PROJECT REPORT

Ritik Gupta (IMT2018518)
Rutvi Padhy (IMT2018519)

INTRODUCTION:

A lot of machine learning and image processing models use exponential and logarithmic functions as part of their activation functions to process data. Since they are complex functions, it becomes difficult at the hardware level to be power-efficient compared to other functions as there remains a lot of multiplication and additional overhead (which can be seen from the Taylor Series implementation of the functions). The applications where these functions are used though are fault-tolerant and can indeed resist a bit of error compared to other applications. Hence, we decided to take up the task of designing custom modules for the exponential and logarithmic functions and explore how much efficient the custom modules can be made for a bit of accuracy trade-off, and reducing the total resource utilization of the said functions.

To make a formal comparison between the existing floating-point exponential and logarithmic functions, with our own designs of the modules, we worked with a Basys3 Artix-7 FPGA board and gathered the relevant data by testing the functions with the pre-designed modules on the same.

EXPONENTIAL MODULE:

Before proceeding towards designing a custom module, we tested the pre-built floating point exponential module available on Vivado and got the following result:

Slice LUTs	Slice Registers	Slice	LUT as logic	LUT as memory	Block Ram	DSP
8.83%	5.80%	9.50%	8.16%	1.46%	5.80%	2.00%

With a worst negative slack (WNS) of 4.46 ns (tested for x=0.5).
This does not seem much but in the context of the applications we are talking about, a nonlinear function taking up 5-10% of available resources is a lot. Hence, we decided to start working on a new approach.

Taylor Series Expansion:
The Taylor series expansion for the exponential function is:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

For complete accuracy, the function goes till infinity. Graphically, as the number of terms in the series are decreased, the plot of the series starts deviating from the function after a certain time. The more terms are reduced, earlier the deviations take place in the series from the actual function. For testing purposes, we decided to take a small interval of values which the module can take and concurrently take only a few terms in the series and design the custom module using the pre-built addition and multiplication modules.

As a starting point, we decided to go till 3 terms in the Taylor series expansion and worked our way up till 6 terms in the Taylor series. In the context of Vivado, all we did was just adding multiplication and addition floating-point units to multiply the previous term's x^n with x and do the same with the denominator, get their division and perform a rolling addition of the terms.

The results of this exercise (exponential.v) were not as good as we expected. As the number of terms in the series increased from **3 to 6**, the total resource utilization increased, as expected. Total **slice LUT** utilization went from **10.79% to 29.80%** at 6 terms whereas the total **Block RAM** usage went from **8.51%** at 3 terms to **21.81%** at 6 terms. There was a similar trend seen in all the other resource utilization parameters. The error, as expected, decreased as we moved from 3 terms to 6 from an **absolute error proximity of 0.02372 to 2.33e-05** (for x=0.5). One thing to note is that, with an increasing number of terms, the interval within which a value close to the exponential function can be generated by the Taylor series is also increased (for example, for terms=4, x E [-1,1]). The in-depth results of this exercise are available at exponential_comparison.xlsx.

As can be seen from the above results, the resource utilization increased a lot and hence was not the right approach to tackle this problem. It performed way worse on both the accuracy and the resource parameters. We learned that using a lot of multiplication, addition and division floating point units is not helping the task in any way and will only worsen the utilization as we try to increase our interval within which x can be given.

Piecewise Linear Function [1]:
To counteract the problem of growing floating-point units with each addition of term, we decided to use an altogether different approach wherein the number of floating-point units remain constant irrespective of the range of x which is inputted into the module to improve the resource utilization parameter of our problem albeit at a decreased accuracy.

The plot of the exponential function is divided into small slices of lines and the slope, and the intercept of each such slice is pre-computed. The length of each slice depends on the step size which the user wants to keep, where, a lower step-size corresponds to a higher accuracy as it covers only a small range of points for which the line gives a constant value (y=mx+c).

We developed a python script which would generate all of the slopes and the constants required for this operation and all these values would be saved as 32-bit floating point units in the LUTs of the FPGA. This is done so as LUTs are closest to the processing unit of the FPGA and hence it saves on time and thus enhances the computational speed of the FPGA.

For a particular x, between an interval of [a, b], the Vivado code constantly adds step size s to a till a+ns-x>0 which signifies that a+(n-1) s is the closest value smaller than x which is required for computing the slice and therefore the final value of the function. After getting the value, the slope m and the intercept c, corresponding to the closest value is extracted from the precomputed LUT and the final value is calculated using a simple floating-point addition and multiplication of those terms with x.

Compared to the previous method, this method proved to be a lot more effective in reducing resource allocation. As expected, the resources used decreased as the step size increased (as increase in step size ->decrease in number of values which needs to be stored to calculate e^x).
For a **step size of 0.1**, the percentage of **LUT slices** used was **0.65%**, **slice registers** and **block ram** used were **0.29%,** whereas, for a **step size of 0.5**, the percentage of **Slice LUTs** further decreased to **0.42%**, and **slice registers** decreased to **0.17%**. More in-depth data for this exercise is available at exponential_LUT_comparison.xlsx.

This was a big improvement over the previous approach and also performed a lot better over the conventional exponential floating-point unit available in Vivado. The **proximity error** computed for **step size of 0.1** was **7.99e-05** which increased to **0.02** for a **step size of 0.5**. This, though, is not an issue as the resource utilization is already so low that the step size could be further reduced to improve accuracy.

LOGARITHMIC MODULE:

Just like in the exponential module, first, the pre-built logarithmic floating-point module was tested as a benchmark, and we got the following results:

Slice LUTs	Slice Registers	Slice	LUT as logic	LUT as memory	Block Ram	DSP
2.04%	1.75%	2.11%	1.97%	0.16%	0.00%	4.44%

With a worst negative slack of 5.877 ns.

Having realized that the Taylor Swift expansion for exponential function did not work as expected. We decided not to use the same approach for logarithmic function as it would have involved the same method of adding addition, multiplication and division modules with each increasing term in the series, eating up on the available resources for the sake of improving accuracy for longer intervals.

Thus, we decided to use another piecewise function approach called the ICSILog Algorithm.

ICSILog Algorithm [2]:

A 32-bit floating point number is made of a sign bit, 8-bit exponent (exp-128 for the exact number) and remaining 23 bits of mantissa where the mantissa represents the fractional part of a number (1+fractional part to be exact). Basically:

$$val = 2^{exp}.mantissa$$

Using a series of mathematical tricks, to get $\log_e(val)$, we can convert it into following:

$$\log_e(val) = (exp + \log_2(man)) \cdot \log_e(2) =$$

$$exp \cdot \log_e(2) + \log_2(man) \cdot \log_e(2)$$

We already know the value of $\log_e 2 = 0.693$ and exp can easily be retrieved from the input x, thus, the left part of the expression can be easily calculated. As for the right part, the mantissa term and its $\log_2(\text{value})$ is stored in LUTs. To optimize the function, i.e., to improve area and power savings, the number of bits in mantissa can be reduced which would take a hit on accuracy. This can be done by removing q bits at the end from the mantissa and using the remaining bits to calculate log and then finding the final term as shown above.

For filling the LUTs, q has to be decided first and based on that, [0,1] is divided such that each bit change represents a number within that range. This can be easily done by first calculating what each slice's value would be depending on each bit change and then assigning each bit to a value (using the method of handling fractions in binary from decimal). The script for the same has been added in the project.

To calculate $\log_e(\text{val})$, first, $\log_2(\text{man}[23\text{-}q])$ is found (which is already present in LUTs), and added with exp, and finally the term is multiplied $\log_e 2$ to get the final result.

As for resource allocation, we checked how much resource was being utilized for q=9, 13, 17 and noticed something interesting. Irrespective of the value of q (and x), the resources utilized by the module remained constant. They were:

Slice LUTs	Slice Registers	Slice	LUT as Logic	LUT as Memory	Block RAM	DSPs	Bonded IPADs

0.15%	0.07%	0.13 %	0.14%	0.03%	0.07 %	0.00%	1.89%

As for error, the proximity error increased with increasing value of q (as number of mantissa bits used for quantization decreased) from 2.45e-05 at q=9 to 0.079 at q=17.

The resource utilization is very low compared to the conventional logarithmic module available in Xilinx and the accuracy hit is not too sharp which makes the module usable to a certain degree.

Optimized Settings for the modules:
Having done the above exercises, we wrote a script to determine what step size (in case of exponential module) and what value of q works best for the above two modules so as to minimize error and at the same time use it to save power and area.

The script ran all of the step sizes from 0.01 to 1 for the exponential module and generated all of the values in the interval and tested them against the algorithm discussed above and calculated the error for each value. With no surprise, for the lowest step size, the mean average error was the lowest. Hence, for a step size of 0.01, the algorithm worked the best. This gives us an indication that to attain the best possible settings for a use-case, the step size has to be kept as low as possible given the resources available.

Similarly, for the logarithmic module, we tested all of the values of q for which the mean average error was calculated for all the values between a certain interval, and, again, with no surprise, we found out that for q=0, the mean average error was the least with the highest error being for q=22. This again gives an indication that q has to be kept as low as possible given the specifications of the system.

CONCLUSION:

We started out to create custom modules for both the non-linear functions and managed to develop designs that saved up a lot of resources without taking a major hit on resources. These designs can be used to create more complex activation functions (for e.g., SoftMax) which are very often used in machine learning and image processing applications. These complex functions, with our modules, should work a lot faster and take up a lot less resources compared to a normal SoftMax module created for 100% accuracy (which is not really required).

Alternatively, these designs can be used to develop custom instructions in a RISC-V instruction set (where, to our knowledge, they do not exist yet), which dramatically reduce a lot of instruction overhead if someone had to calculate a log or exponential values.

References:

[1] X. Geng, J. Lin, B. Zhao, A. Kong, M. M. S. Aly, and V. Chandrasekhar, "Hardware-aware softmax approximation for deep neural networks," in Computer Vision – ACCV 2018, C. Jawahar, H. Li, G. Mori, and K. Schindler, Eds. Cham: Springer International Publishing, 2019, pp. 107–122.

[2] O. Vinyals and G. Friedland, "A hardware-independent fast logarithm approximation with adjustable accuracy," in 2008 Tenth IEEE International Symposium on Multimedia, Dec 2008, pp. 61–65.