High-Throughput Multi-Threaded Sum-Product Network Inference in the Reconfigurable Cloud

Micha Ober, Jaco A. Hofmann, Lukas Sommer, Lukas Weber, Andreas Koch
Embedded Systems and Applications Group, TU Darmstadt, Germany.
{ober, hofmann, sommer, weber, koch}@esa.tu-darmstadt.de

***Abstract*—Large cloud providers have started to make powerful FPGAs available as part of their public cloud offers. One promising application area for this kind of instances is the acceleration of machine learning tasks.**

This work presents an accelerator architecture that uses multiple accelerator cores for the inference in so-called Sum-Product Networks and complements it with a host software interface that overlaps data-transfer and actual computation.

The evaluation shows that, the proposed architecture deployed to Amazon AWS F1 instances is able to outperform a 12-core Xeon processor by a factor of up to 1.9x and a Nvidia Tesla V100 GPU by a factor of up to 6.6x.

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

I. Introduction

FPGAs have received increasing attention as a potential platform for the implementation of application-specific accelerators for datacenter-workload in recent years. As a consequence, large cloud providers have started to make FPGAs available in their public cloud offers, such as Amazon AWS with the F1-instances.

Next to genomics research, financial analysis and high-throughput image and video-processing, FPGAs in the cloud are also used to solve a wide range of machine learning problems. Starting with projects such as Microsoft's Brainwave [1], [2], FPGAs have established themselves as a platform for the acceleration of machine learning tasks, next to GPUs and custom ASICs such as Google's TPU [3].

While much of the existing work on the acceleration of ML tasks on FPGAs has been devoted to neural networks, e.g., for speech recognition or image classification using convolutional neural networks (CNN), a completely different problem is tackled in prior work such as [4], [5], namely the inference in so-called *Sum-Product Networks* (SPN).

Sum-Product Networks are one of the first models from the class of Probabilistic Graphical Models that can provide *tractable* inference, and, in contrast to neural networks, compute *exact* probability values. This difference also poses interesting challenges to the hardware implementation with regard to the arithmetic precision, and many optimization techniques often employed for the mapping of neural networks to FPGA accelerators, such as weight quantization, cannot be applied to Sum-Product Networks [4].

Yet, the evaluation in the prior work has shown some promising results, with a pipelined datapath architecture and memory interface outperforming CPUs and a Tensorflow-based GPU-implementation of SPN-inference [5].

This work aims at extending the existing framework to efficiently map and run the inference in Sum-Product Networks on the publicly available FPGA cloud-offer Amazon AWS F1.

We present the following contributions:

- An extension to the open-source SoC generation framework TaPaSCo [6] to support Amazon AWS F1. TaPaSCo is then used to automatically generate all infrastructure necessary to run the proposed accelerators on F1 and provides convenient software interfaces.
- The existing SPN accelerator generator is extended to allow for concurrent execution of *multiple* accelerators, promising better resource utilization through overlapping processing and memory transfers.
- The software infrastructure is improved to incorporate the parallel execution and preloading of data.

The paper is structured as follows: In Section II, background on Sum-Product Networks is given. Section III shows the existing framework for the acceleration of SPN inference on FPGAs. Section IV describes the accelerators, the implementation of support for Amazon AWS F1 instances in TaPaSCo, and the extensions to the existing framework and software interface. In Section V, three different SoC-architectures are evaluated with regards to their FPGA implementation. Additionally, the FPGA's performance is compared with a CPU- and GPU-based implementation of SPN-inference. Finally, Section VI concludes this paper and gives an outlook to future work.

II. Sum-Product Networks

Sum-Product Networks (SPN) [7] are a type of models from the class of *Probabilistic Graphical Models* (PGM), which have received increasing attention in recent time. PGMs capture statistical information and relations over the variables in a dataset. By using probabilistic queries, PGMs can then be used to solve a wide range of machine learning problems, such as classification or regression. They can also be used to derive statistical properties, such as marginals or conditionals, for a given dataset and input values. For instance, on the NeurIPS corpus, they can be used to compute the probability of different words to occur with a certain frequency in a text.

In contrast to neural networks, which are the currently dominating models in the machine learning (ML) domain, PGMs are capable of computing *exact* probability values. However, earlier approaches to probabilistic graphical models, such as Bayesian Networks or Markov Random Fields, suffer from the fact that the inference in unrestricted PGMs is intractable in general.

But Sum-Product Networks overcome this limitation and combine the ability to compute exact probabilities with *tractable* inference in linear time with respect to the network size [8]. SPNs not only allow to efficiently solve classification or regression problems, but can also be used to calculate additional properties of the underlying probability distribution, such as entropy or mutual information. Examples for the use of SPNs in real-world applications include classification of the characters in a handwritten sequence [9], or path planning algorithms for mobile robots [10]. Because SPNs compute exact probabilities, they are also able to express uncertainty over their output in such applications (e.g. in sequence labeling), a capability not present on neural networks.

A. Model Representation

Sum-Product Networks capture the joint probability distribution over a set of random variables as a rooted, directed acyclic graph (DAG), with three different kinds of nodes:

- Leaf nodes represent univariate distributions. For an efficient mapping to the FPGA, these can be represented using *histograms*.
- Product nodes correspond to a factorization over independent distributions, and are therefore always defined over different scopes (i.e., random variables).
- Weighted sum nodes represent a mixture of multiple distributions over the same scope as a convex combination.

An example for a valid Sum-Product Network, which captures the joint probability $\mathcal{P}(x_1, x_2, x_3)$ for three random variables, is given in Fig. 1.

B. Learning

Similar to many other machine learning models, the structure and parameters of a Sum-Product Network can be learned from training data. As a brief introduction, a short description of the top-down learning algorithm proposed by Molina *et al.* [11] follows. Next to this algorithm, a number of other approaches to learn the structure and/or parameters from data exists.

The algorithm's recursive base case is reached, if only a *single* variable is left. In this case, the algorithm learns a histogram representing the univariate distribution of this variable.

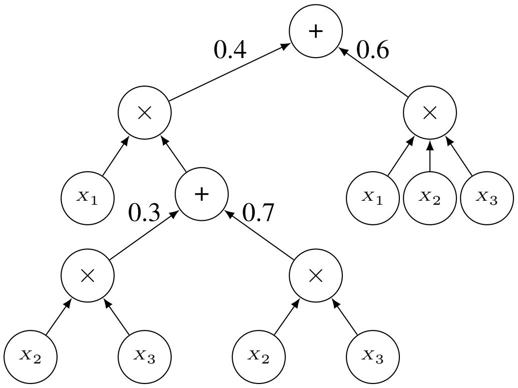


Fig. 1: Example of a valid Sum-Product Network, capturing the joint probability distribution of the variables x_1, x_2 and x_3.

If more than one variable is still left to process, the algorithm tries to find independent sets of variables using a pair-wise, parametric independency test. If the test succeeds, a product node is created and the algorithm recurses on the independent sets. If the independence test fails, the training data is partitioned using clustering. This induces a weighted sum node, where the weights correspond to the normalized, proportional size of the associated clusters.

This work focuses on the *inference* process in a given, previously trained and optimized SPN. Therefore, learning of the SPN structure can happen offline on a traditional CPU.

C. Inference

As stated earlier, SPNs can be used to compute a range of different probabilistic queries to solve different ML problems, such as multi-class classification. However, independent of the actual probabilistic query, the inference process boils down to a bottom-up evaluation of the SPN graph with given (partial) evidence. By indexing the histogram, the probability value associated with the given evidence for the input variable can be determined. These probability values are then propagated upwards through the tree. At product nodes, the probabilities for the different child nodes are multiplied. In case of a sum-node, the probabilities are first multiplied with the associated weight and then summed up. Eventually, at the root node of the SPN, the inference process will yield a single probability value as the answer to our probabilistic query.

This work focuses on the computation of joint probabilities, which corresponds to a single evaluation of the SPN with full evidence per input sample. However, the datapath architecture could easily be extended to support other kinds of probabilistic queries on SPNs, for example to compute marginals, where the histograms for the marginalized variables are replaced by the value 1. With the value for joint probability and the result from a marginalized evaluation, conditional probabilities can easily be computed as $\mathcal{P}(\mathcal{Y}|\mathcal{X}) = \frac{\mathcal{P}(\mathcal{Y},\mathcal{X})}{\mathcal{P}(\mathcal{X})}$.

III. Prior Work

To the best of our knowledge, the work we presented in [4], [5] is the only work to date on the acceleration of SPN inference on FPGAs. The goal of the prior work was the acceleration of the computation of the joint probability over the variables represented by an SPN by its evaluation, and was done by processing small batches of input samples with high throughput.

To this end, an automatic flow that maps textual representations of SPNs to fully custom datapaths is already available as a result of the prior work. The generated datapath for an SPN captures the computation tree represented by the SPN and is completely pipelined by using a fully spatial implementation, where each operation in the SPN tree directly corresponds to a hardware operator in the datapath. For the hardware operator implementation the FloPoCo framework [12] is used, with a format similar to IEEE-754 double precision (apart from subnormals).

The automatically generated datapath is supplied with data through a pipelined memory infrastructure. The memory infrastructure uses AXI4 burst requests to supply the datapath with a continuous stream of input data from the DDR3-memory attached to the FPGA and writes back results to memory. In both the load- and store-unit, a MIMO (Multi-in-Multi-out) unit is used to translate between the external AXI4 datawidth, and the internal input- and result bitwidth of the datapath, respectively.

The open-source TaPaSCo-framework [6] is used to integrate the accelerator core into a heterogeneous system. The automatic SoC composition capability of TaPaSCo can be used to automatically connect the accelerator(s) to the FPGA's external memory and the PCIe-based host interface. Using the TaPaSCo software API, execution of the accelerator(s) can be controlled and data, such as, input & results, can be transferred between host memory and FPGA external memory.

IV. Extension for High-Throughput Inference

This work builds on the automatic mapping flow from [5] to generate pipelined datapaths for SPNs. In the current work, generated datapaths are coupled with a completely new memory infrastructure, designed from the ground up to support the simultaneous execution on *multiple* accelerator cores, and concurrent data-transfers from/to the host. Beyond that, to incorporate the necessary infrastructure, the open-source TaPaSCo framework has been extended to support Amazon AWS F1 instances, and provided with a new multi-threaded SW-interface for simultaneous SPN inference.

Accordingly, the following sections describe the two necessary steps: (1) Extend TaPaSCo with AWS capabilities. (2) Ensure multi-threading compatibility by implementing a new, high-performance memory infrastructure to feed the datapath.

A. Extending the TaPaSCo Framework for the Reconfigurable Cloud

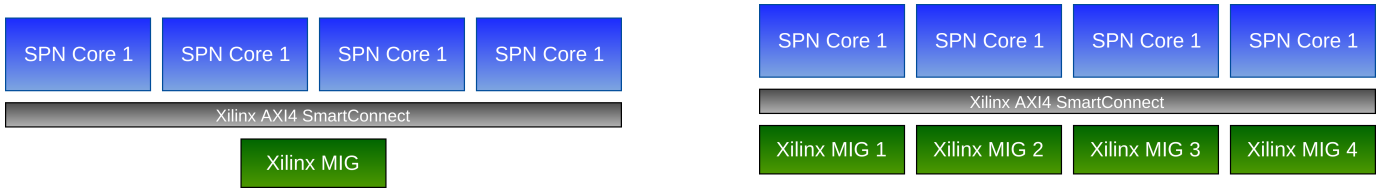
Amazon's first generation FPGA instances actually run in a virtualized environment, with only limited access to the hardware. Amazon uses the partial reconfiguration feature to split the area of the Virtex UltraScale+ FPGA into an user accessible area (called custom logic) and the so-called *Shell*. The Shell is provided by Amazon as an encrypted design checkpoint and acts as an interface between the custom logic and the external peripherals such as the PCI Express interface or memory. A DMA engine for data transfer between host and FPGA is provided by the Shell in the form of the Xilinx XDMA IP core. However, TaPaSCo's own DMA engine is used here, as it achieves better throughput and requires fewer driver changes to be used.

The Shell provides up to 16 MSI based interrupts that can be used by the developer. TaPaSCo, however, requires more than 16 interrupts for optimal performance. A custom interrupt controller is thus used to translate between the needs of TaPaSCo and the interfaces of the AWS Shell. The interrupt controller has a FIFO buffer on each interrupt input and an additional input for ACK signals coming from the Shell. Furthermore, interrupts are ACKed from the host to avoid situations where the interrupt service routine is already active, which would result in dropped interrupts.

Each FPGA on an F1 instance has access to up to four channels of ECC DDR4 memory, each with a size of 16 GiB. One channel is provided by a MIG memory controller inside the Shell, while the MIGs for the other three channels are placed inside the custom logic. The developer can decide how many memory channels are enabled, offering the possibility to trade-off between a larger number of memory channels, and the available area on the device.

Amazon does provide an Amazon Machine Image (AMI) for FPGA development, which includes all required Xilinx tools and licenses. For users, owning the required licenses, on-premise development is supported as well. In both cases, the design flow ends with a design checkpoint containing both the custom logic and the Shell. This design flow differs in many ways from any existing flow in TaPaSCo, as it includes writing out checkpoints for the custom logic, adding the checkpoint of the Shell to the project, linking the custom logic and Shell together, or adding a top-level wrapper provided by Amazon. The Amazon design flow has now been integrated into TaPaSCo, so a final checkpoint is created together with a mandatory manifest file, which are then both sent to Amazon servers. There, in an automated process, the submitted checkpoint is verified to not contain any combinatorial loops or unrouted nets. On success, the bitstream can be referenced by a globally unique identifier, which can in turn be used to load the bitstream into a device.

As Amazon limits the power draw of the FPGA to 85 watts by gating the clock to the custom logic should this limit



(a) Multi-core architecture with single memory interface.

(b) Multi-core architecture with four independent memory interfaces.

Fig. 2: Multi-core SoC architectures with multiple SPN accelerator cores.

be exceeded, clocking was another important topic when integrating the F1 platform into TaPaSCo. The Shell provides up to three different clock groups with different, but not arbitrary frequencies. As the design space exploration of TaPaSCo requires the design frequency to be varied in 5 MHz steps, an additional clock generator (MMCM) is used. The placement of that MMCM is tricky, however, as it cannot be placed near the Shell's MMCM, as this area is not accessible to user logic. This suboptimal placement of the clocking resources put some additional constraints on the design to optimize timing. For example the internal logic of all AXI Interconnects is always clocked by the *Shell's* main clock, even if the *design clock* is faster.

B. Improvements of the Accelerator Architecture

The existing memory infrastructure in the previous work was clearly designed to provide the maximum memory throughput in a scenario with only a single accelerator and where the execution of a job would *not* start before the previous job had completed. As a result, the prior load infrastructure would completely occupy the AXI4-bus by having a high number of transfers with maximum burst length in flight, ensuring that a continuous stream of input data gets supplied to the datapath. On the other side of the accelerator, the data store unit did not buffer many results, but would instead use long-lasting burst requests, occupying the AXI4 write-channel for a long time.

This behavior is problematic, as the accelerator presented in this work executes inference on *multiple* SPN instances *simultaneously*, which requires *overlapping* of data transfer with accelerator execution.

First, the large number of incomplete memory transactions can lead to deadlocks with the simultaneous execution of multiple accelerators: If one core is blocked from writing because another core is occupying the *write* channel, it will not be able to complete potentially incomplete *read* transactions due to back-pressure in the datapath. If the core occupying the write channel is blocked from reading due to the first core's outstanding read request, it won't be able to compute the results necessary to complete the write request, effectively resulting in a deadlock.

The occupation of the write channel is problematic for another reason: TaPaSCo uses a DMA engine to transfer data from the host memory to the FPGA's external memory. However, if the memory interface's write channel is occupied by the accelerator for a long time span, the DMA engine cannot execute the transfer, forcing computation and data transfer to effectively execute sequentially.

To overcome this problem, a completely new memory infrastructure for the accelerator is implemented. In the course of the re-design, the input MIMO unit was moved to the datapath, in order to unify the interface of the accelerator core across different Sum-Product Networks.

The new load interface has fewer incomplete transfers in flight, and now uses an internal buffer that allows to store a substantial amount of input data. A new request is only issued if all data that will be loaded in the course of this request can be buffered internally. In this manner, it can be ensured that even if in the presence of back-pressure from the store interface, all incomplete load requests can be completed to avoid deadlocks. Additionally, this buffer allows the accelerator to continue its computation for some time, even if the current load request incurs some delay.

The store interface is also redesigned to be able to buffer results for a complete burst request internally. Only after enough results for a complete burst request have been calculated, a new write request is issued. Because all write data is available, the write request can be completed in the shortest time possible, not only avoiding deadlock, but also allowing the TaPaSCo DMA engine to transfer data in parallel to the computation in the accelerator cores.

C. SoC Architecture

The TaPaSCo-framework, with the extensions described in Section IV-A, is able to automatically construct and generate SoC-designs for the F1 FPGA instances provided by Amazon AWS. The AXI4 slave interface of the accelerator core, which is used for control signalling (e.g., launch execution) is attached to the PCIe-based host interface. The AXI4 master interfaces of each core are automatically connected to the memory interface of the FPGA's external DDR-memory via a shared bus. As described in the previous section, the SPN accelerator core uses a single AXI4 master interface to read and write data and results from/to external memory.

The obligatory Shell environment provided by Amazon as part of the AWS F1 HDK by default contains a single interface to FPGA external memory. For the baseline architecture a single SPN accelerator core is connected to this interface via AXI4, allowing the accelerator core to read and write data from/to the external memory. As described in the previous section, the load/store interface of the accelerator is re-designed in a way that allows to operate the DMA-unit used by TaPaSCo to transfer data from host memory to FPGA memory concurrently to the accelerator.

Although this baseline architecture is already a fully functional accelerator for SPN inference, further improvements to the throughput can be made: The vast amount of hardware resources available on the Xilinx Virtex UltraScale+ devices, with which the AWS F1 instances are equipped, allows multiple instances of the SPN accelerator cores, which can compute multiple inferences simultaneously. In the multi-core architecture, depicted in Fig. 2a, up to four SPN accelerator are connected to the memory interface via a shared bus. Due to the changes made to the memory infrastructure (see Section IV-B), the different cores can now operate concurrently on the same bus.

However, depending on the number of input values of the SPN and the memory bandwidth requirement that goes along with that (remember that the datapath architecture is fully pipelined and can process a new sample in every clock cycle), the single interface to memory can become a bottleneck. To overcome this limitation, and fully exploit the memory bandwidth provided by the four independent memory banks on the Virtex UltraScale+ device, and additional three *more* memory interfaces can be used. Just as in the multi-core architecture, up to four SPN accelerator cores are connected to the four memory interfaces via an AXI4 SmartConnect, depicted in Fig. 2b. Note that as long as the four accelerator cores access distinct memory regions located on the different memory banks, they can be served *simultaneously* through the SmartConnect, so this bus infrastructure will not become a bottleneck.

D. Multi-threaded Software-Interface

The existing host software uses the TaPaSCo software API to launch the entire computation in a single, large job. However, to fully exploit the available throughput, the software interface needs to reflect the multi-core architecture in the FPGA-hardware and launch multiple, independent jobs that execute concurrently on the different SPN accelerator cores.

Although it would be sufficient to launch four independent jobs in different threads, there are more improvements to be had: In the prior work, the overhead for transferring data between host- and FPGA-memory turned out to be relatively large and contributed significantly to the overall execution time.

Therefore, a goal is to extend the software interface to allow overlapping of computation on the SPN accelerator cores and data transfer to reduce the overall execution time of SPN inference. The computation is split into independent blocks (chunks), which can be independently preloaded. Fig. 3 shows how computation and data transfer are overlapped with this change: While block n is computed on the FPGA SPN accelerator core 1, the input data for block $n+1$ is concurrently transferred to the FPGA memory by another thread. As soon as the computation of block n is completed, the computation for block $n+1$ can start, while the results for block n are copied back to host memory. At the same time, the same execution scheme can be used for another block of samples on another SPN accelerator core.

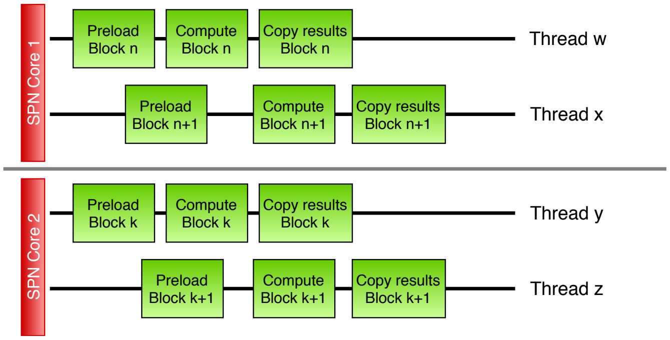


Fig. 3: Block-wise execution overlapping actual computation and data-transfer between host- and FPGA-memory.

This execution scheme is implemented based on the TaPaSCo software API using OpenMP on the host, the block size and number of threads are configurable parameters for the host software. As described in the previous section, it is important that the different SPN accelerator cores operate on *distinct* memory regions located on different memory banks. To ensure this behaviour, each host thread has fixed device memory addresses to evenly distribute the workload across the different SPN accelerator cores in the multi-core architectures (cf. Fig. 2).

V. Evaluation

A. Benchmarks

The approach is evaluated through a set of eight different benchmark SPN models derived from the NeurIPS corpus [13], which has also been used before for evaluation purposes in [4], [5].

The SPN networks derived from the NeurIPS corpus capture statistical properties about the number of occurrences (frequency) of different words in the texts contained in the corpus. Using inference in this networks, it is for example possible to compute the probability that certain words each occur with a specific frequency in a text.

The increasing number of inputs of the networks in the benchmark make them particularly interesting for the evaluation: Not only does the required memory bandwidth increase with the number of inputs, but also the size of the SPN networks

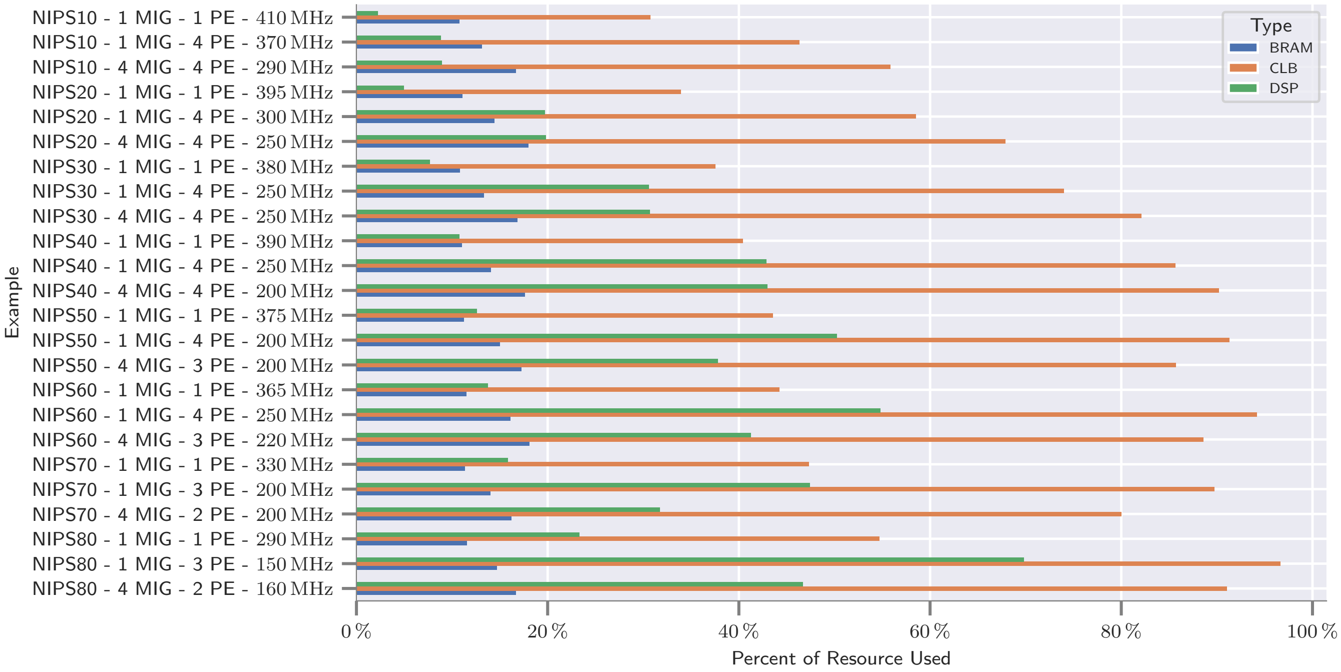


Fig. 4: Utilization of the main FPGA resources on the AWS F1 instances. The designs are mainly limited by available CLBs. DSPs and BRAM play a minor role in the overall resource requirements. A high CLB utilization results in difficult routing and a tendency for lower clock frequencies.

themselves increases, leading to more computational demand and, with the fully spatial implementation of the datapaths, also more FPGA resource consumption.

All the following performance evaluations have been performed using a dataset containing ten million samples per benchmark.

B. FPGA Implementation

The FPGA resource usage and implementation results are evaluated by implementing each of the three different SoC-architectures described in Section IV-C for each of the benchmarks on the xcvu9pflgb2104-2 FPGA, with which the Amazon AWS F1 instances are equipped. Xilinx Vivado version 2018.3 and TaPaSCo version 2019.10 (pre-release), extended as described in Section IV-A, are used. All reported numbers are taken from the post-place&route reports generated by Vivado. The resource usages for each SoC-architecture and each benchmark are given in Fig. 4. For brevity, the numbers are given relative to the number of available resources. For reference, the VU9P FPGA has in total 148×10^3 CLB, 2×10^3 BRAM and 7×10^3 DSP.

Even with a single memory interface and only one SPN accelerator core, the design already requires a significant amount of logic resources (CLB), partially also due to the obligatory AWS Shell and basic platform components (e.g., PCIe interface). However, there is still sufficient headroom left to implement multiple SPN accelerator cores. The operating frequencies for this configuration reach up to 410 MHz.

The resource usage increases noticeably for the first multi-core architecture, which uses up to four SPN accelerator cores coupled with a single memory interface. For the two largest networks, NIPS70 and NIPS80, only three accelerator cores fit onto the FPGA device. The DSP usage correlates linearly to the number of SPN accelerator cores, but for the other two kinds of resources, the relative increase is much smaller: Logic resources (CLB) required increase just by a factor 1.5x to 2.13x, and only between 1.2x and 1.4x more BRAMs are used. The higher resource usage also results in a noticeable degradation of the operating frequency, with this configuration the highest achievable frequency is 370 MHz.

When using four memory interfaces in the multi-core architecture, the available resources limit the number of SPN accelerators that can be implemented for some examples: For NIPS50 and NIPS60 only three cores fit the device and for NIPS70 and NIPS80 only two cores can be placed. Regarding the resource usage, the increase compared to the baseline architecture with one core and one memory interface is comparable to the first multi-core architecture. The number of DSPs used scales linearly with the number of SPN accelerator cores, CLBs increase by factor 1.6x to 2.2x and BRAM usage increases by 1.43x to 1.6x. The direct comparison of the two multi-core architectures shows that the additional memory interfaces only consume a relatively small number of resources.

However, with four memory interfaces, three of them are implemented in the custom logic. Here, timing closure can become problematic. The achievable frequency drops by 30%-50% across all benchmarks, partly because of the high resource usage (more than 80% of logic resources for most examples) and the IO-restricted locations for the memory interfaces.

C. Performance Evaluation

In this section, the performance of the three different architectures is compared to CPU- and GPU-based implementations. Furthermore, the FPGA-implementations have different choices for block-size and number of threads to determine the optimal configuration (cf. Section IV-D). For brevity detailed charts have been omitted. For most benchmarks, a block-size of 400,000 samples per block, and three to four host software threads per SPN accelerator result in the best performance.

1) CPU Baseline: For the CPU baseline, a custom C++ compilation flow is used, sharing some of the infrastructure (intermediate representation, parser) of the datapath generator. The flow automatically generates optimized C++ code for each example and invokes the compiler (gcc) with flags `-O3` and `-ffast-math`. To efficiently parallelize the workloads, the data-set is split into blocks, which can be processed concurrently using OpenMP with twelve threads on a 12-core Xeon E5-2680 v3 CPU of the Lichtenberg high-performance computing cluster.

2) GPU Baseline: The evaluation in prior work [5] showed that Tensorflow, tailored towards standard neural networks, is not able to efficiently map the inference in SPNs to GPUs. Therefore, for a fair comparison, a custom, optimized CUDA compilation flow, again based on the common infrastructure we developed, is used. Within each block, the processing of samples is parallelized across the available GPU processing units. Eight threads on the host CPU are utilized to parallelize processing of blocks. The evaluation is done on the top-end Nvidia Tesla V100 GPUs available in the Amazon AWS cloud with CUDA version 9. The evaluation shows that our custom SPN-to-CUDA compilation flow improves the throughput by a factor of up to 109x (geo.-mean 96x) over the original Tensorflow-based mapping.

3) Performance Comparison: Fig. 5 shows the throughput of the CPU- and GPU-baseline and the three different SoC-architectures.

Even though the performance of the GPU-baseline improves up to factor of 109x over the old Tensorflow-based baseline, the GPU is still left behind and provides the least throughput. The computational density of the SPN inference in the examples is not sufficient to compensate for the data-transfer overheads and fully exploit the GPU's computational power. This can also be seen in our observation that the GPU performance is almost independent of the SPN network size. In contrast, the CPU performance *is* highly dependent on the network size, and decreases as the number of operations in the SPN's tree increases. Still, for small examples, the CPU provides the best throughput of all architectures, since it does not incur data transfer overheads.

The baseline FPGA-architecture with a single SPN accelerator core and memory interface is outperformed by the CPU for the examples up to NIPS30, but for *larger* networks, the pipelined processing yields significant speedups of up to factor 1.6x (geo.-mean 1.01x) over the CPU, and 4.6x over the GPU (geo.-mean 4.3x).

Despite using up to four cores with only a single memory interface, the first multi-core architecture is also able to outperform the CPU by a factor of up to 1.47x (geo.-mean 0.99x) for larger examples and the GPU by a factor of up to 5.15x (geo.-mean 4.23x) on all examples. The overall performance is slightly lower compared to the single-core architecture due to the operating frequency drop (cf. Section V-B) and the saturation of the memory interface. Therefore, the DMA transfers of data from/to host and computation effectively happen sequentially, leading to a degradation in performance, in particular for the larger examples.

With up to four SPN cores and four distinct memory interfaces in the second multi-core architecture, the FPGA is already on par with the CPU for NIPS20 and even more significant speedups can be observed for all bigger networks. The speedup reaches as high as 1.9x over CPU (geo.-mean 1.28x) and 6.6x over GPU (geo.-mean 5.47x).

When directly comparing the baseline FPGA-architecture with the second multi-core architecture with up to four SPN cores and four distinct memory interfaces, it can be seen that the performance advantage *decreases* for the larger networks. Whereas the relative speedup of the multi-core architecture is 1.7x for NIPS10, it decreases to only 1.15x for NIPS80. This is due to the fact that four SPN cores do not fit on the device for the larger examples (cf. Section V-B), and the high penalty on the lowered operating frequency incurred when using four distinct memory interfaces. Note that this is most likely an artifact specific to the AWS F1 system architecture. Still, this multi-core architecture is the best performing FPGA accelerator architecture.

VI. Conclusion & Outlook

This work presents an accelerator architecture, based on a framework developed in prior work, for the efficient inference of so-called Sum-Product Networks on FPGA instances in the cloud based Amazon AWS F1 instances.

The open-source TaPaSCo-framework has been extended to support Amazon AWS F1 instances, enabling automatic SoC generation based on the proposed accelerator including integration into a heterogeneous system.

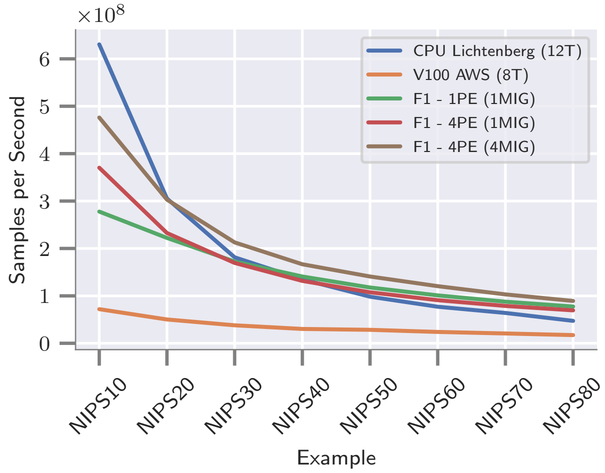


Fig. 5: Samples processed per second by the CPU, GPU and the FPGA architecture for increasing problem sizes. For very small problem sizes, the CPU is faster as the transfer overhead to PCIE-based accelerators is dominant. However, for larger problem sizes, the FPGA pulls ahead. The GPU cannot keep up with either of the other two competing approaches.

To make use of the resources available on the FPGA-instances, three different architectures have been investigated that allow concurrent processing on multiple accelerator cores. This required a re-design of the accelerator memory interface compared to prior work. Lastly, the accelerator architecture is complemented with a multi-threaded host software interface that is able to efficiently overlap data-transfer and actual computation in order to improve end-to-end execution times.

The evaluation shows that the developed architectures are able to outperform a 12-core Xeon CPU and a Tesla V100 GPU by up to a factor of 1.9x and 6.6x, respectively.

In the future, we plan to extend the mapping toolflow with the ability to share operators between multiple operations in the SPN graph, allowing us to map even bigger networks to FPGAs.

The extension to the TaPaSCo-framework developed in the course of this work will also become an official part of the TaPaSCo open-source release on Github [14]. Just as any other platform in the TaPaSCo-framework, the AWS support will allow to automatically compose a SoC-design around any accelerator core with a suitable AXI4-interface and connect to it from the host CPU via the TaPaSCo software API.

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, as well as extensive insight into the matter of Sum-Product Networks.

References

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

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

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

[4] 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.

[5] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch, "Automatic mapping of the sum-product network inference problem to fpga-based accelerators," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct. 2018, pp. 350–357. DOI: 10.1109/ICCD.2018.00060.

[6] 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 Reconfigurable Computing*, 2019.

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

[8] R. Peharz, S. Tschiatschek, F. Pernkopf, and P. Domingos, "On theoretical properties of sum-product networks," in *Proc. of AISTATS*, 2015.

[9] M. Ratajczak, S. Tschiatschek, and F. Pernkopf, "Sum-product networks for sequence labeling," *CoRR*, vol. abs/1807.02324, 2018. arXiv: 1807.02324.

[10] 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.

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

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

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

[14] *TaPaSCo framework on Github*, https://github.com/esa-tu-darmstadt/tapasco, 2019.