# SPNC: Accelerating Sum-Product Network Inference on CPUs and GPUs

Lukas Sommer*, Michael Halkenhäuser*, Cristian Axenie†, Andreas Koch*
*Embedded Systems and Applications Group, TU Darmstadt, Germany
†Intelligent Cloud Technologies Laboratory, Huawei Munich Research Center, Munich, Germany
*{sommer, halkenhaeuser, koch}@esa.tu-darmstadt.de, †cristian.axenie@huawei.com

***Abstract*—Probabilistic models are receiving increasing attention as a complementary alternative to more widespread machine learning approaches, such as neural networks. One particularly interesting class of models are so-called *Sum-Product Networks* (SPN), which combine the expressiveness of probabilistic models with tractable inference, making them an interesting candidate for use in real-world applications.**

**Yet, as Sum-Product Networks are a young class of machine learning models, the software ecosystem is comparably sparse. In this work, we enhance the ecosystem with a domain-specific compiler that allows to easily and efficiently target CPUs and GPUs for Sum-Product Network inference.**

**Using a real-world application of Sum-Product Networks, a robust speaker identification model, we showcase the performance improvements our compiler can achieve for SPN inference on CPUs and GPUs.**

***Index Terms*—Sum-Product Networks, Machine Learning, MLIR, LLVM, CPU, GPU**

## I. Introduction

Similar to other probabilistic models, Sum-Product Networks (SPN) are receiving increasing attention for their ability to handle the *uncertainty* found in real-world scenarios better. Yet, in contrast to more popular machine learning models such as neural networks (NN), the software ecosystem for Sum-Product Networks is comparably sparse, hindering their deployment in actual applications.

Research libraries such as SPFlow by Molina et al. [1] allow to train and construct Sum-Product Networks from data, but their inference implementation in pure Python does not leverage all hardware features of the target platform, e.g., CPU vector extensions.

To address this problem, in this work, we enhance the SPN ecosystem by developing a domain-specific compiler for performing fast Sum-Product Network *inference* on CPUs and GPUs. To this end, we develop a complete compilation flow (Section III) based on the open-source MLIR [2] and LLVM [3] frameworks. The flow is currently able to compile for two main targets, namely x86 CPUs with vector extensions and CUDA GPUs. On both platforms, the mapping strategies implemented in the compilation pipeline make sure to employ the target hardware's features for fast inference.

To facilitate the integration of the compiler into the machine learning experts' workflows, we develop an easy-to-use Python interface that seamlessly interacts with the open-source SPFlow [1] library for SPN training and modelling.

In the evaluation in Section IV, using a real-world application of SPNs for robust automatic speaker identification, we demonstrate how the compilation can accelerate inference by up to a factor of 814x.

Furthermore, we provide necessary background information on Sum-Product Networks in the next section, and discuss related works in Section V.

## II. Sum-Product Networks

SPNs [4] are a relatively young class of probabilistic models. Similar to other probabilistic graphical models (PGM), SPNs are able to efficiently handle real-world uncertainties, such as missing feature values. In contrast to most neural network architectures, SPNs are also able to quantify uncertainty over the output. An overview of modelling approaches, learning algorithms and practical usage examples of SPNs can be found in the survey by Paris et al. [5].

Sum-Product Networks capture the joint probability of a set of variables (i.e., features) in the form of a directed acyclic graph (DAG). Regardless of the application and the underlying data, the DAG is always composed from three different types of nodes. At the bottom of the DAG, so-called *leaf nodes* capture the univariate probability distribution of a single variable/feature.

Further up in the graph, a combination of product nodes and weighted sum nodes is used to capture the joint probability distribution. Product nodes represent factorizations of independent variables, weighted sum nodes, on the other hand, represent a mixture of distributions. The structure of the SPN depends on the distribution of the underlying data, and can either be learned from data, or be hand-crafted, followed by just parameter learning. A small example of an SPN graph is shown in Fig. 1.

After a valid SPN graph has been obtained, the SPN can be used to solve machine learning tasks, such as classification, by performing inference on the graph. To this end, the SPN DAG is traversed bottom-up (starting at the leaf nodes). The univariate leaf nodes are queried using the partial or full evidence specified as input. After that, the probability values are propagated upwards, until eventually a single probability value is obtained at the root node.

The compiler developed in this work aims to accelerate the *inference* in Sum-Product Networks by efficiently mapping them to different hardware targets. Learning of the SPN is

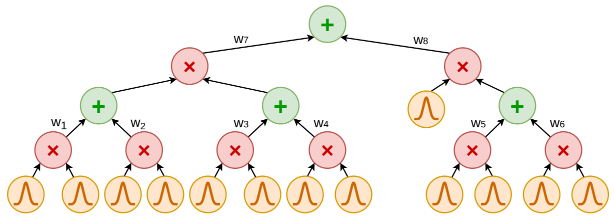

Fig. 1. Example of a Sum-Product Network graph.

assumed to have taken place beforehand, using a standard Sum-Product Network framework such as SPFlow [1].

## III. Compilation Flow

The general approach in this work for generating code for the fast inference in Sum-Product Networks is to compile the inference on a given SPN down to target-specific code with a compilation flow based on the open-source MLIR framework[1]. A detailed description and background information of MLIR and its design can be found in [2].

Independent of the compilation target, the same interface (Section III-A) and target-agnostic part of the MLIR pipeline (Section III-B) are used. After that, the remaining, target-specific MLIR pipeline and compilation steps are used to produce an executable for the respective target (Section III-C & and Section III-D), which can then be loaded by the runtime component (Section III-E).

### A. Python Interface

To seamlessly integrate with the SPFlow library [1], the compiler offers a Python-based interface. The Python interface is implemented using Pybind11[2]. For efficient exchange of SPN models between the Python interface and the compiler, implemented in C++, a binary serialization based on the open-source Cap'n Proto[3] library was implemented.

The Python interface is also used for the runtime component (Section III-E). As Pybind11 has full support for numpy arrays, input data for execution can simply be provided as numpy arrays, and the result data will likewise be returned as a numpy array.

### B. Target-Agnostic Pipeline

The target-agnostic part of the MLIR pipeline uses two SPN-specific dialects. The **HiSPN** dialect's design closely resembles the representation used internally by SPFlow and captures an SPN model on a high-level of abstraction. The binary serialized SPN model is directly de-serialized to the HiSPN dialect.

After that, the HiSPN module is lowered to the second SPN-specific dialect called **LoSPN**. This dialect represents the computation necessary for computing an SPN inference query on the given DAG as a *Kernel* comprising one or multiple *Tasks*, which in turn contain the SPN operations. The LoSPN module is the starting point for the target-specific flows.

[1]https://mlir.llvm.org
[2]https://github.com/pybind/pybind11
[3]https://capnproto.org/

### C. CPU Compilation Flow

For CPU compilation, the LoSPN Tasks and Kernel are lowered to functions, which contain code to process the query for a single or a batch of input samples. To this end, a combination of multiple dialects, which are part of the MLIR framework, is used. If requested by the user, the CPU compilation flow will also perform vectorization. Here, the use of the MLIR framework and its ability to capture an application's high-level structure allow the compiler to reliably perform vectorization and optimizations. Afterwards, the MLIR module is translated to LLVM IR and the LLVM backends are used to generate an executable, which, if requested, is linked with vector libraries such as Intel SVML or Libmvec for efficient implementations of elementary functions (e.g., `log`). The use of LLVM allows the compiler to target any CPU for which a LLVM backend is available, vectorization support is currently limited to x86 (AVX, AVX2).

### D. GPU Compilation Flow

When targeting GPUs, the LoSPN Tasks are lowered into GPU device functions, while the Kernel is lowered to host code that controls GPU device execution and memory transfers between host and device. Similar to the CPU flow, a combination of multiple MLIR dialects is used to represent the computation and SIMT execution model on the GPU. Later, the GPU and host portions of the code are separated. While the host portion undergoes a similar process as the CPU flow, eventually yielding a executable that, at runtime, will load and execute the GPU code, the GPU portion is first translated to NVVM IR. Using LLVM's PTXAS backend, this NVVM IR is translated to PTX assembly and, after that, to a CUDA GPU binary (CUBIN format) through Nvidia's CUDA tools.

### E. Runtime Component

As described in Section III-A, triggering the actual inference, using the compiled kernel, is also possible through the Python interface. For seamless integration with the Python workflow, the interface supports numpy arrays. The runtime component is then responsible for loading the compiled kernel and managing execution.

When targeting CPUs, the runtime component will also break down the set of inputs into multiple subsets and manage concurrent execution on independent subsets using OpenMP.

The compiled kernels are also cached in the runtime to accelerate repeated invocations of inference.

## IV. Evaluation

Our goal is to evaluate our compiler with a *real-world* application of Sum-Product Networks, therefore we are using the SPNs from [6] for our evaluation. In this work, Sum-Product Networks are used to perform automatic speaker identification, outperforming two CNN-based approaches for speaker identification with regard to robustness.

For our evaluation, we are going to use two different scenarios from the paper by Nicolson et al. [6], namely the identification on clean speech samples (245567 samples) and

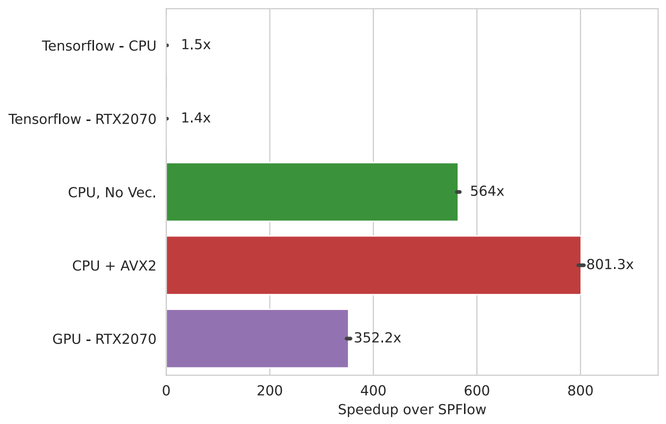

Fig. 2. Performance comparison for clean speech samples, given as speedup over execution in SPFlow.

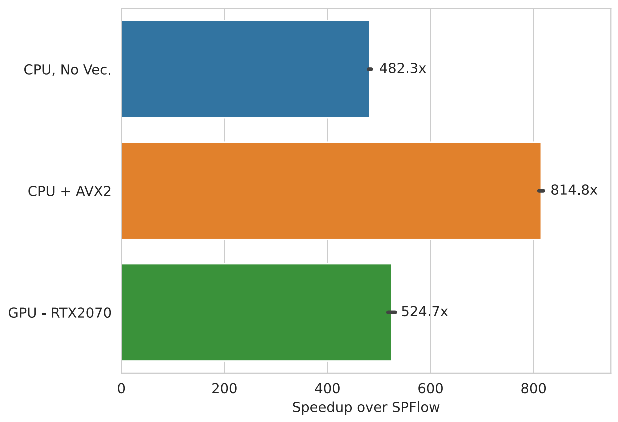

Fig. 3. Performance comparison for noisy speech samples, given as speedup over execution in SPFlow.

identification on noisy speech samples with marginalization (1227835 samples). In both cases, 628 SPN models are evaluated, one for each speaker, which have been reproduced using the open-source release of the speaker identification by Nicolson et al.[4]. A sample comprises 26 features, each encoded as single-precision floating point value. We use computation in log-space to avoid deviation from the original result, using single-precision floats as the underlying data type.

All experiments are conducted on a machine with an AMD Ryzen 9 3900XT CPU equipped with 32 GB RAM and an Nvidia RTX 2070 Super GPU with 8 GB RAM, running Ubuntu 20.04 with kernel version 5.8, CUDA 11.2 and the CUDA driver version 460. We use GLIBC Libmvec version 2.31 as vector library.

In all experiments using our compiler, we measure the execution time from Python, i.e., the execution time always also includes the invocation overhead of the Python interface in addition to the actual execution time. We track compilation time and execution time separately (also for Tensorflow). The average compilation time for CPU is 3.3s (max. 18s) and for GPU 1.7s (max. 4.1s). The translation of the SPFlow graph to a Tensorflow graph takes 8.6s on average (max. 14.5s).

## A. *Performance Comparison*

For performance evaluation, we will compare the execution time of the compiled kernel against SPFlow's performance when executing in Python and when translating and executing a Tensorflow graph (on both CPU and GPU).

For the CPU compilation flow, the comparison includes the configuration not using vectorization on CPU, and compiling for CPU using vectorization and a vector library for optimized implementations of elementary functions (e.g., `log`) on AVX2. For the GPU, a batch size of 64 is used, determined through a simple grid-search. Fig. 2 shows the performance comparison for the clean speech samples, the plot gives the speedup over the Python execution in SPFlow. The speedup achieved by translating the SPFlow graph to a Tensorflow graph is relatively low on both CPU (geo.-mean 1.5x) and GPU (geo.-mean 1.38x), as the graph is still broken down into individual operations that are launched through the Tensorflow runtime.

In contrast, SPNC achieves an average speedup of 564x, even without vectorization. While the speedup does not increase linearly with the vector size, because the initial loading of values into vector registers requires significant effort, vectorization still increases the speedup to 801x with AVX2. Compilation for the GPU achieves an average speedup of 352x.

Fig. 3 shows the same comparison for the noisy speech samples, using marginalized inference. Unfortunately, the translation to Tensorflow graphs, which is currently part of SPFlow, does not support the marginalization necessary for the noisy speech samples, so no bars for Tensorflow can be included in this plot. When compiling for CPU, SPNC again achieves large speedups, with an average of 482x without vectorization and 814x with vectorization for AVX2. In this comparison, the GPU executable outperforms the CPU executable without vectorization with a mean speedup of 524x, as more samples are available for simultaneous processing.

The reason why the executable for the GPU drops behind the executable with vectorization in both comparisons is data movements between host and device, which in both cases make up for more than 60% of the execution time. So even though the execution on the GPU itself is very fast, the data movement overhead, which is not present when compiling for CPU, leads to a higher overall execution time.

Despite that, use of SPNC for CPU and GPU provides speedups of multiple orders of magnitude in comparison to SPFlow's Python-only evaluation, benefiting ML experts when running inference on Sum-Product Networks.

## V. Related Work

To the best of our knowledge, the compiler presented in this work is the *first* compiler for Sum-Product Networks, enabling efficient inference on multiple hardware platforms.

[4]https://github.com/anicolson/SPN-ASI

For creation, training, inference, and experimentation with Sum-Product Networks, a number of libraries have been proposed over the years. The two most popular ones, according to the survey conducted by Paris et al. [5], are SPFlow[5] [1] and libspn[6] [7].

SPFlow allows users to either programmatically create an SPN or learn it, including its structure, from data. It also supports inference on the obtained SPN, either in pure Python, or, for a limited number of cases, through a translation to a Tensorflow graph and execution of that graph. As our evaluation in Section IV has shown, our compiler significantly outperforms both variants. Because SPFlow is so popular among SPN researchers, we have decided to integrate our compiler with it, allowing users to feed SPNs learned with SPFlow directly to our compiler, and perform inference with the compiled kernel, using our Python interface for both steps (cf. Section III-A).

Libspn also allows to perform parameter learning and inference for SPNs, again through translation to a Tensorflow graph, which has yielded suboptimal performance in our evaluation in Section IV.

Another interesting approach to efficient training and inference for SPNs is through tensorization of the SPN graph, as shown in [8] or [9]. However, these implementations are limited to weight learning, with the structure of the SPNs being subject to additional constraints, whereas our compiler can process SPNs with *arbitrary* DAG structure.

As custom hardware architectures for inference in machine learning models such as neural networks have already demonstrated great potential, we have developed a custom, FPGA-based inference accelerator for Sum-Product Networks in prior work [10], [11]. However, as the automatically generated accelerator uses a fully spatial hardware layout, the maximum size of SPNs that can be mapped to the FPGA is limited by the available hardware resources to sizes significantly smaller than the SPNs evaluated in this work, and the flow does not support Gaussian distributions.

## VI. Conclusion & Outlook

In this work, we have presented SPNC, a domain-specific compiler for fast inference in Sum-Product Networks. The implementation of SPNC is based on the open-source MLIR framework, which, by providing common infrastructure for the design of intermediate representations and suitable abstractions for different hardware platforms, facilitates the implementation of domain-specific compilers.

SPNC was designed as a valuable addition to the Sum-Product Network ecosystem, and to provide fast inference to ML experts and researchers working with Sum-Product Network models. To this end, SPNC can seamlessly integrate with SPFlow, a popular open-source library for SPN construction, learning, and representation, through its Python interface.

In our evaluation, using an SPN-based robust automatic speaker identification as an example of a real-world application of Sum-Product Networks for an ML task, the comparison with the currently available inference mechanisms in SPFlow showed that SPNC can achieve a speedup over SPFlow of a factor of up to 814x when compiling for CPU with AVX-2 vector extensions, and up to a factor of 524x when compiling for CUDA GPUs.

In the future, we plan to support even more hardware platforms, such as ARM SVE and AMD GPUs through SPNC's MLIR-based compile flow.

## Availability

SPNC is available as open-source software under the Apache v2 License on Github[7]. In the releases section on Github, pre-built packages for Linux systems can be found for download and installation via Python `pip`.

## Acknowledgements

Calculations for this research were conducted on the Lichtenberg high performance computer of TU Darmstadt. This research was partially funded by the German Federal Ministry for Education and Research (BMBF) with the funding ID ZN 01|S17050.

## References

[1] A. Molina, A. Vergari, K. Stelzner, R. Peharz, P. Subramani, N. D. Mauro, P. Poupart, and K. Kersting, "Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks," 2019.

[2] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. A. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *CGO 2021*, 2021.

[3] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[4] H. Poon and P. Domingos, "Sum-product networks: A new deep architecture," in *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, 2011.

[5] I. Paris, R. Sanchez-Cauce, and F. J. Diez, "Sum-product networks: A survey," 2020.

[6] A. Nicolson and K. K. Paliwal, "Sum-product networks for robust automatic speaker identification," 2020.

[7] A. Pronobis, A. Ranganath, and R. P. Rao, "Libspn: A library for learning and inference with sum-product networks and tensorflow," in *Principled Approaches to Deep Learning Workshop*, 2017.

[8] R. Peharz, A. Vergari, K. Stelzner, A. Molina, X. Shao, M. Trapp, K. Kersting, and Z. Ghahramani, "Random sum-product networks: A simple but effective approach to probabilistic deep learning," in *Proceedings of UAI*, 2019.

[9] J. van de Wolfshaar and A. Pronobis, "Deep Generalized Convolutional Sum-Product Networks for Probabilistic Image Representations," *arXiv:1902.06155 [cs, stat]*, Sep. 2019.

[10] 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 *IEEE International Conference on Computer Design (ICCD)*. IEEE, 2018.

[11] L. Sommer, L. Weber, M. Kumm, and A. Koch, "Comparison of arithmetic number formats for inference in sum-product networks on fpgas," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 1–10.

[5]https://spflow.github.io/SPFlow/

[6]https://www.libspn.org/

[7]https://github.com/esa-tu-darmstadt/spn-compiler