Automatic Synthesis of FPGA-based Accelerators for the Sum-Product Network Inference Problem

Lukas Sommer [1] **Julian Oppermann** [1] **Alejandro Molina** [2] **Carsten Binnig** [3] **Kristian Kersting** [4] **Andreas Koch** [1]

Abstract

In recent years, FPGAs have been successfully employed for the implementation of efficient, application-specific accelerators for a wide range of machine learning tasks. In this work, we consider probabilistic models, namely, (Mixed) Sum-Product Networks (SPN), a deep architecture that can provide tractable inference for multivariate distributions over mixed data-sources. We develop a fully pipelined FPGA accelerator architecture, including a pipelined interface to external memory, for the inference in (mixed) SPNs. To meet the precision constraints of SPNs, all computations are conducted using double-precision floating point arithmetic. Starting from an input description, the custom FPGA-accelerator is synthesized fully automatically by our tool flow. To the best of our knowledge, this work is the first approach to offload the SPN inference problem to FPGA-based accelerators. Our evaluation shows that the SPN inference problem benefits from offloading to our pipelined FPGA accelerator architecture.

1. Introduction

The computational demand of many deep learning approaches, the currently predominating branch of machine learning (ML), can only be satisfied by specialized accelerators. Besides GPUs, which provide massive parallelism for regular computations, and custom processors such as Google's TPU (Jouppi et al., 2017), which supports operations typical for ML now, but is unable to adapt to advances in ML algorithms, FPGAs have shown promise as an energy-efficient-yet-flexible alternative (Nurvitadhi et al., 2017).

A truly intelligent system, however, should be able to deal with uncertain inputs (e.g. missing features) as well as express its uncertainty over outputs. It is, therefore, no surprise that probabilistic approaches have recently gained tremendous momentum within deep learning. Corresponding approaches such as variational autoencoders, deep generative models, and generative adversarial nets (GANs), however, have limited capabilities when it comes to *probabilistic inference*. Consider, e.g. implicit likelihood models like GANs (Goodfellow et al., 2014). Even when successful in capturing the data distribution, they do not allow to compute the probability of a test sample. In contrast, Sum-Product Networks (SPNs) (Poon & Domingos, 2011) are a deep architecture that permit *exact* and *efficient* probabilistic inference. More precisely, they can compute any marginalization and conditioning query in time linear to the model's representation size, by evaluating its computational graph representation—consisting of computational nodes (addition & multiplication) and distribution nodes—in a bottom-up fashion. This computational structure and the need for efficient processing of batches of queries in different AI application scenarios make SPNs a promising candidate for FPGA-based acceleration.

Thus, the goal of the present paper is to develop a new, pipelined FPGA accelerator architecture for the sum-product network inference problem. This opens the door to local (on-chip) model inference and in future work, even model learning. The dynamic range of the probabilities and the need to represent values with very small magnitude make the use of high-precision floating-point arithmetic necessary. Besides the pipelined accelerator architecture, we also develop an automatic synthesis flow for SPNs. Our tool flow is a turn-key solution that not only encompasses the actual synthesis of a hardware datapath from an SPN description but also provides crucial parts for any practically-usable FPGA accelerator, i.e., a high-throughput memory system and accompanying software APIs. In our experimental evaluation, we show that our pipelined architecture can handle different model sizes and still provide high throughput which is an essential feature of scalable inference. To the best of our knowledge, this is

[1]Embedded Systems and Applications Group, TU Darmstadt, Germany [2]Machine Learning Group, TU Darmstadt, Germany [3]Data Management Lab, TU Darmstadt, Germany [4]Machine Learning Group and Centre for Cognitive Science, TU Darmstadt, Germany. Correspondence to: Alejandro Molina <molina@cs.tu-darmstadt.de>.

Working Notes of a ICML/FAIM 2018 Workshop, Stockholm, Sweden, 2018. Copyright 2018 by the author(s).

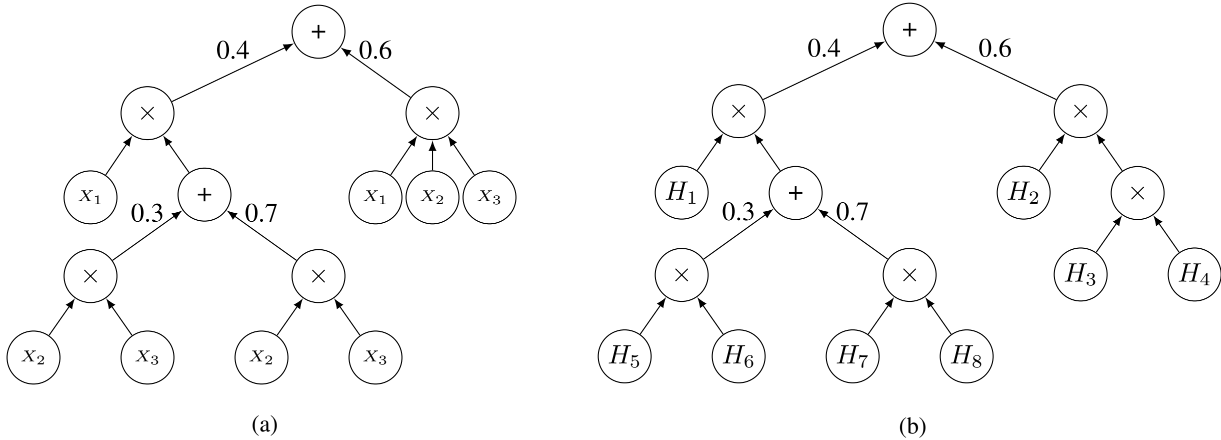

Figure 1: **(a)** An example of a valid SPN. Here, x_1, x_2 and x_3 are random variables. The structure represents the joint distribution $P(x_1, x_2, x_3)$. **(b)** Intermediate representation of the SPN. Operators with n inputs are split into balanced trees of operators with 2 inputs. Univariate distributions are modelled by histograms H_i.

the first approach to offload the SPN inference problem to FPGA-based accelerators.

We proceed as follows. Section 2 briefly reviews SPNs, how they are learned from training data and how inference works. Afterwards, we touch upon existing approaches to accelerate probabilistic graphical models on various architectures. Section 4 then presents our pipelined accelerator architecture, our automatic tool flow, and the integration into a heterogeneous system. In section Section 5 we evaluate our accelerator and compare it to other architectures and section Section 6 concludes the paper and looks forward to future work.

2. Sum-Product Networks

Probabilistic Graphical Models (PGMs) have had a broad impact on machine learning, both in academia and industry. They can solve many ML problems by simply answering probabilistic queries. Consider, e.g., predictive modeling. One trains a PGM which then answers probabilistic queries; for multi-class classification answering the query $\arg\max_c P(\text{class} = c|\text{data})$ gives us the most likely class according to our model. Alternatively, as in one of our benchmark datasets, we can ask which is the most likely plant to grow in a particular location: $\arg\max_p P(\text{plant} = p|\text{location})$. Unfortunately, inference in unrestricted PGMs is often intractable. Motivated by the importance of efficient inference for large-scale applications, a substantial amount of work has been devoted to learning probabilistic models for which inference is guaranteed to be tractable. Examples of these model classes include sum-product networks (SPNs), hinge-loss Markov random fields, and tractable higher-order potentials. Being instances of Arithmetic Circuits (ACs), see (Choi & Darwiche, 2017) for a discussion, SPNs are a deep architecture that can represent high-treewidth models (Zhao et al., 2015) and facilitate *exact* inference for a range of queries in time *linear* in the network size (Poon & Domingos, 2011; Bekker et al., 2015). They inherit universal approximation properties from mixture models—a mixture model is simply a "shallow" SPN with a single sum node. Consequently, SPNs can represent any prediction function, very much like deep neural networks. However, having exact probabilities offers an advantage not present in other PGMs and deep neural networks. One can compare the probabilities computed by different models and not only solve classification or regression problems, but also do anomaly detection at the same time while taking into account the statistical nature of the data. Furthermore, any measures based on probabilities can be computed exactly. Among those measures, we find Entropy, Mutual Information, Information Gain, etc. Moreover, the Mixed Sum Product Network (MSPN) proposed by Molina *et al.* (Molina et al., 2018), is a non-parametric version of SPNs that opens the door for an efficient FPGA implementation based on histograms. This MSPN maintains the expressiveness while representing a wide range of statistical data types.

2.1. Definition of SPNs

Formally, an SPN is a rooted directed acyclic graph, comprising *sum*, *product*, and *leaf* nodes as seen in Fig. 1. The scope of an SPN is the set of random variables appearing on the network. An SPN can be defined recursively as follows: (1) a tractable univariate distribution is an SPN; (2) a product of SPNs defined over different scopes is an SPN; and (3), a convex combination of SPNs over the same scope is an SPN. Thus, a product node in an SPN represents a factorization over independent dis-

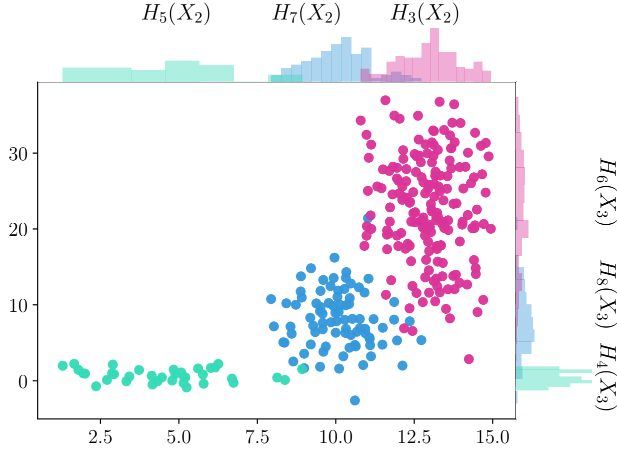

Figure 2: Example of a synthetic dataset, fitted using the SPN in Fig. 1b. For illustration purposes, we present here $P(X_2, X3)$ marginalizing X_1 out. $H_i(X_j)$ represent the different histograms in the MSPN. The histograms are free to overlap, but the greedy learning algorithm attempts to represent different subsets of the data. Deeper MSPNs can represent more complex datasets.

tributions defined over different random variables, while a sum node stands for a mixture of distributions defined over the same variables. From this definition, it follows that the joint distribution modeled by such an SPN is a valid probability distribution, i.e., each complete and partial evidence inference query produces a consistent probability value (Poon & Domingos, 2011; Peharz et al., 2015). This also implies that we can construct multivariate distributions from simpler univariate ones. Furthermore, any node in the network could be replaced by any tractable multivariate distribution over the same scope, obtaining still a valid SPN. Computationally, the number of arithmetic operations is different for the given nodes. For product nodes, we have $|\text{children}| - 1$ number of multiplications. For sum nodes, we have $|\text{children}| - 1$ number of additions and $|\text{children}|$ number of multiplications. The leave nodes require as many operations needed for a look-up table of size $|\text{domain}(\text{Variable}_i)|$. The SPNs make no restriction on reusing sub-structures as long as the consistency rules are preserved.

2.2. Tractable Inference in SPNs

To answer probabilistic queries in an SPN, we evaluate the nodes starting at the leaves. Given some evidence, the probability output of querying leaf distributions is propagated bottom up. For product nodes, the values of the child nodes are multiplied and propagated to their parents. For sum nodes, instead, we sum the weighted values of the child nodes. The value at the root indicates the probability of the asked query. To compute marginals, i.e., the probability of partial configurations, we set the probability at the leaves for those variables to 1 and then proceed as before. An example of such marginalization is shown in Fig. 2, where we obtain a new SPN that computes $P(X_2, X_3) = \sum_x P(X_1 = x, X_2, X_3)$ and show how it can be used to fit a randomly generated dataset. All these operations traverse the tree at most twice and therefore can be achieved in linear time w.r.t. the size of the SPN. In this work, we consider only datasets with discrete variables, as they can be easily implemented with fast look-up tables, however extending the histograms to the continuous case is not difficult. We also focus on joint computations as they are the basis for all other inference algorithms. These operations at the leaves are then executed in constant time, maintaining the tractability of the SPN.

2.3. Learning SPNs

While it is possible to craft the structure of a valid SPN by hand, doing so requires domain knowledge and weight learning afterward (Poon & Domingos, 2011). Here, we use the greedy, top-down approach of the MSPNs that directly learns both the structure and weights of (tree) SPNs at once while making few assumptions on the data.

It consists of three steps: (1) base case, (2) decomposition and (3) conditioning. In the base case, if only one variable remains, the algorithm learns a univariate distribution and terminates. Here, we represent univariate distributions by normalized histograms. These histograms are then converted to look-up tables, as they can be efficiently implemented on FPGAs. We use the same implementation for the traditional CPU code to keep the experiments as similar as possible. In the decomposition step, it tries to partition the variables into independent components. This decomposition is based on a non-parametric independency test (**?**) that is run for every random variable $V_j \subset \mathbf{V}$ in a pair-wise fashion, creating a graph of interactions among all the variables. We then obtain the disconnected components of this graph, which indicates that variables inside a component are tightly coupled and variables among different components are independent. From these components, we induce a product node such that $P(\mathbf{V}) = \prod_j P(V_j)$ and recurse on each child.

If the base case is not applicable and the decomposition step cannot find independencies, then the algorithm partitions the training samples into clusters (conditioning). This clustering procedure first transforms the data to a higher-dimensional space so that discrete variables fit the assumptions of normality of the clustering algorithm. From the clusters, we induce a sum node, and the algorithm recurses on each cluster. The weights of the sum nodes then repre-

sent the data proportions in the clusters.

This learning algorithm does not reuse sub-structures, however, the intermediate language representation used keeps track of sub-tree references and the FPGA pipeline is aware of them. Using a different algorithm or a pruning or compression step can enable this capability.

This learning algorithm is typically pre-computed on a traditional CPU. Then the resulting SPN structure can be compiled for inference into FPGAs, C++ code or even TensorFlow graphs, as shown in section 5.

2.4. Size of SPNs

The size of the SPN depends on the training dataset and the learning parameters. The smallest multi-variate SPN is a *product* node over all the random variables, giving a lower bound on the network size of $|\text{Variables}| + 1$. Introducing binary *sum* nodes doubles the size of the sub-graph by the number of random variables in the node. The learning algorithm then creates SPNs whose size is constrained by heuristics for early stopping and the number of independencies recovered from the data. Deeper SPNs are more expressive, but also more computationally intensive, while shallower SPNs have fewer parameters and use fewer resources. Controlling the depth of the SPN impacts how well it fits the data. A very deep structure tends to overfit, while a shallow structure does not have enough expressive power to represent the training data.

In this work, we focus on the largest SPNs that we can fit entirely in FPGAs in a fully spatial realization.

3. Related Work

To the best of our knowledge, our work is the first automatic synthesis tool to accelerate SPN inference on FPGAs.

Previous research has studied the FPGA acceleration of other kinds of PGMs such as Bayesian Networks (BN) (Alves et al., 2015) and Markov Random Fields (MRF) (Choi & Rutenbar, 2016). These approaches are orthogonal to ours, as the inference problem, in general, is not tractable for BNs and MRFs. A common theme in BN accelerators is to design specialized processors, as inference in large tree-width models is expensive. In contrast, the arithmetic circuit (AC) representation of BNs (Darwiche, 2003) resembles a datapath similar to an SPN, which Zermani *et al.* (Zermani et al., 2015) first compiled to C code and then used Vivado HLS to synthesize an IP core to be used on a Zynq SoC. Other ACs implementations for FPGAs can be found in ((Dormiani et al., 2005), (Geist et al., 2014)). However, ACs are restricted to binary random variables, whereas our SPN-based approach has no such restriction. Lastly, the recently presented LibSPN (Pronobis et al., 2017) aims to bring multi-core and GPU acceleration to SPNs by translating them to TensorFlow (Abadi et al., 2015), but is not yet publicly available and does not support histogram-based representation at the leaf-nodes. For evaluation purposes, we implemented our own TensorFlow-based GPU backend.

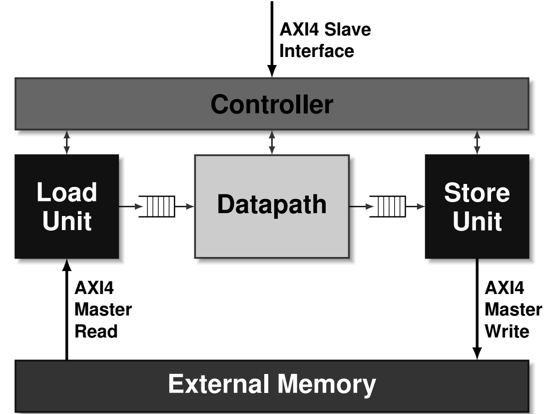

Figure 3: Overview of the system architecture.

4. Approach & Implementation

The goal of this work is to synthesize efficient accelerators for the inference problem in sum-product networks. In this section, we present our accelerator architecture, the compilation flow that generates the accelerator using the SPN structure learned as described in Section 2.3 as input and the integration of our accelerator into a heterogeneous system.

4.1. System Overview

In this paper, we aim for a fully pipelined accelerator architecture since we want to process a large number of queries efficiently and each query is independent of other queries.

The vast amount of data required for the input values of each query makes it inevitable to provide the accelerator with access to the external memory on the FPGA board. We use the open-source TaPaSCo framework (Korinth et al., 2015), which provides a standardized memory interface as well as an interface and host-side API for the integration into a heterogeneous system (cf. Section 4.2).

Our accelerator architecture consists of four main components, depicted in Fig. 3. The controller is responsible for managing the other components and provides the necessary TaPaSCo slave interface (Section 4.2). Load and store unit together make up the memory interface of the accelerator, described in more detail in Section 4.4. The datapath implements the computation represented by the sum-product network, see Section 4.3 for details on its construction.

Load unit, datapath, and store unit are decoupled by queues to allow for the latency-insensitive, independent operation of the different components.

4.2. TaPaSCo & Heterogeneous system integration

In order to integrate our accelerator into a heterogeneous system, we use the TaPaSCo-framework (Korinth et al., 2015).

The open-source TaPaSCo toolchain and framework has been developed to fast-track the prototyping of FPGA-based accelerators. TaPaSCo defines AXI-based, standardized interfaces, which are used to control the execution of the accelerator (slave interface) and provide the accelerator with memory access (master interface). TaPaSCo also includes an automated tool-flow to assemble multiple instances of these accelerators (*processing elements*, PEs) into a complete top-level design, called *threadpool*, which is then wrapped with the connectivity to host and memory.

Regardless of its composition, every threadpool can be controlled by an unified software interface, the so-called TaPaSCO-API. This API provides basic functions to transfer data to/from the device and launch jobs on the accelerators in the threadpool.

To integrate our accelerator into the threadpool, we implement the necessary interfaces. Our controller (cf. Fig. 3) implements an AXI4 slave interface, which is used by TaPaSCo to transmit commands (e.g. the start-command) from the host by reading/writing configuration register values and control signals. Load and store unit in combination implement an AXI4 master interface, which is connected to the external memory on the FPGA board through the TaPaSCo infrastructure. The implementation of this interface is described in more detail in Section 4.4.

The host can transfer data to/from the external memory on the device by using the TaPaSCo API. We also use the API to control the execution of our accelerator in the FPGA. This allows us to seamlessly integrate the offloading to the FPGA accelerator into the host software for SPN inference.

4.3. Compile flow & Datapath architecture

Our automatic tool-flow that maps SPNs to FPGA accelerators starts from a textual representation of the SPN, which describes the nodes of the SPN, their individual configuration, and the connections between the different nodes in the tree-structure of the SPN.

In a first step, we parse the textual input and construct a graph-based intermediate representation of the SPN, as shown in Fig. 1b for the example SPN from Fig. 1a. During the construction of the graph IR, we decompose additions or multiplications with more than two operands into balanced trees of two-input operators, as can be seen for the three-input multiplication on the right side of the example SPN.

As mentioned earlier in Section 2.3, the random variables with univariate distributions at the leaf nodes of the SPN tree-structure are modeled with histograms. Each histogram consists of multiple bins with corresponding probabilities, and the bins match adjacent intervals of input feature values. If two neighboring bins share the same probability value, our tool-flow merges these bins, resulting in a bin which covers the two adjacent intervals of the original bins.

The configuration of the histograms is also part of another optimization implemented in our tool-flow: The sample values stored in memory are used to index into the bins of the different histograms. Based on the largest possible value for the input features, which is determined by the upper bound of the last bin, we can optimize the input bitwidth. Across all histograms, we calculate the minimum number of bits n, necessary to represent the highest possible input value. All input values are then stored in memory using n bits. This optimization reduces the pressure on the memory interface, as less bandwidth is required to read the input values for each sample from memory. If, for example, the highest possible input values across all histograms was 212, we could represent all input values with only eight bit, reducing the memory bandwith requirement by a factor of four compared to a generic representation with 32 bit.

Before we map the tree to hardware operators, we decompose weighted adders into corresponding combinations of multipliers and adders. In the next step, we now map the SPN tree structure to the actual hardware datapath. As we aim for maximum throughput in this first version of our synthesis tool, we use a fully-spatial, statically scheduled microarchitecture for our datapaths. That is, every operation in the SPN is implemented by its own operator module on the device, and no resource-sharing occurs. For the scheduling of the fully-spatial microarchitecture, a simple as-soon-as-possible strategy suffices to obtain an optimal (concerning the latency) static schedule for the datapath. Our pipelined schedule assumes that the memory interface can provide an input sample in every clock cycle. However, depending on the number of input features and the bitwidth of the memory interface (cf. Section 4.4), this might not be the case. We therefore introduce a shift register into our datapath, which keeps track of valid samples in the pipeline to make sure we only write back correct output values to memory.

For the implementation of the histograms at the leaf-nodes of the SPN, we use a simple look-up table approach. The values for each index are computed at compile time, and the discrete values of the input features are used to index

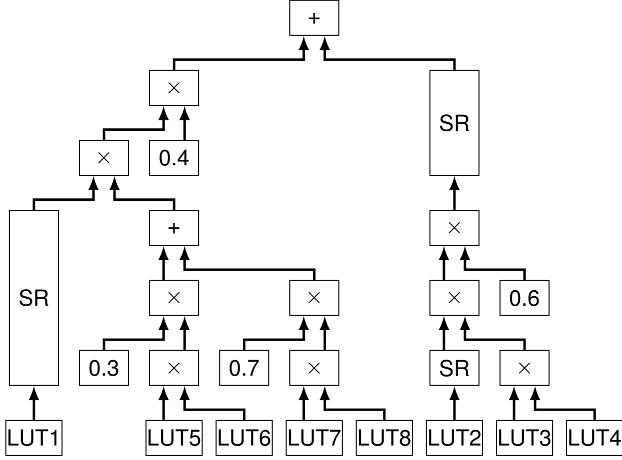

Figure 4: Example SPN mapped to HW operators. To balance the different branches of the tree, some intermediate results need to be preserved in shift registers. Look-up tables for the histograms are indexed with the input feature values read from memory.

the look-up table.

For the realization of the arithmetic operations inside the tree (additions and multiplications), we use operators from the FloPoCo-library (de Dinechin & Pasca, 2011). Regarding the precision, a worst case scenario arises from the inference of a low-probability instance such as an anomaly or an outlier data point evaluated on a shallow SPN consisting of one product over all the univariate features. Furthermore, the number of discrete values or bins in the look-up table can have a negative impact as they represent a normalized distribution. This has a lower bound of $\prod_{i=0}^{\#\text{features}} \min(\mathrm{histogram_i(x)})$. To represent these values, which have a very small magnitude, the whole computation within the datapath is completely done in the FloPoCo floating point format, using 11 bits for the exponent and 52 bits for the mantissa. Aside from subnormal numbers, this format is equivalent to a double-precision floating-point format. The final conversion to IEEE-754 double precision format for use in the host is done right before the values are written back to memory.

In practice, the SPNs used in the experimental section did not suffer underflows, however, bigger SPNs might have to do computations in log-space. The size of the SPNs is limited to the FPGA capabilities and implementing exp and log functions, would reduce the size of the SPNs that we could implement on-chip.

The mapping to the hardware datapath for the example from Fig. 1 is given in Fig. 4. The height of the operators in the diagram indicates their scheduled start time, with the first time step at the bottom. All look-up tables used to implement the histograms are placed in the first time step and will be indexed by the discrete values for the three input variables read from memory. As one can see from the IR-graph in Fig. 1b, the different branches of the tree have different heights/latencies. We need to balance the different branches to match partial results from the different branches at the merge points. To this end, we insert pipelined shift registers into the datapath that work as intermediate storages for partial results (labeled *SR* in the diagram).

4.4. Memory Interface

As explained in Section 4.2, TaPaSCo provides each processing element, such as our accelerator, with a standardized AXI4-interface which we use to connect our accelerator to the external memory on the FPGA-board.

With our fully pipelined accelerator architecture, the goal is to feed the pipeline with a new query every clock cycle. This requires an efficient, pipelined load infrastructure, in particular as the number of bits at the input of the datapath is typically higher than at the output (due to the tree-structure of SPNs). For an efficient supply of input data, we use AXI4 burst requests to read the query input values from memory. The load unit computes the addresses for the requests, relative to a base address configurable from the host. The independence of the different AXI4 channels allows us to issue the next burst request before all data from the current one has been sent, resulting in a continuous stream of input data.

Inside the load unit, a re-alignment unit converts from the AXI4 data width (e.g. 512 bit in case of our evaluation on the VC709 board) to the input width of the datapath. Here, we have to consider three different cases: If the bitwidth of a sample matches the bitwidth of the memory interface, we can simply forward the read data. If the input bitwidth of the datapath is smaller than the memory interface bitwidth, the realignment unit buffers the read data and forwards chunks of appropriate size to the datapath. In case the bitwidth at the input of the datapath exceeds the maximum bitwidth of the memory interface, multiple beats must be buffered, before a complete sample can be forwarded to the datapath. In all three cases, the realignment unit was designed to forward a complete sample to the datapath as soon as a sufficient amount of data was read/buffered. In the latter case, the bitwidth of the memory interface limits the performance of our accelerator, and we cannot start the computation for a new sample in every clock cycle. As explained in the previous section, our datapath is equipped with a shift register to keep track of valid values in this case.

After the computation inside the datapath has completed, we need to store the results back to memory. Here we buffer multiple output values and again use AXI4 burst re-

Table 1: FPGA implementation results, all numbers are post-place&route.

Dataset	Cols	Adds	Muls	Depth	Freq. (MHz)	LUT (%)	Reg (%)	Slices (%)	BRAM (%)	DSP (%)
Accidents	111	27	217	99	200	61.49	32.92	75.87	4.66	36.17
Audio	100	12	275	99	200	72.26	37.65	89.32	4.66	45.83
Netflix	100	11	231	115	190	67.60	36.12	87.84	4.66	38.50
MSNBC200	17	30	165	245	200	51.64	27.24	66.01	4.66	27.50
MSNBC300	17	17	102	163	200	36.82	21.76	46.52	4.66	17.00
NLTCS	16	27	152	150	200	47.91	26.72	61.99	4.66	25.28
Plants	69	14	256	206	200	68.89	34.88	86.80	4.66	42.67
NIPS5	5	1	10	38	200	17.96	8.87	24.12	4.80	1.67
NIPS10	10	3	25	76	200	21.20	11.01	29.55	4.80	4.33
NIPS20	20	7	56	82	200	28.84	14.67	38.93	5.27	9.67
NIPS30	30	10	87	94	199	36.47	18.31	46.27	4.86	14.83
NIPS40	40	16	122	94	200	44.79	23.67	57.67	5.14	21.17
NIPS50	50	16	143	100	200	51.78	25.06	62.97	5.48	24.50
NIPS60	60	13	156	94	200	54.71	27.84	68.15	5.88	26.67
NIPS70	70	14	180	106	200	61.97	28.90	75.58	5.61	30.50
NIPS80	80	32	265	143	180	79.72	38.99	96.17	5.95	44.17

quests, which can be handled more efficiently by the memory interface, to write back a batch of values.

5. Experimental Evaluation

Our intention here is to investigate the performance of our synthesis pipline.

5.1. Datasets

For the performance evaluation, we focused on datasets of count and binary data types. For count data we used the NIPS[1] corpus, containing 1,500 documents over the 100 most frequent words. For binary data we evaluated our implementation on a range of six different datasets as pre-processed and provided by (Lowd & Davis, 2010) and (Van Haaren & Davis, 2012). *Accidents* is a dataset of traffic accidents in Belgium for the period 1991-2000. The *Audio* dataset consists of information about users that listened or did not listen to a set of top 100 artists. The *Netflix* dataset is a random subset of the Netflix challenge, focused on the 100 most frequently rated movies and whether a user rated a movie. The anonymized *MSNBC* data contains information about whether a user visited a top-level MSNBC page during a particular browsing session. The National Long Term Care Survey (NLTCS) dataset contains variables that measure whether a person can perform a set of daily living activities. The *Plants* dataset indicates whether a given plant can be found in a particular location.

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

5.2. FPGA implementation

For the evaluation of our FPGA implementation, we target a Xilinx VC709 evaluation board, containing a Virtex7-device (xc7vx690) and 4 GiB of RAM. The heterogeneous system that we use for the performance evaluation of the FPGA in Section 5.4 combines the FPGA-board with an Intel i7 6700K CPU. We used Xilinx Vivado 2017.4 for the FPGA implementation with a target frequency of 200 MHz. The results are given in Table 1. Column *Cols* gives the number of inputs to the datapath, *Adds* and *Muls* give the number of two-input adders and multipliers in the datapath and *Depth* indicates the depth of our computation pipeline. Besides the achieved clock frequency, we also state the resource requirements on the FPGA, for brevity those numbers are given relative to the entire FPGA in percent[2].

We encounter a relatively low demand for BRAMs and a moderate usage of registers. For the other numbers, the resource requirements roughly correspond to the size of the network. This can be seen from the NIPS-examples in particular: Here, we add an increasing number of input features (i.e. words) to the SPN, resulting in a bigger network and thus in a higher resource consumption.

Most of the examples achieve the target frequency, but in some examples the routing to the DSP-blocks used for the realization of the floating-point multiplication in FloPoCo becomes critical. We were able to improve the routing by

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

Table 2: Performance comparison. Best end-to-end throughputs (T), excluding the cycle counter measurements, are denoted bold.

Dataset	Rows	CPU (μs)	T-CPU (rows/ μs)	CPUF (μs)	T-CPUF (rows/ μs)	GPU (μs)	T-GPU (rows/ μs)	FPGA Cycle Counter	FPGAC (μs)	T-FPGAC (rows/ μs)	FPGA (μs)	T-FPGA (rows/ μs)
Accidents	17,009	2,798.27	6.08	2,162.59	7.87	63,090	0.27	17,249	86.25	197.22	696.00	**24.44**
Audio	20,000	4,271.78	4.68	3,683.71	5.43	78,253	0.26	20,317	101.59	196.88	761.00	**26.28**
Netflix	20,000	4,892.22	4.09	4,098.88	4.88	67,172	0.30	20,322	106.95	187.00	654.00	**30.58**
MSNBC200	388,434	15,476.05	25.10	12,713.55	30.55	62,349	6.23	388,900	1,944.50	199.76	5,008.00	**77.56**
MSNBC300	388,434	10,060.78	38.61	9,418.29	41.24	50,558	7.68	388,810	1,944.05	199.81	4,933.00	**78.74**
NLTCS	21,574	791.80	27.25	687.25	31.39	35,544	0.61	21,904	109.52	196.99	566.00	**38.12**
Plants	23,215	3,621.71	6.41	3,521.04	6.59	67,004	0.35	23,592	117.96	196.80	778.00	**29.84**
NIPS5	10,000	25.11	**398.31**	26.37	379.23	8,210	1.22	10,236	51.18	195.39	337.30	29.65
NIPS10	10,000	83.60	**119.61**	84.39	118.49	11,550	0.87	10,279	51.40	194.57	464.30	21.54
NIPS20	10,000	191.30	52.27	182.73	**54.72**	18,689	0.54	10,285	51.43	194.46	543.60	18.40
NIPS30	10,000	387.61	25.80	349.84	**28.58**	25,355	0.39	10,308	51.80	193.06	592.30	16.88
NIPS40	10,000	551.64	18.13	471.26	**21.22**	30,820	0.32	10,306	51.53	194.06	632.20	15.82
NIPS50	10,000	812.44	12.31	792.13	12.62	36,355	0.28	10,559	52.80	189.41	720.60	**13.88**
NIPS60	10,000	1046.38	9.56	662.53	**15.09**	40,778	0.25	12,271	61.36	162.99	799.20	12.51
NIPS70	10,000	1,148.17	8.71	1,134.80	8.81	46,759	0.21	14,022	70.11	142.63	858.60	**11.65**
NIPS80	10,000	1,556.99	6.42	1,277.81	7.83	63,217	0.16	14,275	78.51	127.37	961.80	**10.40**

adding one or two more pipeline stages to the multiplier; however, we see a notable degradation of the operating frequency for *NIPS80* and *Netflix*.

5.3. CPU & GPU Implementation

To have a complete performance comparison, we compiled the same SPNs used in the FPGAs to both C++ and TensorFlow (Abadi et al., 2015). Both implementations were executed on a Linux workstation with an AMD Ryzen 1950X Processor, 128GB of RAM and an NVIDIA 1080Ti GPU with 11GB of memory. We implemented the C++ version via code generation, writing inlined functions with look-up tables for the leaves. The rest of the SPN was expressed as a single function of additions and multiplications of the leaf functions. We compiled the generated C++ source code using GCC 7.2.0 and the flag -O3 and created two versions, one with the flag -ffast-math enabled, called **CPUF**, and one without it, called **CPU**. Both C++ implementations load all the data in memory and evaluate each complete dataset 1000 times. We report the average time for each instance of the dataset. For the **GPU** version, we implemented a TensorFlow graph of additions and multiplications. For the leaves, we used a look-up table implemented as a *tf.gather* operation over placeholders containing the data. We then executed each complete dataset 1000 times and measured execution times including data-transfer to the GPU, just as for the heterogeneous system with the FPGA.

5.4. Performance evaluation

To compare the performance of our FPGA implementation with the CPU and GPU, we report two different execution times for the FPGA: One only for the actual SPN computation, measured using a performance counter (**Cycle Counter**) inside the accelerator, with the corresponding actual time shown as (**FPGAC**). The second time (**FPGA**) is measured on the host side and includes the time for data transfer to/from the device memory and the launch of HW-execution. The performance counter also allows us to assess the effectiveness of our pipelining. The performance results are given in Table 2, *Rows* gives the number of samples processed for each example. Besides the execution time in microseconds, we also report the throughput in samples per microsecond (e.g. **T-GPU**).

The GPU performance clearly shows that the naive TensorFlow parallelization model is not very suitable for the tree-structure of the SPNs. The analysis of the traces shows that most lanes are only used for a few operations and are idle most of the time. Additionally, a lot of inter-lane communication takes place, as the computation of the tree is spread across multiple lanes. This a general problem of the TensorFlow GPU-model, which is tailored more towards neural networks, and not linked to a specific GPU.

The comparison of **CPU** and **CPUF** shows that, except for the two smallest networks, the CPU execution profits from the compilation with the *fastmath*-flag.

Comparing the performance of CPU and the performance measured using the FPGA performance counter, one can see that, aside from NIPS5, the pipelined computation in the FPGA outperforms the CPU implementation regarding execution time and throughput. The values reported by the performance counter also demonstrate the effectiveness of our pipelining: Especially for the cases with binary input data, we are able to achieve an almost perfect pipelining, where we process a sample per clock cycle (equivalent to a throughput of 200 samples (rows) per μs). From the development of the NIPS-examples, one can also see that the accelerator is memory bound: With a larger number of input values, more data has to be loaded from memory, and the pipeline cannot be fed every clock cycle.

The comparison of the performance counter and the total FPGA execution time including interaction in the heterogeneous system shows that there is significant overhead for data transfers to/from FPGA memory and the HW-launch. However, the FPGA is still able to outperform the CPU for all binary examples and most of the larger NIPS-examples (NIPS50, NIPS70, NIPS80), demonstrating the potential of offloading the inference in SPNs to the FPGA, in particular for larger SPNs.

Note that for platforms having true shared memory between the CPU and the FPGA, such as the Xilinx Zynq devices, or the Intel HARP2 systems, these explicit data transfers between CPU and FPGA memory can be completely avoided and the full speed-up (based on the **FPGAC** measurement) realized.

6. Conclusion and Future Work

We have presented the first FPGA-based accelerator architecture for the inference problem in sum-product networks (SPNs), a deep architecture for probability distributions. Our automatic synthesis flow generates a fully-pipelined accelerator from an input description of the SPN and also provides a software interface for interaction with the accelerator in a heterogeneous system. The accelerator architecture features pipelined access to the external memory on the FPGA-board and double-precision floating-point computation. The results of our experimental evaluation demonstrate that the pipelined computation in the FPGA can outperform CPU- and TensorFlow-GPU-implementations, almost processing a complete input sample per cycle for many examples.

There are several interesting avenues for future work. One could extend our synthesis flow and hardware implementation for resource-sharing of operators, in order to be able to map bigger networks. One could also further optimize the arithmetic operators, e.g., by using log-space computations, a very common arithmetic optimization in the ML-domain. Another interesting research avenue is pre-compiling a randomly generated structure and doing weight optimization in the FPGA, with the aim of having a full implementation of a PGM in a chip. If the random structure is large enough it could be retrained on different domains to fit any kind of data. This could also account for domains with concept drifting. Finally, other potential usage scenarios should be explored, such as computing *mutual information*, *maximum a posteriori estimation* and *approximate queries* within databases. These scenarios require fast processing of many input combinations but require less data-transfer from host to FPGA.

Acknowledgements. The authors would like to thank Xilinx Inc. for supporting their work by donations of hard- and software.

References

Abadi, M. et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Alves, J., Ferreira, J., Lobo, J., and Dias, J. Brief survey on computational solutions for bayesian inference. In *Workshop on Unconventional computing for Bayesian inference*, 2015.

Bekker, J., Davis, J., Choi, A., Darwiche, A., and Van den Broeck, G. Tractable learning for complex probability queries. In *Proc. of NIPS*, 2015.

Choi, A. and Darwiche, A. On relaxing determinism in arithmetic circuits. In *Proceedings of ICML*, pp. 825–833, 2017.

Choi, J. and Rutenbar, R. A. 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.

Darwiche, A. A differential approach to inference in bayesian networks. *J. ACM*, 50(3):280–305, 2003.

de Dinechin, F. and Pasca, B. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011.

Dormiani, P., Omoto, D., Adharapurapu, P., and Ercegovac, M. D. A design of online scheme for evaluation of multinomials. In *Advanced Signal Processing Algorithms, Architectures, and Implementations XV*, volume

5910, pp. 59100S. International Society for Optics and Photonics, 2005.

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

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Proceedings of NIPS*, pp. 2672–2680, 2014.

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

Korinth, J., d. l. Chevallerie, D., and Koch, A. An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, May 2015.

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

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

Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., et al. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017*, pp. 5–14. ACM, 2017.

Peharz, R., Tschiatschek, S., Pernkopf, F., and Domingos, P. On theoretical properties of sum-product networks. In *Proc. of AISTATS*, 2015.

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

Pronobis, A., Ranganath, A., and Rao, R. LibSPN: A Library for Learning and Inference with Sum-Product Networks and TensorFlow. In *Principled Approaches to Deep Learning Workshop*, 2017.

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

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

Zhao, H., Melibari, M., and Poupart, P. On the Relationship between Sum-Product Networks and Bayesian Networks. In *Proc. of ICML*, 2015.