accuracy of the model and complexity in implementation.

$$a^{(l+1)} = g(z^{(l)}) \tag{2}$$

2.2. Activation function

The function $g(x)$ is called the activation function. The activation function calculates the final output value of the neuron and introduces non-linearity to the neural network model. Non-linearity is the key characteristics that allow the network to fit into the multidimensional input data set. This project uses two types of activation functions, $ReLU$ and $Sigmoid$.

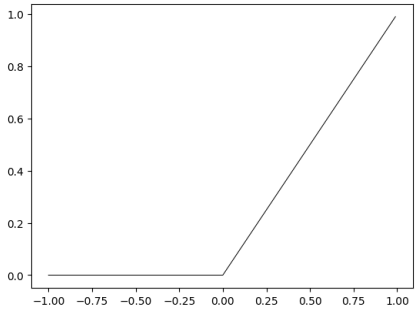
2.2.1. ReLU. The $ReLU$ activation function is given by the following equation (3).

$$ReLU(x) = max(0, x) \tag{3}$$

The function propagates all non-negative values and returns zero for the negative value of an argument. The $ReLU$ function is chosen for two reasons, it is computationally simple compared to other activation functions and does not affect the learning speed of gradient descent.

The $ReLU$ function is easy to implement in software as well as in hardware. For non-negative numbers, it is a simple pass-through function, and for any negative values of arguments, the function always returns zero. All that is needed to implement $ReLU$ is logic capable of detecting negative numbers. Unlike other activation functions, $ReLU$ does not perform any arithmetic operations.

Figure 2. ReLU activation function



Another property that makes $ReLU$ a suitable candidate for activation function is that it does not slow down the learning process with the gradient descent method. The gradient descent uses partial derivatives to determine the direction and length of the next step progression in the multidimensional solution space. When the partial derivative becomes smaller, the calculated step also becomes shorter. In the activation functions like Sigmoid, the slope in the vicinity of $arg = 1$ or $arg = -1$ becomes close to zero which results in slow progression of gradient descent. The slope of $ReLU$ is always equal to one for positive values and zero negative values coming from neurons which do not contribute to the specific input. Therefore, the gradient descent progression is not slowed down, and the learning process executes faster compared to traditional activation functions.

2.2.2. Sigmoid. The second type of activation function is the Sigmoid(Eq.4). Sigmoid is the classical activation function used in early research and mostly theoretical work.

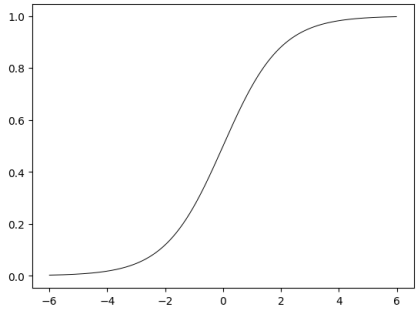
$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

The sigmoid allows for a higher level of non-linearity and potentially higher accuracy of the trained network. However, sigmoid is much more complicated to implement in hardware. The function characteristics is shown in figure 3. In the vicinity of zero, the function resembles linear characteristics. Further, from the centre point, the curvature starts to asymptotically approach one for positive x and zero for negative values.

2.2.3. Parameters scaling and Sigmoid. The problem of scaling arguments and weights is an essential issue for number representations based on fixed point precision. In order to correctly utilize the fractional representation, all number must be scaled to fit the range shown in Eq.5.

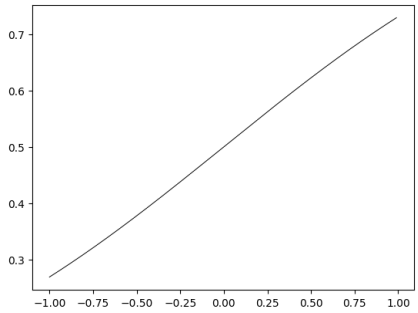
$$-1 < x < 1 \tag{5}$$

Figure 3. Sigmoid activation function



When operating within the +/-1 range, the sigmoid function assumes the shape shown in Fig.4. On the extremes, the max value of sigmoid is close to 0.7 and the min 0.25. In this state, the function never results in zero value and loses the ability to turn off neurons causing a drastic drop in the accuracy of the DNN.

Figure 4. Sigmoid normalized range



The examination of trained weights has shown a significant number of values being below -1 which indicates the importance of the proper scaling of the sigmoid function. The arguments to the sigmoid function cannot be simply scaled up to the previous range because of the value clamping necessary to preserve the proper range. The solution is to horizontally compress the sigmoid function by combining both scalers used for arguments and weights within the layer.

Assuming the A is the scaler applied to arguments such that a/A and W is the scaler applied to weights such that w/W, the scaled sigmoid takes the form of:

$$Sigmoid(x) = \frac{1}{1 + e^{-AWx}} \tag{6}$$

Now the domain of the sigmoid function correctly spans within the range of $+/-1$. The arguments coming from MAC can now be directly passed to the LUT generated from the modified version of the sigmoid.

An additional difficulty comes from the amount of scaling applied in each layer. If the scaling factors of each layer differ, then a multiple separate copies LUT are required to accommodate each layer.

2.3. DSP focused number representation

The default accuracy available on general purpose machines like amd64 os ARM architecture provides hardware support for very high precision number representations. The currently default floating point representation used in languages like Matlab or Python is the 64-bit version of floating point numbers called double precision.

However, using IEEE 745 floating point standard has its drawbacks and is not always an optimal solution especially when the project requirements allow for reduced accuracy. As a tradeoff for wide number range and high accuracy, the floating-point representation requires complicated logic to perform addition and multiplication [3].

An alternative to the floating point is the fixed point representation [4], [5]. The fixed point representation is particularly effective when the number range is restricted to values from the range shown in Eq.5. Such range assumption allows for uniform bit assignment where a single MSB is dedicated to being the sign bit, and the rest bits are translated as positive numbers. In practice, the fixed point representation is a 2's Complement system interpreted as a fraction.

The second alternative to the floating point representation is the u-law scheme [6]. The u-law is a form of further compressing the fixed point representation using a non-uniform number distribution space. Numbers represented using u-law encoding exhibit increased accuracy in the vicinity of zero while the values further from the origin gradually lose accuracy. The u-law representation is a compression mechanism from 14-bit fixed point to 8-bit u-law number.

Despite the complexity of floating point hardware, the floating point representation offers high accuracy and is not bounded by the range requirement(Eq5). However, the default double precision representation might turn out to be too demanding regarding memory requirements while offering precision that is unnecessarily high. The presented research evaluates a half precision floating point variant that is a 16-bits instead of the default 64-bit double precision.

Figure 5. Number representation quantization

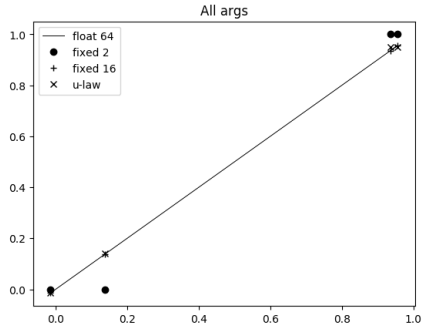


Figure 5 shows example quantization errors for the number represented within the range from zero to one. The result shows small discrepancies from 64-bit double precision for all representations except the 2-bit fixed point precision.

3. Simulation model

The implementation is divided into two modules, number representation and deep neural network. Both modules are used together first to represent the input data and then perform calculations of a single neuron. The input data consist of the sets of input arguments and weights. Both modules support the main simulation code used to calculate the result of every case and process the simulation data.

The number representation part contains classes simulating bit accurate number in a specific category.

- float_hp.py: half precision floating point.
- fixed.py: fixed point representation with parametrized precision.
- u_law.py: u-law number representation.

Each package allows for conversion from and to 64-bit floating point number. One instance of the class corresponds to one value. The classes are equipped with overloaded operators which perform basic addition, subtraction and multiplication operations. All classes except the half-precision floating point, support value clamping if the initial number given in the constructor exceeds the range(Eq.5). Fixed point and u-law modules operate only on fractional representation. The represented range of fixed-point an u-law values is $-1 < x < 1$.

If the input set of numbers contain any value outside of the supported range, the entire dataset must be pre-scaled. The implementation provides a Scaler class to ensure uniform and consistent scaling of the numbers. The scaler class accepts two arguments the scaler and bias. The class operates only on the Python float64 data type. Calling the scale method performs the scaling on the entire set. The scaler class also provides the restore method to reverse the scaling operation.

The deep neural network module contains the implementation of the neuron and the activation functions. One Neuron class instance represents a single neuron in the network. The class is initialized with the set of weights and the activation function. Optionally the layer number and neuron index in the layer can be assigned to the class for further identification.

The process function of class Neuron performs the calculation described by equations 1 and 2. The equation 1 can result in value outside of the range (Eq.5) supported by number representation.

During summation, if the value exceeds allowed range it is then clamped by the number representation class. Finally the activation function executes Eq.2 returning the final value.

Errors like type mismatch or unexpected occurrence of NaN value are complicated to debug. To prevent errors resulting from such cases, the process function runs a series of validation tests and terminates execution with an exception if such problems occur. Future versions will support the stronger identification of the place of error allowing for efficient debugging.

The DNN_FC classes implement the fully connected DNN segment. It is the primary object that performs simulation on a given neural network. The infer method takes the input arguments as a list of floats and returns the list of outputs from the last layer of the network. The return type of infer function matches the configured number representation.

The DNN_FC class is fully parameterizable and adjusts to the given neural network configuration. The configuration is done by passing the DNN_FC_DESC descriptor class as an argument during initialization. The descriptor contains information about the number of layers, neurons per layer and corresponding activation function used by the network. In this implementation, the descriptor data is read from the descriptor.txt file.

The inference is performed by calling the infer method of the DNN_FC class. The method takes a list of input parameters and performs all the calculations. The calculation occurs layer after layer, the arguments and weights are scaled, and then a neuron is created. The data is calculated for each neuron and then passed to use in the next layer. In the end, the infer method returns a list of values which are the outputs of each neuron in the last stage of the network.

4. RTL implementation

The RTL implementation developed using Xilinx Vivado tools and synthesized for the Zynq XC7Z20CLG400C-1 device. The RTL codebase is developed using SystemVerilog and supported with a series of scripts implemented in Python. The inference engine device requires access to random access memory used to store neural network data such as weights, input values and lookup tables. The top module contains one memory addressing port, one memory data input and series of registers representing values of computed output neuron values. To control the working of the device, a clock signal and start signal is available. When the inference completes, the line representing work done is brought high.

The internal design of the device is split into two major parts, datapath and state machine controller. The data path implements the data flow and computations where the state machine handles the execution of the inference algorithm driving the data path. Both components work together with a single run, accepting one set of 400 input arguments and giving back the set of calculated predictions.

For the given size of the network, in order to fit the entire structure, the device must be capable of accommodating a place for 10720 values not including look-up tables. In the case of XC7Z20 chip, the required network size exceeds the capacity of the device.

Due to the size requirements exceeding the device capacity, an approach in the form of multi-function datapath is applied, and the inference engine requires external random access memory. The data path operation focuses on calculating a single input argument at a time. The design takes advantage of the fact that a single input argument is

processed by every neuron in the layer allowing some degree of parallelism.

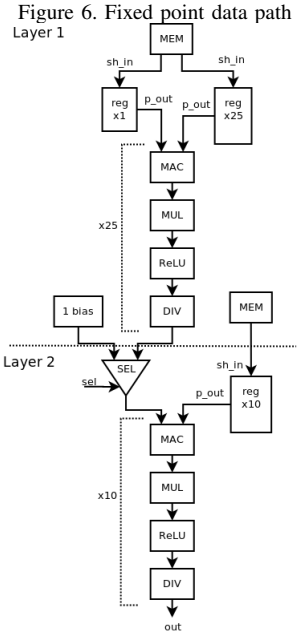
For example, in the first layer, the first action is to load an argument from memory. Then the corresponding weights are shifted into the weights register. When all values are in place a single cycle of multiply and accumulate occurs calculating the partial sum for every neuron in the layer. The datapath components implement computational elements supported by a set of working registers. The structures of data-paths differ between each other depending on the activation function an number representation used by the network.

4.1. ReLU datapath

The ReLU datapath provides support for the network based on the ReLU activation function. The ReLU activation function(Eq.3) is a simple pass-through function for all non negative values while negative arguments are replaced by zero value. The function is shown in Fig.2. The special characteristic of the ReLU activation function is its low complexity in implementation in hardware. Implementation of ReLU requires only the ability to determine the sign of an argument, and it does not perform any mathematical computations.

The implementations of different number representation require slightly different structures of data paths on top of support for given activation function. Assuming the elements in data path are specialized in computing specific number representation, on top of common element names every data path shows differences in structure. The data path shown in Fig.6 implements inference engine based on fixed point precision numbers.

Figure 6. Fixed point data path



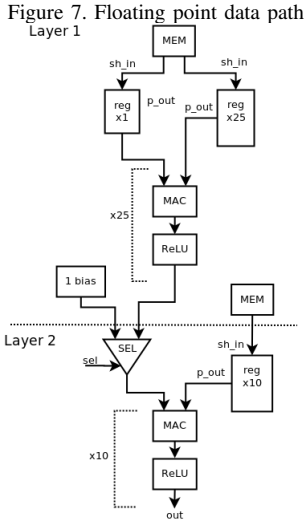
The Layer 1 contains a single value register that stores an argument loaded from the memory and another register for the weights with a total capacity of 25 values. The component block spanning from MAC to DIV component is replicated 25 times to accommodate all neurons in the layer and calculate partial sums in parallel. The MAC module performs the multiply and accumulate operation and serves as storage element holding value which is further processed and fed to the second layer. The MUL, ReLU and DIV components are purely combinational modules. The weights and arguments are pre-scaled to the range shown in Eq.5 to conform to the requirements of fixed-point number representation.

After all input values in the first layer get processed the MAC module stores the complete sum(Eq.1) Keeping the values in place for the second layer. The sum is scaled back to its original range and clamped if overflow occurs, then processed by ReLU and scaled down by a factor specific to the second layer.

In the second layer, the row role of single value register is replaced by selector indexing output among 25 previous stage neurons. The cycle is the same as in the first layer, but the number of neurons is now 10 with each neuron value becoming an output from the neural network.

The floating point data path shown in Fig.7 is the simplest of all the ReLU implementations from the perspective of the data path structure. The half precision floating point is capable of storing values outside of the $+/-1$ range and thus does not require additional scaling.

Figure 7. Floating point data path

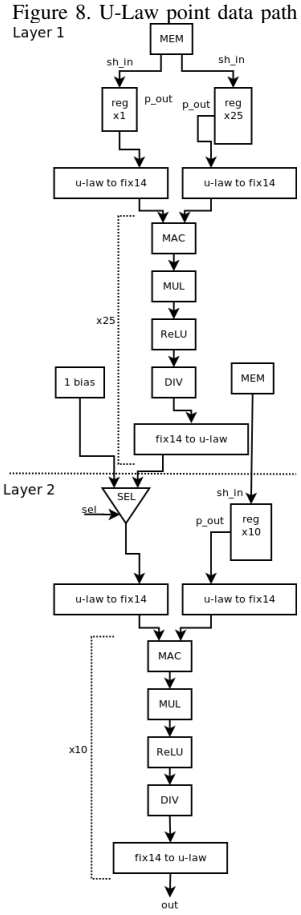


The downside of floating point implementation is relatively complex addition and multiplication compared to the fixed point representation. However, the floating point allows for highest accuracy because no value clamping occurs. Otherwise, the data path structure is the same but without additional scaling elements.

The u-law implementation shown in Fig.8 is the implementation specialized in computing using the u-law number representation. The u-law datapath is nearly identical to fixed point representation with additional data converters. All the arithmetic operations performed on u-law numbers

are done using the fixed point 14-bit representation as an intermediate form.

Figure 8. U-Law point data path



4.2. Sigmoid datapath

The second type of data path is the implementation using the Sigmoid activation function. The sigmoid function allows for greater non-linearity due to the shape of the function(Eq.4).

The function takes the shape of the letter S(Fig.3) with the function asymptotically reaching value of 1 for growing positive arguments and 0 for negative. The sigmoid function allows for greater accuracy of the network, but it is significantly more challenging to implement in hardware directly. The division and power operations are the main difficulties in the RTL implementation. This design implements the sigmoid function using the look-up tables stored in memory. The input argument indexes a look-up table in whole or as a part — the more accurate the indexing, the higher accuracy of the operation but at the additional cost of memory.

The sigmoid implementation directly affects the structure of the datapath. Unlike in the ReLU implementation, the second layer input data is obtained from memory storing the look-up table rather than from the previous layer. The

Figure 9. Fixed and floating point data path

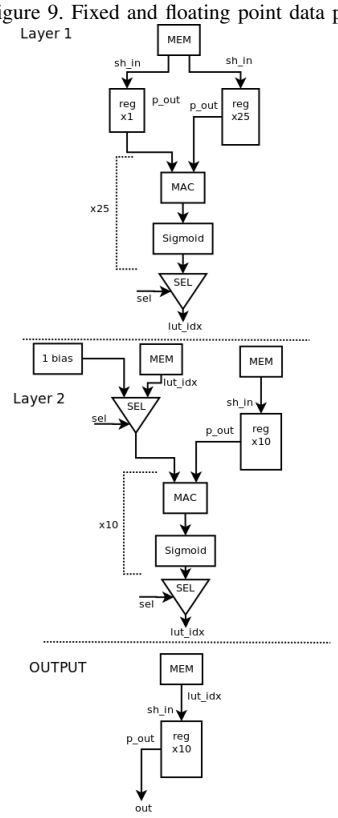


Figure.9 shows datapath for fixed point and floating point implementations.

The values stored in MAC modules are fed directly to the sigmoid element. The sigmoid operation is split into two parts. The hardware part present in the data path is tasked with producing the index for LUT. Then the memory is addressed with the base address of LUT added to the index and the result of a sigmoid operation is read from memory in the next stage. Similarly to the ReLU datapath, a selector is used to select given index of current argument input to the second layer. After the second layer is done, an additional step is performed which is going through indexes produced by stage two one by one and reading the resulting values back to the output register.

The u-law version of datapath shown in Fig.10 is similar to the other number representations but incorporates encoding and decoding modules between u-law and fixed-point representations. After the full cycle of MAC(eq.1), the resulting value is converted back to u-law representation and used as the index for LUT.

4.3. State machine

The state machine is the second main module in the device and is responsible for the execution of the inference algorithm. The computation algorithm works in multiple stages reusing the registers for different parts of the network. The state machine shown in Figure 11 implements computation using the ReLU activation function.

Figure 10. U-Law point data path

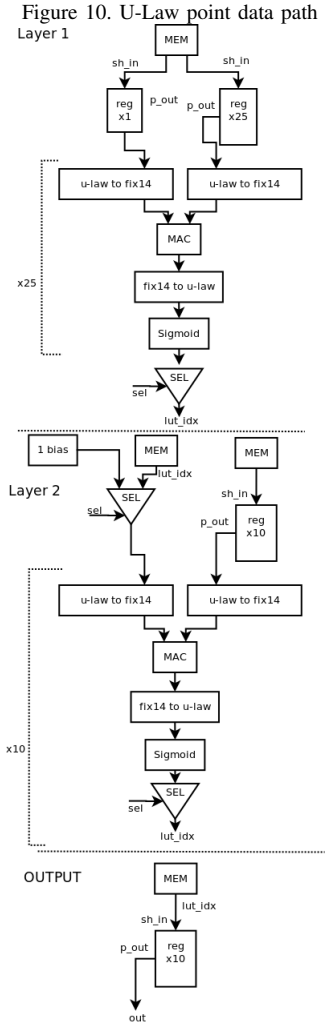
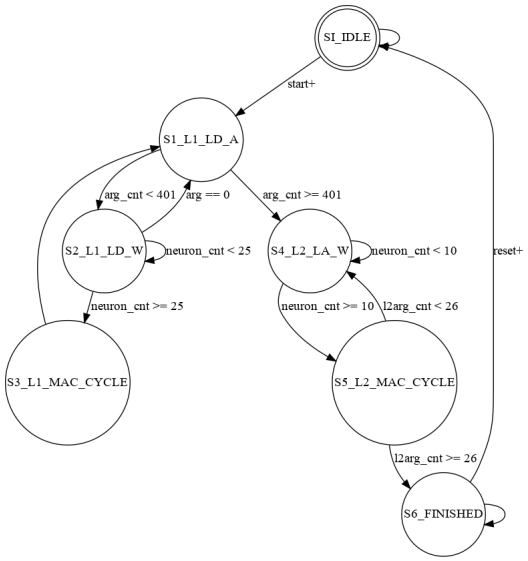


Figure 11. State machine for ReLU workflow



The shown states perform the following functions:

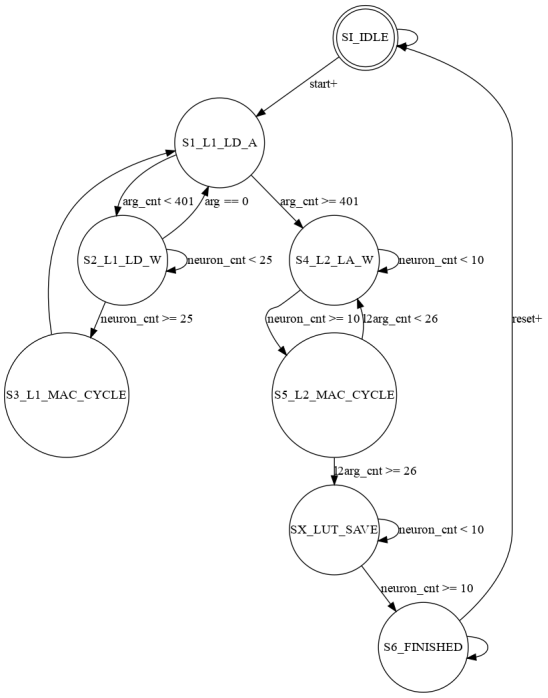
- SI_IDLE: The state loops back and waits until the start signal is given. When the start signal appears the memory content populated with weights and arguments is expected to be valid, and the state machine begins processing.
- S1_L1_LD_A: The primary function of the state is to load an argument into the register from memory after the argument is loaded the state machine transitions to loading the corresponding weights. However, of the device counters indicate that all values were already loaded, a transition occurs to the second layer processing.
- S2_L1_LD_W: The state loads all corresponding weights into register holding one weight per the neuron and the transition to the MAC cycle. The state also implements an optimization. When the loaded argument turned out to be a value of 0, the loading of weights and MAC operations are skipped, and the state machine begins to load the next argument.
- S3_L1_MAC_CYCLE: The state implements the multiply and accumulate step where all neurons in the layer compute the partial sums in parallel. Then the state machine transitions to loading the next argument from memory.
- S4_L2_LA_W the state performs the same functionality as the S2_L1_LD_W but loads the weights for the second layer. No argument loading occurs as those are immediately available from stage 1 registers.
- S5_L2_MAC_CYCLE performs the same function as S3_L1_MAC_CYCLE however when the counters indicate all arguments being processed the state machine transition to the finished state.
- S6_FINISHED: The state indicates an end of the computation. All the output registers are valid and contain the predicted result. The state loops until the reset signal are given to the device.

The state machine implementing sigmoid function(Fig.12) operates based on the same principle as the ReLU version. However, the sigmoid requires access to LUT to obtain the result. The existence of LUT requires additional steps:

- Memory access to LUT before the S5_L2_MAC_CYCLE state to obtain argument.
- SX_LUT_SAVE state that loads the final results from LUT using output of second layer as indexes.

This implementation assumes a simplified model of memory. The assumption is that memory returns any value within one clock cycle. However, for practical memory implementations, an additional set of states is required to accommodate for the memory delay in both ReLU and Sigmoid versions.

Figure 12. State machine for Sigmoid workflow



5. Results

The evaluation procedure used for all representations is broken into two sections, a section that evaluates the accuracy of the implemented engine and evaluation of hardware resources utilization.

$$Matlab\ Accuracy:\ 95.00\% \tag{7}$$

The tests use the accuracy achieved by the original Matlab model(Eq.7) as the reference for the evaluation of each of the network performance. The accuracy tests are applied to the bit-accurate implementation in Python as well as to the RTL in the form of behavioral simulation. During the accuracy tests, a dataset of 5000 test cases is fed to the network and compared against the expected value. The results show the absolute percentage of correct predictions for all test cases. However, the evaluation compares the result to the absolute maximum achieved by reference implementation (Eq.7).

The utilization tests are applied only to the RTL model. Each variation is synthesized for the Xilinx Zynq XC7Z020-1CLG400C [7] device implemented in the Zybo Z7-20 board. Every variation of design is separately synthesized and implemented and final usage results used to compile final comparison between design.

5.1. High Level Bit Accurate Simulation

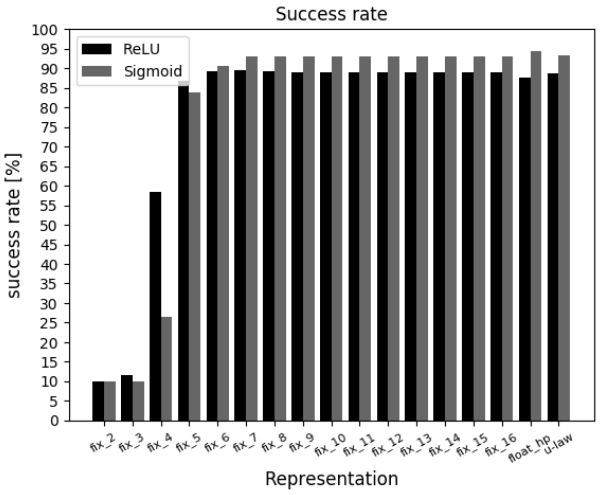
The high-level simulation performed using bit accurate number representation and simulated DNN model show results closely following the reference accuracy for most of the network variations.

Representation	ReLU [%]	Sigmoid [%]
fix2	9.90	10.00
fix3	11.56	10.00
fix4	58.44	26.60
fix5	86.94	83.96
fix6	89.20	90.48
fix7	89.62	92.92
fix8	89.22	93.06
fix9	89.10	93.16
fix10	88.94	93.12
fix11	88.84	93.16
fix12	88.86	93.14
fix13	88.86	93.08
fix14	88.84	93.12
fix15	88.86	93.12
fix16	88.84	93.12
fp16	87.58	94.44
u-law	88.76	93.40

TABLE 1. ACCURACY OF THE BIT ACCURATE MODELS

The Sigmoid implementation gives noticeably better results compared to the ReLU version. One of the factors that determine the better accuracy of Sigmoid is potentially the fact that the original weights of the network were trained using the Sigmoid activation function and the ReLU implementation used the same weights directly. The series of tests using Sigmoid function is performed using high-precision version which internally converts into Float64 to perform the calculations and then converts the result back to original representation.

Figure 13. Accuracy

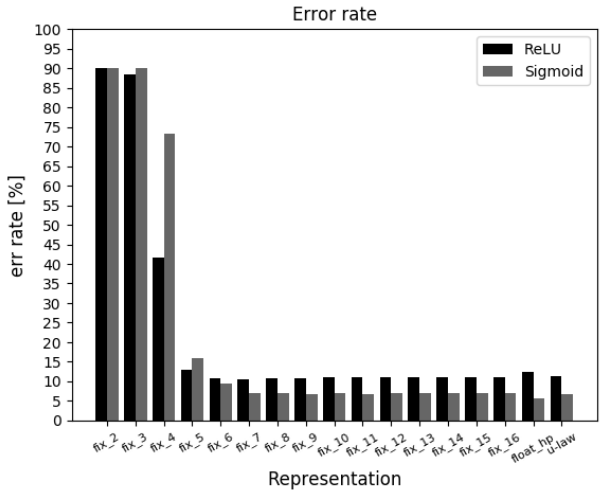


An apparent spike in the accuracy(Fig.13) is present starting from the 5-bit fixed point representation. Beginning at the 7-bit fixed point, the prediction accuracy differences between the representations are nearly unnoticeable.

As expected the highest accuracy is delivered by the half-precision floating point representation lagging by 0.56% behind the reference implementation while requiring only 25% of the memory to store the weights and arguments.

The 8-bit fixed point representation shows nearly the same accuracy as the reference implementation differing by 2% while one-sixth fo the total memory. The u-law representation also shows a performance close to 8-bit fixed-point representation at the same memory requirements. However, the u-law requires conversion between the 8-bit u-law and 14-bit fixed point precision to perform the addition and multiplication operation which complicates the overall circuit.

Figure 14. Error rate



The error rate graph(Fig.14) shows continuously decreasing error with growing width of number representations. An apparent difference is present between the ReLU and Sigmoid implementations. The presented results show the potential of the ReLU activation function. Further tests are required to properly evaluate the ReLU implementation since the network was originally trained only for the Sigmoid activation function.

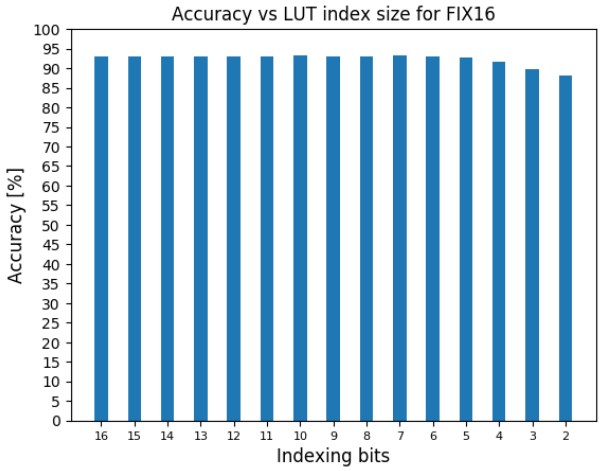
5.2. Reduced accuracy of LUT for Sigmoid

While the sigmoid activation function shows accuracy close to the reference implementation, its implementation in hardware is much more problematic than the ReLU activation function. The Sigmoid requires a look-up table which quickly grows in size for an increasing number of bits used in specific number representation.

In the worst case scenario, assuming all bits of specific number representation are used as the indexing bits, for the FIX16 and Float16 the size of LUT would require $2^{16} * 2bytes = 128KiB$. In some cases the reduction of LUT table size is necessary. The following experiment attempts to investigate the accuracy loss due to reduced accuracy of LUT while removing indexing bits.

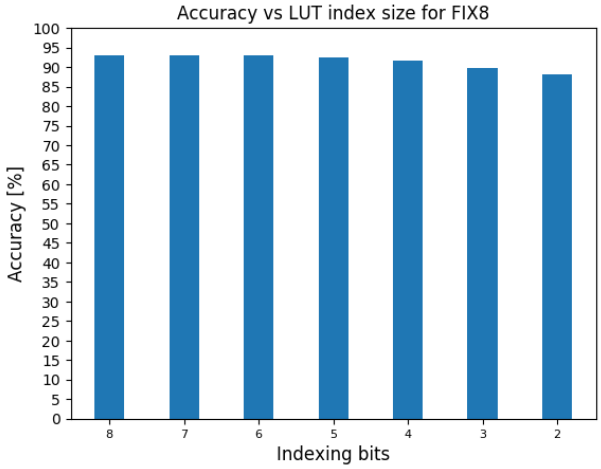
The results show in Fig.15 and Fig.16 present the accuracy loss while the least significant bits are discarded as indexing bits. The removal starts from the LSB.

Figure 15. FIX16 accuracy vs index bits



In the 16-bit fixed point representation, no significant accuracy drop is present until the indexing width falls below 6-bits. The results indicate more than half of LUT memory savings while maintaining accuracy close to the reference implementation. The similar trend is present for the 8-bit fixed point representation. The accuracy appears to stay the same up until the indexing bits fall below 6.

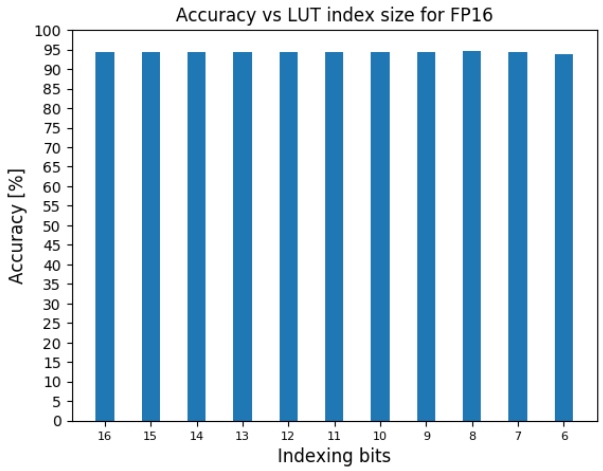
Figure 16. FIX8 accuracy vs index bits



In the floating point representation, the indexing bits are removed only from the mantissa starting from the LSB. The achieved accuracy appears to stay approximately the same even for the entire mantissa removed from indexing. The results show that the floating point representation delivers higher accuracy than the 16-bit fixed point representation at the same size of the lookup table.

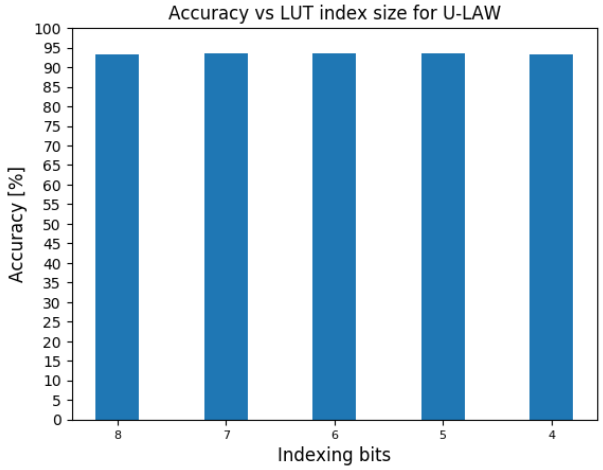
For the u-law representation, the bits are removed from the step section while preserving the chord and sign bits. The accuracy does not drop even if all step bits are removed

Figure 17. FP16 accuracy vs index bits



from indexing. The u-law representation is capable of delivering higher accuracy than the 16-bits fixed precision while significantly reducing the memory requirements.

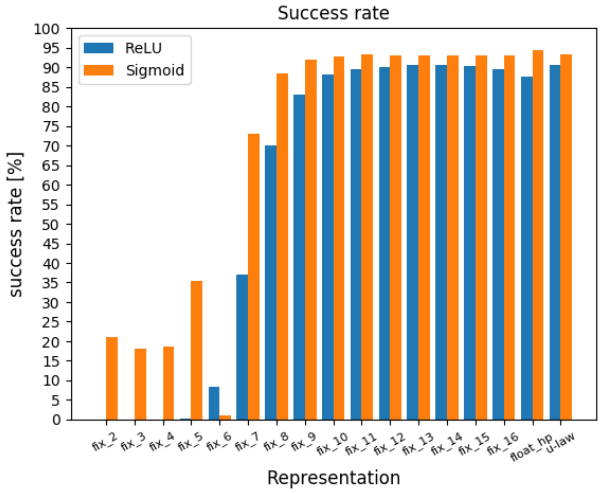
Figure 18. U-LAW accuracy vs index bits



Representation	ReLU	Sigmoid
fix2	0	1057
fix3	0	900
fix4	0	931
fix5	8	1773
fix6	416	50
fix7	1854	3654
fix8	3501	4419
fix9	4149	4592
fix10	4405	4640
fix11	4474	4659
fix12	4507	4655
fix13	4524	4652
fix14	4526	4653
fix15	4522	4654
fix16	4473	4656
fp16	4380	4721
u-law	4529	4659

TABLE 2. CORRECT PREDICTIONS FROM TOTAL OF 5000 TEST CASES

Figure 19. RTL Accuracy



5.3. RTL Simulation

All the simulation tests are presented as total correct predictions from the set of 5000 test cases. The estimated model accuracy limitation is assumed to be the accuracy achieved by the reference Matlab implementation(eq.7) which amounts to 4750 correct predictions.

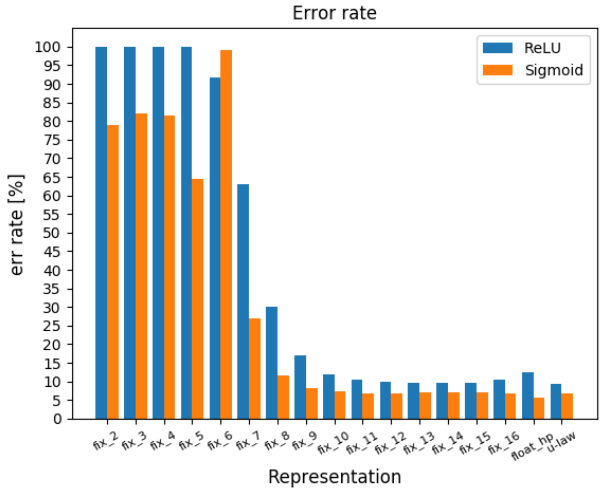
The Table.2 shows the correct predictions for each network configuration. Each row shows the single configuration, and there are two columns one dedicated to the results of ReLU implementation other to sigmoid. A significant spike in the accuracy is present for fixed-point number representation larger or equal to 8-bits width.

The sigmoid implementation leads in the accuracy. It is an expected outcome since the original model of the network were trained using sigmoid. At 9-bits of width, the fixed point representation achieves accuracy nearly identical to the original Matlab model with 55 fewer bits. However, the representation in this configuration uses double full resolution 9-bits indexed LUT requiring 1.152KiB od memory. In contrast, the currently used sigmoid datapath using 64-bit representation would require 3.840KiB. The 9-bit fixed point implementation shows nearly 60% memory savings at comparable accuracy for sigmoid activation function.

The floating point implementation shows the best accuracy of all lagging behind the Matlab reference by 29 correct predictions. However the double, full resolution LUTs require 2MiB of memory.

The u-law representation shows the best accuracy vs memory requirements. The sigmoid version missed 51 more test cases than the reference. The memory requirements for

Figure 20. RTL Error rate

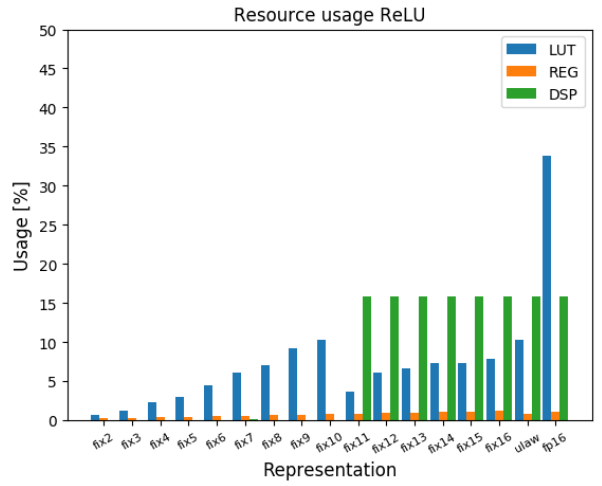


the LUT is 512 bytes.

5.4. Resource usage

The synthesis results show steady growth in required resources with the growing size of the number representation. There is a significant spike in resources usage present in the floating point representation due to the complexity of addition and multiplication.

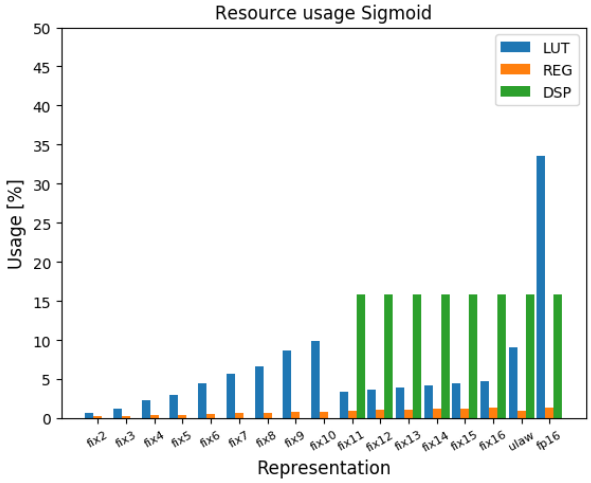
Figure 21. Resources usage for ReLU architecture



At the representation width of 11-bits, the synthesizer starts to use DSP blocks while drastically lowering the usage of general purpose LUT and register elements.

The Vivado environment reported inability to use DSP blocks for some elements due to the asynchronous reset feature built into the registers. Applying synchronous reset could potentially allow the synthesized to use the DSP blocks more effectively.

Figure 22. Resources usage for Sigmoid architecture



6. Future recommendations

While the project results shown consistent and satisfactory results, some improvements could allow for better insight into the investigated task. The first improvement is to provide a reference model trained specifically for the ReLU activation function. While the current results show high accuracy, the relative success rate is low compared to Sigmoid version. It is unclear whether the performance shown by ReLU version is limited by the properties of the activation function or the lower accuracy is caused by adopting weights trained for different activation function.

The second improvement relates to the RTL implementation of the network registers. The asynchronous reset prevented the synthesized from translating a number of elements into the DSP modules. An introduction of synchronous reset to the registers is expected to allow for better usage of DSP elements resulting in improved FPGA resources usage and potential performance boost due to better utilization of optimized blocks.

Third recommendation spans from the architecture of the datapath module. The replication of hardware used to implement both layers of the network seems to be unnecessary. While one stage is active the second always remains idle. A different approach involving reuse of layer logic can provide a significant reduction in resources usage.

The fourth recommendation involves optimized access to network data in memory. The approach based on single port memory and shift registers cost the majority of the clock cycles spent only on data movement. A more efficient approach possibly based on multi-port memory might improve the inference time. Also, taking advantage of significant gains from reducing the indexing bits in Sigmoid LUT present an opportunity to store entire LUT within chip registers allowing for instant evaluation bypassing memory cycles.

7. Conclusion

The goal of the research is to evaluate the effect of the use of multiple alternatives to Float64 number representations on DNN accuracy and FPGA resource usage. The research focuses on implementing hardware DNN inference engine using reference Matlab model of fully connected neural network segment trained to recognize handwritten digits. The reference implementation sourced from a high-level model written in Matlab and the trained weights used to test the inference engine.

Two types of models are developed, one high-level bit-accurate model developed in Python and one synthesizable RTL model in SystemVerilog using Xilinx Vivado tools. Both models are evaluated for accuracy using the set of 5000 test cases. A total of 34 different variations of neural networks are generated. The instances vary in used number representation and activation function. The used number representations span from FIX2 to Fix16, u-law and Float16, each set instantiated in two versions one ReLU and one Sigmoid.

The simulation tests show accuracy close to the reference for Sigmoid, starting at 8-bit fixed-point and up including u-law and Float16. The ReLU based representation shows consistently lower accuracy. The potential reason for lower performance is the fact that the network was initially trained for Sigmoid and the weights were directly used in ReLU.

The synthesis and implementation showed constant increasing resources usage with growing number representation accuracy. At 11-bit fixed point number representation, the synthesized starts to use the DSP modules drastically lowering the usage if general purpose resources. The floating point representation shows a significant spike in general purpose resources usage due to complex addition and multiplication logic.

The sigmoid activation function while delivering better accuracy also complicates the implementation. The sigmoid based architecture requires a look-up table which increases memory requirements — however, a simulation with reduced bit width indexing shown minimal od no lost in accuracy allowing for memory optimizations.

Overall results meet the expectation; the models show accuracy close the reference implementation. The 8-bit fixed point and u-law models show potential for further use while delivering good trade-off between resources requirements and accuracy.

References

[1] A. Ng. Machine learning. [Online]. Available: https://www.coursera.org/learn/machine-learning

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[3] J. L. Hennessy, *Computer architecture: a quantitative approach*, 5th ed., ser. The Morgan Kaufmann Series in Computer Architecture and Design. Amsterdam ; Boston: Morgan Kaufmann/Elsevier, 2012.

[4] R. Yates, "Fixed-point arithmetic: An introduction," Jan 2013. [Online]. Available: http://www.digitalsignallabs.com/fp.pdf

[5] E. L. Oberstar, "Fixed-point representation & fractional math," Aug 2007.

[6] C. W. Brokish and M. Lewis, "A-law and mu-law companding implementations using the tms320c54x." [Online]. Available: http://www.ti.com/lit/an/spra163a/spra163a.pdf

[7] L. H. Crockett, R. A. Elliot, and M. A. Enderwitz, *The Zynq Book Tutorials for Zybo and ZedBoard*. UK: Strathclyde Academic Media, 2015.