Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs

Lukas Sommer*, Lukas Weber*, Martin Kumm†, Andreas Koch*
*Embedded Systems and Applications Group, TU Darmstadt, Germany
†Faculty of Applied Computer Science, Fulda University of Applied Sciences, Germany
*{sommer, weber, koch}@esa.tu-darmstadt.de, †martin.kumm@cs.hs-fulda.de

***Abstract*—Probabilistic Graphical Models (PGM) have recently received increasing attention for various machine learning tasks and approaches for their acceleration on FPGAs have been presented.**

In this work, we investigate three different arithmetic formats, namely customized floating-point, Posit and logarithmic number systems with regard to their suitability for the inference in PGMs, specifically so-called Sum-Product Networks (SPN). Based on results from an automatic design-space exploration developed in this work, we implement hardware arithmetic operators for each format, optimized for SPN inference.

Our evaluation shows that the choice of the most area-efficient solution depends on the relation between the numbers of adders to multipliers in the network. Up to 57% and 68% of Slice and DSP reductions, respectively, could be obtained compared to previous work. With regard to performance, all formats achieve similar results and outperform CPU and GPU-based implementations of SPN inference by factors up to 12x and 4.6x, respectively.

I. Introduction

Next to GPUs and custom ASICs, such as Google's TPU, FPGAs have established themselves as a succesful implementation platform for the acceleration of machine learning (ML) tasks, in particular for inference. Besides numerous works on the acceleration of the inference in neural networks, for example convolutional neural networks (CNN) for computer vision applications, new approaches to accelerate inference in *probabilistic models* on FPGAs have recently been presented.

One such approach for the inference in so-called *Sum-Product Networks* (SPN) was developed in [1], [2], [3]. Compared to neural networks, Sum-Product Networks, which belong to the class of *tractable* Probabilistic Graphical Models (PGM), can better deal with missing input features and, as SPNs compute *exact* probability values, are also able to express uncertainty over their outputs.

However, this ability also poses new challenges to the implementation of such networks on FPGAs. In [1], [2], the authors used a double-precision floating-point format to preserve accuracy. Such an arithmetic format is expensive to implement on FPGAs. Therefore, in this work, we seek to optimize the hardware arithmetic operators to *reduce* resource usage, while *preserving* sufficient accuracy. To this end, we will investigate three different arithmetic formats, namely "traditional" but customized floating point, logarithmic number system (LNS) and Posit, with regard to their suitability for FPGA-based accelerators for SPN inference.

We exploit an automatic and efficient design-space exploration (DSE) flow, based on software-only emulation of the arithmetic formats for SPN inference, to determine the minimal bit-widths required to preserve accuracy with each of the formats prior to hardware generation.

Based on the findings from our DSE, we then implement hardware arithmetic operators for each of the three investigated arithmetic formats, optimized for the inference in Sum-Product Networks on FPGAs. The optimized arithmetic operators are used to generate fully pipelined datapaths, which are integrated into a SoC-design providing the host-CPU software interface. In our extensive evaluation, we investigate which arithmetic format is most suited for SPN inference on FPGAs and compare the performance of the generated datapaths with CPU and GPU-based implementations of SPN inference.

II. SPN Background

Sum-Product Networks [4] belong to the class of *probabilistic models*, which can be used for a range of different machine learning tasks. As they are also able to take the statistical nature of the data into account, and deal well with uncertainty and missing features, this class of models has received increasing attention recently.

After a probabilistic model has been trained from data, different machine learning problems, such as classification and regression, can be solved by using probabilistic queries on the trained model. An example for such a query would be to determine which news-article a user is most likely interested in, based on information on whether or not he or she has looked at other articles before.

In comparison with other probabilistic models and other ML-techniques, such as deep neural networks, SPNs exhibit a number of interesting characteristics, that makes them attractive for use in a range of different applications. For example, SPNs have already been used succesfully for sequence labeling [5], i.e., classifying the characters in a handwritten sequence, or in path planning algorithms for mobile robots [6].

One very important property of SPNs for their practical usage is the *efficiency* of the inference: While in general, inference for unrestricted PGMs is *intractable*, the inference in SPNs is guaranteed to be *linear* w.r.t. the number of nodes [4], [7]. This *tractable* inference is key to efficiently answering probabilistic queries in practical applications.

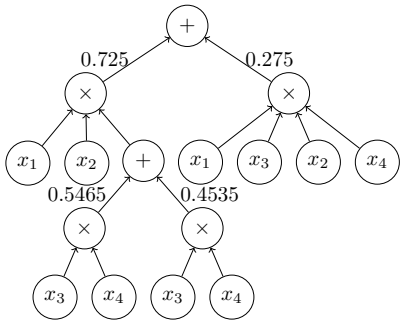


Fig. 1. Example of a valid SPN representing the joint probability $\mathcal{P}(x_1, x_2, x_3, x_4)$.

Another interesting property of SPNs is their expressiveness: From mixture models, which can easily be represented by a shallow Sum-Product Network with a single sum-node, SPNs inherit the *universal approximation property* [8]. This means that Sum-Product Networks can represent *any prediction* function, similar to deep neural networks.

One of the most interesting properties about Sum-Product Networks, that also makes SPNs stand out from other ML-techniques such as deep neural networks, is the *precision* of the inference process. Whereas neural networks generally compute *approximate* values, Sum-Product Networks are instances of Arithmetic Circuits [9] and therefore facilitate the computation of *exact* probability values. Beyond more precise answers to queries, this also offers the advantage that the inference process can be combined with *anomaly detection* by comparing the respective probabilities from different SPNs, and also better account for the statistical nature of the data.

In this work, we focus on the *inference* process in a pre-trained SPN. In this case, the learning has taken place offline on a traditional CPU-based machine.

A. Model Representation

A Sum-Product Network captures the joint probability $\mathcal{P}(X, Y, Z)$ over a set of variables $\{X, Y, Z\}$ in the form of a rooted, directed acyclic graph (DAG). An example for a valid SPN over the variables $\{x_1, x_2, x_3, x_4\}$ can be found in Fig. 1. The graph representation of SPNs is composed from three different kinds of nodes, with some additional restrictions to guarantee the validity of the SPN:

- Leaf nodes represent univariate distributions over a single variable. In this work, based on the approach proposed by Molina et al. [10], we represent these univariate distributions by histograms for an efficient mapping to the FPGA.
- Factorizations over independent distributions are represented by product nodes in the graph. The child nodes of a product node are defined over different scopes, i.e., each sub-tree uses a distinct set of variables.
- Mixtures over distributions defined over the same set of variables are represented by sum-nodes, where each child node is additionally associated with a weight. The child nodes of a sum node are defined over the same scope, i.e., the same set of variables appears in each subtree.

B. Inference

The inference process depends on the kind of probabilistic query that should be answered. Common to all kinds of inference is the bottom-up evaluation of the SPN graph, eventually yielding a probability value at the root of the graph.

The most basic kind of inference in an SPN is the joint computation, yielding the joint probability for given input values, i.e., full evidence. In the first step, the leaf nodes are queried with the value of the associated input variable, yielding a probability value. In this work, the univariate distributions at leaf nodes associated with an input variable are modeled using histograms, which are simply indexed with the input value. The resulting probability values are then propagated upwards through the tree. At product nodes, the child node values are multiplied with each other. When a sum node is reached, the child node values are first multiplied with the corresponding weight and then summed up.

Marginalization [8] of variables is another possible kind of query that can be answered by inference. To this end, the leaf nodes associated with the marginalized input variables are replaced by the probability 1. The remaining leaf nodes are just queried with the associated input values from the partial evidence. The rest of the inference process is identical to the joint computation. Through the combination of joint computation and marginalization, it is also possible to compute conditional probabilities using the following equation, where the numerator of the fraction corresponds to the joint computation and the denominator can be computed by marginalization of Y: $\mathcal{P}(Y|X) = \frac{\mathcal{P}(Y,X)}{\mathcal{P}(X)}$.

In this work, we focus on joint computation, but the datapath architecture can easily be extended to support other kinds of inference, such as marginalization.

In prior work, accelerators for the inference in other Probabilistic Graphical Models such as Bayesian Networks (BN) [11] or Markov Random Fields (MRF) [12] were developed. However, as discussed in the previous section, the inference in these kinds of PGMs differs significantly from Sum-Product Networks and the techniques used in these works cannot be applied to SPNs without further ado.

To the best of our knowledge, the only approach to accelerate SPN inference on FPGAs was presented in [1], [2]. In this work, we seek to extend the automatic toolflow from this work with three different arithmetic formats.

III. Arithmetic Number Formats

A. Fixed Point

Fixed-point arithmetic can be implemented very efficiently in FPGAs. Yet, we do not consider fixed-point further in this work, because with SPNs, very small numbers can still represent significant results. In [10], the authors reported on relevant *log-likelihoods* as small as -144 and a first analysis of the dynamic range of the results of our benchmark networks showed that the smallest numbers are as small as $1.85 \cdot 10^{-88}$.

As each number can also be as large as one, at least 292 bits would be necessary to encode this number. A binary multiplier of corresponding size would require over 200 DSP-slices and is thus not a viable option.

The fact that such small numbers can still be significant for the outcome of the SPN and the result of the ML-task is also the reason why we use comparisons in *log-space* to compute the deviation from reference results in the rest of this work.

B. Floating Point

As motivated above, for applications requiring a large dynamic range, the word length w of fixed-point numbers may get excessively large. *Floating point* (FP) numbers provide a much wider dynamic range, at the cost of a reduced precision, for the same number of bits. An FP number X according to the IEEE 754 standard is represented as $X = (-1)^s \times 1.f \times 2^{e-e_0}$, where s is the sign bit (0 for positive, 1 for negative), f is the fraction and e is the exponent field. The exponent field e is a w_e bit unsigned integer that represents the signed exponent $e - e_0$, where e_0 is called the bias defined as $e_0 = 2^{w_\mathrm{e}-1} - 1$. As the FP format is normalized such that the leading bit of the significant is equal to '1', only the fractional bits of the mantissa are stored in w_m bits.

C. Posit

The Posit arithmetic format is a comparably young format, introduced in 2017 as an implementation of type-3 unum (universal number) arithmetic [13]. The Posit format is characterized by two parameters, the total number of bits in the format w and the number of bits used to represent the exponent w_es.

As shown in Fig. 2, the Posit number representation is composed of four parts.

Negative numbers are encoded as 2's complement where the most significant bit (s) indicates the sign of the number.

The next component, the so-called *regime*, distinguishes Posit from traditional floating point formats. The regime is represented using a variable run-length (or thermometer) encoding, i.e., a sequence of bits with identical value terminated by a bit of the opposite value, where the length of the sequence represents the encoded value. As an example, the sequence 0001 encodes the value -3, whereas the sequence 110 encodes the value 2.

The third component, the *exponent*, is encoded as a binary number using a fixed size of w_es bits. In contrast to IEEE754 floating point, the exponent only encodes *positive* numbers and no bias is used.

The last component is the *mantissa*, which is stored just as in IEEE754 floating point, with an implicit leading 1 omitted. The mantissa occupies the remaining w_m bits, that are left after the run-length encoding of the regime and the fixed-size exponent.

Because the length of the regime is only limited by $w - 1$, the mantissa and also the exponent may not be present at all.

Given the sign bit s, a regime value r, the exponent e and the mantissa f, the number represented in Posit can be computed as follows: $(-1)^S \times \mathrm{useed}^r \times 2^e \times 1.f$, where $\mathrm{useed} = 2^{2^{w_\mathrm{es}}}$. As an example, with $w = 7$ and $w_\mathrm{es} = 2$, the bit-sequence $0\_01\_11\_10$ encodes the decimal value $(-1)^0 \times (2^{2^2})^{-1} \times 2^3 \times 1.10_2 = 0.75$.

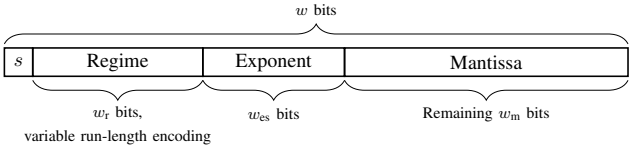


Fig. 2. Posit binary format.

Multiple previous works have developed Posit arithmetic hardware operators for FPGAs. While [14] and [15] found that Posit incurred a significant area overhead over traditional floating point, the operators developed in [16] required resources comparable to FP implementations and for the particular application investigated in this work, floating point could be replaced with a smaller bit-width and more area-efficient Posit format. As only the operators from [14] are available open-source, we build on this library for the implementation of the Posit hardware operators in this work. We extend the operators to meet our requirements as described in Section V-C.

D. Logarithmic Number System

Originally, Logarithmic Number Systems (LNS) were developed as an alternative to floating point numbers. The general idea behind LNS is that instead of storing a real number as a combination of an integer exponent and a fixed-point number, only the *logarithm* $\log_2(A) = E_A$ is stored as a fixed-point *exponent*. In general purpose applications, LNS-numbers are then encoded as follows: $A = -1^{S_A} \times 2^{E_A}$ and a flag is used for zero values [17], [18].

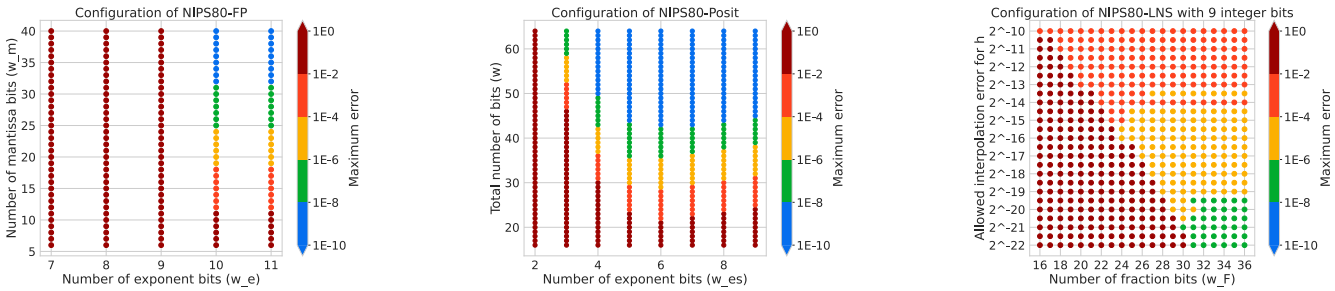
Due to the logarithmic nature of the encoding, all calculations are performed in a *logarithmic scale*. Thus, logarithmic properties apply and $\log_2(a \times b) = \log_2(a) + \log_2(b)$, greatly simplifying multiplicative calculations.

In contrast to this, additive arithmetic operations become more complex. Assuming that $x > y$ holds, addition and subtraction are given by $\log_2(x \pm y) = \log_2(x) + \log_2(1 \pm 2^{(\log_2(y) - \log_2(x))})$. The second part of the equation is usually implemented through a helper function h, and the allowed interpolation error determines how this function is implemented in hardware. In this work, we adapt the approach from [3], which was optimized for SPNs and uses a quadratic spline interpolation for h.

IV. Design-Space Exploration using Software Emulation

A fair comparison of the three arithmetic formats considered requires that the individual parameters of the different formats (e.g. overall bitwidth) are *optimized* as much as possible.

To this end, prior work such as [19] has often used theoretical worst-case analyses based on error-models for fixed- and floating-point arithmetic operators. However, these analyses tend to *overestimate* the error that occurs during



(a) Floating point. (b) Posit. (c) LNS with $w_{\mathrm{I}} = 9$.

Fig. 3. Development of the maximum error depending on the configuration of the arithmetic formats. Best viewed in color.

actual computation. Besides that, error analysis models for Posit and LNS are not readily available and many of the application-specific optimizations to the hardware operator implementations described in Section V cannot easily be modelled in such error models.

Therefore, we take a different approach: Using a C++-based software emulation of the individual SPN and the different arithmetic formats, the design-space is traversed to determine the best *viable* configuration for each arithmetic format on a per-benchmark basis. We use the available benchmark data to run the software emulation with each configuration and only accept a configuration, if it maintains a given error-threshold. As the representativeness of the training data is key to the ML training itself, the design-space exploration will yield configurations that work for all relevant input combinations. This approach is also common when quantizing neural networks [20].

A. Implementation

Using a graph-based intermediate representation and an abstract syntax tree (AST) infrastructure, we generate C++ code emulating the behavior of each of the different arithmetic formats in hardware as closely as possible.

The design-space of possible configurations is then automatically traversed. For each configuration, we generate and compile the C++ code and run the SPN inference on a CPU. If the maximum error does not exceed the configurable error threshold, we accept the configuration.

The performance of the DSE can be improved significantly by investigating multiple configurations in parallel and additionally parallelizing the CPU-based execution using OpenMP. This way, we could reduce the time required to determine the correct floating-point configuration for the largest benchmark instance *NIPS80* from 656 seconds to only 138 seconds.

B. Accuracy Results

For the following accuracy evaluation, an error threshold of 1×10^{-6} was used. Note that we compute the error in log-space to determine the error independently from the magnitude of the values. For each of the arithmetic formats, different parameters can be chosen: For floating point, the number of bits in the mantissa (w_{m}) and the exponent (w_{e}) can be configured. The

TABLE I
CONFIGURATION OF THE THREE ARITHMETIC FORMATS FOR EACH OF THE BENCHMARKS MAINTAINING AN ERROR OF 1×10^{-6}.

Benchmark	FP		Posit		LNS		
	w_{e}	w_{m}	w	w_{es}	w_{I}	w_{F}	h-Error
Accidents	8	26	36	4	7	32	21.5
Audio	9	28	36	4	8	30	20.5
MSNBC 200	8	26	32	4	7	31	19.5
MSNBC 300	8	24	32	4	7	31	20.5
Netflix	9	26	36	4	8	30	20.5
NLTCS	7	26	32	3	6	30	19.5
Plants	8	28	36	5	7	31	20.5
NIPS5	7	24	30	3	5	26	18.5
NIPS10	7	24	32	3	6	27	20.0
NIPS20	8	24	34	3	7	29	19.5
NIPS30	8	26	34	4	7	29	19.5
NIPS40	9	26	34	4	7	30	19.5
NIPS50	9	26	34	5	8	30	19.5
NIPS60	9	26	36	5	8	30	19.5
NIPS70	9	26	36	5	8	30	20.5
NIPS80	10	26	36	5	9	31	19.5

Posit format is parameterized by the total number of bits (w) and the number of bits used for the exponent (w_{es}). The LNS format can be configured by three parameters: The number of integer (w_{I}) and fraction (w_{F}) bits in the fixed-point format of the exponent and the maximum error allowed for the interpolation (Error) of the helper function h used in LNS-addition.

The configurations identified through our design-space exploration for each benchmark can be found in Table I. The plots in Fig. 3 show how the maximum error develops across different configurations for each arithmetic format in the NIPS80 benchmark, the largest instance in our benchmark set.

For floating point, a minimum number of exponent bits (w_{e}) is required to be able to represent small but significant values in the first place. Beyond that, a certain number of mantissa bits (w_{m}) is required to represent numbers sufficiently accurate so the error will not accumulate beyond the error threshold.

With Posit, a minimum number of bits for the exponent (w_{es}) and the total size of the format (w) is required. However, if the size of the exponent is increased *beyond* that minimum number, the total number of bits *also* has to be increased, otherwise the number of bits remaining for the mantissa (max.

$w - w_{\text{es}} - 3$) is no longer sufficient. So for Posit, the sweet spot is reached when w_{es} is just large enough to encode all relevant exponents.

The direct comparison of floating-point and Posit shows, that the total bitwidth of the formats is typically relatively close. This result aligns with the findings in [21]. The probabilistic values computed inside the SPN tree are very small, and lie outside of the *golden range* identified in [21]. In that range, relatively small Posit formats can be used to replace significantly larger floating-point formats.

The LNS format will only produce correct results, if the number of integer bits (w_{I}) is sufficiently large to represent all relevant exponents, therefore the plot in Fig. 3 shows the development of the error depending on the fraction bits (w_{F}) of the exponent and the interpolation error of the addition helper function h for $w_{\text{I}} = 9$. The number of fraction bits must be sufficiently large to represent numbers with a certain accuracy and, at the same time, the allowed interpolation error of h must be sufficiently small so the LNS addition does not introduce excessive error.

V. Implementation of Hardware Arithmetic Operators

Based on the findings from the automatic DSE presented in the previous section, specialized hardware arithmetic operators for SPN inference were developed. This section details the implementation for each arithmetic format. An overview of the resource requirements of the individual operators can be found in Table II. The operators were designed as drop-in replacement for the operators in [2] to enable reuse of the automatic toolflow in this work.

A. Floating Point

The floating point implementations used in this work are based on the FloPoCo tool [22] which was extended for the specifics of SPN. All of our extensions have been made publicly available in the FloPoCo git repository [23]. Note that subnormal numbers are not supported in FloPoCo as they are very costly to implement and the loss in dynamic range can be easily compensated by adding one additional mantissa bit.

1) Floating Point Adder: Addition in FP is a much more time and resource consuming operation compared to FP multiplication. The basic algorithm to perform a floating point addition requires the following computation steps: 1) computing the exponent difference, 2) alignment of the operands, 3) mantissa addition, 4) alignment and rounding of the result, and, 5) handling of special values. All these computations lie on the critical path where the large bit shifters required for the two alignment steps are among the most demanding. Also, faithful rounding does not help much for addition. However, a well-known technique to reduce the delay is the dual-path (DP) architecture [24]. The observation here is that two cases exist that can be treated separately: 1) when subtracting two numbers with similar magnitude, only a *small* operand shift is necessary while a *full* result shifter is required; 2) in all other cases, the operand shift has to be *large* while a *small* shift for the result is sufficient. In the DP adder, the computations for both cases are computed *in parallel*, and the correct result is selected at the end. Fig. 4 shows the data path for processing the mantissa, omitting the control signals for brevity. The first case is called the *close-path* (shown on the left in blue) and the second case the *far-path* (shown on the right in green). While for the *operand* alignment only a 1-bit right shift (R1-Shift) is necessary in the close-path, a full right shifter (R-Shift) is necessary in the far-path. In contrast, the *result* of the close-path requires a leading zero counter (LZC) and full right shifter, while the the far-path only requires a 2-bit shift (R2-Shift) for normalization and rounding.

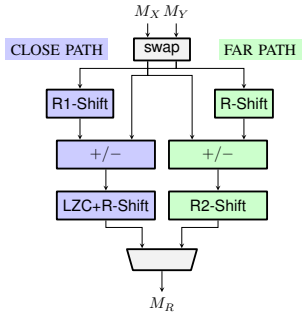


Fig. 4. FP Adder dual path mantissa processing

TABLE II
Comparison of the per-operator resource requirements and pipeline depth, using configurations for *NIPS80* (cf. Table I).

Format	Op.	Slice	DSP	BRAM	pipeline depth
FP	Adder	106	0	0	5
	Mult.	86	2	0	5
Posit	Adder	374	0	0	7
	Mult.	340	4	0	12
LNS	Adder	757	20	1.5	64
	Mult.	36	0	0	3

To implement SPNs, we can make use of the dual-path idea by exploiting the fact that all values in SPNs are restricted to be *positive* and only *additions* occur. Hence, the close-path in a dual-path architecture will *never* be active in an SPN. To this end, we extended the dual-path implementation of the `FPAdd` operator in FloPoCo with an option to optimize the adder only for positive numbers, which omits all components from the close-path as well as the output multiplexer.

To gauge the effects of this optimization, we performed a synthesis experiment on the single operators (using the same setup later described in Section VI-B). The results are given in Table III, showing the logic resources, the pipeline depth (PD) as well as the max. clock frequency. As there are two options for the FP adder in FloPoCo, a single path and a dual path, we synthesized both. As expected, the dual path has one pipeline stage less compared to the single path, but at the expense of a larger chip area. Remarkably, our optimization for only positive operands (listed as "only pos. args.") leads to a slice reduction of 23.2% and 42.4% compared to the single

TABLE III
DIRECT COMPARISON OF THE FP ADDER BEFORE AND AFTER THEIR OPTIMIZATION USING THE CONFIGURATIONS FOR *NIPS80* (CF. TABLE I).

Operator	Slice	DSP	PD	Freq. [MHz]
FP Adder Single Path	138	0	8	384
FP Adder Dual Path	184	0	7	274
FP Adder (only pos. args.)	106	0	5	389

and dual path options, respectively, while reducing the pipeline depth by 3 and 2 cycles at the same time.

2) Floating Point Multiplier: The computation of an FP multiplication is much simpler compared to addition: 1) the mantissas are multiplied, 2) the exponents are added, and, finally 3) the result is normalized and rounded. This normalization requires only a small shift by one bit position and can usually be merged with the output MUX that is necessary for the special values. Besides this, the rounding mode has the most influence on the used resources. In contrast to correct rounding, faithful rounding requires only about half the number of bits plus some guard bits of the mantissa multiplication result [25]. Hence, a truncated integer multiplier can be used for the mantissa which requires less chip area. Therefore, the FP multipliers in this work use faithful rounding based on the work in [25].

B. Logarithmic Number System

For the implementation of the LNS hardware operators, we employ the implementation of Weber et al., presented in [3]. They developed pipelined and parameterized LNS adders and multipliers targeted towards SPNs.

As discussed earlier, multiplication in the logarithmic space can be implemented as a simple binary addition, and consequently consumes less than half (36 vs. 86, cf. Table II) of the slices compared to the floating-point multiplier, and no DSPs.

On the other hand, the much more complex calculation for addition in logarithmic space, for which a quadratic spline interpolation was used in [3], results in a larger chip area for the logarithmic adder, which consumes 757 slices and 20 DSPs, compared to 106 slices and no DSPs for FP.

C. Posit

For the implementation of the Posit hardware operators, we build upon PACoGen [14], an open-source project providing Posit basic arithmetic operators. These implementations are generally only realized as combinatorial circuits.

To ensure a fair comparison between the arithmetic formats regarding operating frequency, we introduced pipelining into the existing, parameterized implementation. The resulting multiplication operator requires almost five times the logic resources (340 vs. 86 slices), and, even though we adopted the optimal DSP allocation scheme from [26], twice the number of DSPs (4 vs. 2), as the floating-point multiplier. In case of the addition, the additional decoding logic for the regime and the higher internal precision cause the Posit adder to use significantly more resources than its floating-point counterpart (374 vs. 106 slices, cf. Table II).

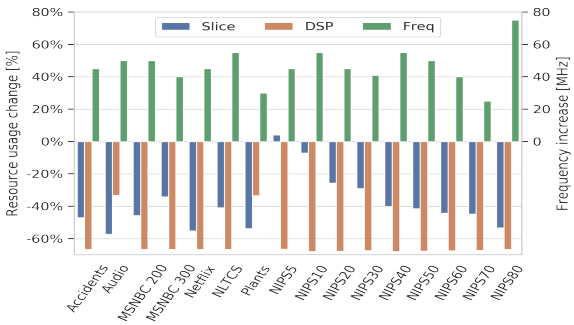


Fig. 5. Improved resource and maximum frequency for floating-point arithmetic in comparison with prior work [2].

VI. EVALUATION

A. Benchmarks

In order to be able to compare the performance and FPGA resource usage directly to [2], we use the same set of benchmarks. The set contains two kinds of benchmarks: Count-based examples, which are taken from the NeurIPS corpus [27] and capture information about the frequency of words in texts, and examples with binary input variables, which were preprocessed by [28] and [29] and capture statistical data, such as usage statistics of services. More detailed information on the individual benchmarks can be found in [2].

B. FPGA Implementation Results

We first compare the resource usage of the three different arithmetic formats for the benchmark set, using the configurations from Table I. Xilinx Vivado 2019.1 and TaPaSCo 2019.10 (pre-release) are used to generate bitstreams for a Xilinx Virtex 7 FPGA device (xc7vx690), all numbers given here are taken from the post-place&route reports. We use the automatic design-space exploration feature of TaPaSCo [30] to determine the best possible frequency. All bitstreams are tested in actual hardware on a Xilinx VC709 development board, verifying that the configurations determined by our DSE (cf. Section IV) maintain the given error bound of 1×10^{-6}.

The FPGA implementation results are given in Table IV. For brevity, numbers are given relative to the entire FPGA, the absolute number of resources available are 108,300 (Slices), 1,470 (BRAM) and 3,600 (DSP), respectively.

Through our automatic design-space exploration to determine the minimum viable configuration and the optimization to the floating point operators described in Section V-A, the resource usage compared to the results reported in [2], decreases by up to 57% in logic slices (avg. 38.5%) and up to 68% in DSP (avg. 62.9%). Additionally, the clock frequency increases by up to 75 MHz (avg. 46.6 MHz). The decrease in resource consumption is also depicted in Fig. 5.

The comparison between customized floating-point (CFP) and Posit shows that the latter requires significantly more logic (avg. +53%) and, except for benchmarks *Audio* and *Plants*, which contain a low number of adders in comparison

TABLE IV
FPGA IMPLEMENTATION RESULTS FOR ALL BENCHMARKS, USING THE CONFIGURATIONS FROM TABLE I. BEST VALUES BOLD.

Benchmark	M/A	Slices [%]			DSP [%]			BRAM [%]			Frequency [MHz]			Pipeline Depth		
		CFP	Posit	LNS	CFP	Posit	LNS	CFP	Posit	LNS	CFP	Posit	LNS	CFP	Posit	LNS
Accidents	8	**40.19**	70.81	41.31	**12.06**	24.11	15	**3.71**	3.71	7.52	**245**	200	222	**73**	161	169
Audio	22.9	**38.22**	77.02	38.63	30.56	30.56	**6.33**	**3.71**	3.71	5.75	**250**	200	210	**73**	161	169
MSNBC 200	5.5	**35.82**	53.5	41.69	**9.17**	18.33	15.83	**3.71**	3.71	6.77	**250**	215	245	**143**	299	577
MSNBC 300	6	**30.61**	43.59	38.35	**5.67**	11.33	8.97	**3.71**	3.71	6.77	240	225	**255**	**89**	213	431
Netflix	21	39.3	67.98	**37.33**	12.83	25.67	**5.81**	**3.71**	3.71	5.75	**235**	215	220	**68**	149	166
NLTCS	5.6	**36.64**	51.38	40.51	**8.44**	16.89	13.5	**3.71**	3.71	6.46	255	220	**270**	**98**	206	342
Plants	18.3	**40.14**	74.78	41.17	28.44	28.44	**7.39**	**3.71**	3.71	6.09	230	205	**250**	**128**	278	385
NIPS5	10	**25.11**	26.59	26.17	0.56	1.11	**0.44**	3.74	**3.71**	3.84	**245**	240	240	**24**	58	60
NIPS10	8.3	**27.44**	30.71	28.09	1.39	2.78	**1.33**	3.74	**3.71**	4.18	**255**	240	**255**	**42**	96	182
NIPS20	8	**28.93**	37.88	31.28	**3.11**	6.22	3.5	**3.81**	4.01	4.69	245	200	**270**	**46**	108	203
NIPS30	8.7	**32.85**	44.87	35.43	**4.83**	9.67	5	**3.74**	4.01	4.93	240	220	**250**	**63**	132	209
NIPS40	7.6	**34.48**	50.35	39.42	**6.78**	13.56	7.56	**3.91**	4.08	5.95	255	215	**265**	**63**	132	224
NIPS50	8.9	**36.86**	54.99	42.42	**7.94**	15.89	8.44	**3.98**	4.15	6.09	**250**	215	245	**68**	144	227
NIPS60	12	**38**	59.91	40.34	8.67	17.33	**6.86**	**4.32**	4.46	6.19	**240**	215	235	**63**	132	224
NIPS70	12.9	**41.75**	67.86	43.12	10	20	**7.39**	**4.05**	4.35	6.97	**225**	203	210	**73**	156	230
NIPS80	8.3	**44.83**	82.84	47.52	**14.72**	29.44	20.44	**3.98**	4.63	7.72	**255**	195	230	**83**	211	306

to the number of multipliers, also twice the number of DSPs. The BRAM utilization is almost identical, the frequency is typically lower for Posit (avg. 30 MHz less) and the pipelines are notably deeper. Overall, one can conclude that Posit is less suitable for SPN inference than floating-point, probably because the numbers involved in SPN inference lie outside of the *golden range* (cf. Section IV-B), where Posit could make up for the additional decoding logic by using much narrower bitwidths. However, the Posit-based arithmetic still outperforms the double-precision arithmetic used in [2] by up to 22.6% in slices (avg. 9.5%) and 36% in DSP (avg. 34.2%).

When compared with floating-point, LNS requires slightly more slices (avg. +7.57%) and significantly (avg. +56%) more BRAM, which, however, is not a critical resource in our case. The frequencies are comparable, with winners in both formats. The pipelines are much deeper, mainly due to the long latency (64 cycles) of the LNS adder. The DSP usage comparison between floating-point and LNS is highly dependent on the multiplier/adder-ratio (given as *M/A* in Table IV) of the examples. Only if there are roughly nine times more multipliers than adders, LNS outperforms floating-point with regard to the DSP usage (NIPS10 is an outlier, probably due to the very low DSP usage in both formats). Overall, it seems that LNS is only suitable for such SPNs with a much higher number of multipliers than adders. Yet, the LNS-based arithmetic is able to outperform the FloPoCo double-arithmetic results from [2] by up to 57.5% in slices (avg. 33.7%) and 86% in DSP (avg. 66%), in particular for examples with only a few adders.

To further validate our results, we also tested relaxed error conditions, namely 1×10^{-4} and 1×10^{-2}, for benchmarks *Accidents* and *Audio*, which were chosen because of their very different adder/multiplier-ratio. We have to omit detailed results for brevity here, but overall, the relation between LNS- and floating-point format found in the evaluation for 1×10^{-6} *persists* for relaxed error conditions: LNS is only able to save resources in comparison to floating-point, if the SPN contains very few adders compared to the number of multipliers.

C. Power Evaluation

Next to the required chip area, we are also interested in the impact of the arithmetic format onto power consumption.

In order to investigate the power consumption of the different arithmetic formats, we consider only the datapath itself, leaving out the memory infrastructure and TaPaSCo platform infrastructure. We again run synthesis and P&R for the Xilinx VC709 board using Vivado 2019.1. Afterwards, we use Mentor Questasim 2019.2 to run a *post-implementation timing simulation* to capture signal activity information from a run with actual inference input data. Using this activity information, we then use the Vivado 2019.1 power analysis for an estimate of the power consumption of the datapath.

As the post-implementation timing simulation can take several days for larger circuits, we again limit our investigation to the two benchmark instances *Accidents* and *Audio*, that we selected for the reasons described in the previous section. In addition to the three arithmetic formats investigated in this work, we also conduct the measurement for the double-precision FloPoCo-format from prior work [2].

TABLE V
POWER CONSUMPTION OF THE DATAPATH.

Benchmark	Power Consumption [Watt]			
	[2]	CFP	Posit	LNS
Accidents	12.493	**3.069**	5.267	3.721
Audio	18.427	5.518	8.358	**2.818**

The results from the power analysis (Table V) align with our findings for the chip area in the previous sections: In the benchmark instance *Accidents*, where the customized floating-point was the most area-efficient format, it also requires the least power, followed by LNS. For the benchmark instance *Audio*, where LNS was the most area-efficient format due to the low number of adders in the SPN, LNS also requires the least power. Just as before, Posit is not able to keep up with the two other formats with regard to power usage.

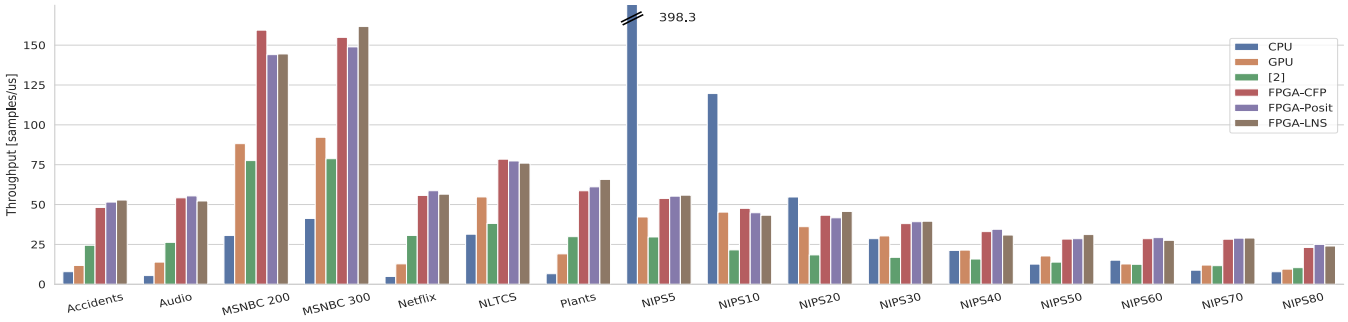


Fig. 6. Throughput of the CPU, GPU and FPGA-implementations in samples/μs. Each group represents an example SPN. The single outlier is the CPU throughput for example NIPS5 which amounts to 398.8 samples/μs.

Compared to the double-precision format from prior work, the SPN-optimized arithmetic formats developed in this work are able to save significant amounts of power.

D. Performance Evaluation

In this section, we evaluate the performance of three arithmetic formats implemented on the FPGA and compare it to a CPU and GPU-based implementation of SPN inference.

1) CPU & GPU Baseline: Based on the compiler infrastructure that we created for the design-space exploration (cf. Section IV), we additionally built a custom compilation flow mapping an SPN description to optimized C++ and CUDA-code, both using double-precision floating-point arithmetic. In both cases, we compiled using `-O3` and `-ffast-math` to enable aggressive compiler optimizations. Our C++ compilation flow on an AMD Ryzen 1600X performs on par with the CPU-baseline from [2], and our CUDA compilation flow is able to *outperform* the original Tensorflow-based GPU-mapping from [2] by a factor of up to 90x on a Nvidia 1080Ti GPU.

2) Performance Comparison: For the comparison, we run the inference on the VC709 development board, coupled with an AMD Ryzen 1600X. Our measurements of the throughput in Fig. 6 also include the time required to *transfer* the data between host and FPGA.

For the three smallest count-based samples (NIPS5-20), the CPU provides the best throughput. For these small networks the overhead for data-transfer to the accelerator (GPU or FPGA) clearly dominates the execution time. With our optimized CUDA compilation flow, the GPU provides better throughput than the CPU for the remaining benchmarks, in particular for the binary examples.

Despite the large differences in the pipeline-depth (cf. Table IV), the performance for the three arithmetic formats implemented on the FPGA varies only slightly. Overall, all three versions deliver very similar performance (with an overall difference of less than 2%). Compared to the previous FPGA implementation in [2], the new formats provide better throughput (geo.-mean. 2.1x speedup). This is partly due to the higher operator frequencies, but also caused by improvements to the underlying TaPaSCo framework.

All three formats significantly *outperform* the CPU. Except for the three benchmarks mentioned earlier, the speedup reaches as high as factor 12x (geo.-mean 2.5x). The three FPGA versions also provide significantly *higher throughput* than the GPU-based implementation, here, the speedups reach up to 4.6x (geo.-mean. 2.1x).

Again, note that our measurements include the PCIe data-transfer to the FPGA memory. On shared-memory systems such as Zynq MPSoC, the speedup over the CPU and the GPU would reach up to 37x and 14x, respectively.

VII. Conclusion & Outlook

In this work, we have investigated three different arithmetic formats with regard to their suitability for Sum-Product Network Inference on FPGAs. We have developed an automatic design-space exploration framework, which allows us to efficiently identify the minimum bitwidth required for each of the formats to maintain a given error margin. Based on the findings from the DSE, hardware arithmetic operators, optimized for SPN inference, for each of the formats were implemented.

Our evaluation shows that customized floating-point is the most resource-efficient format for SPN inference, and is only outperformed by a logarithmic number system format for SPNs with *very few* adders compared to the number of multipliers. All three investigated arithmetic formats deliver almost identical performance and significantly outperform CPU and GPU-based implementations of SPN inference, by factors up to 12x and 4.6x, respectively.

In future work, we will investigate how the hardware arithmetic operators can be optimized further, e.g., by using fused operators.

Acknowledgements

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

Finally, the authors would like to thank Kristian Kersting and Alejandro Molina, for much appreciated discussions of the subject and insights into the matter of Sum-Product Networks.

Lukas Sommer and Lukas Weber contributed equally to this work.

REFERENCES

[1] 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.

[2] 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 *36th Intl. Conf. on Computer Design (ICCD)*, Oct 2018, pp. 350–357.

[3] L. Weber, L. Sommer, J. Oppermann, A. Molina, K. Kersting, and A. Koch, "Resource-Efficient Logarithmic Number Scale Arithmetic for SPN Inference on FPGAs," in *International Conference on Field-Programmable Technology (FPT)*, 2019.

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

[5] M. Ratajczak, S. Tschiatschek, and F. Pernkopf, "Sum-Product Networks for Sequence Labeling," *CoRR*, vol. abs/1807.02324, 2018.

[6] A. Pronobis, F. Riccio, and R. P. Rao, "Deep spatial affordance hierarchy: Spatial knowledge representation for planning in large-scale environments," in *ICAPS 2017 Workshop on Planning and Robotics*, 2017.

[7] J. Bekker, J. Davis, A. Choi, A. Darwiche, and G. Van den Broeck, "Tractable Learning for Complex Probability Queries," in *NIPS*, 2015.

[8] R. Peharz, S. Tschiatschek, F. Pernkopf, and P. Domingos, "On Theoretical Properties of Sum-Product Networks," in *Proc. of AISTATS*, 2015.

[9] H. Zhao, M. Melibari, and P. Poupart, "On the Relationship between Sum-Product Networks and Bayesian Networks," in *Proc. of ICML*, 2015.

[10] A. Molina, A. Vergari, N. D. Mauro, F. Esposito, S. Natarajan, and K. Kersting, "Mixed Sum-Product Networks: A Deep Architecture for Hybrid Domains," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[11] J. Alves, J. Ferreira, J. Lobo, and J. Dias, "Brief survey on computational solutions for Bayesian inference," in *Unconventional comp. for Bayesian inference*, 2015.

[12] 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.

[13] J. L. Gustafson and I. T. Yonemoto, "Beating Floating Point at its Own Game: Posit Arithmetic," vol. 4, Apr. 2017.

[14] M. K. Jaiswal and H. K. H. So, "PACoGen: A Hardware Posit Arithmetic Core Generator," *IEEE Access*, vol. 7, pp. 74 586–74 601, 2019.

[15] A. Podobas and S. Matsuoka, "Hardware Implementation of POSITs and Their Application in FPGAs," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018.

[16] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, "Parameterized Posit Arithmetic Hardware Generator," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct. 2018.

[17] 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.

[18] J. Detrey and F. de Dinechin, "A VHDL library of LNS operators," in *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003*, Nov 2003.

[19] N. Shah, L. I. G. Olascoaga, W. Meert, and M. Verhelst, "ProbLP: A Framework for Low-precision Probabilistic Inference," in *56th Annual Design Automation Conference*, ser. DAC '19. New York, NY, USA: ACM, 2019.

[20] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Neep Neural Networks with Pruning, Trained Quantization and Huffman coding," 2015.

[21] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, "Posits: The Good, the Bad and the Ugly," in *Proceedings of the Conference for Next Generation Arithmetic 2019*, ser. CoNGA'19. New York, NY, USA: ACM, 2019, pp. 6:1–6:10.

[22] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

[23] F. de Dinechin. FloPoCo Project Website. [Online]. Available: http://flopoco.gforge.inria.fr

[24] P. M. Farmwald, "On the design of high performance digital arithmetic units," Ph.D. dissertation, Stanford University, 1981.

[25] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran, "Multipliers for Floating-Point Double Precision and Beyond on FPGAs," *SIGARCH Computer Architecture News*, vol. 38, no. 4, pp. 73–79, Sep. 2010.

[26] M. Kumm, J. Kappauf, M. Istoan, and P. Zipf, "Resource Optimal Design of Large Multipliers for FPGAs," in *2017 24th Symp. on Computer Arithmetic*, July 2017.

[27] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[28] D. Lowd and J. Davis, "Learning Markov network structure with decision trees," in *Data Mining (ICDM), 2010 IEEE 10th International Conf.*, 2010.

[29] J. Van Haaren and J. Davis, "Markov Network Structure Learning: A Randomized Feature Generation Approach." in *AAAI*, 2012, pp. 1148–1154.

[30] J. Korinth, J. Hofmann, C. Heinz, and A. Koch, "The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems," in *Applied Reconfig. Comp.*, 2019.