SPNC: Fast Sum-Product Network Inference

Lukas Sommer[1] [0000−0003−1918−3911], Cristian Axenie[2], and Andreas Koch[1] [0000−0002−1164−3082]

[1] Embedded Systems and Applications Group[⋆]
Technical University Darmstadt, Germany
{sommer, koch}@esa.tu-darmstadt.de
[2] Intelligent Cloud Technologies Laboratory
Huawei Munich Research Center, Germany
cristian.axenie@huawei.com

Abstract. Sum-Product Networks have received increasing attention from academia and industry alike, but the software ecosystem is comparably sparse. In this work, we enhance the ecosystem with an open-source, domain-specific compiler that allows to easily and efficiently target CPUs and GPUs for Sum-Product Network inference. The implementation of the compiler is based on the open-source MLIR framework. 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.

Keywords: Sum-Product Networks · Compiler · MLIR · LLVM

1 Introduction

Probabilistic models are receiving increasing attention from both academia and industry, being a complementary alternative to more widespread machine learning approaches such as (deep) neural networks (NN). Probabilistic models can handle the *uncertainty* found in real-world scenarios better, and are also, in contrast to NNs, able to *express uncertainty* over their output.

However, in contrast to neural networks, for which a rich ecosystem with a variety of frameworks, libraries and compilers, such as Tensorflow's XLA, or Facebook's Glow, is available, the ecosystem for probabilistic models such as Sum-Product Networks (SPN) is comparatively sparse, due to them being a relatively young class of models.

One of the most popular libraries for research with Sum-Product Networks is SPFlow by Molina et al. [3], which provides a programmatic representation of Sum-Product Network models and allows to learn their structure and parameters from data. SPFlow also allows to perform inference on the models obtained through learning, but is implemented in pure Python and can therefore not

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

leverage the full feature set of CPUs or GPUs. However, exploiting all available hardware features is crucial for the deployment of SPN models on embedded-grade devices and for efficient inference in real-world applications, e.g., to fulfill real-time requirements.

Therefore, in this work, we enhance the SPN ecosystem by developing *SPNC*, an *open-source* domain-specific, multi-platform compiler for performing fast Sum-Product Network inference on CPUs and GPUs, and present the following contributions:

– Based on the MLIR framework [2], we develop custom high-level intermediate representations capturing the semantics of Sum-Product Networks in the compiler (Section 3.1).
– We define efficient strategies to map Sum-Product Network inference to CPUs with vector extensions and CUDA GPUs. The mapping strategies make use of the underlying hardware's specific features for efficient inference (Section 3.2 & Section 3.3).
– Using a real-world application of Sum-Product Networks, we evaluate our approach in detail, and compare it to the currently available framework (Section 5).
– We develop a Python interface to our compiler, which seamlessly integrates with SPFlow [3] and allows to target CPUs and GPUs with ease (Section 4).

Furthermore, we provide necessary background information on Sum-Product Networks and the open-source MLIR framework in the next section, and discuss related works in Section 6.

2 Background

2.1 Sum-Product Networks

SPNs [7] are a relatively young class of probabilistic graphical models (PGM). In principle, such class of models can be considered a unified approach combining Bayesian network representation formalism and Markov random field computation. Such a computational configuration enables SPNs to efficiently reason under incompleteness and uncertainty, which is a challenging task in many real-world scenarios [11]. Unfortunately, inference requires intense computation that introduces a long delay. This is a core motivation of our work.

Additionally, in contrast to most neural network architectures, SPNs are also able to quantify uncertainty over the output. An example for this property can be found in [6], where SPNs, when confronted with out-of-domain images, indicate this through a low likelihood for the output class, in contrast to the multi-layer perceptron undergoing the same test. An overview of other 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

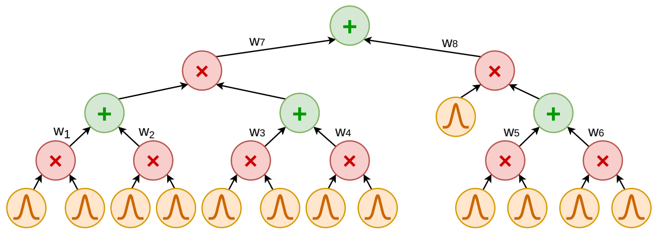


Fig. 1. Example of a Sum-Product Network graph.

different types of nodes. At the bottom of the DAG, so-called *leaf nodes* capture the univariate probability distribution of a single variable/feature. Depending on the type of data (e.g., continuous vs. discrete), and underlying distribution of the data, different probability distributions can be used, e.g., Gaussian distribution for continuous variables, or a categorical distribution for discrete values. The so-called *scope* of a leaf node is the single variable associated with it. 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. For the SPN to be valid, the scopes of the different child nodes of a product node must be *disjoint*. Weighted sum nodes, on the other hand, represent a mixture of distributions and the scopes of all child nodes must be *identical* in a valid SPN. The structure of the SPN depends on the distribution of the underlying data, and can either be learned from data, or be hand-crafted, just followed by parameter learning. An overview of SPN learning algorithms can be found in [5]. A small example of an SPN graph is shown in Fig. 1. As a core functional principle, the SPN decomposes complex multivariate "global" functions by exploiting the way in which the global function factors into a product of simpler "local" functions of a subset of the variables [1]. SPN can be used to solve machine learning tasks, such as classification, by performing inference on the underlying DAG. In general, SPNs support multiple different types of inference. In this study, we are focusing on two of these types, namely joint probability inference and marginal inference. Joint probability inference is used to obtain the joint probability given *full* evidence (i.e., a value for each variable). To this end, the evaluation of the SPN DAG starts by evaluating the distribution of the leaf nodes, given the value of the variable associated with each of them. After that, the values are propagated upwards through the DAG, performing multiplication or weighted addition at the product and sum nodes, until a final probability value is obtained at the root node of the SPN. Marginal inference, on the other hand, is used when only *partial* evidence is available. Leaf nodes for which no evidence is available are set to 1, the remaining ones are evaluated just as in joint inference, and the propagation of values through the SPN is performed analogously to the description above.

The compiler developed in this study aims to accelerate the *inference* in Sum-Product Networks by efficiently mapping them to different hardware targets.

Learning of the SPN is assumed to have taken place beforehand, using a standard Sum-Product Network framework such as SPFlow [3].

2.2 MLIR

The implementation of SPNC in this work is heavily based on the open-source MLIR framework [2][3]. Therefore this section presents a brief overview of MLIR.

MLIR aims to facilitate the implementation of compilers by providing an *extensible* framework for the implementation of multi-level IRs. The main reason for adding multiple levels of abstractions into compilers is that an early lowering to a *low-level* intermediate representation such as LLVM IR loses too much of the high-level structure of the program, which later on must be reconstructed using often fragile approaches based on the low-level IR in order to perform transformations, e.g., on loops. Capturing additional information and potentially domain-specific semantics in one or multiple high-level IRs enables the compiler to perform more powerful program transformations.

Because MLIR provides common components for the implementation of IRs, such as pass managers and common transformations, users can focus on the design of the IR itself.

In order to not impose too many constraints on the semantics of different IRs, MLIR defines a minimal set of *generic* abstractions that must be used by all IRs. Similar to most modern compilers, MLIR uses the static single assigment (SSA) form, with *operations* (short: Ops) consuming and producing *values*. All values are typed, with the type system being extensible, while also defining a number of common types. So-called *attributes*, which are also typed, can additionally be used to attach compile-time information to operations.

Operations, types, and attributes are organized in so-called *dialects*, which do not add any semantics, but are a mere logical unit for the organization of the IR. Dialects can be mixed in the same logical unit, and so-called *lowerings* translate *between* different dialects, with intra-dialect transformations also being available. Typically, a progressive, step-wise lowering from a high-level dialect to lower-level dialects is used to compile for a specific target, e.g., a CPU. To enable the implementation of common transformations, MLIR uses the notion of *traits* and *interfaces* that can be attached to operations, and provides generic interfaces for transformations such as constant folding.

MLIR's extensible nature allows us to design *custom* high-level IRs. We use these to represent Sum-Product Networks in the compiler developed in this work, while we can rely on the *common* infrastructure and dialects provided with MLIR to efficiently target different hardware platforms.

3 Approach

The aim of SPNC is to automatically compile Sum-Product Networks and probabilistic queries operating on them to executable kernels. Compiling individual

[3] https://mlir.llvm.org

SPNs allows to employ *all* hardware features available on the target platform for fast inference, for example vector extensions present on most modern CPUs. Currently, SPNC supports two main targets:

– **CPUs:** Being based on MLIR and LLVM, SPNC can target any CPU for which a backend is present in LLVM. Vector extensions are currently supported on x86 (AVX, AVX2, AVX-512) and Arm (Neon Advanced SIMD) CPUs.
– **GPUs:** The flow currently supports Nvidia CUDA GPUs, but the generic GPU abstractions of MLIR would allow to target other GPUs with comparably few changes.

The compilation flow for both targets is based on MLIR. To this end, two SPN-specific MLIR dialects have been designed and implemented, which will be described in Section 3.1. Starting from these dialects, the target-specific lowerings will create an executable, using the flows described in Section 3.2 and Section 3.3. The user interface and some implementation details are described in Section 4.

3.1 MLIR Dialect Design

The first of the two SPN-specific dialects that SPNC employs during compilation, called **HiSPN**, captures the DAG structure of a Sum-Product Network and the information about the query to perform on a high level of abstraction. It was designed to closely match the representation used internally by the SPFlow framework [3], similar to how an abstract syntax tree captures a general-purpose programming language on a high level of abstraction.

In the HiSPN dialect, an abstract probability type is used for values inside the SPN DAG, allowing SPNC to delay the decision on the actual data type used for computation and take graph characteristics into account.

The second SPN-specific dialect, called **LoSPN**, represents the actual computation that needs to be performed to process the requested query on the SPN. The top-level unit in this dialect is a *Kernel*, comprising one or multiple *Tasks*. A Task does not only represent (parts of) the SPN DAG structure, including weighted sum, product, and leaf nodes, but also contains information about which inputs values will need to be accessed and which outputs will be produced as intermediate or final result. In contrast to HiSPN, LoSPN uses a concrete type for the values. To represent the computation in log-space, which commonly used to avoid arithmetic underflow in Sum-Product Network inference, a SPN-specific data type was added to the LoSPN dialect.

The lowering from HiSPN to LoSPN is currently identical for both flows, targeting CPU and GPU. In this step, the necessary computation for the query is derived from the SPN DAG structure and query information captured by HiSPN and is the lowered into the operations of the LoSPN dialect. After lowering, the LoSPN representation undergoes a number of transformations, including steps such as common subexpression elimination (CSE), followed by the target-specific lowerings to dialects provided by the MLIR framework described in the next two sections.

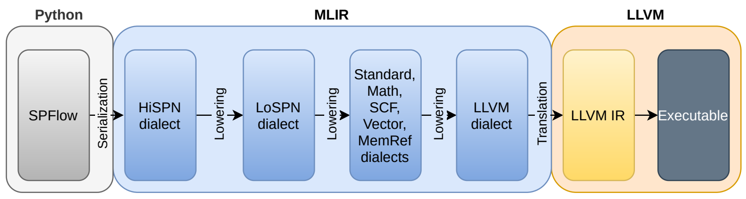


Fig. 2. CPU Compilation Flow.

3.2 CPU Compilation Flow

The compilation flow for the CPU starts with the serialized SPN model (cf. Section 4), an overview is shown in Fig. 2. After deserialization to the HiSPN dialect, lowering to LoSPN, and the transformations on the LoSPN dialect have been performed, the IR is again lowered to dialects provided as part of the MLIR framework. The Kernel and the Tasks in the LoSPN dialect are lowered to functions, with the Kernel function calling the functions for the individual Tasks and the Task functions iterating multiple inputs for batch processing.

The operations contained inside each Task are lowered to a combination of different dialects (Note that MLIR allows to mix operations from different dialects in the same function/module):

- **Standard dialect:** Contains operations such as simple addition or multiplication on arbitrary data-types, including vectors.
- **Math dialect:** Elementary math functions, such as the `exp` and `log` function, are represented by operations from this dialect.
- **SCF dialect:** Operations from this dialect represent structured control flow, e.g. for-loops.
- **MemRef dialect:** Contains facilities to handle memory, e.g., allocation or store/load operations.
- **Vector dialect:** Vector specific operations, e.g., vector lane shuffling.

The combination of the Vector dialect and the Standard operation's ability to handle vector data-types lets the compiler exploit the CPU's SIMD extensions, if present, for maximum efficiency. In contrast to a generic loop vectorization, a domain-specific compiler such as SPNC can, thanks to the MLIR framework, leverage high-level information to generate more efficient code, e.g., by employing a combination of simple vector loads and shuffles instead of expensive gather loads.

After some transformation passes provided by the MLIR framework, all dialects are lowered to the LLVM dialect and then translated to LLVM IR, so LLVM can produce the final executable. As part of this process, the executable is also linked with vector libraries, providing optimized implementations of elementary math functions (e.g., `exp`) for vector code. The currently supported vector libraries are Intel SVML and Libmvec for x86 CPUs, and ARM Optimized Routines for ARM Neon.

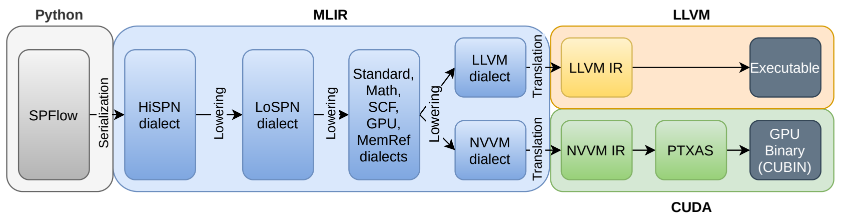


Fig. 3. GPU Compilation Flow.

Although it is technically possible to perform within the MLIR framework, we have decided to implement *multi-threading* in the runtime-component (cf. Section 4) rather than directly in the generated code. This allows to adopt the threading behavior dynamically, e.g., when executing multiple compiled kernels concurrently.

3.3 GPU Compilation Flow

Similar to the CPU compilation flow, the GPU compilation flow also starts from the serialized SPN model and performs the same steps up to the lowering of LoSPN to dialects from the MLIR framework. Here, for the GPU, the Kernel is lowered into a function, which will remain on the host CPU and will be responsible for GPU/CPU data transfers and the invocation of the Tasks, which, in contrast to the CPU flow, are lowered into *device* functions executing on the GPU.

For the operations inside the Tasks, a similar combination of MLIR-provided dialects is used, with one notable difference: Instead of the Vector dialect, the **GPU** dialect is used to represent the SIMT execution model, with operations for access to block and thread identifiers and for representation of GPU device functions and runtime functions for memory & execution management.

After that step, the GPU- and host portion of the IR are separated into two compilation units. While the flow for the host portion via LLVM is very similar to the CPU flow, eventually resulting in an executable, the GPU portion of the code is translated in multiple steps to NVVM IR, PTX assembly and a GPU binary (CUBIN format). This GPU binary is then loaded at runtime by the host function to execute inference on the GPU.

An overview of the overall compilation flow for the GPU is shown in Fig. 3.

4 Python Interface & Implementation

In order to make the compiler and the execution of the compiled binaries via the runtime component accessible to machine learning experts working with the SPFlow library, SPNC offers a Python-based interface to the compiler and runtime. In this manner, machine learning experts can create the SPNs using their familiar tools from the SPFlow library and feed their results to the compiler.

```
import numpy as np
from spn.structure... import ...

# Create an example SPN
p0 = Product(children=[Categorical(p=[0.3, 0.7], scope=1), Categorical(p=[0.4, 0.6], scope=2)])
...
spn = Sum(weights=[0.4, 0.6], children=[p2, p4])

# Create some random test data
...
test_data = np.c_[a, b, c].astype("float32")

# Perform inference using SPFlow
from spn.algorithms.Inference import log_likelihood
spflow_results = log_likelihood(spn, test_data)                # Location (1)

# Compile for CPU and perform inference
from spnc.cpu import CPUCompiler
cpu_results = CPUCompiler().log_likelihood(spn, test_data)     # Location (2)

# Compile for CUDA GPU and perform inference
from spnc.gpu import CUDACompiler
gpu_results = CUDACompiler().log_likelihood(spn, test_data)    # Location (3)
```

Fig. 4. Python interface usage example.

Fig. 4 shows an usage example of the Python interface, the example SPN is taken from SPFlow's documentation. Location (1) in the code shows how inference is usually performed in SPFlow, by invoking `log_likelihood`.

The other two locations show the invocation of SPNC for compilation and execution on the CPU (2) or CUDA GPU (3). In both cases, the invocation of `log_likelihood` on the compiler will first compile the SPN using the respective flow described in Section 3.2 and Section 3.3 and then execute inference using the compiled kernel. A small runtime component part of SPNC is responsible for loading the compiled kernel and executing inference. In case of CPU execution, the runtime is also responsible for multi-threaded execution using OpenMP. The Python interface also supports separate compilation and execution, so an SPN only needs to be compiled once to repeatedly perform inference.

Similar to SPFlow, the compiled kernels also support marginalized inference by passing `NaN` as input value for marginalized variables.

The Python interface is implemented using Pybind11[4]. 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. 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[5] library was implemented.

[4] https://github.com/pybind/pybind11
[5] https://capnproto.org/

5 Evaluation

To demonstrate SPNC's ability to target different heterogeneous systems, we are evaluating it on two different systems: As an example of an embedded-grade device, a Nvidia Jetson AGX Xavier device with 6-core ARM v8 CPU and Volta GPU will be used. As a non-embedded device, 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 will be used. As the Ryzen processor does not support AVX-512, experiments for AVX-512 will be performed on a dual-socket system with two Intel Xeon Platinum 9242 CPUs and 384 GB RAM.

As a real-world application of SPNs, an SPN-based automatic speaker identification from [4] is used as example application. Based on the open-source release by Nicolson et al.[6], we evaluate two different scenarios, namely the clean speech samples (245567 samples) and noisy speech samples with marginalization (1227835 samples). 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. The implementation by Nicolson et al. contains an SPN *per speaker*, so a set of 628 different SPNs is used for evaluation.

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 across all platforms for CPU is 7 seconds (max. 33s) and for GPU 2s (max. 5s). The translation of the SPFlow graph to a Tensorflow graph, which is provided by the SPFlow framework, takes 18 seconds on average (max. 61s).

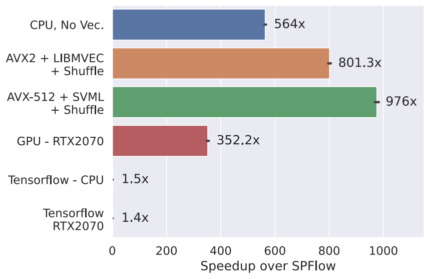
5.1 Non-Embedded Systems

Fig. 5 shows the performance comparison for the non-embedded systems, the numbers are given as speedup over the inference execution with SPFlow.

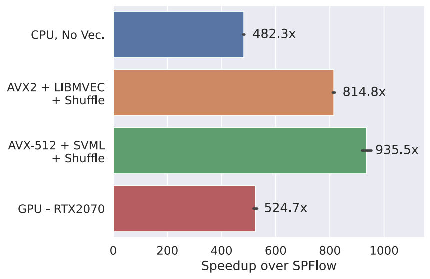
The speedup achieved by translating the SPFlow graph to a Tensorflow graph is relatively low on both CPU (geo.-mean 1.5x) and GPU (1.38x), as the graph is still broken down into individual operations that are launched through the Tensorflow runtime. Marginalization is currently not supported by the Tensorflow translation in SPFlow, therefore no bars are shown for Tensorflow in Fig. 5b.

SPNC on the other hand achieves speedups of 564x and 482x by compiling for the CPU and multithreaded execution, without employing vector extensions. If the vector extensions and vector libraries for elementary functions (Libmvec for AVX-2 and Intel SVML for AVX-512) are used additionally, the speedup increases to 801/814x and 976/935x, respectively. The compilation for the GPU also achieves a significant speedup of 352x and 524x, but data movements between host and device in both cases make up for more than 60% of the execution

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



(a) Clean Speech



(b) Noisy Speech with Marginalization

Fig. 5. Performance comparison on non-embedded systems, given as speedup over execution in SPFlow.

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, limits the speedup.

5.2 Embedded System

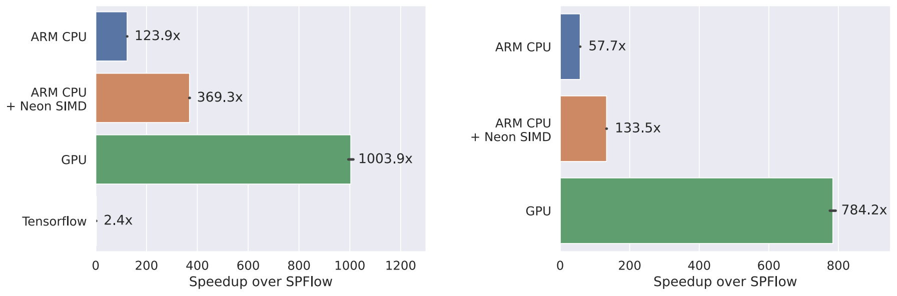
Fig. 6 shows the same comparison for the embedded-grade Jetson Xavier platform. As there are fewer CPU cores available than on the Ryzen/Xeon CPU, the speedup achieved by compilation for CPU is smaller compared to Fig. 5, but still reaches 124x (clean) and 58x (noisy) compared to SPFlow. When using the Neon Advanced SIMD extensions, the speedup increases by 2.9x/2.3x to 369x and 133x. In contrast to Fig. 5, the GPU compilation on the Xavier platform provides better performance than the CPU compilation. This is due to the fact that GPU and CPU *physically* share the same memory and no memory transfers between host CPU and GPU are necessary. With the memory transfers eliminated, our GPU compilation achieves speedups of 1004x and 784x.

Another important aspect on embedded systems is memory usage: For the noisy speech samples, it is not possible to process all samples in one batch with SPFlow, as the SPFlow inference runs out of memory (16 GB) and the input has to be processed in multiple blocks sequentially. The compiled kernels are much more memory-efficient and allow to process all samples in a single invocation.

For the Tensorflow comparison on this platform, the Tensorflow package officially provided by Nvidia for Jetson platforms is used. Similar to Fig. 5, the translation provides a speedup over SPFlow (2.36x), but is still significantly slower than the compiled executables.

6 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.



(a) Clean Speech (b) Noisy Speech with Marginalization

Fig. 6. Performance comparison on embedded systems, given as speedup over execution in SPFlow.

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 [3] and libspn [8].

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 has shown, our compiler significantly outperforms both variants.

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 5.

Another interesting approach to efficient training and inference for SPNs is through tensorization of the SPN graph, as shown in [6] or [12]. 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.

In previous work [9, 10], we have developed a custom, FPGA-based inference accelerator for Sum-Product Networks. 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 currently does not support Gaussian distributions.

7 Conclusion

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 facilitates the implementation of domain-specific compilers.

SPNC was designed to 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 a real-world example of Sum-Product Networks, we have demonstrated how SPNC can target different heterogeneous systems and can achieve a speedup over SPFlow of a factor of up to 978x when compiling for CPUs with vector extensions, and up to a factor of 1003x when targeting CUDA GPUs.

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`.

References

1. Kschischang, F.R., Frey, B.J., Loeliger, H.A.: Factor graphs and the sum-product algorithm. IEEE Transactions on information theory **47**(2), 498–519 (2001)
2. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J.A., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: Mlir: Scaling compiler infrastructure for domain specific computation. In: CGO 2021 (2021)
3. Molina, A., Vergari, A., Stelzner, K., Peharz, R., Subramani, P., Mauro, N.D., Poupart, P., Kersting, K.: Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks (2019)
4. Nicolson, A., Paliwal, K.K.: Sum-product networks for robust automatic speaker identification (2020)
5. Paris, I., Sanchez-Cauce, R., Diez, F.J.: Sum-product networks: A survey (2020)
6. Peharz, R., Vergari, A., Stelzner, K., Molina, A., Shao, X., Trapp, M., Kersting, K., Ghahramani, Z.: Random sum-product networks: A simple but effective approach to probabilistic deep learning. In: Proceedings of UAI (2019)
7. Poon, H., Domingos, P.: Sum-product networks: A new deep architecture. In: 2011 IEEE Intl. Conf. on Computer Vision Workshops (ICCV Workshops) (2011)
8. Pronobis, A., Ranganath, A., Rao, R.P.: Libspn: A library for learning and inference with sum-product networks and tensorflow. In: Principled Approaches to Deep Learning Workshop (2017)
9. Sommer, L., Oppermann, J., Molina, A., Binnig, C., Kersting, K., Koch, A.: Automatic mapping of the sum-product network inference problem to fpga-based accelerators. In: IEEE Intl. Conf. on Computer Design (ICCD). IEEE (2018)
10. Sommer, L., Weber, L., Kumm, M., Koch, A.: Comparison of arithmetic number formats for inference in sum-product networks on fpgas. In: 2020 IEEE 28th Annual Intl. Symp. on Field-Programmable Custom Computing Machines (FCCM) (2020)
11. Sugiarto, I., Axenie, C., Conradt, J.: Fpga-based hardware accelerator for an embedded factor graph with configurable optimization. Journal of Circuits, Systems and Computers **28**(02), 1950031 (2019)
12. van de Wolfshaar, J., Pronobis, A.: Deep Generalized Convolutional Sum-Product Networks for Probabilistic Image Representations. arXiv:1902.06155 (Sep 2019)

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