SPNC: An Open-Source MLIR-Based Compiler for Fast Sum-Product Network Inference on CPUs and GPUs

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

***Abstract*—Sum-Product Networks (SPNs) are an alternative to the widely used Neural Networks (NNs) for machine learning. SPNs can not only reason about (un)certainty by qualifying their output with a probability, they also allow fast (tractable) inference by having run-times that are just linear w.r.t. the network size.**

We present *SPNC*, the first tool flow for generating fast native code for SPN inference on both CPUs and GPUs, including the use of vectorized/SIMD execution. To this end, we add two SPN-specific dialects to the MLIR framework and discuss their lowering towards the execution targets.

We evaluate our approach on two applications, for which we consider performance, scaling to very large SPNs, and compile vs execution-time trade-offs. In this manner, we achieve multiple orders of magnitude in speed-ups over existing SPN support libraries.

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

I. Introduction

Sum-Product-Networks (SPN) [1], a recent class of probabilistic graphical models, are a machine-learning approach with a number of advantages over the widely used neural networks (NN). These include the capability to reason about the (un)certainty of their output by providing a probability value, as well as being able to perform inference in linear time w.r.t. the size of the SPN.

However, tool support for using SPNs is more limited than for NNs. As an example, for the latter, a number of tools is available to compile the NNs down to native code for inference, such as Tensorflow's XLA [2] or Facebook's Glow [3], for quickly performing the inference operation. While software support for SPNs is improving, it is generally not aimed at high performance yet, e.g., the commonly used SPN framework SPFlow [4] performs inference in Python code.

We present *SPNC*, the first multi-target tool chain capable of mapping the SPN inference operation into high-performance native code for both CPUs and GPUs, including support for advanced processor features such as vectorized/SIMD operations. SPNC, which is available as open-source[1], is based on the modern MLIR framework.

Two custom MLIR dialects have been created to represent the high-level semantics of Sum-Product Network inference. The design of these dialects is described in Section III. Starting from these two dialects, we develop a number of target-independent transformation steps that include techniques to handle even very large Sum-Product Network models during compilation (Section IV-A). The result of these transformation steps then serves as the input to the two lowering pipelines, targeting CPUs (Section IV-B) and GPUs (Section IV-C), respectively.

We introduce SPNs and the MLIR framework in Section II, and survey related work in Section VI. Our approach is evaluated in Section V on two different applications. One targeting the use of generic (full featured) SPNs, the second one focusing on processing very large SPNs. The latter kind of SPN occurs in practice, when more restricted SPN models are used. Our evaluation includes a design-space exploration of different optimization options and compares to performing SPN inference in existing frameworks.

II. Background

A. Sum-Product Networks

Handling uncertainties and imperfections, such as missing feature values in real-world data, is an important requirement for machine learning models. Probabilistic models and, more specifically, Sum-Product Networks [1], which are a relatively recent representative of this class of models, are well suited to handle these uncertainties. One mechanism that allows Sum-Product Networks to cope with uncertainties is the ability to actually quantify uncertainty over their own output in the form of probabilities, an ability not found in most traditional neural network architectures.

At the heart of a Sum-Product Network model lies a directed acyclic graph (DAG), capturing the joint probability distribution over a set of random variables. The DAG is composed from three different types of nodes: Leaf nodes represent univariate distributions over a single random variable and can model continuous (e.g., Gaussian) as well as discrete (e.g., categorical) distributions. Apart from the leaf nodes, the DAG contains weighted sum nodes and product nodes, which represent a mixture of distributions and the factorization of independent variables, respectively. The structure of the DAG can either be learned directly from data, or can be constructed beforehand, followed by weight learning. An extensive overview of learning algorithms for Sum-Product Networks is given in [5], an example of an SPN DAG is shown in Fig. 1.

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

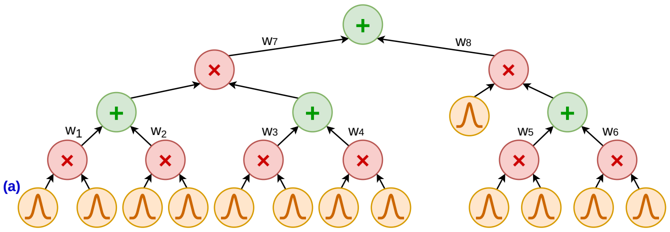

Fig. 1. Example of a Sum-Product Network graph.

After learning the DAG, it can be used to perform inference to solve machine learning tasks such as classification. For most probabilistic queries, the inference requires a single bottom-up evaluation of the SPN DAG, starting at the leaf nodes. This also holds true for the two kinds of inference we focus on in this work, namely joint probability and marginal inference. In both cases, the inference starts by querying the univariate leaf distributions using the full (joint) or partial (marginal) evidence, i.e., input values. The probabilities obtained from the leaf nodes are then propagated upwards through the DAG, performing multiplication and weighted addition at the corresponding nodes. Eventually, the final result probability is obtained at the root node of the DAG.

As only a *single* traversal of the DAG is needed for inference, the complexity of inference is *linear* in the DAG size [6], making SPN *tractable*. This tractable inference, a property not found on many other probabilistic (graphical) models such as Bayesian Networks, combined with their expressiveness and ability to handle uncertainties, make Sum-Product Networks an interesting candidate for many applications. An overview of applications, ranging from medical imaging [7] to robotics [8], can also be found in the survey [5].

The compiler developed in this work aims to further speed-up inference in Sum-Product Networks through compilation for different target platforms. Training of the SPN is assumed to have taken place beforehand, using a standard Sum-Product Network framework such as SPFlow [4].

B. MLIR

For the implementation of the compiler, we use the open-source MLIR framework [9]. MLIR is motivated by the observation that the early lowering to a low-level intermediate representation (IR) in a conventional compiler looses much of the high-level semantics and structure of an application, because it cannot be represented in the low-level IR.

In order to allow the compiler to reason about the high-level semantics and to achieve better compilation results, it would be desirable to represent the application by a hierarchy of intermediate representations, and only gradually lower these to a low-level representation, similar to how this is already implemented in Tensorflow's XLA, or the Swift and Rust compilers.

To facilitate the implementation of such IRs at various abstraction levels and avoid the repeated implementation of infrastructure components, the MLIR framework aims to provide a framework for the re-usable implementation of such IRs, as well as compilers based on them.

The basic organization unit for IRs are the *Dialects*, which can define a set of *Operations* and *Types*. While the MLIR framework also uses the static single assignment (SSA) form, its representations are more flexible than the traditional control-flow graph (CFG) representations found in many compilers, such as in LLVM IR. In contrast to these CFG structures, MLIR allows for *nesting* inside the IR. This means, that each *Operation* does not only produce typed (SSA) values, but can also have one or multiple *Regions* attached to it. Each Region then contains one or multiple *Blocks*, which, in turn, contain Operations. This flexibility allows representing different structures, e.g., "traditional" CFGs, as well as more application/domain-specific structures. In addition, *Attributes* can be attached to operations to represent additional compile-time information.

A typical compile flow will use multiple Dialects, and mixtures of dialects, to represent the application at different abstraction levels. As the flow progresses, and moves to lower-level dialects, so-called *Lowerings* are used to translate between different dialects. MLIR also provides, in addition to other common infrastructure such as pass managers, a number of generic transformations, such as constant folding, enabled through *Traits* and *Interfaces* that can be attached to operations.

In the compiler developed in this work, MLIR is used to represent the high-level semantics of Sum-Product Network inference by two SPN-specific dialects (cf. Section III), and to target CPUs and GPUs via a mixture of dialects provided by the MLIR framework (cf. Sections IV-B and IV-C).

III. MLIR Dialect Design

The goal of this work is to develop a domain-specific tool for the compilation of probabilistic queries on Sum-Product Networks. The compilation should make best use of the various hardware features available on the different target platforms, e.g., SIMD vector extensions on CPUs.

The implementation of SPNC is based on the open-source MLIR-framework [9]. The first step is the design of two SPN-specific dialects that allow the compiler to reason about SPN semantics. Both of these will be described throughout this section.

A. HiSPN Dialect

The first SPN-specific dialect is called *HiSPN*. It was designed to closely match the SPN representation used by the open-source framework SPFlow [4], a popular library for SPN modelling and training. As such, it captures SPN query and model information on a high level of abstraction.

Fig. 2 shows an excerpt of the HiSPN dialect for the example SPN in Fig. 1 and a query operating on that SPN. The HiSPN dialect fulfills two main tasks. The first task, represented by the `hi_spn.graph` operation, and the operations nested inside its region, is to capture the SPN's directed acyclic graph structure. To this end, the HiSPN dialect contains operations directly corresponding to sum, product and leaf nodes in an

TABLE I

OPERATIONS OF THE HISPN DIALECT. *ProbType* IS USED TO REFER TO THE ABSTRACT PROBABILITY TYPE DEFINED IN THE HISPN DIALECT. *InputType* CAN BE ANY FLOAT OR INTEGER TYPE.

Name	**Short Description**	**Operands**	**Attributes**	**Results**	**Nested Regions**
`joint_query`	Joint probability query	-	numFeatures, inputType, batchSize, supportMarginal	-	yes
`graph`	SPN DAG container	-	numFeatures	-	yes
`root`	SPN DAG root marker	rootValue (ProbType)	-	-	no
`product`	Product node	operands (ProbType, variadic)	-	ProbType	no
`sum`	Weighted sum node	operands (ProbType, variadic)	weights	ProbType	no
`histogram`	Histogram leaf	index (InputType)	buckets, bucketCount	ProbType	no
`categorical`	Categorical leaf	index (InputType)	probabilities	ProbType	no
`gaussian`	Gaussian leaf	evidence (InputType)	mean, stddev	ProbType	no

```
module  {
  "hi_spn.joint_query"() ( {
    "hi_spn.graph"() ( {
    ^bb0(%arg0: f32, %arg1: f32):
      %0 = "hi_spn.gaussian"(%arg0) {mean = 5.0e-01 : f64, stddev = 1.0e+00 : f64} : (f32) -> ...
      %1 = "hi_spn.gaussian"(%arg1) {mean = 2.5e-01 : f64, stddev = 5.0e-01 : f64} : (f32) -> ...
      %2 = "hi_spn.product"(%0, %1) : (!hi_spn.probability, !hi_spn.probability) -> !hi_spn.probability
      ...
      %6 = "hi_spn.sum"(%2, %5) {weights = [2.5e-01, 7.5e-01]} : (!hi_spn.probability, ...) -> ...
      "hi_spn.root"(%6) : (!hi_spn.probability) -> ()
    }) {numFeatures = 2 : ui32} : () -> ()
  }) {batchSize = 96 : ui32, inputType = f32, numFeatures = 2 : ui32, supportMarginal = true} : () -> ()
}
```

Fig. 2. Excerpt of the HiSPN representation of the SPN DAG and a joint probability query for the example SPN in Fig. 1. For example, `%0` directly corresponds to the node labeled with (a) in Fig. 1.

SPN. The graph structure is modeled by the dataflow between those nodes.

The second task is to capture information about the type of query that should be performed. This is represented by an operation *surrounding* the graph, in this case the `hi_spn.joint_query`, to which additional information such as the batch size is attached in the form of attributes. Keeping the representation of the DAG independent from the query in HiSPN allows to reuse the same graph representation for different queries.

The HiSPN dialect also uses the abstract `hi_spn.probability` type instead of a concrete type for computation. This allows the compiler to defer the decision which datatype will be used for computation until the lowering. The decision can then be based on characteristics, e.g., the depth of the graph, of the SPN.

An overview of all HiSPN dialect operations is given in Table I.

B. LoSPN Dialect

The second SPN-specific dialect, called *LoSPN*. It is designed as the lowering target for HiSPN and represents the actual computations necessary to process a query. A probabilistic query on a batch of inputs is represented by a *Kernel*, which comprises one or multiple *Tasks*.

The semantics of a Task are the application of the operations contained in its single region to every input sample in a batch. To this end, the entry block of a task has an additional block argument for the batch index, similar to the induction variable in a loop (`%arg2` in line 5 of Fig. 3). Inside the Task region, LoSPN operations for access to inputs and results of the task are present, i.e., the `lo_spn.batch_read` and `lo_spn.batch_write`. Representing the access to individual features of each input sample as dedicated operations allows the compiler to reason about and optimize the memory access pattern, as we will discuss in Sections IV-B and IV-C.

The actual arithmetic operations are nested inside the single region of a *Body* operation. Their representation is still very close to the original SPN DAG representation with sum, product and leaf nodes. The two major differences are that the operations only take two operands (in contrast to the variadic HiSPN operations) and weighted sums are decomposed into sum and product operations.

To represent batches of inputs and outputs, the LoSPN dialect supports the MLIR `tensor` type as well as the `memref` type, and both are used for different purposes at the different stages of the compilation process (cf. Section IV-A5).

As computation in log-space is very common for Sum-Product Networks to avoid arithmetic underflow for small probabilities, the LoSPN dialect also introduces the `lo_spn.log` type to represent computation in log-space. While the actual computation in the generated code will still happen using traditional floating-point arithmetic (32-bit float in the example in Fig. 3), the log-space type instructs the lowering to generate arithmetic instructions for log-space operation. For example, for the multiplication in line 14 of Fig. 3, a simple floating-

TABLE II

CORE OPERATIONS OF THE LOSPN DIALECT. *CT* CAN BE ANY FLOAT OR INTEGER TYPE, OR THE LOG-TYPE DEFINED IN THE LOSPN DIALECT. *IT* CAN BE ANY FLOAT OR INTEGER TYPE. *T* IS SHORT FOR THE MLIR TENSOR TYPE, *M* IS SHORT FOR THE MLIR MEMREF TYPE.

Name	**Short Description**	**Operands**	**Attributes**	**Results**	**Nested Regions**
`kernel`	Entry point for query	Function-like			yes
`task`	Computational task	inputs (T/M of IT/CT, variadic)	batchSize	T of CT (optional)	yes
`body`	Container for arithmetic operations	inputs (IT/CT, variadic)	-	CT (variadic)	yes
`batch_extract/` `batch_read`	Input access from Tensor/MemRef	input (T/M), dynamicIndex (Index)	staticIndex, transposed	IT/CT	no
`batch_collect`	Result storage to Tensor	batchIndex (Index), resultValues (CT, variadic)	transposed	T of CT	no
`batch_write`	Result storage to Memref	batchMem (M), batchIndex (Index), resultValues (CT, variadic)	transposed	-	no
`mul`	Multiplication	left (CT), right (CT)	-	CT	no
`sum`	Addition	left (CT), right (CT)	-	CT	no
`histogram`	Histogram leaf	index (InputType)	buckets, bucketCount	CT	no
`categorical`	Categorical leaf	index (InputType)	probabilities	CT	no
`gaussian`	Gaussian leaf	evidence (InputType)	mean, stddev	CT	no

```
module  {
  "lo_spn.kernel"() ( {
  ^bb0(%arg0: memref<?x2xf32>, %arg1: memref<1x?xf32>):
    "lo_spn.task"(%arg0, %arg1) ( {
    ^bb0(%arg2: index, %arg3: memref<?x2xf32>, %arg4: memref<1x?xf32>):
      %0 = "lo_spn.batch_read"(%arg3, %arg2){staticIndex = 0 : ui32,...} : (memref<?x2xf32>, index)->f32
      ...
      %2 = "lo_spn.body"(%0, %1) ( {
      ^bb0(%arg5: f32, %arg6: f32):
        %3 = "lo_spn.gaussian"(%arg5) {...} : (f32) -> !lo_spn.log<f32>
        ...
        %11 = "lo_spn.constant"(){type = !lo_spn.log<f32>, value = -0.28 : f64} : () -> !lo_spn.log<f32>
        %12 = "lo_spn.mul"(%8, %11) : (!lo_spn.log<f32>, !lo_spn.log<f32>) -> !lo_spn.log<f32>
        %13 = "lo_spn.add"(%10, %12) : (!lo_spn.log<f32>, !lo_spn.log<f32>) -> !lo_spn.log<f32>
        "lo_spn.yield"(%13) : (!lo_spn.log<f32>) -> ()
      }) : (f32, f32) -> f32
      "lo_spn.batch_write"(%arg4, %arg2, %2) {transposed = true} : (memref<1x?xf32>, index, f32) -> ()
    }) {batchSize = 96 : ui32} : (memref<?x2xf32>, memref<1x?xf32>) -> ()
  })
}
```

Fig. 3. Excerpt of the result for the lowering from the HiSPN dialect code in Fig. 2 to the LoSPN dialect. `%3` corresponds to `%0` in Fig. 2.

point addition will be generated, as multiplication in log-space is equivalent to addition of the inputs.

An overview of the most important operations of the LoSPN dialect is given in Table II.

IV. COMPILATION FLOW

Based on the two new SPN-specific dialects described in the previous section, and the various existing dialects provided by the MLIR framework, we construct a complete end-to-end compilation flow for probabilistic queries operating on Sum-Product Networks.

At first, a number of *target-independent* compilation steps is applied to the input Sum-Product Network. These steps are identical for all targets and are presented in the first part of this section. Afterwards, the lowering used to generate code for the two main targets of the compiler, namely CPUs and GPUs, is described in more detail.

A. Target-Independent Compilation Steps

The aim of the *target-independent* compilation steps is to perform transformation and early optimization steps to generate a LoSPN module, that can then serve as the input to the *target-specific* lowerings described in the next subsections.

1) Input Interface: In order to provide an easy-to-use interface for machine learning experts, the compiler tightly integrates with the popular SPFlow library. Unfortunately, SPFlow does not yet support binary serialization of SPNs, therefore a custom binary serialization based on Cap'n Proto[2] is used to serialize the SPNs for communication with the compiler. The Python interface of the compiler also allows to start the compilation and execution of the compiled query directly from Python with as little as a single API call.

2) HiSPN Translation: During de-serialization of the binary format, the query and the corresponding SPN DAG are translated to the HiSPN dialect. As the HiSPN dialect is designed to closely resemble SPFlow's internal representation of SPNs (cf. Section III-A), this translation is fairly straightforward. The translation to HiSPN is also the entry point to the MLIR framework, and subsequently, the MLIR framework can be

[2] https://capnproto.org/

used for some early optimizations, e.g., the transformation of DAG nodes with only a single input.

3) Lowering to LoSPN: After these initial transformations, the module is lowered to the LoSPN dialect. The HiSPN representation of the SPN DAG and the requested probabilistic query is used to generate a LoSPN Kernel with a Task containing the necessary computations.

At this point, the LoSPN module uses the MLIR `tensor` type to represent batches of input/output, as this representation facilitates tracking of values flowing across tasks.

4) Graph Partitioning: While Sum-Product Networks can provide a compact representation for many tasks, they can also grow to several hundred thousand nodes for other applications, e.g., for imaging applications.

In such cases, compilation of the whole graph in one piece can quickly become infeasible, as we will demonstrate in Section V-B. Therefore, graph partitioning is used to partition large LoSPN Tasks into multiple smaller Tasks. The algorithm for partitioning is based on the idea for heuristic acyclic graph partitioning from [10]. This algorithm was chosen for two main reasons: (1) the acyclicity constraint is crucial, as Tasks dependencies must not form a cycle and (2) the heuristic nature of the algorithm keeps the runtime of the graph partitioning itself within a feasible limit.

Similar to [10], the algorithm first constructs an initial partitioning. In contrast to [10], we do not use a random topological ordering. Instead, to account for the tree-like DAG structure of SPNs, that tapers towards the root node, we use an ordering similar to a depth-first traversal of the DAG. A node is added to an ordering as soon as all its child-nodes have been processed, making it more likely that they end up in the same initial partition. Our ordering still preserves the condition that no node in partition V_j has an outgoing edge to a node in V_i with $i < j$, which is important for acyclicity.

Furthermore, our partitioning is also less strict about the balancing of partition sizes and allows up to 1% of slack. This enables more moves during the refinement, as slightly unbalanced tasks only have a negligible impact on application execution.

The cost model for the partitioning was also adapted to reflect that communication across partitions occurs via load/stores to and from intermediate buffers. In [10], each edge crossing partitions has identical cost. In our case, all edges from partition V_j to V_i have a *combined* cost of 1, as the value needs to be stored only *once* in the Task corresponding to V_j, and loaded only *once* in the Task for V_i.

After the initial partitioning, the *Simple Moves* heuristic from [10] is used to further refine the partitioning. While this heuristic only uses moves between neighbouring partitions for the refinement, it is also lightweight and does not have a huge impact on compilation time.

5) Bufferization: So far, the LoSPN module used MLIR's `tensor` type to represent batches of input and output, because tensor *values* are easier to reason about than the *side-effects* of operations operating on MLIR `memrefs`.

As preparation for the lowering to the different targets, the bufferization now transforms the LoSPN Kernel and Tasks to use actual buffers in the form of `memrefs` instead of abstract `tensors`. To this end, the signature of the Kernel and Tasks is transformed to use `memrefs`, `tensor`-typed results are replaced by stores to buffers provided as output arguments. An additional pass also optimizes the bufferized LoSPN to avoid unnecessary copies, e.g., by writing directly to the final output buffer of the Kernel instead of copying an intermediate result buffer. For the eventual de-allocation of all intermediate buffers, the MLIR-provided *BufferDeallocation* pass is used.

Next to bufferization, the LoSPN module is also optimized by leveraging a combination of LoSPN-specific optimizations (e.g., constant folding) through the MLIR canonicalization framework, and dialect-agnostic optimizations such as common subexpression eliminiation (CSE).

The result of all the compilation steps described in this section is the transformation from a HiSPN input as given in Fig. 2 to the LoSPN example in Fig. 3, which would then serve as input for the further target-specific lowerings.

B. CPU Target Lowering

For further lowering towards the CPU target, the LoSPN Kernels and Tasks are each lowered to a function, and the Kernel function will call the Task functions in an appropriate order. Inside the Task functions, a loop is generated to process all inputs in a batch. The actual SPN operations are lowered to the corresponding arithmetic operations, depending on the type used for computation, e.g., if computation on log-space was requested. For discrete distributions as leaf nodes, a table look-up is generated, and for continuous distributions, the probability density function is calculated. Fig. 4 shows an excerpt for the CPU target lowering for the LoSPN code in Fig. 3.

While this lowering to native code is already able to provide significant speedups over the Python/numpy-based inference of the SPFlow library (cf. Section V), it still leaves much performance potential unused. Many modern CPUs come with a SIMD unit for short vector computations. To leverage the performance potential of these units, the compiler can optionally generate *vectorized* code through explicit vectorization in MLIR, using the MLIR vector type and dialect. During vectorization, the loop inside each task function is vectorized using the *data-parallel* approach, and computes the results for multiple input samples in parallel, using the maximum number of elements possible for the data type on the CPU's hardware. The vectorized loop is complemented by a scalar epilogue loop that handles the left-over elements in case the number of elements in the vector does not evenly divide the number of inputs. For vectorization, the information about the access pattern present in the LoSPN dialect can also be used to generate a more efficient combination of regular vector loads and shuffle instructions instead of gather loads.

The second major source of parallelism on modern CPUs, namely multi-threading, is not directly used in the generated code. Instead, the runtime component, which will load and execute the generated code, will split the input data into

```
1  module {
2    func @task_0(%arg0: memref<?x2xf32>,
3                  %arg1: memref<1x?xf32>) {
4      %0 = memref.dim %arg0, %c0 : memref<?x2xf32>
5      %c1 = constant 1 : index
6      scf.for %arg2 = %c0 to %0 step %c1 {
7        %1 = memref.load %arg0[%arg2, %c0] ...
8        ...
9        %37 = addf %32, %36 : f32
10       memref.store %37, %arg1[%c0, %arg2] ...
11     }
12     return
13   }
14   func @spn_cpu(%arg0: memref<?x2xf32>,
15                 %arg1: memref<1x?xf32>) {
16     call @task_0(%arg0, %arg1) ...
17     return
18   }
19 }
```

Fig. 4. Excerpt of the result for the lowering of the LoSPN code in Fig. 3 for the CPU target, without vectorization. For example, lines 8 and 11 directly correspond to lines 6 and 18 in Fig. 3, respectively.

multiple chunks and use multiple threads to process these chunks in parallel. In this case, the user-provided *batch size* is used as size for the chunks. Note that the batch size is a mere optimization hint, the generated kernel can still process an arbitrary number of inputs.

The generated code after the lowering is represented by a combination of multiple dialects provided by the MLIR framework, namely the *Standard*, *Math*, *SCF*, *Vector*, and *MemRef* dialects. This combination of dialects is lowered to the LLVM dialect through a series of MLIR-provided lowering passes, and then translated to LLVM IR. This LLVM IR is then optimized further by the LLVM framework, before eventually being translated to object code, which the runtime component can load for execution. In case of vectorization, the object code is also linked with vector libraries such as Intel SVML or GLIBC Libmvec, which provide optimized vector implementations for elementary functions, e.g., `log`. As the LLVM framework is used for compilation, the compiler can support any CPU featured by the LLVM framework. Vectorization, though, is currently only supported on x86 and ARM Neon.

C. GPU Target Lowering

When lowering the LoSPN dialect for the GPU target, a GPU kernel function is generated for each Task. The LoSPN Kernel, on the other hand, is lowered to a regular function, which will remain on the host and will coordinate data transfers between host and GPU as well as execution of the GPU kernels.

Inside each GPU kernel, code for the computation of a single input of the corresponding Task is generated. This way, the computation for a batch of inputs is explicitly parallelized across the large number of threads available on a GPU. The lowering for the SPN DAG nodes is similar to the CPU case, with the exception of discrete univariate distributions, which are lowered into a cascade of select operations, instead of a table-based lookup. Fig. 5 lists an excerpt for the lowering of the LoSPN code in Fig. 3 and shows the host function for the LoSPN Kernel and a GPU kernel for a Task.

```
1  module attributes {gpu.container_module} {
2    func @spn_gpu(%arg0: memref<?x2xf32>,
3                      %arg1: memref<1x?xf32>) {
4      %memref = gpu.alloc  (%0) : memref<?x2xf32>
5      %memref_0 = gpu.alloc  (%1) : memref<1x?xf32>
6      gpu.memcpy  %memref, %arg0 : memref<?x2xf32>,
7                                  memref<?x2xf32>
8      gpu.launch_func  @spn_gpu::@spn_gpu_kernel
9              blocks in (%3, %c1, %c1)
10             threads in (%c64, %c1, %c1)
11             args(%2 : index,
12                     %memref : memref<?x2xf32>,
13                     %memref_0 : memref<1x?xf32>)
14     gpu.dealloc  %memref : memref<?x2xf32>
15     gpu.memcpy  %arg1, %memref_0 : ...
16     gpu.dealloc  %memref_0 : memref<1x?xf32>
17     return
18   }
19   gpu.module @spn_gpu {
20     gpu.func @spn_gpu_kernel(
21         %arg0: index,
22         %arg1: memref<?x2xf32>,
23         %arg2: memref<1x?xf32>) kernel {
24       %0 = "gpu.block_id"() {dim = "x"} ...
25       %3 = "gpu.thread_id"() {dim = "x"} ...
26       br ^bb1
27     ^bb1:  // pred: ^bb0
28       scf.if %14 {
29         %15 = memref.load %arg1[%13, %c0]
30         ...
31         %52 = addf %46, %51 : f32
32         memref.store %52, %arg2[%c0, %13]
33       }
34       gpu.return
35     }
36   }
37 }
```

Fig. 5. Excerpt of the result for the lowering of the LoSPN code in Fig. 3 for the GPU target, showing the Kernel function remaining on the host and one GPU kernel corresponding to a LoSPN Task. For example, lines 32 and 35 directly correspond to lines 6 and 18 in Fig. 3, respectively.

Also during the GPU target lowering, the memory access semantics of the LoSPN dialect can be used for optimization. The compiler tries to eliminate unnecessary copies of intermediate results buffers between GPU and host. Instead of copying an intermediate result to the host after computing it and back to the GPU before using it again, a compiler pass will eliminate these copy operations and instead re-use the result buffer already present on the GPU for tasks consuming this result. This approach can remove a significant number of expensive copy operations, resulting in improved performance.

The compiler currently only supports CUDA GPUs for the execution, although the result of the lowering from LoSPN uses generic GPU abstractions and could also be used to target GPUs from other vendors.

Similar to the CPU target lowering, a combination of MLIR dialects is used to represent the result of the lowering, in this case the *Standard*, *Math*, *SCF*, *GPU*, and *MemRef* dialect.

Later on, the GPU and host portion of the generated code are separated from each other. The GPU portion is lowered to NVVM IR through a series of MLIR-provided lowerings and translations. This NVVM IR is linked with `libdevice`, which contains optimized implementations for elementary functions (e.g., `log`), similar to the vector libraries used for the CPU compilation.

Later, the NVVM IR is compiled into PTX assembly using the LLVM framework, and eventually down to the CUBIN binary format. The content of the CUBIN module is then attached to the host module in binary format. The remaining host module undergoes a similar series of transformations as described for the CPU target above, resulting in a object file that can be loaded and executed by the runtime component.

V. Evaluation

In our experimental evaluation, we are going to investigate two different applications of Sum-Product Networks.

We use two different systems for the evaluation: Most of the 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. As this system does not have AVX-512 ISA extensions, we use a second system to evaluate CPU vectorization on AVX-512, which is a dual-socket system with two Intel Xeon Platinum 9242 CPUs, 384 GB RAM, running CentOS 8 with kernel 4.18. For the Ryzen machine, we are going to use GLIBC Libmvec version 2.31 as vector library, for the Xeon system, we use Intel SVML version 2021.1.

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.

A. Application 1: Robust Automatic Speaker Identification

Our aim is to evaluate the compiler with a *real-world* application of Sum-Product Networks, therefore we are using the SPNs from [11] 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 each potential speaker, a separate SPN is trained using speech samples. To determine the speaker of a particular speech sample, one can evaluate the SPN for each speaker with the speech sample, and then compare the output likelihood to determine the speaker for the sample. For our evaluation, we are going to use two different scenarios from the paper by Nicolson et al. [11], namely the identification on clean speech samples (245567 samples) and identification on noisy speech samples with marginalization (1227835 samples). Using the open-source release of the speaker identification by Nicolson et al. [12], the SPNs were reproduced. We validate the results produced by the compiled kernel through direct comparison with the original output data for each sample and each speaker. 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 average size of the SPN is 2569 operations and around 49% of the operations are Gaussian leaf nodes.

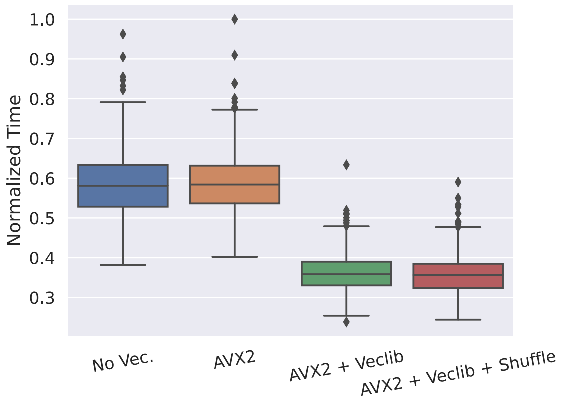

Fig. 6. Comparison of different vectorization configurations for clean speech samples, normalized against the maximum execution time across all configurations.

1) Compiler Configuration: In a first step, the best configuration for the compiler is determined. When compiling for CPUs, the mapping strategy and execution can be configured by a number of parameters, and we evaluate multiple variants to determine the best configuration.

- *No Vec.*: No vectorization is performed.
- *AVX2*: Vectorization for AVX2 ISA extension.
- *+Veclib*: Use a vector library (Intel SVML, GLIBC Libmvec) for optimized implementations of primitive math functions (e.g., `log`) in vector code.
- *+Shuffle*: Use a combination of vector loads and shuffles instead of separate gather loads (cf. Section IV-B) for vectorization.

As shown in Fig. 6, vectorization without use of a vectorized function library actually leads to an *increase* in execution time, as all individual values need to be extracted from the vector, the function (e.g., log) is invoked with scalar values, and the results are then inserted back into the vector again. Using the optimized functions from the vector library leads to a significant improvement. The use of loads and shuffles, instead of gather loads, yields another, though small, reduction in execution time.

When compiling for GPU, the most important parameter is the user-provided batch size, which will be used as the constant block size for the GPU kernel launches. After evaluating a range of different batch sizes, it becomes clear that a small block size of 64 is preferable.

2) Performance Comparison: Now that the best configurations have been determined for the compile flow, we will compare the execution time of the compiled kernel against SPFlow's performance when executing in Python with numpy. In addition, we use SPFlow's feature to also translate to a Tensorflow graph to compare with the execution of the translated Tensorflow graph (on both CPU and GPU).

For the compiler, the comparison includes the configuration with no vectorization on CPU, and compiling for CPU using

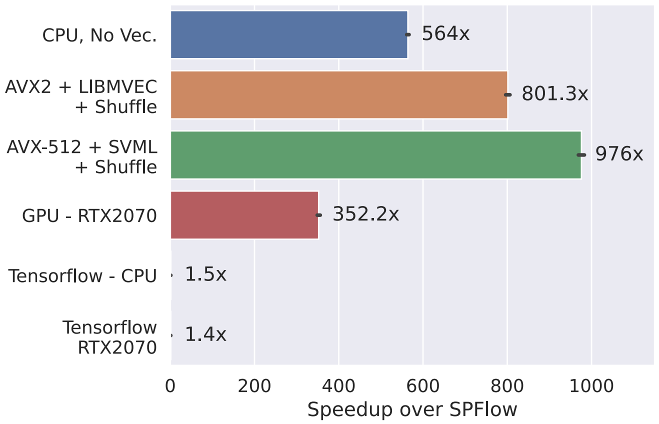

Fig. 7. Performance comparison for clean speech samples, given as speedup over execution in SPFlow.

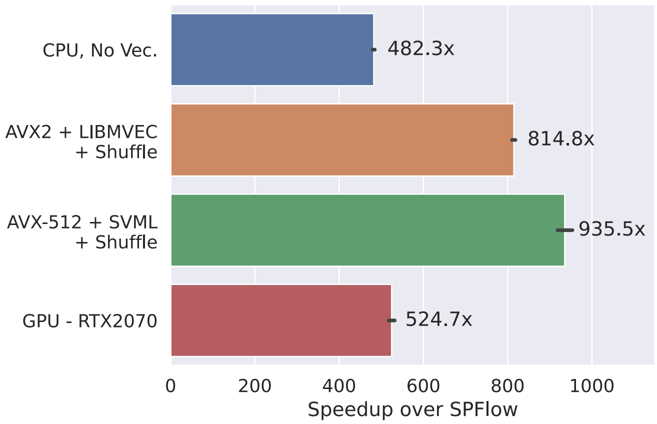

Fig. 8. Performance comparison for noisy speech samples, given as speedup over execution in SPFlow.

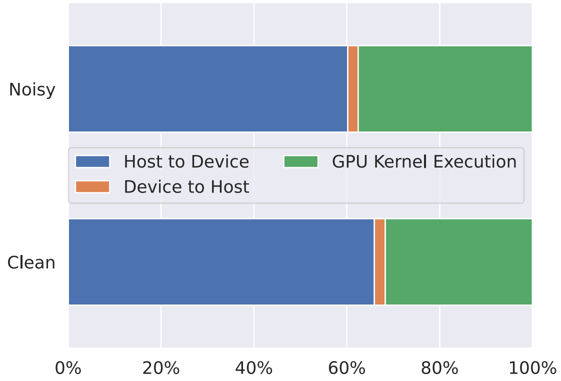

Fig. 9. Portion of time spent on data movement and execution for GPU execution.

vectorization, a vector library, and loads-and-shuffles instead of gathers, on both AVX2 and AVX-512. All CPU configurations are using a batch size of 4096. For the GPU, a batch size of 64 is used.

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).

Fig. 7 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, our compiler 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 (AVX2) and 976x (AVX-512), respectively. Compilation for the GPU achieves an average speedup of 352x.

Fig. 8 shows the same comparison for the noisy speech samples. 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, our compiler again achieves large speedups, with an average of 482x without vectorization and 814x (AVX2) and 935x (AVX-512), respectively, with vectorization. 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, can be found in Fig. 9: Data movements between host and device 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, the use of our compiler for CPU and GPU achieves speedups of multiple orders of magnitude in comparison to SPFlow's Python-only evaluation, benefiting ML experts when running inference on Sum-Product Networks.

B. Application 2: Random Sum-Product Networks

As a second application for Sum-Product Networks, we will investigate *Random* Sum-Product Networks, as presented by Peharz et al. in [13]. The special structural requirements of general Sum-Product Networks can make (structure) learning complicated and costly. To overcome this limitation, Peharz et al. construct a *random* SPN structure with additional structure constraints, and employ techniques from “classical” machine learning, such as automatic differentation for faster weight learning. Peharz et al. apply this approach to multiple machine learning tasks and achieve prediction results comparable to deep neural networks.

This benchmark was mainly chosen as a stress test for the compiler, since the resulting large SPNs allow us to evaluate

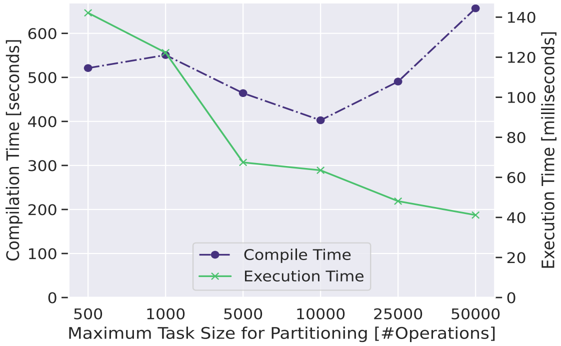

Fig. 10. Impact of the maximum partition size during graph partitioning on compilation time and execution time for CPUs.

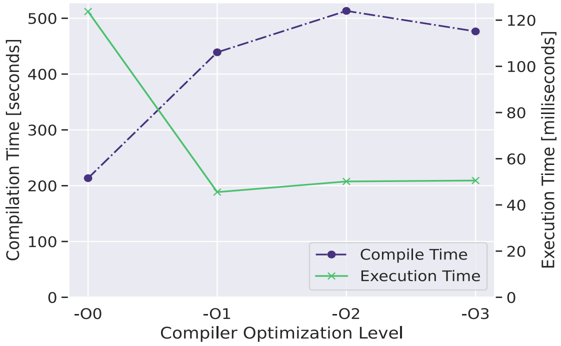

Fig. 11. Impact of the compiler optimization level on compilation time and execution time for CPUs.

the scaling of the tool to very large networks.

Based on the original implementation by Peharz et al. [14], we train RAT-SPNs (Random Tensorized SPNs) for the popular MNIST and fashion-MNIST dataset, which were also used in [13]. The trained models achieve an accuracy of 94.3% and 86.3%, respectively.

1) Compilation Time: While the size of the SPN DAGs for the speaker identification above averaged around 2500 operations, and compilation typically required a few seconds, the RAT-SPNs are much larger. After conversion to SPFlow, the RAT-SPN contains a *separate* SPN for each of the ten possible output classes for the benchmarks. Each of these SPNs, which we will compile separately, contains about 165,000 leaf nodes, around 170,000 product nodes and more than 3,000 sum nodes.

To keep compilation times within a reasonable limit, graph partitioning (cf. Section IV-A4) needs to be used to split the large input DAG into multiple LoSPN Tasks. This graph partitioning can be configured with the maximum size of allowed for each partition, i.e., the number of SPN DAG operations inside this partition.

Figs. 10 and 12 show the impact of the maximum partition size on compilation time and execution time. For the CPU, increasing the maximum partition size first decreases the compilation time up until 10,000 operations, after which it increases again. The execution time improves with increasing partition size, as fewer partitions means that fewer intermediate results have to be communicated via memory buffers between Tasks. For the further investigation of the CPU compilation, we select a max. partition size of 25k operations as the best trade-off.

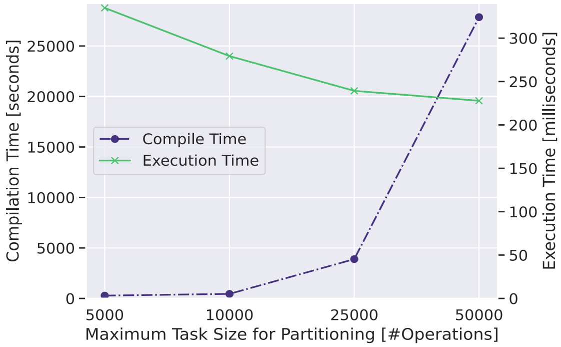

Fig. 12. Impact of the maximum partition size during graph partitioning on compilation time and execution time for GPUs. Note the significantly different different scale cf. Fig. 10.

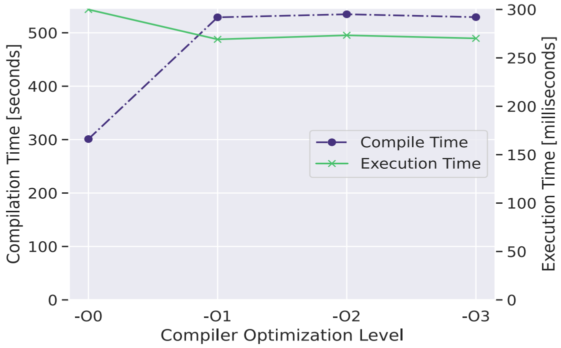

Fig. 13. Impact of the compiler optimization level on compilation time and execution time for GPUs.

For the GPU, a smaller number of partition sizes are investigated, as small GPU kernels incur too much overhead for launch and communication between host and GPU. With increasing partition size, the compilation time increases drastically, while the execution time improves at a much slower rate. We therefore choose a maximum partition size of 10,000 operations as the best trade-off for GPUs.

Another parameter that has a significant impact on the compilation time is the compiler optimization level, which is mainly used for the LLVM IR optimization and translation to object code. Figs. 11 and 13 show the impact of different optimization levels on compilation time and execution time. In both cases, `-O0` yields the shortest compilation time, but also the longest execution time. Optimization levels `-O1` to `-O3` significantly increase the compilation time, but also improve execution time at a similar rate. The differences between these three optimization levels is relatively small, therefore we pick `-O1` as the optimization levels for further experiments.

With the chosen configurations, the compilation time for both platforms amounts to about 500 seconds (roughly 8 minutes). For the CPU compilation, most of this time (around 75%) is spent on the translation to object code. Here, the LLVM DAG Instruction Selection (27%) and the Greedy Register Allocator (25%) require most of the time.

For the GPU compilation, the vast majority of the compilation time (around 95%) is spent on the translation of the PTX assembly to the CUBIN format via the CUDA API.

As both parameters, the maximum partition size and the compiler optimization level, can easily be controlled by users through the Python interface, this gives the user the opportunity to trade compilation time for performance of the generated executable, depending on their current stage of development.

2) Performance Comparison: In contrast to the generic SPN models in Section V-A, RAT-SPNs are natively implemented in Tensorflow. Therefore, the Tensorflow-based execution yields *much* better performance for these models than for generic SPNs. The Tensorflow execution on the GPU performs the classification of 10,000 images in 0.427 and 0.426 seconds, for MNIST and fashion-MNIST, respectively. As the random structure for both tasks is identical and only the weights differ, the performance is also virtually identical.

Our compilation for CPUs performs on-par with Tensorflow on the GPU, with 0.444 and 0.437 seconds. Our compilation for GPUs is slower, completing the tasks in 1.299 and 1.31 seconds. When comparing our compiler to the Tensorflow execution, one has to consider, though, that Tensorflow executes the entire RAT-SPN in one run, whereas our compiler needs to run ten distinct SPNs (one per class) after the conversion to SPFLow. For the compilation for the GPU, in particular, this means that the input data has to transferred ten times and ten distinct launches have to take place. This is one of the reasons our GPU compilation is not quite able to perform on-par with Tensorflow for the more restricted RAT-SPNs.

Nevertheless, the compilation for both platforms still outperforms the Tensorflow execution on the CPU, which requires 1.72 and 1.742 seconds to complete the task.

When comparing the performance of the native Tensorflow implementation and our compiler, one has to consider that the high performance of Tensorflow for RAT-SPNs is achieved only since the structure of RAT-SPNs is more constrained than that of general SPNs. If an application, such as the automatic speaker identification [11] in Section V-A, needs more general SPNs that cannot be constrained appropriately, our SPN compiler will allow much faster inference.

VI. Related Work

The most popular open-source library for use of and experimentation with Sum-Product Networks is SPFlow [4]. It supports to learn Sum-Product Networks, including their structure, directly from data, as well as the manual construction of Sum-Product Networks using a DSL-like syntax embedded in Python. SPFlow also supports inference through a Python implementation which employs numpy and, for cases where no marginalization is required, through a translation to a Tensorflow graph. As our evaluation in Section V-A has shown, the compiler can provide multiple orders of magnitude faster inference than these mechanisms.

Still, given the popularity of SPFlow, our compiler tightly integrates with SPFlow and its internal representation is used for the Python interface of the compiler. This way, machine learning experts can easily leverage the compilation presented in this work after learning an SPN using SPFlow.

Another approach for efficient training and inference for Sum-Product Networks, that was used by the second application [13] in our evaluation (cf. Section V-B) and also by [15], [16], is the *direct* implementation of Sum-Product Networks in a framework such as Tensorflow or Pytorch. As demonstrated in Section V-B, our compiler still outperforms the direct implementation in Tensorflow for inference on the CPU, with the compiled CPU executables being competitive with Tensorflow's GPU execution.

A custom, FPGA-based hardware accelerator for Sum-Product Network inference was developed by Sommer et al. [17] and was able to demonstrate significant improvements in inference throughput. However, their accelerator only supports discrete, histogram-based leaf node distributions, which map to FPGA look-up tables naturally, and, as such, is not suitable for the two applications investigated in the evaluation, as those use continuous, Gaussian leaf node distributions.

VII. Conclusion

In this work, we have presented an MLIR-based compiler for fast Sum-Product Network inference on CPUs and GPUs. Two SPN-specific dialects were developed to capture the high-level semantics of Sum-Product Network inference in the MLIR framework, and serve as the starting point for a number of target-independent transformations. These transformations, which include a graph partitioning, enable the compiler to handle even very large SPN graphs, by preparing the input for the automated lowering to CPUs and GPUs. The lowering enables the compiler to make best use of the available platform features, e.g., SIMD vector units on the CPU.

In the first part of our evaluation, using an SPN-based automatic speaker identification as example application, we have demonstrated that the compiler can achieve speedups of up to 976x (CPU) and 524x (GPU) over the existing inference implementations in the popular open-source library for SPN implementation SPFlow.

In the second part, using image classification as an example task, we have investigated the impact of the compiler parameters on compilation and execution time and also demonstrated that the compiler is able to provide performance on-par with a direct implementation of Sum-Product Networks in Tensorflow.

The compiler tightly integrates with the popular SPFlow library and allows machine learning experts to automatically compile Sum-Product Networks for fast inference on CPUs and GPUs after training or constructing them using SPFlow.

Acknowledgements

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.

References

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

[2] "XLA - TensorFlow, compiled." [Online]. Available: https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html

[3] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, "Glow: Graph lowering compiler techniques for neural networks," 2019.

[4] 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.

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

[6] R. Peharz, S. Tschiatschek, F. Pernkopf, and P. Domingos, "On Theoretical Properties of Sum-Product Networks," in *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, G. Lebanon and S. V. N. Vishwanathan, Eds., vol. 38. San Diego, California, USA: PMLR, 09–12 May 2015, pp. 744–752. [Online]. Available: https://proceedings.mlr.press/v38/peharz15.html

[7] F. Rathke, M. Desana, and C. Schnörr, "Locally adaptive probabilistic models for global segmentation of pathological oct scans," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2017, pp. 177–184.

[8] K. Zheng, A. Pronobis, and R. P. N. Rao, "Learning semantic maps with topological spatial relations using graph-structured sum-product networks," *CoRR*, vol. abs/1709.08274, 2017. [Online]. Available: http://arxiv.org/abs/1709.08274

[9] 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.

[10] O. Moreira, M. Popp, and C. Schulz, "Evolutionary multi-level acyclic graph partitioning," *Journal of Heuristics*, vol. 26, no. 5, pp. 771–799, 2020.

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

[12] ——, "Spn-asi," https://github.com/anicolson/SPN-ASI, 2020.

[13] 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.

[14] ——, "Rat-spn," https://github.com/cambridge-mlg/RAT-SPN, 2019.

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

[16] 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.

[17] 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.