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 successfully been used for the implementation of dedicated accelerators for a wide range of machine learning problems. Also the inference in so-called Sum-Product Networks, a subclass of Probabilistic Graphical Models, can 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 scale 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 and almost identical performance.

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

I. Introduction

The widespread use of machine learning techniques in academia and industry would not be possible without specialized accelerators that make it feasible to run training and inference for large datasets. Next to GPUs and custom ASIC solutions such as Google's TPU [1], field-programmable gate arrays (FPGAs) have established as an important platform for the implementation of accelerators for machine learning workloads, in particular for inference. Front-runner for this development were projects such as Brainwave by Microsoft [2], [3].

In the past, most of the work on FPGA-based accelerators for machine learning inference has focused on the acceleration of (artificial) neural networks [4], such as the very popular convolutional neural networks [5].

In contrast, Sommer et al. [6], [7] in their work developed an automatic tool-flow for mapping the inference in so-called *Sum-Product Networks* (SPN) to an FPGA-based accelerator. Sum-Product Networks are a very promising type of machine learning network, that belong to the class of Probabilistic Graphical Models (PGM).

Compared to "classical" neural networks, SPNs offer the advantage of *exact* inference, which allows them to deal much better with uncertain inputs and express uncertainty over their output. In contrast to other PGMs, such as generative adversarial nets (GAN), the inference in Sum-Product Networks is *tractable*, allowing users to efficiently compute any probabilistic query such as marginalization or conditioning.

The fact that Sum-Product Networks compute *exact* probabilities poses a number of unique challenges to the hardware implementation and most of the optimization techniques employed for the hardware mapping of neural networks, such as quantization of weights [8], are not readily applicable to SPNs.

In order to preserve the exactness property of SPNs, Sommer et al. in their work used a full-blown double-precision floating-point format based on the floating-point format provided by the FloPoCo library [9]. The use of such a high-precision, high-bitwidth floating-point formats on FPGAs comes at the price of high resource usage per arithmetic operator, because hard-float DSP-blocks are only rarely available on today's FPGAs and the arithmetic operators need to be implemented using large amounts of general logic such as LUTs.

A possible solution to this problem is the use of a *logarithmic* number format for the arithmetic on the FPGA. The use of logarithmic number scale (LNS) arithmetic is very common in the implementation of probabilistic graphical models on CPUs and GPUs. While on these platforms LNS can only be *emulated* with IEEE-754 floating point units or integer units, the flexibility of FPGAs enables us to implement hardware arithmetic operators that operate directly on LNS-encoded numbers. Using LNS representation allows us to preserve the dynamic range and precision for probability values while working with much smaller bitwidths and therefore much less ressources.

In this work, based on prior work [10], we propose a logarithmic number scale format and corresponding hardware arithmetic operators, optimized for the handling of probability values in the inference of Sum-Product Networks. We conduct a detailed analysis to determine the correct bit-widths for our number format and the best algorithm and configuration for the interpolation of the helper function used in logarithmic addition [10]. Our operators act as a drop-in replacement for the FloPoCo operators in the pipelined architecture developed by Sommer et al. Our evaluation, where we compare our LNS-based arithmetic to the previously used double-precision operators, shows that with logarithmic number scaling, we are able to save up to 50% of hardware resources while maintaining sufficient precision for SPN inference and an almost identical performance, superior to CPU- and GPU-based implementations.

We proceed as follows: In Section II we provide necessary background information on Sum-Product Networks and the inference process. Afterwards, we present previous work on the use of LNS arithmetic on FPGAs and FPGA-based accelerators for SPN inference. Section IV presents our optimized number format and provides details on the implementation of the required arithmetic operators. In Section V we evaluate our approach and compare it against prior work with regard to performance and resource consumption and Section VI concludes this paper and gives an outlook to future work.

II. Sum-Product Networks

Sum-Product Networks [11] belong to the class of Probabilistic Graphical Models (PGM), who have had a broad impact on machine learning, both in academia and industry.

Probabilistic Graphical Models can be used to solve a wide range of machine learning problems, using probabilistic queries on the model. 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., $arg\ max_c\ \mathcal{P}(class = c|evidence)$. This allows us to answer questions such as which movie or, as in one of our benchmark datasets, news a user is most likely interested in, given a history of movies or respectively news he has previously consumed and potentially other user information.

Despite all their advantages and potential applications, there is a downside to the use of PGMs: In general, the inference in unrestricted PGMs, such as Bayesian Networks or Markov Random Fields, is intractable.

Sum-Product Networks overcome this limitation and allow to compute *exact* probabilities in time *linear* wrt. to the network size. This tractable inference and the ability to compute exact probabilities enable SPNs to combine the computation of regression or classification problems with *anomaly detection*, by comparing the probabilities computed by different models.

In addition, Sum-Product Networks inherit the universal approximation property from mixture models (actually 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. Moreover, given a trained SPN, this model can be used to calculate additional properties of the underlying probability distribution, such as entropy or mutual information.

A. Model Representation

Basically, 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:

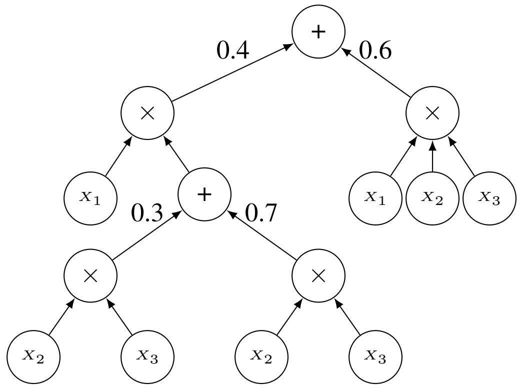

Fig. 1: Example of a valid Sum-Product Network, capturing the joint probability distribution of the variables x_1, x_2 and x_3.

- A tractable, univariate distribution is an SPN. This corresponds to the leaf nodes in the network.
- 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.
- 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 of a valid Sum-Product Network, capturing the joint probability distribution of the three variables x_1, x_2 and x_3, is given in Fig. 1.

B. Learning

Just as for other deep architectures such as deep neural networks, it is possible to learn the structure and weights of a Sum-Product Network from training data. In the following explanation, we will focus on the top-down approach proposed by Molina et al. [12], that was also used to train the benchmark networks in the work by Sommer et al. [6]. Next to this algorithm, there are other training algorithms and, with sufficient domain knowledge, the structure of an SPN can also be handcrafted, followed by weight learning [11].

Similarly to the definition of SPNs in the previous section, the algorithm works in a recursive manner. In the base case, which is reached when only a single random variable remains, a histogram representing the univariate distribution of this random variable is learned from data.

If there is still more than one variable left, the algorithm first tries to further decompose the variable set into sets of independent variables, using a non-parametric independence test [13]. If the algorithm succeeds in identifying independent sets, a product node is constructed, with the results of the recursion on each of the sets as child nodes.

If the algorithm fails to identify independent sets, the training data is partitioned using clustering. The clustering step (also called *conditioning*) induces a sum-node. The child nodes of the sum node are the results of the recursion on each of the clusters and the associated weight corresponds to the proportional size of the cluster.

An important parameter, that can be controlled during learning, is the depth of the network. Shallow networks typically suffer from high bias, whereas very deep SPNs tend to exhibit high variance.

As we focus on *inference* in this work, the learning of the SPN structure and weights happens offline on a traditional CPU.

C. Inference

A Sum-Product Network can be used to answer a range of different probabilistic queries. Independent of the actual application, the inference scheme always remains the same: Given (partial) evidence, a bottom-up evaluation of the SPN, starting at the leaf nodes, is used to compute a probability. For this purpose, the leaf nodes, in our case histograms, are indexed with the input values (note that, similarly to Sommer et al., we focus on discrete input values in this work), resulting in a probability value. This value is propagated upwards to the parent nodes. At product nodes, the probabilities of the child nodes are simply multiplied with each other. In case of a sum node, the child-node values are first multiplied with the associated weight and then summed up. The bottom-up evaluation results in a single probability value at the root node of the SPN.

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 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 [14]. This resulted in the development of an *arithmetic unit* (AU) which, at the time, outperformed all similar floating point-based AUs [15].

The same interpolation-based approach for calculating the logarithmic addition was later used by Haselman et al.. 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.

In a larger research project, Coleman et al. also developed a microprocessor completely based on logarithmic arithmetic [16].

B. SPN Inference on FPGA

In previous work, FPGA-based accelerators have been developed for *Arithmetic Circuits* (AC) [17], [18], [19], which are related to SPNs but less expressive. There is also work on the mapping of other PGMs, such as *Bayesian Networks* (BN) [20] and *Markov Random Fields* (MRF) [21], to FPGA accelerators, but the inference problem for these PGMs is very different to the inference in SPNs, as for these models inference is intractable in general.

To the best of our knowledge, the work presented by Sommer et al. [6] is the first and to date only approach to accelerate SPN inference on FPGAs. They developed an automatic toolflow that maps the inference in Sum-Product Networks to a fully pipelined FPGA-accelerator. Their 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, they used the FloPoCo framework [9] with a numeric format similar to IEE-754 double precision.

In their evaluation, Sommer et al. report high resource usage for larger examples, with some benchmarks reaching to the limit of available resources, resulting in degradation of the achievable operating frequency. The performance comparison with a x86-CPU and a Tensorflow-based GPU-implementation shows up to 6x/38x speedup in end-to-end computation time over CPU and GPU, respectively.

In this work, we will extend the framework developed by Sommer et al.. By developing LNS-based hardware arithmetic operators specialized for SPN inference as drop-in replacement of the FloPoCo-based operators, we seek to significantly reduce the hardware resource consumption, while maintaining sufficient precision and similar performance. In Section IV-D, we describe how the specialized operators developed in this work integrate into the existing framework and what changes were made to this end.

IV. Approach

Using the framework provided by Sommer et al. we develop a logarithmic adder and a logarithmic multiplier which are used as drop-in replacements for the previously used Flopoco-operators. In addition, we adapt the framework to suit our operators in regards to scheduling and integration with the existing infrastructure.

A. Number Format

For the basic number format, we rely on the work of Haselman et al.. Similar to their approach and considering a probability-value A, we use a fixed-point number to encode the exponent E_A. To increase the adaptability of the implementation, we use variable bitwidths for the integer and fraction portion of the number. 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, 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. The resulting encoding is shown in Table I.

Name	Zero	Sign	Exponent (fixed point)	
Symbol	Z_A	S_A	E_A	
Bit-Width	1 bit	1 bit	k bits	l bits

TABLE I: LNS-Encoding after optimization for probabilities

Using this encoding the following functions show how an LNS-encoding and the original value $A \in [0,1]$ relate:

$$A = \begin{cases} 0 & \text{if } Z_A = 1 \\ 1 & \text{if } S_A = 0 \wedge Z_A = 0 \\ 2^{-E_A} & \text{else.} \end{cases}$$

$$E_A = \begin{cases} 0 & \text{if } A = 0 \vee A = 1 \\ |\log_2(A)| & \text{else.} \end{cases}$$

$$Z_A = \begin{cases} 1 & \text{if } A = 0 \\ 0 & \text{else.} \end{cases} \qquad S_A = \begin{cases} 1 & \text{if } A \in \,]0,1[\\ 0 & \text{else.} \end{cases}$$

In difference to the encodings used by Haselman et al. [10] or Detrey et al. [22], the encoding removes the additional sign-bit which is used to encode the overall sign which is normally used to allow the encoding of negative numbers. In addition, we chose to change from 2's complement encoding of the exponent to encoding with an explicit sign-flag. Additionally, we do not encode special cases like 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 [10].

B. LNS Multiplication

The multiplication of values in linear scaling corresponds to an addition in logarithmic scale. This is also visible in the corresponding logarithmic properterty:

$$log_2(x \times y) = log_2(x) + log_2(y)$$

Assuming correct input values, this implies that the standard case (when both numbers are neither 0 nor 1) is simply an addition of the exponents E_A. If either one of the operands has its zero-flag set, this implies a linear-scale multiplication by 0, thus the result is zero. If either of the operands has its sign-flag unset, this corresponds to a multiplication by 1 in linear scale, thus the result is equal to the second operand.

Since a fixed-point number is used for encoding the exponent, we may also have to handle the case where the addition overflows. This corresponds to the case, where two really small values are multiplied and the result is too small to be encoded. In this case, the result is simply saturated towards zero.

The aforementioned calculations are represented by the following equations:

$$Z_R = Z_A \vee Z_B \vee overflow(E_A + E_B)$$
$$S_R = S_A \vee S_B$$

$$E_R = \begin{cases} 0 & \text{if } Z_R \\ E_A + E_B & \text{else.} \end{cases}$$

In actual hardware, we perform these calculations in 3 steps. First, the inputs are divided into exponents and flags. Then the addition of the exponents is performed and depending on the results of the addition and the flags, we lastly perform a special case handling. The resulting hardware-unit has a delay of 3 clock cycles due to this design. This could in general be reduced, but since timing closure was an issue for Sommer et al. [6] the addition of pipeline-stages may improve the obtainable maximum clock frequencies.

C. LNS Addition

In contrast to the multiplication, the 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 problem with the logarithmic addition is also obvious in the corresponding logarithmic property:

$$log_2(x + y) = log_2(x) + log_2(1 + 2^{(log_2(y) - log2(x))})$$

The major problem with this equation lies in the necessity to either calculate or approximate the function $f(z) = log_2(1 + 2^z)$, where $z = log_2(y) - log_2(x)$. While the calculation of z is a simple subtraction of the exponents, the calculation of $f(Z)$ requires evaluation of a logarithm and an exponentiation. While recent research has taken interest in actually computing the result of this function [23], most relevant LNS-implementation rely on some form of polynomial interpolation for approximating $f(x)$ [10], [22].

For the interpolation of $f(x)$ the adder relies on a simple piecewise polynomial interpolation of second degree, similar to [10], [22].

The implementation of the interpolation poses an additional challenge, since it relies on binary additions and multiplications. The bitwidth of these operations depends on the bitwidth of the exonent E_A. To increase the accuracy of these operations, the operations internal to the interpolater are up to twice the bitwidth of E_A. To achieve acceptable clock frequencies, we have to pipeline these operations and optimally use 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 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. [24], which relies on *integer linear programming* (ILP) to calculate resource-optimal multiplier compositions. Using specifically calculated costs for LUT-based multipliers and a given number n of DSP-slices to use a composition is calculated that uses n DSP-slices and minimizes the number of LUTs.

Since Sommer et al. already utilized almost all available LUTs for some examples, it was necessary to adapt the ILP to solely use DSPs. The resulting configurations are represented by Chisel3 modules [25]. During synthesis, we can either let Vivado infer DSP-slices or enforce their use. If let to infer, Vivado will generally use DSP-slices if a certain number of inputs are actually used. To achieve the most optimal results for our use case, we evaluated the differences between both variations.

As expected, using forced DSP-slices increases the number of used DSPs but the corresponding decrease in LUTs is marginal. According to Sommer et al. higher utilization of DSP-slices may result in timing issues due to problems during Place & Route [6], we thus opted for the variation using infered DSP-slices.

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 only store the fraction-bits and a single integer bit due to the observations of Vouzis et al. [26].

Using the interpolation unit we can now compose a unit for logarithmic addition by considering all possible cases under the assumption that $|A| \geq |B|$:

$$x = interpolate(E_B - E_A)$$

$$F_u = underflow(x)$$

$$Z_R = \begin{cases} 1 & \text{if } Z_A \wedge Z_B \\ 0 & \text{else.} \end{cases} \qquad S_R = \begin{cases} 0 & \text{if } Z_R \vee F_u \\ 1 & \text{else.} \end{cases}$$

$$E_R = \begin{cases} 0 & \text{if } Z_R \vee \ S_R \\ E_A - x & \text{else.} \end{cases}$$

To actually implement this, a pipelined approach is used. After dividing the operands into exponents and flags an additional stage ensures that $|A| \geq |B|$ holds, by 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 special cases and handle them accordingly.

D. Framework Integration

To integrate our operators into the existing framework, some additional changes were necessary to suit our needs. Firstly, all occurring constants that were previously encoded as floating point numbers have to be converted to the given LNS-encoding. This means, that all weights in the weighted additions, as well as the entries of the histogram distributions have to be transformed. Moreover, the datapath is generated with a final conversion which originally converted from the FloPoCo-specific format to regular IEEE754 floating point numbers. In our case, the conversion has to transform an LNS-encoded number to a IEEE754 format, to ensure interoperabilitiy with the existing software interface.

Fortunately, the conversion is relatively simple. Using a priority encoder we find the highest set bit in the exponent and determine the shift amount necessary to shift this bit to the position of the implicit 1 used in IEEE754. Using this shift amount, we can shift the exponent accordingly to effectively transform it into a valid mantissa. The exponent can then be calculated from the corresponding bias of the floating point format and the shift amount. The resulting value is an IEEE754 double that is still in logarithmic scale.

Both the arithmetic operators as well as the conversion operator are fully-pipelined designs, similar to the FloPoCo-operators.

The last important change to the framework were some minor changes to the scheduling. Since the logarithmic adder can have varying delays depending on the used bitwidths, we had to make the scheduling more variable to ensure that encodings with less bitwidth would fully benefit from potential reductions in the delay.

Using the toolflow proposed by Somer et al. [6] we can now simply generate accelerator designs as IP-xact cores. Using the current version of TaPaSCO [**?**], we compose fully-integrated accelerator designs around the IP-xact cores. Note that since the work of Sommer et al., the TaPaSCo-generated architecture-wrappers have improved significantly.

V. Evaluation

A. Benchmarks

For full comparability, we use the same set of benchmark datasets as Sommer et al. did in their work [6]. The set contains benchmarks of two different types, count-based and binary.

All count-based examples are taken from the NIPS[1] corpus and provide insights about the frequency of words in texts, with the input values being integer numbers.

The binary benchmarks were pre-processed by [27] and [28] and contain different datasets about usage statistics of services and other statistical data, based on binary input variables. A more detailed description of each of the datasets can be found in [6].

B. Parameters

In this section, we fine-tune the parameters of our LNS implementation, i.e. the integer- and fractional bit-width of the fixed-point format and the maximum error of the function interpolation. For the fine-tuning, we use a simulation with a subset of the input-data for each benchmark and validate the parameters in our evaluation on the actual FPGA-board later on in Section V-C.

Together with domain experts, a value of 1×10^{-6} was determined as the maximum deviation in *log-space* from the reference values. We chose to evaluate the deviation in log-space, because the datapath operates completely in log-space and the results are used in log-space and also the reference values were provided in log-space and conversion to regular space before evaluation would introduce an additional source of error. Note that in the for probability values relevant range $[0, 1]$, the given error-bound also corresponds to a *maximum* error of 1×10^{-6} in regular (i.e., non-log) space.

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

Since the bitwidth of the LNS-encoding also determines the bitwidths of the internal binary arithmetic, it is necessary to use as few bits as possible while still upholding the error margin. To achieve this, a step-wise evaluation approach was employed. Starting with reasonable numbers for the interpolation error (1×10^{-8}) and fraction bits (32), the simulation was run using different numbers of integer bits. The simulation determined, that 7 integer bits are only sufficient for the smallest examples, while 8 bits satisfy the error margin for all examples. Thus, the using 8 integer bits, 32 fraction bits and an interpolation error of 1×10^{-8} presents a configuration, that is able to uphold the error margin.

Next we aimed to reduce the number of fraction bits to the necessary minimum. Starting with 8 integer bits and the previous interpolation error, each evaluation step was to remove a fraction bit until the first violations of the error margin occured. Even the first step (using 31 fraction bits) produces violations. Thus we determined the required number of fraction bits to be 32.

To find a suitable configuration for the interpolation error, each example was considered individually. Using the same simulation runs, the overall error (e_{max}) was evaluated dependent on the interpolation error. Starting with the number $x = 18$ and increasing by .5 in each iteration, all interpolation errors $e = 2^{-x}$ were considered, until the error margin was met. Thus, each example has a specific configuration associated with it.

Figure 2 shows the resulting maximum overall errors produced by the examples over the evaluation subset. Each point corresponds to a single example. It also shows that some of the example-SPNs react readily to a decreased interpolation error, while the reduction in overall error stagnates for others. In addition it shows, that an interpolation error of $e = 2^{-21.5}$ is sufficient for all examples.

For the FP-variations, we use the FloPoCo-encoding which is closest to regular IEEE754 doubles to ensure easy conversions. The resulting configuration uses 11 exponent-bits and 52 mantissa-bits. In difference to regular doubles, special cases are encoded using flag-bits and thus the FloPoCo-encoding uses 66 bits instead of 64.

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. Because the underlying TaPaSCo framework has changed significantly and new versions of the FPGA implementation tools are available, we reproduce the FPGA results from [6] with the new tool-versions. We use Xilinx Vivado 2019.1 for FPGA implementation and version 2019.10 (pre-release) of TaPaSCo.

To achieve the best possible results, we opted to use the design-space exploration feature provided by TaPaSCo. This feature allows an easy evaluation of maximum reachable

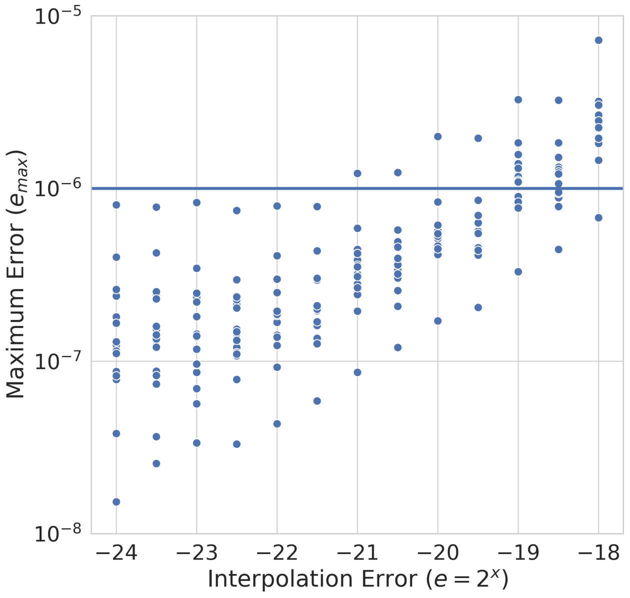

Fig. 2: Maximum Error in dependence of the used Interpolation Error.

clock frequency by automatically trying different possible frequencies until a maximum is found. The resulting operating frequencies and resource consumptions can be found in Table II. For brevity those numbers are given relative to the entire FPGA in percent[2].

Throughout the complete set of Benchmarks, the LNS-variant requires less Slices and less DSPs than its FP-counterpart. Due to the use of ROMs for storing the coefficients for the interpolation, BRAM utilization is higher in LNS-variants. This BRAM requirement is slightly more than doubled for examples *Accidents* and *NIPS80*. The increase is still almost irrelevant, since the original BRAM requirements were always below 5% and thus even the worst-case example (*NIPS80*) only requires 10% of available BRAM.

Opposing this, the resulting utilization of Slices and DSP-slices are always reduced, depending on the size of the example-SPN and its adder:multiplier-ratio. For small examples like *NIPS10*, the utilization is reduced by 1.87% and 2.25% for slices and DSP-slices respectively. In the biggest example (*NIPS80*) the reduction amounts to 44.83% less slices and 23.73% less DSP-slices.

Figure 3 gives a good overview of this change and also shows the insignificance of BRAM-changes in contrast to the changes in slices and DSP-slices.

D. Performance Evaluation

Similar to Sommer et al., we compare the performance of our FPGA-accelerator to CPU- & GPU-implementations of the benchmarks. For CPU, we use the numbers reported in

[2]The absolute number of resources available are 433200 (LUT), 866400 (Register), 108300 (Slices), 1470 (BRAM) and 3600 (DSP), respectively.

Benchmark	LUT [%] FP	LUT [%] LNS	Register [%] FP	Register [%] LNS	Slices [%] FP	Slices [%] LNS	BRAM [%] FP	BRAM [%] LNS	DSP [%] FP	DSP [%] LNS	Frequency [MHz.] FP	Frequency [MHz.] LNS
Accidents	54.28	**31.09**	32.61	**22.68**	69.48	**42.07**	**3.57**	7.38	36.17	**17.25**	205	**230**
Audio	60.75	**24.95**	36.33	**17.58**	85.23	**34.86**	**3.57**	4.80	45.83	**7.67**	205	**229**
Netflix	53.58	**24.45**	31.43	**16.62**	73.78	**38.28**	**3.57**	4.69	38.50	**7.03**	199	**250**
MSNBC200	44.59	**31.52**	26.43	**23.19**	59.84	**42.75**	**3.57**	6.63	27.5	**19.17**	**250**	225
MSNBC300	33.8	**25.46**	19.49	**17.2**	43.99	**39.59**	**3.57**	5.31	17.00	**10.86**	**250**	**250**
NLTCS	42.66	**30.41**	25	**21.75**	57.69	**40.85**	**3.57**	6.33	25.28	**17.25**	250	**265**
Plants	58.36	**25.97**	35.11	**18.14**	82.02	**35.88**	**3.57**	5.95	42.67	**8.94**	205	**233**
NIPS5	19.61	**18.07**	10.33	**9.76**	28.40	**25.75**	**3.71**	3.74	1.67	**0.64**	250	**255**
NIPS10	22.03	**19.68**	11.73	**11.07**	31.47	**29.60**	**3.71**	4.15	4.17	**1.92**	255	**270**
NIPS20	28.2	**21.51**	15.11	**13.13**	40.00	**32.21**	**4.18**	4.83	9.33	**4.47**	**255**	250
NIPS30	35.01	**24.27**	19.09	**14.69**	43.84	**36.58**	**3.78**	5.07	14.50	**6.39**	220	**230**
NIPS40	40.99	**26.73**	23.03	**17.69**	51.32	**41.21**	**4.05**	5.75	20.33	**10.22**	226	**255**
NIPS50	45.72	**28.72**	25.37	**18.07**	57.99	**43.12**	**4.39**	6.02	23.83	**10.22**	210	**250**
NIPS60	48.04	**26.99**	26.29	**17.25**	66.92	**41.32**	**4.80**	6.05	26.00	**8.31**	217	**240**
NIPS70	54.94	**29.66**	28.63	**18.11**	73.39	**43.38**	**4.52**	5.95	30.00	**8.94**	210	**215**
NIPS80	68.28	**36.81**	38.89	**26.56**	93.08	**48.25**	**4.86**	10.00	44.17	**20.44**	190	**240**

TABLE II: FPGA implementation results using floating-point (FP) and logarithmic number scale (LNS) arithmetic operators, respectively. For brevity those numbers are given relative to the entire FPGA in percent.

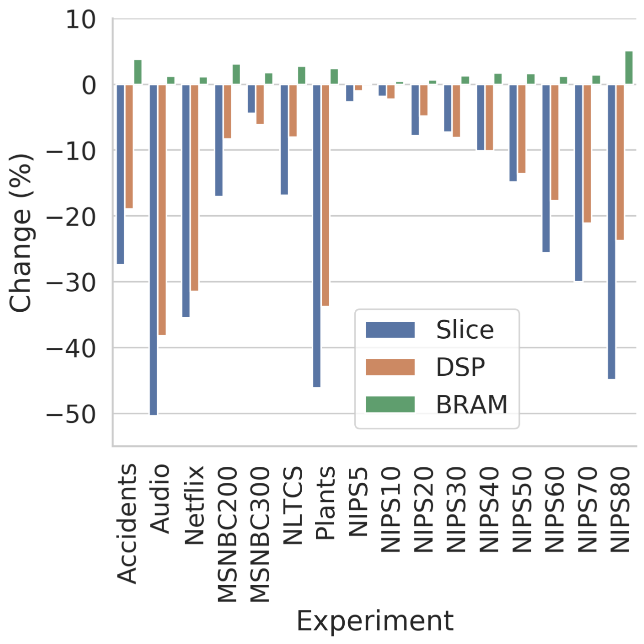

Fig. 3: Increases and Reductions to the utilization of Slices, DSP-slices and BRAM. Changes are absolute changes to the utilization of LNS over the baseline FP.

[6], merging the columns for compilation with and without -ffast-math flag using the best value.

As the naive Tensorflow-based GPU-implementation did not perform very well in [6], we implemented a custom, CUDA-based compilation-flow for SPNs, using the Nvidia CUDA compiler nvcc in version 10.0.130 and a Nvidia 1080Ti GPU with 11GB of memory. This results in speedups up to 90x compared to the original Tensorflow-based implementation.

In regards to the FP-variant, the FPGA-implementation has also seen a speedup of approximately 2x in comparison to [6] due to the changes to TaPaSCo, which mainly affect the data-transfers from main memory to device memory and back, which are performed via PCIe. In addition, our improvements to the pipelining of the FP-operators has also improved most of the clock frequencies of the FP-variants.

Considering the throughput, the FPGA-implementations will generally outperform CPU and GPU, unless the overhead incurred from data transfers exceeds a certain threshold. Examples for this can be seen in *NIPS5*, *NIPS10* and *NIPS20*. For these examples, the throughput of the CPU exceeds the corresponding throughput of all other implementations. As soon as the networks increase in overall size, the FPGA-implementations will outperform CPU and GPU manifold. For the example *Netflix*, the throughput of both FPGA-implementations is more than 11.4x of CPU and 4.7x of the GPU. The following Fig. 4 shows the throughput of all variations in comparison.

Comparing LNS- and FP-variants of the same example, it is also visible that there are only minor differences in throughput between them. Additionally, there are some LNS-variants that have above 17% increased throughput, while others have a decrease by about 10%. On average, the LNS-variants have a 1.1% reduced throughput compared to FP-variants, but this difference is only marginal.

An additional point to note is the overhead that is incurred by the data transfers via PCIe, which only concerns FPGA- and GPU-implementations. For almost all examples, excluding only *MSNBC200* and *MSNBC300* this overhead amounts to at least 60% of execution time. Especially smaller count-based examples are hit by this overhead, some reaching almost 80%. This also in part reason for the respectively lower throughput of these examples. Figure 5 illustrates this further. The data-transfers alone take more time than the complete execution on the CPU. Despite this major disadvantage, the FPGA-implementations are still outperforming CPU and GPU on 13 of 16 examples.

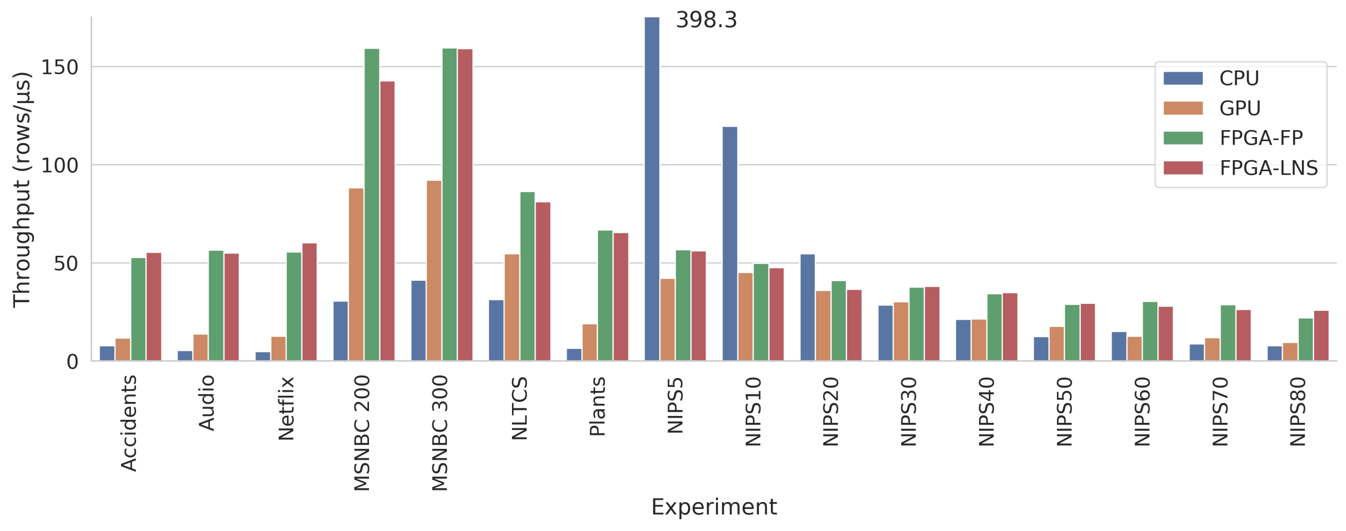

Fig. 4: Throughput of the CPU-, GPU- and both FPGA-implementations in $rows/\mu s$. Each group represents an example-SPN. Single outlier is the CPU-Throughput for example NIPS5 which amounts to $398.8 rows/\mu s$.

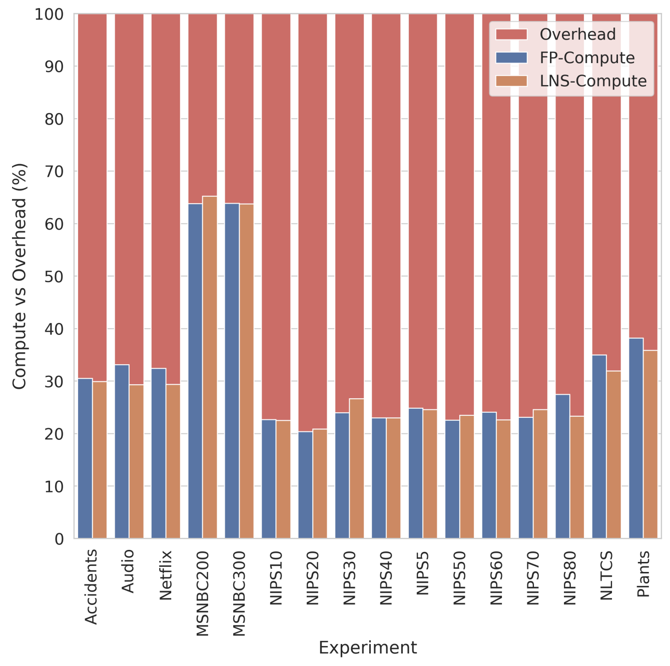

Fig. 5: Percentage of execution time divided into computation and overhead for all example-SPNs.

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 by Sommer et al. [6], which allows to automatically generate fully pipelined FPGA-accelerators for SPN inference.

After fine-tuning the parameters of our implementation, i.e., the bit-widths of the internal fixed-point representation and the parameters for function interpolation as part of the addition, we compared our implementation against the existing work by Sommer et al. 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 (LUTs, registers, DSPs) of up to 50%.

At the same time, we are able to maintain an almost identical performance to Sommer et al., significantly outperforming the GPU-based and CPU-based implementations in thirteen out of sixteen examples.

In the future, we plan to extend the synthesis flow for resource sharing of operators and other types of queries, such as marginalization and *maximum a posteriori estimation*. Besides that, we want to investigate how FPGAs can be used to accelerate weight learning for randomly generated Sum-Product Networks [29].

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 Prof. Dr.-Ing. Martin Kumm, for well appreciated discussions of subject, as well as extensive insight into the matter.

References

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017*. ACM, 2017, pp. 1–12.

[2] E. Chung, J. Fowers *et al.*, "Accelerating Persistent Neural Networks at Datacenter Scale," in *Hot Chips 29: A Symposium on High-Performance Chips*, 2017.

[3] K. Freund, "Microsoft: FPGA Wins Versus Google TPUs For AI," https://www.forbes.com/sites/moorinsights/2017/08/28/microsoft-fpga-wins-versus-google-tpus-for-ai, 2017, accessed April 5, 2018.

[4] J. Misra and I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress," *Neurocomputing*, vol. 74, no. 1, pp. 239–255, Dec. 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S092523121000216X

[5] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 56:1–56:39, Jun. 2018. [Online]. Available: http://doi.acm.org/10.1145/3186332

[6] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch, "Automatic mapping of the sum-product network inference problem to fpga-based accelerators," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct 2018, pp. 350–357.

[7] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch, "Automatic synthesis of fpga-based accelerators for the sum-product network inference problem," in *ICML 2018 Workshop on Tractable Probabilistic Models (TPM)*, 2018.

[8] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "Fp-bnn: Binarized neural network on fpga," *Neurocomputing*, vol. 275, pp. 1072 – 1086, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231217315655

[9] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, Jul. 2011.

[10] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. Underwood, and K. S. Hemmert, "A comparison of floating point and logarithmic number systems for FPGAs," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, Apr. 2005, pp. 181–190.

[11] H. Poon and P. Domingos, "Sum-Product Networks: a New Deep Architecture," *Proc. of UAI*, 2011.

[12] A. Molina, A. Vergari, N. Di Mauro, S. Natarajan, F. Esposito, and K. Kersting, "Mixed sum-product networks: A deep architecture for hybrid domains," 2018.

[13] D. Lopez-Paz, P. Hennig, and B. Schölkopf, "The randomized dependence coefficient," in *Advances in neural information processing systems*, 2013, pp. 1–9.

[14] D. M. Lewis, "An accurate lns arithmetic unit using interleaved memory function interpolator," in *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, June 1993, pp. 2–9.

[15] ——, "114 mflops logarithmic number system arithmetic unit for dsp applications," in *Proceedings ISSCC '95 - International Solid-State Circuits Conference*, Feb 1995, pp. 86–87.

[16] J. N. Coleman, C. I. Softley, J. Kadlec, R. Matousek, M. Tichy, Z. Pohl, A. Hermanek, and N. F. Benschop, "The european logarithmic microprocesor," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 532–546, April 2008.

[17] P. Dormiani, D. Omoto, P. Adharapurapu, and M. D. Ercegovac, "A design of online scheme for evaluation of multinomials," in *Advanced Signal Processing Algorithms, Architectures, and Implementations XV*, vol. 5910. International Society for Optics and Photonics, 2005.

[18] J. Geist, K. Y. Rozier, and J. Schumann, "Runtime observer pairs and bayesian network reasoners on-board fpgas: flight-certifiable system health management for embedded systems," in *International Conference on Runtime Verification*. Springer, 2014, pp. 215–230.

[19] S. Zermani, C. Dezan, H. Chenini, R. Euler, and J. Diguet, "FPGA implementation of bayesian network inference for an embedded diagnosis," in *2015 IEEE Conference on Prognostics and Health Management, ICPHM 2015*. IEEE, 2015, pp. 1–10.

[20] J. Alves, J. Ferreira, J. Lobo, and J. Dias, "Brief survey on computational solutions for bayesian inference," in *Workshop on Unconventional computing for Bayesian inference*, 2015.

[21] J. Choi and R. A. Rutenbar, "Video-rate stereo matching using markov random field TRW-S inference on a hybrid CPU+FPGA computing platform," *IEEE Trans. Circuits Syst. Video Techn.*, 2016.

[22] J. Detrey and F. de Dinechin, "A vhdl library of lns operators," in *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003*, vol. 2, Nov 2003, pp. 2227–2231 Vol.2.

[23] R. Chen and C. Chen, "Pipelined computation of very large word-length lns addition/subtraction computation with exponential convergence rate," in *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, Dec 2009, pp. 69–73.

[24] M. Kumm, J. Kappauf, M. Istoan, and P. Zipf, "Resource optimal design of large multipliers for fpgas," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, July 2017, pp. 131–138.

[25] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviienis, J. Wawrzynek, and K. Asanovi, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, June 2012, pp. 1212–1221.

[26] P. D. Vouzis, S. Collange, and M. G. Arnold, "Cotransformation provides area and accuracy improvement in an hdl library for lns subtraction," in *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, Aug 2007, pp. 85–93.

[27] D. Lowd and J. Davis, "Learning markov network structure with decision trees," in *Data Mining (ICDM), 2010 IEEE 10th International Conference on*. IEEE, 2010, pp. 334–343.

[28] J. Van Haaren and J. Davis, "Markov network structure learning: A randomized feature generation approach." in *AAAI*, 2012, pp. 1148–1154.

[29] R. Peharz, A. Vergari, K. Stelzner, A. Molina, X. Shao, M. Trapp, K. Kersting, and Z. Ghahramani, "Random sum-product networks: A simple but effective approach to probabilistic deep learning," in *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI); a previous version also as arXiv preprint arXiv:1806.01910*, 2019.