LogGen: A Parameterized Generator for Designing Floating-Point Logarithm Units for Deep Learning

Pragnesh Patel*, Aman Arora†, Earl Swartzlander‡, Lizy John§
The University of Texas at Austin, United States
*prp1998@utexas.edu, †aman.kbm@utexas.edu, ‡eswartzla@utexas.edu , §ljohn@ece.utexas.edu

***Abstract*—Several applications like Deep Learning (DL), Image Processing, and Digital Signal Processing (DSP) rely on the frequent and efficient computation of the logarithm function. Many of these applications use lower precision floating-point datatypes like IEEE half-precision (FP16), bfloat16 (BF16), tensorfloat32 (TF32) instead of single-precision (FP32) and double-precision (FP64). This is because lower precision reduces the computational complexity and memory bandwidth required, albeit with a small degradation in accuracy. While developing logarithm units for FP32 and FP64 datatypes has received a lot of attention, not a lot of effort has been put into the designs of logarithm units for smaller datatypes. Also, different DL applications have different area, delay, memory, accuracy, and datatype requirements. A one-size-fits-all design cannot satisfy all these requirements. This paper presents an open-source, parameterized generator, called LogGen, for generating logarithm unit implementations optimized for smaller floating-point datatypes. LogGen enables generation of designs by varying multiple knobs - precision, accuracy, base of logarithm, storage, and latency. It uses a flexible and efficient Look-Up Table (LUT) based architecture that leverages the small size of datatypes to optimize this architecture. Design space exploration using LogGen is presented. The experimental results show that LogGen designs can outperform commercial IPs (Synopsys DesignWare, Xilinx) and open-source IPs (FloPoCo) in terms of area and delay metrics.**

***Index Terms*—Floating-point Arithmetic, Logarithm, Deep Learning Hardware, Tools, Generator, FPGA, ASIC**

I. Introduction

Elementary functions like the logarithm find use in many areas such as Deep Learning (DL), Digital Signal Processing (DSP), image processing, and bioinformatics.Complex arithmetic operations like multiplications, divisions, and reciprocals can be simplified by incorporating a logarithmic arithmetic circuit. In DL, logarithmic data with arbitrary log-bases has been used to accelerate convolutional neural networks in hardware [13]. Efficient hardware implementation of neural network layers like Softmax requires logarithm units of various fixed and floating-point datatypes [14]. Thus, for DL and several other applications, there is an increasing need to design efficient hardware for computing logarithm.

Most of the DL applications use smaller floating-point datatypes like half-precision (FP16), bfloat16 (BF16) and tensorfloat32 (TF32) instead of single-precision (FP32) and double-precision (FP64). BF16 (or Brain Floating Point) is a 16-bit truncated version of the 32-bit IEEE single-precision floating-point datatype used in Intel DL processors, FPGAs and Google TPUs. TF32 is an emerging floating-point datatype that occupies 19 bits and is used in Nvidia A100 GPUs.

The precision used for DL algorithms impacts the hardware's area and performance metrics. A lower precision reduces computational complexity and memory bandwidth required, but leads to lower accuracy. It has been shown that a lower internal computational accuracy does not lead to a significant overall accuracy loss in neural networks [9]. The NPU architecture for Microsoft Brainwave used a narrow precision datatype called Microsoft Floating-Point (MSFP) with 5-bit exponents and mantissas as low as 2 to 5 bits with a negligible impact on accuracy [6]. Thus, hardware for DL can be optimized for area and delay at the cost of accuracy.

Different DL accelerators require support for different datatypes and have different budgets for area, delay, memory and latency. Different applications have different tolerance for accuracy. Some applications might even require logarithm computation for arbitrary bases. A one-size-fits-all design cannot satisfy all the requirements in such a diverse application space. Thus, in this paper, a tunable generator called LogGen is presented that can generate different LOG units that are fine-tuned to fit user requirements (precision, accuracy, base of logarithm, storage and latency). Ad hoc techniques of exploration can miss out efficient implementations leading to inefficient accelerators. Thus, LogGen provides a valuable and convenient tool to explore the LOG unit design for different application scenarios and requirements. No such tool currently exists in the open-source community.

The base architecture of the LOG unit used by LogGen must accommodate the range of features required by the generator and deliver satisfactory performance, resource usage, and accuracy metrics. To achieve this, the base architecture uses a LUT-based approximation based on a simple formula that is mentioned in Section II-B. The LUT-based design also enables efficient mapping to both ASIC and FPGA architectures.

The contributions of this paper are summarized as follows:

- A generator called LogGen that is a tool to dump Verilog code for the required LOG units. It is controlled by 5 knobs - precision, accuracy, base of logarithm, storage and latency - that can take different values.
- A base architecture that has been optimized to compute the logarithm of small floating-point datatypes (like FP16, BF16 and TF32) and is flexible in terms of area, accuracy, delay and latency.
- Design space exploration using LogGen and analysis of trade-offs between area, delay and accuracy. The various generated LOG unit designs are evaluated, and compared with commercial (Synopsys DesignWare, Xilinx Floating Point Operator), open-source (FloPoCo) designs and a past published implementation [1]. As an application example, the generated LOG units are integrated into a Softmax design.

II. Background

A. Floating Point

Assume that num is a floating-point number such that $num = (-1)^s * 2^{exp} * mant$, where s indicates the sign of the number (with $s \in \{0, 1\}$), exp indicates the exponent and $mant$ indicates the mantissa. According to the IEEE-754 standard for floating-point arithmetic, $mant$ is a normalized mantissa with $mant \in [1, 2)$, except for the subnormal range. The leading bit of the mantissa is excluded in this representation because it is always a constant one.

Different floating-point datatypes can be created by varying the number of bits used to represent the exponent and mantissa parts. For example, the FP16 datatype has a 5-bit exponent and a 10-bit mantissa, the BF16 datatype has an 8-bit exponent and a 7-bit mantissa. The more recent TF32 datatype uses the same 10-bit mantissa and 8-bit exponent as FP16 and BF16 respectively. These 3 datatypes are shown in Fig. 1.

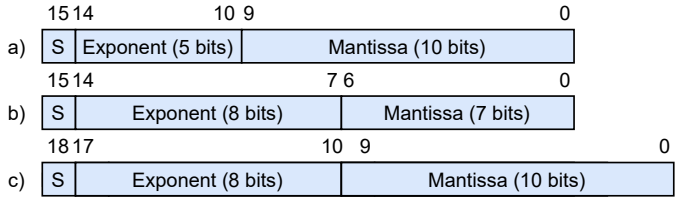

Fig. 1: Floating point representation of a) FP16, b) BF16 and c) TF32

B. Formula

As seen in the Section II-A, a floating-point number consists of three fields: the sign, the exponent (exp), and the mantissa $(mant)$. Since the logarithm function is only defined for positive numbers, the sign bit can be ignored. The final value of $log(num)$ can be calculated using the multiplicative property of the logarithm function as follows:

$$log(num) = exp \times log(2) + log(mant) \quad (1)$$

In this paper, Equation (1) is used to compute the floating-point logarithm function. A LUT-based method is used to evaluate the terms - $exp \times log(2)$ and $log(mant)$ - in hardware. [1] [5] [10] also use this formula. However, different techniques are used by them to calculate the terms of the formula. Further details will be provided in subsequent sections.

III. Previous Work

The problem of implementing logarithm has been thoroughly explored, although a lot of the published works are not optimal for the new generation of applications. A lot of work has been focused on fixed-point datatypes, traditional floating-point (FP32 and FP64) datatypes [10] [1] and parameterized log units [5], but no work specifically focuses on optimizing the designs for small floating-point datatypes.

Certain proprietary IPs like the Synopsys DesignWare Library [12] and the Xilinx Floating-Point Operator [15] provide a parameterized logarithm unit. However, in Section VI, it is shown that the Synopsys DesignWare library logarithm unit is very slow and requires a lot of area. Also, the Xilinx Floating-Point Log Operator consumes a lot of resources and only supports FP16, FP32 and FP64 datatypes.

In [5] and [10], higher order approximations and Taylor series were used to design logarithm units. However, using such techniques for small datatypes is inefficient in terms of resources and latency as it involves several floating-point multiplications and additions. Moreover, Taylor series can only be used to calculate natural logarithm while our base architecture supports any arbitrary logarithm base.

Dinechin, Klein and Pasca presented FloPoCo, an open-source arithmetic core generator [4]. However, there is a lack of compliance with the IEEE-754 standard. In Section VI, it is shown that the FloPoCo LOG units are more accurate, but slower and larger in comparison to the LogGen designs.

Alachiotis and Stamatakis proposed a logarithm unit for FPGAs for FP32 and FP64 datatypes [1]. However, their approach involves additional hardware units like subtractor, multiplexer, fixed to FP32 datatype conversion logic, and floating-point multiplier that can be easily avoided for smaller datatypes using the LUT-based approach shown in this paper.

IV. Architecture

The baseline architecture of the LOG unit for FP16, BF16 and TF32 datatypes is shown in Fig. 2. This section describes the LOG unit architecture for FP16 datatype that has a 5-bit exponent and 10-bit mantissa. The architecture is physically divided into 4 blocks as described below.

A. Look-Up Table for the Exponent (LUT-EXP)

For a naive hardware implementation of Equation (1), the calculation of the first part of the sum: $exp \times log(2)$ requires subtracting the biased 5-bit fixed-point value of the exponent field with the bias (i.e. 15 for FP16), converting the result of subtraction from fixed-point to the respective FP16 value, followed by a floating-point multiplication with $log(2)$. This design uses a faster and smaller LUT-EXP that directly stores the 2^5 FP16 values of the final result of the $exp \times log(2)$ multiplication and requires only 0.5Kb ($2^5 * 16$ bits) of memory.

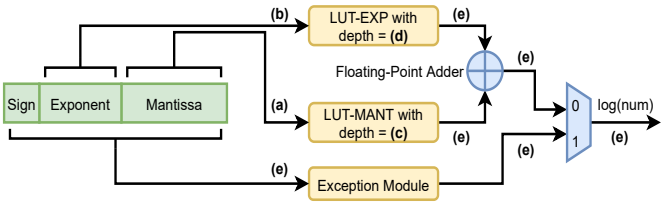

Fig. 2: Proposed Architecture for FP16/BF16/TF32 LOG Units. a/b/e denote bit widths. a/b/c/d/e take different values for different datatypes. a = 10/7/10, b = 5/8/8, c = 1024/128/1024, d = 32/256/256, and e = 16/16/19 for FP16/BF16/TF32 respectively.

B. Look-Up Table for Mantissa (LUT-MANT)

The calculation of $log(mant)$ requires the use of LUT-MANT. The LUT-MANT is initialized with all the pre-computed FP16 values for $log(mant)$. Since FP16 has 10 bits of mantissa, it requires 16Kb ($2^{10} * 16$ bits) of memory as compared to the 256Mb ($2^{23} * 32$ bits) required for a full-size LUT-MANT for FP32. The small size of the LUT for FP16 makes it unnecessary to use other complex approximation

techniques. The Accuracy knob of LogGen allows the user to choose the number of most significant bits of mantissa that are used to index the LUT-MANT. Thus, the user can reduce the size of the LUT in exchange for a lower accuracy. Section VI-B, provides a comparison between the accuracy loss, area reduction and delay for designs with smaller LUT-MANT.

C. Floating-point Adder

The outputs of LUT-EXP and LUT-MANT are added in the floating-point adder to obtain the final logarithm value. The Precision and Pipeline knobs of LogGen allow the user to choose between different implementations of the floating-point adder.

D. Exception Module

The architecture described here does not support subnormals. The logarithm output for both zero and subnormal inputs is $-inf$. An optional exception module can also be added to map the output for negative inputs to NaN (Not a Number) if the application requires support for negative inputs. Then the final result will be obtained by multiplexing the output of the adder with the output of the exception module.

E. Cancellation Problem

The sum of $exp \times log(2)$ and $log(mant)$ in Equation (1) may result in a massive cancellation when num is approaching 1 but is less than 1 ($exp = -1$ and $mant \rightarrow 2$). The technique used to handle this problem leads to a greater resource usage and more complex design [5] [10]. However, in the base architecture, since the values of both the terms of the sum are precomputed with high accuracy and are stored in the 2 LUTs, the stored LUT values are such that cancellation will only occur when $num = 1$. Also, the closest number to 1 that can be represented by FP16 datatype (i.e. s = 0, exp = -1 and mant = 8'hFF) is 0.9995. The absolute error between the expected and observed value of $ln(0.9995)$ is 0.0001. This error is much less compared to the the maximum absolute error of the base LOG unit as shown in Section VI-B. Thus, due to the LUT-based approach and low precision of smaller datatypes like FP16, the architecture presented in this paper avoids this cancellation problem. Moreover, this makes the architecture cheaper in terms of resources as compared to [10] and [5].

F. Other datatypes

The base architectures of the LOG units for BF16 and TF32 datatypes are similar to that of FP16. The differences in the architectures for BF16 and TF32 from FP16 are the sizes of LUT-EXP and LUT-MANT (as shown in Table I) along with the datatype supported by the floating-point adder.

Table I. Size and memory requirements of LUT-EXP and LUT-MANT for base LOG units of FP16, BF16 and TF32 datatypes

LUTs in base LOG unit		**FP16**	**BF16**	**TF32**
LUT-EXP	No. of entries	32	256	256
	Size of memory (Kbits)	0.5	4	4.75
LUT-MANT	No. of entries	1024	128	1024
	Size of memory (Kbits)	16	2	19

V. LogGen

This section describes the flow, structure and knobs of LogGen. Fig. 3 provides a top-level view of the generator.

A. Flow and Structure

The inputs to the generator are the values of various knobs - precision, accuracy, base of logarithm, storage and pipeline. These knobs control the different aspects of the base architecture. The outputs of the generator are a set of Verilog files for the top-level module, LUT-EXP, LUT-MANT and floating-point adder blocks. The Makefile available can be used to generate the designs.

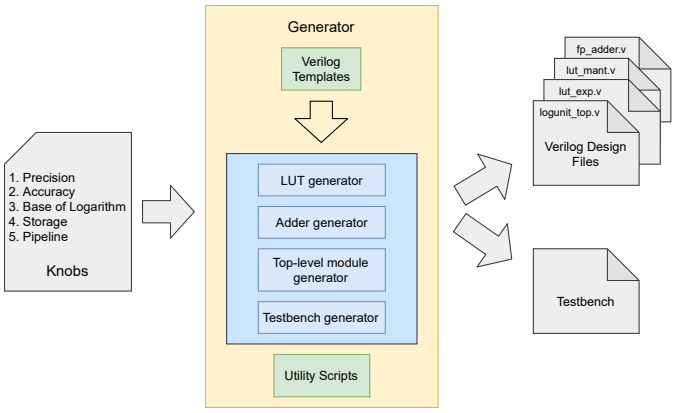

Fig. 3: Flow and structure of LogGen

There are two components inside the generator: Verilog templates and Python scripts. The Verilog templates are skeleton designs with different labels placed at different locations. The Python scripts process the template, replace the labels with Verilog code based on the input knobs and dump the Verilog files. There are separate scripts for the LUT-EXP and LUT-MANT blocks that are used to pre-compute the LUT values based on the input knobs.

B. Knobs

Hardware implementation trade-offs for the LOG unit can be explored using the following 5 knobs:

Precision: This knob defines the floating-point precision (datatype) supported by the generated design. It controls the exponent and mantissa width of the floating-point input. For example, if the input of the knob is {5,10}, then the generated LOG unit will be for FP16. This knob supports any positive exponent and mantissa bit-width values as inputs. However, for datatypes like FP32 and FP64 with large bit-widths, the LUT sizes of the design will become very large and the generated designs will not be efficient in terms of area and delay.

Accuracy: This knob controls the number of most significant bits of mantissa used to index LUT-MANT. Thus, this knob defines the LUT-MANT size. Positive values less than or equal to the mantissa width of the Precision knob will be considered as valid inputs. For example, the mantissa width for FP16 is 10 but the first 8 bits from the most significant bit of the mantissa can be used to index LUT-MANT. The last 2

bits are ignored. This reduces the size of LUT-MANT from 2^{10} to 2^8 entries, but lowers the accuracy of the design.

Base of Logarithm: This knob defines the base of the logarithm function of the design. The base LOG unit has the flexibility to support any arbitrary base as the LUT values can be pre-computed and stored for any given base. All positive numbers are considered as valid inputs. The default value for this knob is e and the LOG unit calculates natural logarithm. Other values like 2 or 10 can be used the generate LOG units that compute $log_2(x)$ or $log_{10}(x)$ respectively. This knob does not affect any of the area, delay, and latency metrics and is mainly dependent on the user's functional requirements.

Storage: The LUT values can either be implemented using logic gates (LG) or can be stored in hard macro RAMs (RAM) or flip-flop based memories (FF) for ASIC designs. For FF, the synchronous single-port, read/write flip-flop based RAM (DW_ram_rw_s_dff) from the Synopsys DesignWare library was used. The simulation and synthesis models for the hard macro RAM were obtained using the OpenRAM tool [8]. For FPGA designs, these values can be implemented using configurable logic blocks (CLB) or stored in Block RAMs (BRAM). This knob can take 5 values for the storage type - LG, SRAM, FF, CLB, and BRAM - and allows the user to choose between an ASIC or FPGA design.

Pipeline: This knob is used to choose between the different floating-point adder implementations. Currently, this knob is only valid for ASIC designs and can take two values - pipe or no_pipe. There are two different implementations of the floating-point adder. The first one has been obtained from the Synopsys DesignWare IP library called DW_fp_addsub and the other one is a custom design called FP_AddSub that has been designed by us. For FP_AddSub, we use the architecture described in [3] (hard multiplier and adder based, not the soft logic based, not the iterative design; also, we use a different pipelining scheme). Compared to FP_AddSub, DW_fp_addsub has a smaller area but is slower as it is not pipelined. If required, users can easily integrate their own adders by modifying the floating-point adder module instantiation in the top-level module of the generated design. For FPGA implementation, the Xilinx Floating-Point Addition Operator [15] was used to perform the floating-point addition.

VI. Experimental Results

This section discusses the methodology and observations of the experiments conducted. Verilog was used to create the designs and testbenches. For the FPGA evaluation, Xilinx Vivado 2019.2 was used for simulation and implementation, with Xilinx Zynq XC7Z020-1CLG400C SoC FPGA. For the ASIC evaluation, Synopsys VCS and Synopsys Design Compiler were used for simulation and synthesis respectively. The area values are post-synthesis and pre-placement/routing areas. The library used for synthesizing the designs is the FreePDK45 [11] that uses a 45nm ASIC process. For the accuracy assessment of the design, the observed output values of the LOG units were compared to the expected output values from a Python-based CPU model for 10^5 randomly generated inputs.

A. Exploration based on Precision knob

For this experiment, the Precision knob of the generator was varied. As expected, the BF16 LOG unit has the smallest area since it has the smallest LUTs. The FP16 LOG unit requires less area than TF32 LOG unit because it has a smaller LUT-EXP. The frequencies of all the 3 LOG units are about the same. This is because the DW_fp_addsub adder is used. The total latency of designs with the Pipeline = no_pipe is 2 clock cycles as there is one pipeline stage between the LUTs and DW_fp_addsub.

Table II. Area-Delay characteristics of ASIC implementations for different configurations of Precision knob (Storage=lg, Pipeline=no_pipe) and scaled-down FP16 version of [1]

LOG Unit	Area (μm^2)	Freq. (MHz)	Latency (cycles)
Design with PR={5,10}, AC=10	9711	201	2
Design with PR={8,7}, AC=7	4538	193	2
Design with PR={8,10}, AC=10	11479	189	2
FP16 version of design in [1]	13602	201	2

Table III shows the FPGA resource utilization for both FP16 and BF16 LOG units with Storage = BRAM. The FP16 LOG Unit utilizes less CLB resources and more DSP Slices than BF16 LOG unit because the Xilinx Add/Subtract Floating-Point Operator IP can be configured to use the DSP slice resources only for FP16, FP32 and FP64 datatypes. Also, the difference in latency is due to the different implementations of FP16 and BF16 adders provided by Xilinx.

Table III. Resource usage for the FPGA implementation of FP16 and BF16 LOG units

Resource	PR={5,10}, AC=10, ST=BRAM	PR={8,7}, AC=7, ST=BRAM	PR={5,10}, AC=10, ST=CLB
LUTs in CLBs	92	154	275
LUTRAM	0	9	0
FF	297	278	289
DSP Slice	2	0	2
36 Kb BRAM	1	1	0
Freq. (MHz)	322	333	322
Latency (cycles)	14	11	14

B. Exploration based on Accuracy knob

To see the effect of reducing the mantissa width on the area and accuracy, the Accuracy knob was varied from 10 to 5. Fig. 4 shows that the average absolute error increases and area decreases exponentially as the LUT-MANT size decreases. This is because the LUT-MANT size gets approximately halved with every reduction in the mantissa bit-width used to index LUT-MANT. Upon examination, the design with Accuracy = 8 provides the best balance between area and accuracy for FP16 LOG units. The design with Accuracy = 8 has 54% less area than the design with Accuracy = 10 while only having a 17.5% increase in the average absolute error.

C. Exploration based on Storage knob

For ASIC designs, the Storage knob can take 3 values - LG, FF and RAM. Table IV shows that the least area was utilized when hard logic gates were used to implement the LUTs. This

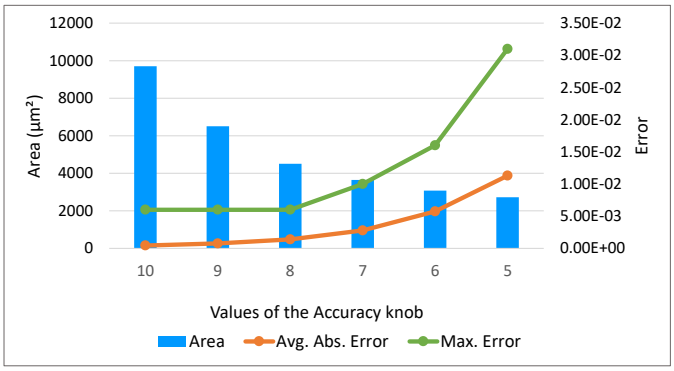

Fig. 4: Trade-off between area, average absolute error and max error with varying values of the Accuracy knob (Precision={5,10}, Storage=LG and Pipeline=no_pipe)

is because using a flip-flop based memory or hard macro RAM block to implement LUTs with small table sizes is not optimal in terms of area and delay [7]. However, hard logic gates are not optimal for large LUTs used in datatypes like FP32 and FP64 as they will have a high combinational delay and area compared to a hard macro RAM.

Table IV. ASIC area comparison for different values of Storage knob with Precision={5,10}, Accuracy=5 and Pipeline=no_pipe

LOG Unit	Area (μm^2)
Design with ST=LG	2727
Design with ST=RAM	6230
Design with ST=FF	20042

For the FPGA implementation, the Storage knob can be either BRAM or CLB. The Xilinx 7 series FPGA BRAMs can store upto 36Kb of data (can be configured as two independent 18Kb blocks too). Table I shows that the largest LUT-EXP and LUT-MANT for FP16, BF16 and TF32 datatypes can be fit into the 18Kb BRAM blocks. Hence, the total BRAM resource required is one irrespective of the Accuracy knob. LUTs in CLBs can also be used. Table III, shows that there is an increase of 183 in LUT usage when CLBs were used to implement the LUT-EXP and LUT-MANT. The user can decide the Storage knob value based on the BRAMs and CLBs availability. The frequency and latency remained same for both designs with BRAM and CLB as those values are dependent on the Xilinx Floating-Point Adder Operator.

D. Exploration based on Pipeline knob

For the ASIC implementation, either the DW_fp_addsub or the FP_AddSub adder can be used. As expected, Table V shows that the designs with the the pipelined FP_AddSub adder (Pipeline = pipe) have higher frequency and latency. However, they are less accurate and consumes more area than designs with the DW_fp_addsub adder (Pipeline = no_pipe).

Table V shows that the frequency of the generated designs with Pipeline = no_pipe is the same. This is because the critical path delay and the frequency of the design is always decided by the DW_fp_addsub.

However, the frequencies for the designs with Pipeline = pipe are different. This is because the critical path of the designs can either be LUT-MANT or FP_AddSub adder. For FP16 designs with Pipeline = pipe, the frequency is decided by LUT-MANT for Accuracy = 9 or 10 and by FP_AddSub for Accuracy <= 8.

Table V. Comparing various metrics between ASIC implementations of generated designs (Precision={5,10}, Storage=lg), DW_fp_ln, and FloPoCo LOG unit

LOG Unit	Area (μm^2)	Freq. (MHz)	Latency (cycles)	Avg. Abs. Error
AC=10, P=no_pipe	9711	201	2	4.45E-04
AC=10, P=pipe	13067	500	11	7.85E-04
AC=8, P=no_pipe	4503	201	2	1.38E-03
AC=8, P=pipe	7859	885	11	1.87E-03
DW_fp_ln	6201	144	1	7.42E-04
FloPoCo	11479	333	9	2.92E-04

E. Comparison with other ASIC implementations

The LOG unit architecture mentioned in [1] has been designed for FP32 and FP64 datatypes. Hence, for a fair comparison their LOG unit was scaled down to FP16 using the same Synopsys DesignWare Library blocks as the base LOG unit. Table II, shows that the generated design (Accuracy = 10, Storage = LG and Pipeline = no_pipe) has a 1.4 times smaller area than the scaled down version. The frequency and latency is the same because DW_fp_addsub is the critical path.

DW_fp_ln is the parameterized LOG unit available in the Synopsys DesignWare library. Table V, shows that the DW_fp_ln design has the lowest frequency among all the designs. This is because it is not pipelined. The generated design with Accuracy = 8 and Pipeline = no_pipe is faster and consumes lesser area but is less accurate than DW_fp_ln. The generated design with Accuracy = 8 and Pipeline = pipe has a 6.1 times higher frequency at the cost of 1.2 times the area of DW_fp_ln.

The FloPoCo LOG unit requires the largest area among all the designs. However, it is the most accurate design. The generated FP16 design (Accuracy = 8, Storage = LG) with Pipeline = pipe has a 2.65 times higher frequency and 1.46 times lesser area. Thus, the user can generate LOG units using LogGen that provide metrics as good as if not better than the existing commercial and open-source LOG units.

F. Comparison with other FPGA implementations

Both Xilinx Vivado HLS and IP Generator can be used to generate the FP16 log unit. However, the HLS design internally used the same log unit as the IP Generator. The IP Generator only supports FP16, FP32, and FP64 datatypes for the log unit. Thus, it doesn't have support for custom datatypes like like LogGen. Table VI shows that the Xilinx FP16 log unit is more accurate, but utilizes more resources than the generated FP16 design with Accuracy = 8 and Storage = CLB. Even though it can operate at a slightly higher frequency, it has a higher latency than the generated design. Table VI also shows that the FloPoCo LOG unit consumes the highest amount of FPGA resources, is the slowest among all the 3 designs, and has nearly the same accuracy as the Xilinx FP16 Log Unit.

Table VI. Various metrics for FPGA implementations of Xilinx IP, FloPoCo and the generated LOG unit (PR={5,10}, AC=8, ST=CLB)

Resource	Xilinx IP	FloPoCo	LogGen
LUTs in CLBs	292	470	146
LUTRAM	25	35	0
FF	451	361	289
DSP Slice	2	1	2
36 Kb BRAM	0	0	0
Freq. (MHz)	344	123	322
Latency (cycles)	18	9	14
Avg. Abs. Error	2.93E-04	2.92E-04	1.42E-03

G. Application Level Evaluation - Softmax

The Softmax function is used in several neural network-sRecent state-of-the-art hardware implementations of Softmax use logarithm to avoid large area consumption and accuracy loss caused by division units [14]. For proof of concept and to demonstrate the trade-off between area and accuracy at an application level, a Softmax design based on [14] was created with LogGen LOG units integrated into it. This design is also part of the Koios benchmark suite [2].

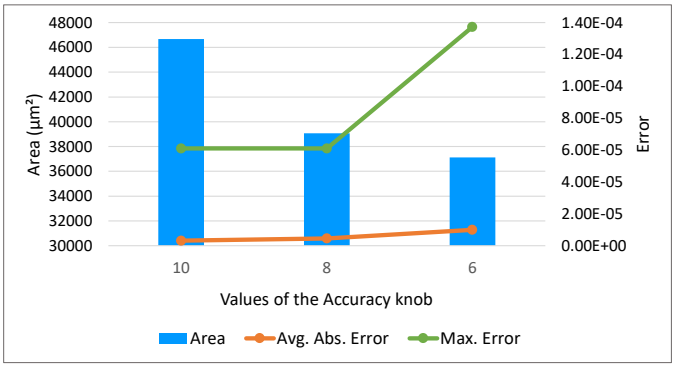

Fig. 5: Area-Accuracy trade-off for Softmax units with different LOG units (Precision={5,10}, Storage=LG, Pipeline=no_pipe)

Fig. 5 shows that the area of design that used the LOG unit with Accuracy = 8 was 16.3% less than the design that used the LOG unit with Accuracy = 10. There was a slight increase in the average absolute error of the design while the maximum error stayed the same. In comparison, the area of design that used the LOG unit with Accuracy = 6 was 20.5% less than the design that used the LOG unit with Accuracy = 10. There was an increase in both average absolute and maximum error as well. This observation is in-line with the observation in Section VI-B, that the design with Accuracy = 8 provides the best balance between area and accuracy for FP16 LOG units. The frequency of all three designs was the same as the critical path was not in the LOG unit.

VII. Conclusion

This paper presents an open-source generator called LogGen that can create LOG unit designs by varying 5 aspects (precision, accuracy, base of logarithm, storage & pipeline) of a flexible base architecture. LogGen is available at: (*https://github.com/pragneshp7/LogGen*). The base architecture has been optimized for small floating-point datatypes.

The goal of LogGen is to enable evaluation of trade-offs between area, delay, latency, and accuracy of the LOG unit. With new datatypes (like TF32 in Nvidia GPUs and MSFP in Microsoft Brainwave) constantly emerging and becoming increasingly popular in DL, LogGen can be useful to quickly generate efficient LOG units for these emerging datatypes. In addition to user-defined datatype, LogGen can be used to generate LOG units for a user-defined logarithm base.

The generated designs were compared to commercial and open-source IPs along with a past implementation. Based on the comparisons, it can be concluded that the LogGen designs are adequately accurate, and the area and delay metrics are as good as if not better compared to the other existing designs.

VIII. Acknowledgement

This research was supported in part by NSF grant number 1763848. We thank anonymous reviewers for the detailed comments on the paper. Authors would also like to acknowledge computational resources from Texas Advanced Computing Center (TACC). Any opinions, findings, conclusions or recommendations are those of the authors and not of the National Science Foundation or other sponsors.

References

[1] N. Alachiotis and A. Stamatakis, "A vector-like reconfigurable floating-point unit for the logarithm," *International Journal of Reconfigurable Computing*, vol. 2011, 2011.

[2] A. Arora, A. Boutros, D. Rauch, A. Rajen, A. Borda, S. A. Damghani, S. Mehta, S. Kate, P. Patel, K. B. Kent, V. Betz, and L. K. John, "Koios: A deep learning benchmark suite for fpga architecture and cad research," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 355–362.

[3] F. Brosser, H. Y. Cheah, and S. A. Fahmy, "Iterative floating point computation using fpga dsp blocks," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–6.

[4] F. De Dinechin, C. Klein, and B. Pasca, "Generating high-performance custom floating-point pipelines," in *2009 International Conference on Field Programmable Logic and Applications*. IEEE, 2009, pp. 59–64.

[5] J. Detrey and F. de Dinechin, "A parameterizable floating-point logarithm operator for FPGAs," in *Conference Record of the Thirty-Ninth Asilomar Conference on Signals, Systems and Computers, 2005.*, 2005, pp. 1186–1190.

[6] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 1–14.

[7] X. Geng, J. Lin, B. Zhao, A. Kong, M. M. S. Aly, and V. Chandrasekhar, "Hardware-Aware Softmax Approximation for Deep Neural Networks," in *Asian Conference on Computer Vision*. Springer, 2018, pp. 107–122.

[8] M. R. Guthaus, J. E. Stine, S. Ataei, Brian Chen, Bin Wu, and M. Sarwar, "OpenRAM: An open-source memory compiler," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–6.

[9] J. Johnson, "Rethinking floating point for deep learning," *arXiv preprint arXiv:1811.01721*, 2018.

[10] M. Langhammer and B. Pasca, "Single precision logarithm and exponential architectures for hard floating-point enabled FPGAs," *IEEE Transactions on Computers*, vol. 66, no. 12, pp. 2031–2043, 2017.

[11] NCSU. (2018) Freepdk45. [Online]. Available: https://www.eda.ncsu.edu/wiki/FreePDK45:Contents

[12] Synopsys. (2018) DesignWare Library - Datapath and Building Block IP. [Online]. Available: https://www.synopsys.com/dw/buildingblock.php

[13] S. Vogel, M. Liang, A. Guntoro, W. Stechele, and G. Ascheid, "Efficient Hardware Acceleration of CNNs Using Logarithmic Data Representation with Arbitrary Log-Base," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3240765.3240803

[14] Z. Wei, A. Arora, P. Patel, and L. John, "Design Space Exploration for Softmax Implementations," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2020, pp. 45–52.

[15] Xilinx, "Floating Point Operator v7.1," https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf, 2019.