Resource-Efficient Logarithmic Number Scale Arithmetic for SPN Inference on FPGAs

Lukas Weber*, Lukas Sommer*, Julian Oppermann*, Alejandro Molina†, Kristian Kersting† and Andreas Koch*
*Embedded Systems and Applications Group, TU Darmstadt, Germany.
{weber, sommer, oppermann, koch}@esa.tu-darmstadt.de
†Machine Learning Lab, TU Darmstadt, Germany.
{molina, kersting}@cs.tu-darmstadt.de

***Abstract*—FPGAs have been successfully used for the implementation of dedicated accelerators for a wide range of machine learning problems. The inference in so-called Sum-Product Networks can also be accelerated efficiently using a pipelined FPGA architecture.**

However, as Sum-Product Networks compute exact probability values, the required arithmetic precision poses different challenges than those encountered with Neural Networks. In previous work, this precision was maintained by using double-precision floating-point number formats, which are expensive to implement in FPGAs.

In this work, we propose the use of a logarithmic number system format tailored specifically towards the inference in Sum-Product Networks. The evaluation of our optimized arithmetic hardware operators shows that the use of logarithmic number formats allows to save up to 50% hardware resources compared to double-precision floating point, while maintaining sufficient precision for SPN inference at almost identical performance.

***Index Terms*—FPGA, SPN, Machine Learning, Graphical Models, Deep Models**

I. Introduction

In the past, most of the work on FPGA-based accelerators for machine learning inference has focused on the acceleration of (artificial) neural networks [1], such as the very popular convolutional neural networks.

In contrast, in [2], [3] an automatic tool-flow for mapping the inference for so-called *Sum-Product Networks* (SPN) to an FPGA-based accelerator was developed. Sum-Product Networks are a very promising type of machine learning network, that belong to the class of tractable probabilistic models and share similarities with probabilistic graphical models (PGM).

Compared to "classical" neural networks, SPNs allow *exact* inference, thereby allowing them to explicitly deal with uncertainty over the inputs. However, the fact that Sum-Product Networks compute *exact* probabilities poses a number of unique challenges to the hardware implementation. Most of the optimization techniques employed for the hardware mapping of neural networks, such as quantization of weights, are not readily applicable to SPNs.

In [2], the use of double-precision allowed the preservation of sufficient accuracy at the cost of a relatively high resource requirement per operator. To circumvent this problem, we seek to use an optimization which is commonly used in ML. The use of logarithmic scaling is often used on CPUs and GPUs even though the scaling has to be emulated. Using the flexibility of FPGAs, we aim to implement the logarithmic scaling through a logarithmic number system (LNS). This allows us to maintain sufficient accurary with smaller bitwidths, leading to significant resource savings.

II. Sum-Product Networks

Probabilistic models can be used to solve a range of machine learning problems. For example, the problem of multi-class classification can be solved with probabilistic queries on a PGM by determining the class with the highest probability, given some evidence, i.e., $arg\ max_c\ \mathcal{P}(class = c|evidence)$.

While PGMs are very versatile, they generally suffer from one disadvantage: In general, inference in unrestricted PGMs (e.g. Baysian Networks) is intractable. SPNs overcome this limitation and allow to compute *exact* probabilities in time *linear* wrt. to the network size. Additionally, SPNs inherit the universal approximation property from mixture models, as mixture models can easily be represented as SPNs using a single sum-node. This means that SPNs are able to represent any prediction function, similar to deep neural networks.

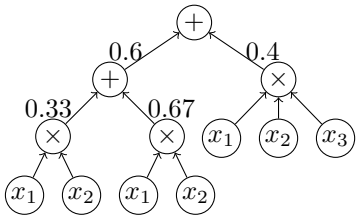

Fig. 1: Example of a valid SPN, capturing the joint probability distribution of the variables x_1, x_2 and x_3.

A. Model Representation

A Sum-Product Network captures the joint probability $\mathcal{P}(X_1, X_2, ...)$ over a set of variables, called the scope of the SPN. The graph structure of the SPN, used to represent this joint probability distribution, is a rooted, directed acyclic graph with three different kinds of nodes: Sum, product and leaf nodes. With these three node types, an SPN can be defined recursively as follows: 1. A tractable, univariate distribution is an SPN. This corresponds to the leaf nodes in the network. 2. A product of SPNs over different scopes

(i.e., random variables) is an SPN, represented by a product node in the network. Essentially, a product node corresponds to a factorization over independent distributions. 3. A convex combination (i.e., weighted sum) of SPNs over the same scope is an SPN. This is equivalent to a mixture of multiple distributions over the same variables and represented by a sum node and the weights associated with each of the child nodes. An example SPN is given in Fig. 1.

B. Inference

To answer probabilistic queries using SPNs the following scheme is used: Given (partial) evidence, histograms at the leaf nodes are evaluated, mapping input values to probabilities. As in [2], based on [4], all leave nodes use discrete input values. Using the probabilities of the leaves, nodes are evaluated bottom-up to calculate the resulting probability. The evaluation always results in a single probability value.

The basic case is the computation of the joint probability $\mathcal{P}(X_1, X_2, ...)$, which corresponds to a single evaluation with full evidence. In order to marginalize out one or multiple variables, it is sufficient to replace the leaf nodes for these variables with the value 1 and evaluate the SPN. Both cases can be combined to compute the conditional probability: $\mathcal{P}(Y|X) = \frac{\mathcal{P}(Y,X)}{\mathcal{P}(X)}$.

In this work, we focus on the joint computation, but the accelerator can be extended to also compute marginals and conditional probabilities.

III. Prior Work

A. Logarithmic Number Scale on FPGA

Regarding the use of LNS, there has been extensive research. Especially for digital signal processing it gained traction through the works of Lewis, which employed a function interpolation scheme using interleaved memory to calculate the logarithmic addition [5]. This resulted in the development of an *arithmetic unit* (AU) which, at the time, outperformed all similar floating point-based AUs [6].

A very similar interpolation-based approach for calculating the logarithmic addition was later used [7]. The work additionally compared FP and LNS and determined, that for LNS to outperform FP in latency and required area, it was necessary that about 70% of operations were multiplicative.

B. SPN Inference on FPGA

To the best of our knowledge, the work presented in [2] is the first and to date only approach to accelerate SPN inference on FPGAs. In that work, an automatic toolflow that maps the inference in Sum-Product Networks to a fully pipelined FPGA-accelerator was developed. The toolflow uses a fully spatial mapping, i.e., for each arithmetic operator in the SPN, there is a corresponding hardware operator in the datapath. For the implementation of the hardware arithmetic operators, the FloPoCo framework [8] was used, with a numeric format similar to IEEE-754 double precision, achieving end-to-end speedups of 6x over an x86-CPU and 38x over a Tensorflow-based GPU-implementation.

In this work, we design LNS operators as drop-in replacement for the FloPoCo-operators in [2] and reuse the automatic toolflow to automatically map SPN inference to the FPGA.

IV. Approach

Similar to Haselman et al. [7], we use a fixed-point number to encode the exponent E_A for a probability value A. In addition to the exponent, a zero-flag Z_A and a sign-flag S_A are used to denote special cases and the sign of the exponent. Since we only consider probabilities (values between 0 and 1), the sign-flag is inherently also a flag that indicates that the linear-scale value is 1, similar to the zero-flag which indicates a linear-scale value of 0.

In contrast to the encodings used by Haselman et al. [7] or Detrey et al. [9], this encoding removes the additional sign-bit, which is used to encode the overall sign for representing negative numbers. In addition, we chose to change from 2's complement encoding of the exponent to an encoding with an explicit sign-flag. Additionally, we do not encode special cases such as NaN or $\pm\infty$. Thus we are able to save a bit, while the magnitude of the exponent gains an additional bit in comparison to [7].

A. LNS Multiplication

The multiplication of values in linear scaling corresponds to an addition in logarithmic scale. This is also visible in the corresponding logarithmic property: $log_2(x \times y) = log_2(x) + log_2(y)$. Assuming correct input values and neither input being 0 nor 1, the calculation is a simple addition of the exponents. If either Zero-flag is set, the result is zero (linear-scale multiplication by 0). Additionally, if the addition of exponents overflows, this results in a value that is too small and thus saturated towards 0. While the resulting sign is $S_R = S_A \vee S_B$, the exponent is $E_R = E_A + E_B$. Additionally, the zero-flag is $Z_R = Z_A \vee Z_B$. The special case handling will override E_R and S_R to zero, if one of the operands zero-flag was set, or if the calculation of E_R overflowed. In actual hardware, this is split into 3 pipeline stages: Decoding, calculation and special-case handling.

B. LNS Addition

In contrast to multiplication, addition is not simplified in the logarithmic scale. Instead, the calculation of an addition in logarithmic scale is *harder* than in linear scale. The main challenge with the logarithmic addition is obvious in the corresponding logarithmic property: $log_2(x + y) = log_2(x) + log_2(1 + 2^{(log_2(y) - log2(x))})$. Similar to [7], [9], we implement this using a simple piecewise polynomial of second degree.

The implementation of the interpolation poses an additional challenge, since it relies on binary arithmetic. The required bitwidth of these operations depends on the bitwidths of the exponents. For increased accuracy of these operations, the operations internal to the interpolator are up to *twice* the regular bitwidth. To achieve acceptable clock frequencies, we have to pipeline these operations and exploit the available special function slices.

For binary additions, this does not pose much of a challenge, since the carry-save chains on modern FPGAs generally allow additions of at least 40 bits without problems. Dividing the operands in chunks of corresponding size allows easy chaining of these adders using intermediary pipeline registers. The resulting adders are similar to pipelined Ripple-Carry Adders.

In contrast, the creation of the corresponding multipliers is much more complex. For FPGA applications, the de-facto standard for generating these multipliers is given by the work of Kumm et al. [10], which relies on *integer linear programming* (ILP) to calculate resource-optimal multiplier compositions. Adapting their approach, it was possible to also avoid the high LUT utilizations of [2]. By solely using DSP-tiles, the resulting ILP solutions are perfectly tiled to map to DSPs only. Depending on the placement of the DSP-tile, Vivado then infers DSPs only if they are used efficiently or if a corresponding directive is used. Evaluating both approaches, we found the inference-option to be more resource efficient.

Using these binary additions and multiplications as primitives, we built the interpolation unit which calculates the interpolating polynomial $ax^2 + bx + c$. The coefficients a, b, c are pre-computed and stored in *read-only memory* (ROM). To reduce the size of these ROMs, we store only the fraction-bits and a single integer bit due to the observation by Vouzis et al. [11], that interpolation coefficients for this interpolated function are positive and in the range $[0, 1]$.

Using the interpolation unit we can now compose a unit for logarithmic addition by considering all possible cases under the assumptions that $|A| \geq |B|$, $x = \textit{interpolate}(E_B - E_A)$ and $F_u = \textit{underflow}(x)$. 1. Z_R is set only if Z_A and Z_B were set. 2. S_R is unset, unless Z_R or F_u. 3. If Z_R is set and S_R is unset, $E_R = 0$. In all other cases, $E_R = E_A - x$.

To actually implement this, a pipelined approach is used. After splitting the operands into exponents and flags, an additional stage ensures that $|A| \geq |B|$ holds, switching the operands if necessary. Then the difference of the exponents is calculated and pushed into the interpolation unit. Using the interpolation result and the flags we can detect the special cases and handle them accordingly.

V. Evaluation

A. Benchmarks

For full comparability, we use the same set of benchmark datasets as in [2]. That set contains benchmarks of two different types, count-based and binary. While the count-based examples are taken from the well-known NIPS[1] corpus, the binary benchmarks are pre-processed and provided by [12] and [13]. A more detailed description of each of the datasets can be found in [2].

B. Parameters

We tuned the parameters of our operators (i.e. integer & fractional bit-width, interpolation error ceiling) using an iterative process. We identified a configuration with eight integer-bits, 32 fraction bits and an interpolation error of $2^{-21.5}$ as minimal configuration to maintain sufficient accuracy across all benchmarks. The acceptable interpolation error depends on the size of each benchmark, with smaller SPNs tolerating higher interpolation error ($2^{-18.5}$ for *NIPS5*) and bigger SPNs requiring smaller interpolation errors ($2^{-21.5}$ for *Accidents*).

Benchmark	Slices [%]		BRAM [%]		DSP [%]		Freq. [MHz.]	
	FP	LNS	FP	LNS	FP	LNS	FP	LNS
Accidents	69.4	**42.0**	**3.5**	7.3	36.1	**17.2**	205	**230**
Audio	85.2	**34.8**	**3.5**	4.8	45.8	**7.6**	205	**229**
Netflix	73.7	**38.2**	**3.5**	4.6	38.5	**7.0**	199	**250**
MSNBC200	59.8	**42.7**	**3.5**	6.6	27.5	**19.1**	**250**	225
MSNBC300	43.9	**39.5**	**3.5**	5.3	17.0	**10.8**	**250**	**250**
NLTCS	57.6	**40.8**	**3.5**	6.3	25.2	**17.2**	250	**265**
Plants	82.0	**35.8**	**3.5**	5.9	42.6	**8.9**	205	**233**
NIPS5	28.4	**25.7**	**3.7**	3.7	1.6	**0.6**	250	**255**
NIPS10	31.4	**29.6**	**3.7**	4.1	4.1	**1.9**	255	**270**
NIPS20	40.0	**32.2**	**4.1**	4.8	9.3	**4.4**	**255**	250
NIPS30	43.8	**36.5**	**3.7**	5.0	14.5	**6.3**	220	**230**
NIPS40	51.3	**41.2**	**4.0**	5.7	20.3	**10.2**	226	**255**
NIPS50	57.9	**43.1**	**4.3**	6.0	23.8	**10.2**	210	**250**
NIPS60	66.9	**41.3**	**4.8**	6.0	26.0	**8.3**	217	**240**
NIPS70	73.3	**43.3**	**4.5**	5.9	30.0	**8.9**	210	**215**
NIPS80	93.0	**48.2**	**4.8**	10.0	44.1	**20.4**	190	**240**

TABLE I: FPGA implementation results using double-precision floating-point (FP) and logarithmic number scale (LNS) arithmetic operators, respectively. For brevity those numbers are given as usage relative to the resources available on the FPGA in percent.

C. FPGA Resource Consumption

As target device for the FPGA evaluation, we select the Xilinx VC709 development board, containing a Virtex7-device (xc7vx690) and 4 GiB of RAM. With the improvements in TaPaSCo and Vivado we opted to reproduce the results from [2] using current tool versions TaPaSCo 2019.10 and Vivado 2019.1, again for better comparability.

To achieve the best possible results, we employed the design-space exploration feature of TaPaSCo, which automatically maximizes the design frequencies. The resulting frequencies and resource utilizations can be found in Table I. For brevity, those numbers are given relative to the entire FPGA in percent[2].

Throughout the complete set of Benchmarks, the LNS-based implementations require *fewer* Slices and *fewer* DSPs than their FP-counterparts. Due to the use of ROMs for storing the coefficients for the interpolation, BRAM utilization is higher in LNS-implementations. The BRAM requirements are slightly more than doubled for the examples *Accidents* and *NIPS80*. However, that increase is almost irrelevant, since the original BRAM requirements were always below 5%, and thus the worst-case example (*NIPS80*) only requires 10% of BRAM.

In contrast, the resulting utilizations of Slices and DSPs are *always reduced*, depending on the size of the SPN and its adder-to-multiplier-ratio. For small examples such as *NIPS10*, the utilization is reduced by 1.8% and 2.2% for slices and

[1] archive.ics.uci.edu/ml/datasets/bag+of+words

[2] The absolute number of resources available are 433,200 (LUT), 866,400 (Register), 108,300 (Slices), 1,470 (BRAM) and 3,600 (DSP), respectively.

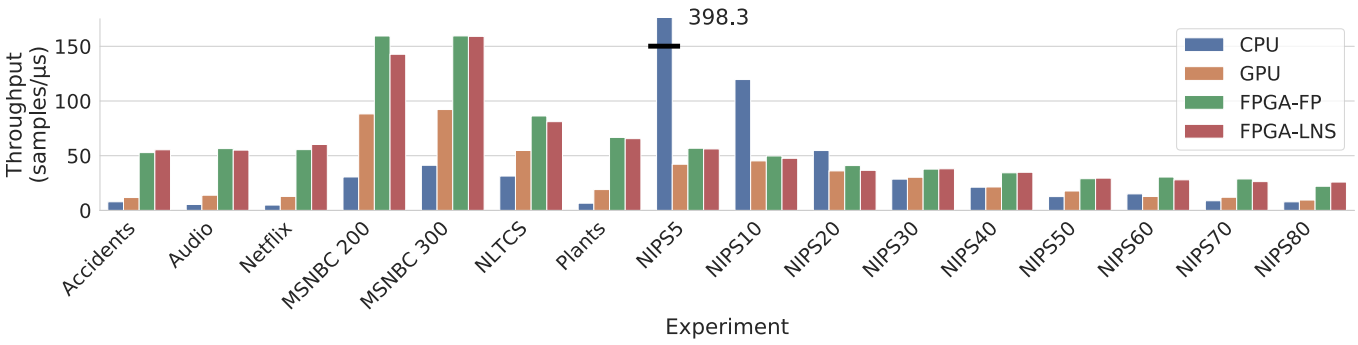

Fig. 2: Throughput of the CPU-, GPU- and both FPGA-implementations in samples/μs. Each group represents an example SPN. The single outlier is the CPU-Throughput for example NIPS5 which is cut off from more than 2x its size to fit.

DSPs, respectively. In the biggest example (*NIPS80*), the reduction grows to 44.8% *fewer* slices and 23.7% *fewer* DSPs.

D. Performance Evaluation

Similar to [2], we compare our accelerators to CPU- & GPU-implementations. For the CPU, we use the best results obtained in [2]. For the GPU, we implemented a custom CUDA-based compilation flow and evaluated it using the Nvidia CUDA compiler nvcc in version 10.0.130 and an Nvidia 1080Ti (11GB). Our new flow is up to 90x faster than the one used in [2].

Regarding throughput, the FPGA-implementations will generally outperform CPU and GPU, unless data transfer overhead exceeds a threshold. Examples for this are *NIPS5*, *NIPS10* and *NIPS20*. For these, the throughput of the CPU exceeds the corresponding throughput of all other implementations. As soon as the networks become larger, the FPGA-implementations will outperform CPU and GPU by many times. For the example *Netflix*, the throughput of both FPGA-implementations is more than 11.4x of CPU and 4.7x of the GPU. Fig. 2 shows the throughput of all implementations.

Comparing the throughputs of both FPGA-implementations, only minor differences exist. On average, the more area-efficient LNS-variants have 1.1% reduced throughput. Note that GPU and FPGA throughput data *includes* PCIe data transfer overhead. Similar to [2], this can be up to 80% of overall compute time (*NIPS20*).

VI. Conclusion & Outlook

In this work, we have developed a specialized logarithmic number format for the use in Sum-Product Network inference and implemented highly efficient, pipelined hardware arithmetic operators for addition and multiplication. Our hardware operators seamlessly integrate with the existing framework developed in [2], which allows to automatically generate fully pipelined FPGA-accelerators for SPN inference.

We compared our implementation with the existing work [2] and CPU- and GPU-implementations of SPN inference. Our evaluation shows that we can maintain sufficient precision with just 42 bits for the LNS format, whereas the FloPoCo-operators in prior work use 66 bits, leading to reductions in logic resource consumption (Slices, DSPs) of up to 50%. At the same time, we are able to maintain *similar* performance to [2], significantly outperforming the GPU- and CPU-based implementations in thirteen out of sixteen examples.

In the future, we plan to extend the synthesis flow for other types of queries, such as marginalization.

Acknowledgements.

The authors would like to thank Xilinx Inc. for supporting their work by donations of hard- and software. Additionally, calculations for this research were conducted on the Lichtenberg high performance computer of the TU Darmstadt.

Finally, the authors would like to thank Martin Kumm, for much appreciated discussions of the subject.

References

[1] J. Misra and I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress," *Neurocomputing*, 2010.
[2] L. Sommer *et al.*, "Automatic mapping of the sum-product network inference problem to fpga-based accelerators," in *IEEE 36th International Conference on Computer Design (ICCD)*, 2018.
[3] L. Sommer *et al.*, "Automatic synthesis of fpga-based accelerators for the sum-product network inference problem," in *ICML Workshop on Tractable Probabilistic Models (TPM)*, 2018.
[4] A. Molina *et al.*, "Mixed sum-product networks: A deep architecture for hybrid domains," in *Proceedings of AAAI*, 2018.
[5] D. M. Lewis, "An accurate lns arithmetic unit using interleaved memory function interpolator," in *11th Symp. on Computer Arith.c*, June 1993.
[6] ——, "114 mflops logarithmic number system arithmetic unit for dsp applications," in *Proceedings ISSCC - International Solid-State Circuits Conference*, 1995.
[7] M. Haselman *et al.*, "A comparison of floating point and logarithmic number systems for FPGAs," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, 2005.
[8] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, 2011.
[9] J. Detrey and F. de Dinechin, "A vhdl library of lns operators," in *Asilomar Conference on Signals, Systems Computers*, 2003.
[10] M. Kumm *et al.*, "Resource optimal design of large multipliers for fpgas," in *IEEE 24th Symposium on Computer Arithmetic*, 2017.
[11] P. D. Vouzis *et al.*, "Cotransformation provides area and accuracy improvement in an hdl library for lns subtraction," in *DSD 2007*, 2007.
[12] D. Lowd and J. Davis, "Learning markov network structure with decision trees," in *IEEE 10th Int. Conf. on Data Mining (ICDM)*, 2010.
[13] J. Van Haaren and J. Davis, "Markov network structure learning: A randomized feature generation approach." in *AAAI*, 2012.