KLay: Accelerating Sparse Arithmetic Circuits

Jaron Maene[1], **Vincent Derkinderen**[1], **Pedro Zuidberg Dos Martires**[2]
[1]KU Leuven, Belgium
[2]Örebro University, Sweden
jaron.maene@kuleuven.be, vincent.derkinderen@kuleuven.be, pedro.zuidberg-dos-martires@oru.se

Abstract

A popular approach to neurosymbolic AI involves mapping logic formulas to arithmetic circuits (computation graphs consisting of sums and products) and passing the outputs of a neural network through these circuits. This approach enforces symbolic constraints onto a neural network in a principled and end-to-end differentiable way. Unfortunately, arithmetic circuits are challenging to run on modern AI accelerators as they exhibit a high degree of irregular sparsity. To address this limitation, we introduce knowledge layers (KLAY), a new data structure to represent arithmetic circuits that can be efficiently parallelized on GPUs. Moreover, we contribute two algorithms used in the translation of traditional circuit representations to KLAY and a further algorithm that exploits parallelization opportunities during circuit evaluations. We empirically show that KLAY achieves speedups of multiple orders of magnitude over the state of the art, thereby paving the way towards scaling neurosymbolic AI to larger real-world applications.

1 Introduction

Probabilistic neurosymbolic models combine neural networks with arithmetic circuits. This approach, pioneered by Xu et al. (2018) and Manhaeve et al. (2018), performs probabilistic inference on the outputs of neural networks in order to enforce logical guarantees and exploit background knowledge during learning.

While arithmetic circuits are end-to-end differentiable, they also pose certain challenges. In particular, arithmetic circuits encoding logical knowledge are challenging to formulate in terms of dense tensor operations as they exhibit a high degree of unstructured sparsity. This means that in contrast to neural networks, existing neurosymbolic models using circuits have struggled to leverage modern GPU or TPU hardware. In this work, we address this challenge using KLAY: a new data structure representing arithmetic circuits as **k**nowledge **lay**ers which can exploit parallel compute.

The main advantage of KLAY is that it reduces arithmetic circuit evaluations to a sequence of index and scatter operations – operations already present in popular tensor libraries. This makes KLAY completely agnostic towards the underlying hardware. By leveraging the compiler stacks of open-source tensor libraries, KLAY can furthermore considerably outperform existing hand-written CUDA kernels.

2 Arithmetic Circuits & Neurosymbolic AI

The symbolic knowledge in neurosymbolic AI is commonly specified as Boolean logic. *Boolean circuits* are a compact representation of Boolean logic using directed acyclic graphs (Darwiche 2021). More specifically, the leaves in Boolean circuits correspond to Boolean variables (or their negation), while inner nodes are either $\wedge$-gates or $\vee$-gates. We make the usual assumption that Boolean circuits do not contain negation, known as *negation normal form* (NNF). Figure 1 (left) contains an example of a Boolean circuit. A circuit can be evaluated for a set of inputs by a simple post-order traversal of the graph, meaning children get evaluated before their parents. More formally, the evaluation of a circuit is a Boolean function $\mathbb{B}^N \to \mathbb{B}$ which maps the Boolean input values to the value of the root node.

In the context of neurosymbolic AI, we commonly perform a so-called knowledge compilation step (Darwiche and Marquis 2002) prior to evaluation. This compilation step transforms an NNF circuit into *deterministic decomposable negation normal form* (d-DNNF), which guarantees that certain computations can be performed tractably. We refer to Vergari et al. (2021) for an in-depth discussion on tractable computations on circuits.

Importantly, d-DNNF circuits allow linear time probabilistic inference. Assume first that the Boolean variables are no longer deterministic but instead constitute Bernoulli random variables. We can then compute the probability of the d-DNNF circuit evaluating to true under the input distribution by labeling the leaves of the circuit with the probabilities of the Boolean variables and replacing $\wedge$- and $\vee$-gates with $\times$ and $+$ operations, respectively. The resulting circuit is also called an arithmetic circuit (Darwiche 2003).

While probabilistic inference on d-DNNF circuits has linear time complexity, transforming an NNF circuit into d-DNNF is #P-hard (Valiant 1979). However, once the d-DNNF structure is obtained we can re-evaluate the circuit with different probabilities for the Boolean variables in the leaves. This compile once and evaluate often paradigm has gained traction in neurosymbolic systems (Manhaeve et al.

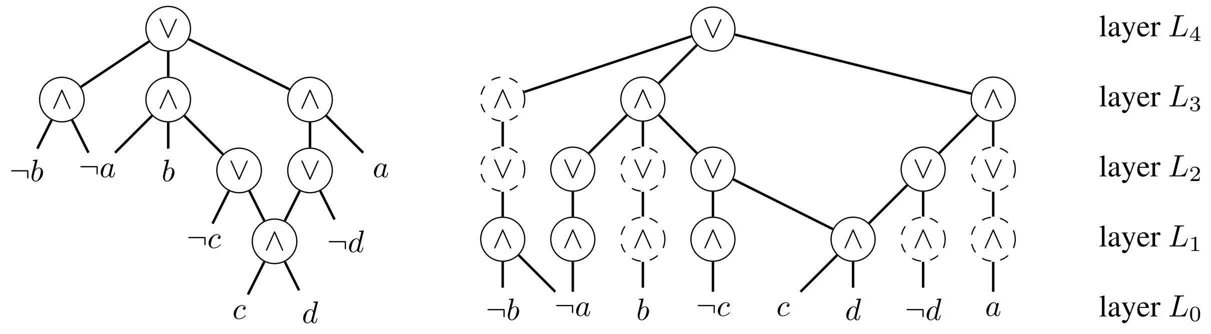

Figure 1: Example of (left) a Boolean d-DNNF circuit, and (right) the corresponding layerized circuit.

2018; Ahmed et al. 2022b; De Smet et al. 2023). The high-level idea behind these neurosymbolic models is to compile the symbolic knowledge once into an arithmetic circuit and let a neural network predict the probabilities to be fed into the arithmetic circuit. Given that the arithmetic circuit consists only of sum and product operations, the resulting computation graph (neural network + arithmetic circuit) is end-to-end differentiable and the parameters can be optimized using standard gradient descent methods.

As discussed in the introduction, arithmetic circuits currently hinder the efficient application of neurosymbolic methods as these circuits are not well-suited to modern AI accelerators. As a matter of fact, the standard way to evaluate an arithmetic circuit in a neurosymbolic context is to naively evaluate every node one by one (Manhaeve et al. 2018; Ahmed et al. 2022a,b) using a naive traversal of the arithmetic circuit. Although such a traversal of the circuit allows for a certain degree of data parallelism, it fails to fully utilize the capacity of modern GPUs.

3 Layerizing Circuits

In this section, we show how to map an arithmetic circuit to our layerized KLay representation. Afterwards, in Section 4, we discuss how the resulting KLay representation can be run efficiently on GPUs.

In order to parallelize a circuit, we group the nodes into sets of nodes that can be evaluated in parallel. We dub these groups layers – reminiscent of layers in neural networks. Concretely, for each node n in a circuit C we compute its height in the circuit h_n, and nodes with the same height are assigned to the same layer.

$$L_i = \{n \in C \mid h_n = i\}$$
$$\text{where} \quad h_n = \begin{cases} 0, & \text{if } n \text{ is a leaf,} \\ \max_{c \in \mathcal{C}_n} h_c + 1 & \text{otherwise.} \end{cases}$$

Here, we use $\mathcal{C}_n$ to denote the set of children of a node n. Note that the height of all nodes can be efficiently evaluated in a single post-order circuit traversal. The initial layer, L_0, comprises all leaf nodes, while the last layer comprises the root node.

Without loss of generality, we can assume that $\vee$-gates only have $\wedge$-gates as children and that $\wedge$-gates have either $\vee$-gates or leaf nodes as children (Choi, Vergari, and Van den Broeck 2020). This implies that $\wedge$- and $\vee$-gates appear in an alternating fashion throughout the circuit and that all nodes in the same layer have the same type.

If nodes are assigned to layers based on their height h_n, the child of a node can be in any of the previous layers. However, to transform the circuit evaluation into a sequence of parallel operations, it is more convenient if all children are in the immediately preceding layer. In such a structure, the next layer can be computed solely using the current layer.

We obtain this layer-by-layer structure by introducing additional unary nodes. Whenever a node $n \in L_{h_n}$ has a child c in a non-immediately preceding layer L_{h_c}, i.e. $h_c + 1 < h_n$, we introduce a chain of unary nodes, one per layer between L_{h_c} and L_{h_n}, to connect n to c via these unary nodes. This is illustrated in Figure 1 (right), where the newly introduced nodes are indicated by dashed circles. Note that the type of a unary node ($\vee$ or $\wedge$) is irrelevant and chosen to satisfy the assumption of alternating node types. Algorithm 2 in Appendix B summarizes this layerization in pseudo-code.

4 Tensorizing Layered Circuits

In the previous section, we organized circuits into layers by assigning each node n in the circuit a height h_n. We proceed in this section with mapping the layered linked nodes to a layered computation graph where layers are evaluated sequentially and computations within a layer can be parallelized.

To this end, we make a simple, yet powerful, observation: the current layer can be computed from the output of the previous layer by only using *indexing and aggregation*. To see this, we first impose an arbitrary order on the nodes within each layer. This means we can write the values of all nodes in a layer L_i as a vector $\mathbf{N}_i$. Now, to specify the computation in a layer we use two vectors of indices: $\mathbf{S}_i$ and $\mathbf{R}_i$. For each edge between $\mathbf{N}_{i-1}$ and $\mathbf{N}_i$, $\mathbf{S}_i$ contains the index of the input node, while $\mathbf{R}_i$ contains the index of the output node. We exemplify this for a single layer in Figure 2.

To compute $\mathbf{N}_i$, we first select relevant values from $\mathbf{N}_{i-1}$ using as index $\mathbf{S}_i$, giving us $\mathbf{E}_i = \mathbf{N}_{i-1}[\mathbf{S}_i]$. The vector $\mathbf{E}_i$ essentially contains the values of all the edges between $\mathbf{N}_i$ and $\mathbf{N}_{i-1}$. Next, we need to correctly segment the edges $\mathbf{E}_i$ and aggregate the individual segments – either by using

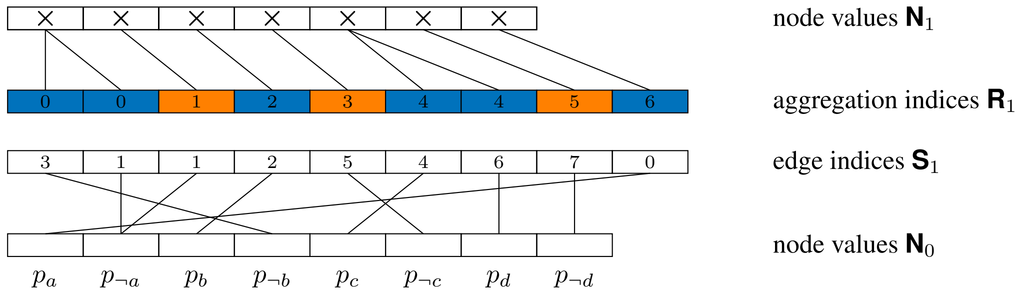
node values $\mathbf{N}_1$
aggregation indices $\mathbf{R}_1$
edge indices $\mathbf{S}_1$
node values $\mathbf{N}_0$
0 0 1 2 3 4 4 5 6
3 1 1 2 5 4 6 7 0
p_a $p_{\neg a}$ p_b $p_{\neg b}$ p_c $p_{\neg c}$ p_d $p_{\neg d}$

Figure 2: The evaluation of the first layer in Figure 1 (right), as an indexing and aggregation operation. The symbols $\mathbf{N}_0$ and $\mathbf{N}_1$ denote the node values at layer L_0 and L_1 respectively. First, we index in $\mathbf{N}_0$ using the edge indices $\mathbf{S}_1$, effectively creating a vector with the values of all edges in between $\mathbf{N}_0$ and $\mathbf{N}_1$. Next, the aggregation indices $\mathbf{R}_1$ determine what edges are reduced together. As a result we obtain the node values $\mathbf{N}_1 = \text{scatter}(\mathbf{N}_0[\mathbf{S}_1], \mathbf{R}_1, \text{reduce} = \text{'product'})$.

sums or products, depending on the layer. This is done using $\mathbf{R}_i$: all elements with the same index in $\mathbf{R}_i$ are reduced together. Fortunately, such segment-reduce operations are implemented as primitives in various tensor libraries such as Jax, TensorFlow, or PyTorch (Abadi et al. 2015; Paszke et al. 2019; Bradbury et al. 2018). In Figure 2, these segments are indicated with alternating colors.

Algorithm 1: KLay Forward Evaluation

```
Input: $\mathbf{N}_0$, selection indices $\mathbf{S}_1, \mathbf{S}_2, \dots, \mathbf{S}_L$,
       reduction indices $\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_L$
for $i \leftarrow 1$ to $L$ do
    $\mathbf{E}_i \leftarrow \mathbf{N}_{i-1}[\mathbf{S}_i]$;
    if $i \bmod 2 = 0$ then
        $\mathbf{N}_i \leftarrow \text{scatter}(\mathbf{E}_i, \mathbf{R}_i, \text{reduce='sum'})$;
    else
        $\mathbf{N}_i \leftarrow \text{scatter}(\mathbf{E}_i, \mathbf{R}_i, \text{reduce='product'})$;
    end
end
return $\mathbf{N}_L$;
```

Algorithm 1 contains pseudo-code for the layerwise circuit evaluations, where we use the common `scatter` function to segment and aggregate the $\mathbf{E}_i$ vectors.

5 Experimental Evaluation

We implement KLAY as a Python library supporting two popular tensor libraries: PyTorch and Jax. We evaluate the runtime performance of KLAY on several synthetic benchmarks and neurosymbolic experiments. All experiments were conducted on the same machine, with an NVIDIA GeForce RTX 4090 as GPU and an Intel i9-13900K as CPU.

Benchmarks We consider the performance of KLAY on a set of synthetic circuits, by randomly generating logical formulas in 3-CNF. We compile the 3-CNF formulas into d-DNNF circuits, more specifically SDD circuits, using the PySDD library (Meert and Choi 2017). By changing the number of variables and clauses in the CNF, we vary the size of the compiled circuits over 5 orders of magnitude. Figure 3 compares the performance of KLAY with the native post-order traversal from PySDD implemented in C. We report results for both the real and logarithmic semiring. As JUICE does not support the logarithmic semiring and Jax does not support backpropagation on scatter multiplication, these are excluded from the respective comparisons.

In Appendix A, we repeat the same experiment using the D4 knowledge compiler (Lagniez and Marquis 2017) instead of PySDD.

Results Our results in Figure 3 indicate that on large circuits, KLAY on GPU outperforms all baselines with over one order of magnitude. Due to SIMD and multi-core parallelization, KLAY on CPU is also considerably faster than the baselines. JUICE does not include results for the largest instances due to a timeout after 30 minutes. KLAY attains best results with Jax, due to its superior JIT compilation and kernel fusion.

6 Related Work

The closest related work is the arithmetic circuit layerization present in JUICE (Dang et al. 2021). Similar to our circuit layerization scheme, JUICE takes a Boolean circuit and maps it to a set of layers that can be evaluated sequentially, although not layer per layer. To this end, Dang et al. (2021) implemented a custom SIMD implementation for the CPU and custom CUDA kernels for the GPU. This is in contrast to KLAY where we reduce circuit evaluations to a sequence of index and scatter-reduce operations, which are already efficiently implemented in the modern deep learning stack. In our experiments, we show that KLAY dramatically outperforms JUICE in terms of run time on CPU and more importantly on GPU as well. Noteworthy here is that our experimental evaluation also shows that JUICE's GPU implementation is slower than their CPU implementation – hinting at missed parallelization opportunities.

The difficulty of running arithmetic circuits on GPUs was also pointed out by Shah et al. (2020, 2021). While they focused on developing hardware accelerators for arithmetic circuits, they also implemented custom circuit evaluations exploiting to a certain degree SIMD instructions and GPU parallelization. They found that CPU circuit evaluations out-

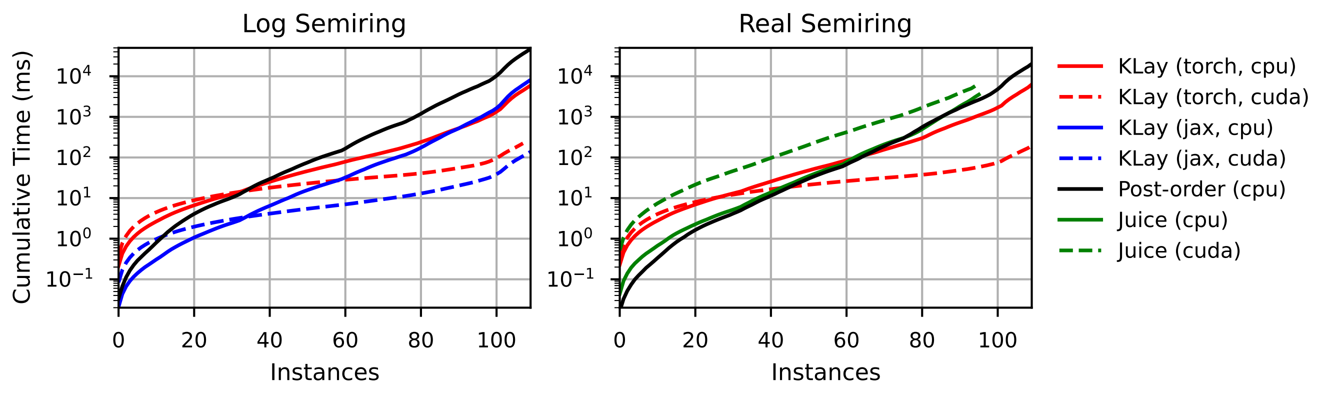

Figure 3: Cumulative runtime in milliseconds for the forward and backward pass on a circuit. That is, we plot the combined run time of the n fastest circuit evaluations against the number of n circuits. Timings are averaged over 10 runs per SDD. The SDDs are randomly generated from 3-CNF, where the difficulty is varied by the number of variables and clauses. The left and right figure show cumulative evaluation times for the logarithmic and real semiring, respectively.

performed the GPU implementations. Shah et al. (2020) concluded that arithmetic circuits were simply too sparse to be run efficiently on GPUs. By layerizing arithmetic circuits and interpreting the product and sum layers as indexing and scatter-reduce operations, we are able to refute this claim. We provide experimental evidence for this in Section 5.

Notable is also the work of Vasimuddin, Chockalingam, and Aluru (2018), who proposed an efficient CPU implementation for evaluating arithmetic circuits deployed on multiple CPU cores. They deemed an efficient GPU implementation to be impractical due to the high demands on memory bandwidth. We can again refute this claim.

Arithmetic circuits are closely related to the model class of probabilistic circuits (Vergari et al. 2021). The main difference is that the sum units for the latter are parameterized using mixture weights, while the former does not contain such weights. In order to run probabilistic circuits efficiently on the GPU, implementations usually rely on casting circuit evaluations as dense matrix-vector product (Peharz et al. 2020b,a; Galindez Olascoaga et al. 2019; Mari, Vessio, and Vergari 2023; Sommer et al. 2021). This idea has recently been generalized by Liu, Ahmed, and Van den Broeck (2024), who allow for the presence of block-sparsity in the matrix that encodes a layer. By exploiting this sparsity via custom kernels, they widened the class of probabilistic circuits that can be run efficiently on GPUs. Nevertheless, probabilistic circuits are usually far more dense than arithmetic circuits that are compiled from a logical theory. Unfortunately, this prevents the techniques developed for probabilistic circuits to be effective in the context of arithmetic circuits. Similarly, having densely parameterized sum nodes also limits the relevance of techniques developed for arithmetic circuits for probabilistic circuits, e.g. the indexing and segmenting scheme of KLAY circuit evaluations.

Besides algorithmic advances, specialized hardware solutions have also been proposed to deal with the irregularity of the computational graph of an arithmetic circuit (Dadu et al. 2019; Shah et al. 2020, 2021). However, these approaches have the drawback that they would require the purchase of non-commoditized hardware. While this could partially be remedied by the use of FPGAs, as done by Sommer et al. (2018); Weber et al. (2022); Choi et al. (2023), any custom hardware retains a communication overhead. Specifically, in the context of neurosymbolic AI, one needs to pass the output of a neural network to an arithmetic circuit. If the neural net and the circuit are on two different devices, e.g. a GPU and an FPGA, the latency of data transfer can counteract gains in evaluation speed.

7 Conclusions

The success of neural networks has been largely attributed to their scale (Kaplan et al. 2020), which is realized by their effective use of hardware accelerators. To compete, novel methods must run efficiently on the available hardware or risk losing out due to what Hooker (2021) coined the hardware lottery. We tackled this issue for probabilistic neurosymbolic AI by introducing KLAY– a new data structure to represent arithmetic circuits that is amenable to efficient evaluations on modern AI accelerators. Along with this representation, we contributed three algorithms for KLAY. The first two algorithms map the traditional linked node representation of arithmetic circuits to the corresponding KLAY representation (Algorithm 2 and 3). This representation is efficiently evaluated using the third algorithm, which exploits the parallelization opportunities (Algorithm 1).

Our experiments demonstrated that arithmetic circuits can be run efficiently on GPUs, despite their high degree of unstructured sparsity. To this end, a key aspect is the reduction of circuit evaluations to a sequence of indexing and scatter-reduce operations, as these can be implemented using highly optimized primitives available in modern tensor libraries. Resolving circuit evaluations as one of the major bottlenecks present in current neurosymbolic architectures allows to further scale neurosymbolic models.

Acknowledgments This research received funding from the Flemish Government (AI Research Program), the Flanders Research Foundation (FWO) under project G097720N, and the KU Leuven Research Fund (C14/18/062). PZD is supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. We thank Luc De Raedt for his valuable feedback.

References

Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Mané, D.; Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talwar, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Viégas, F.; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; and Zheng, X. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org.

Ahmed, K.; Li, T.; Ton, T.; Guo, Q.; Chang, K.-W.; Kordjamshidi, P.; Srikumar, V.; Van den Broeck, G.; and Singh, S. 2022a. Pylon: A pytorch framework for learning with constraints. In *NeurIPS 2021 Competitions and Demonstrations Track*, 319–324. PMLR.

Ahmed, K.; Teso, S.; Chang, K.-W.; Van den Broeck, G.; and Vergari, A. 2022b. Semantic probabilistic layers for neuro-symbolic learning. *Advances in Neural Information Processing Systems*, 35: 29944–29959.

Bradbury, J.; Frostig, R.; Hawkins, P.; Johnson, M. J.; Leary, C.; Maclaurin, D.; Necula, G.; Paszke, A.; VanderPlas, J.; Wanderman-Milne, S.; and Zhang, Q. 2018. JAX: composable transformations of Python+NumPy programs.

Choi, Y.; Vergari, A.; and Van den Broeck, G. 2020. Probabilistic circuits: A unifying framework for tractable probabilistic models. Technical report, University of California Los Angeles.

Choi, Y.-k.; Santillana, C.; Shen, Y.; Darwiche, A.; and Cong, J. 2023. FPGA acceleration of probabilistic sentential decision diagrams with high-level synthesis. *ACM Transactions on Reconfigurable Technology and Systems*, 16(2): 1–22.

Dadu, V.; Weng, J.; Liu, S.; and Nowatzki, T. 2019. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 924–939.

Dang, M.; Khosravi, P.; Liang, Y.; Vergari, A.; and Van den Broeck, G. 2021. Juice: A julia package for logic and probabilistic circuits. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 16020–16023.

Darwiche, A. 2003. A differential approach to inference in Bayesian networks. *Journal of the ACM (JACM)*, 50(3): 280–305.

Darwiche, A. 2021. Tractable Boolean and arithmetic circuits. In *Neuro-Symbolic Artificial Intelligence: The State of the Art*, 146–172. IOS Press.

Darwiche, A.; and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17: 229–264.

De Smet, L.; Dos Martires, P. Z.; Manhaeve, R.; Marra, G.; Kimmig, A.; and De Readt, L. 2023. Neural probabilistic logic programming in discrete-continuous domains. In *Uncertainty in Artificial Intelligence*, 529–538. PMLR.

Galindez Olascoaga, L. I.; Meert, W.; Shah, N.; Verhelst, M.; and Van den Broeck, G. 2019. Towards hardware-aware tractable learning of probabilistic models. *Advances in Neural Information Processing Systems*, 32.

Hooker, S. 2021. The hardware lottery. *Communications of the ACM*, 64(12): 58–65.

Kaplan, J.; McCandlish, S.; Henighan, T.; Brown, T. B.; Chess, B.; Child, R.; Gray, S.; Radford, A.; Wu, J.; and Amodei, D. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Lagniez, J.-M.; and Marquis, P. 2017. An Improved Decision-DNNF Compiler. In *IJCAI*, volume 17, 667–673.

Liu, A.; Ahmed, K.; and Van den Broeck, G. 2024. Scaling Tractable Probabilistic Circuits: A Systems Perspective. In *Forty-first International Conference on Machine Learning*.

Manhaeve, R.; Dumancic, S.; Kimmig, A.; Demeester, T.; and De Raedt, L. 2018. Deepproblog: Neural probabilistic logic programming. *Advances in neural information processing systems*, 31.

Mari, A.; Vessio, G.; and Vergari, A. 2023. Unifying and understanding overparameterized circuit representations via low-rank tensor decompositions. In *The 6th Workshop on Tractable Probabilistic Modeling*.

Meert, W.; and Choi, A. 2017. PySDD. In *Recent trends in knowledge compilation (dagstuhl seminar 17381)*, volume 7.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Peharz, R.; Lang, S.; Vergari, A.; Stelzner, K.; Molina, A.; Trapp, M.; Van den Broeck, G.; Kersting, K.; and Ghahramani, Z. 2020a. Einsum networks: Fast and scalable learning of tractable probabilistic circuits. In *International Conference on Machine Learning*, 7563–7574. PMLR.

Peharz, R.; Vergari, A.; Stelzner, K.; Molina, A.; Shao, X.; Trapp, M.; Kersting, K.; and Ghahramani, Z. 2020b. Random sum-product networks: A simple and effective approach to probabilistic deep learning. In *Uncertainty in Artificial Intelligence*, 334–344. PMLR.

Shah, N.; Olascoaga, L. I. G.; Meert, W.; and Verhelst, M. 2020. Acceleration of probabilistic reasoning through custom processor architecture. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 322–325. IEEE.

Shah, N.; Olascoaga, L. I. G.; Zhao, S.; Meert, W.; and Verhelst, M. 2021. DPU: DAG processing unit for irregular graphs with precision-scalable posit arithmetic in 28 nm. *IEEE Journal of Solid-State Circuits*, 57(8): 2586–2596.

Sommer, L.; Halkenhäuser, M.; Axenie, C.; and Koch, A. 2021. Spnc: Accelerating sum-product network inference on cpus and gpus. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 53–56. IEEE.

Sommer, L.; Oppermann, J.; Molina, A.; Binnig, C.; Kersting, K.; and Koch, A. 2018. Automatic mapping of the sum-product network inference problem to fpga-based accelerators. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 350–357. IEEE.

Valiant, L. G. 1979. The complexity of computing the permanent. *Theoretical computer science*, 8(2): 189–201.

Vasimuddin, M.; Chockalingam, S. P.; and Aluru, S. 2018. A parallel algorithm for Bayesian network inference using arithmetic circuits. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 34–43. IEEE.

Vergari, A.; Choi, Y.; Liu, A.; Teso, S.; and Van den Broeck, G. 2021. A compositional atlas of tractable circuit operations for probabilistic inference. In *Advances in Neural Information Processing Systems*.

Weber, L.; Wirth, J.; Sommer, L.; and Koch, A. 2022. Exploiting High-Bandwidth Memory for FPGA-Acceleration of Inference on Sum-Product Networks. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 112–119. IEEE.

Xu, J.; Zhang, Z.; Friedman, T.; Liang, Y.; and Broeck, G. 2018. A semantic loss function for deep learning with symbolic knowledge. In *International conference on machine learning*, 5502–5511. PMLR.

A Additional Experiments

In Figure 4, we repeat the same synthetic experiment as in Figure 3 but using a top-down d-DNNF knowledge compiler instead of a bottom-up SDD compiler. As a baseline, we use an optimized Rust implementation of the node-by-node post-order evaluation algorithm. These circuits do not contain duplicate nodes and are somewhat less balanced, leading to a larger overhead in terms of extra nodes, as is visible on the right of Figure 4. Nonetheless, KLAY still outperforms the baseline by a large margin.

B Pseudo-code

Algorithm 2 and 3 contain pseudo-code of the previously discussed layerization and tensorization procedures.

Algorithm 2: KLay Layerization

```
Input: Boolean Circuit $C$ as linked nodes.
layers ← [ ];
height ← [ ];
hashes ← [ ];
for node $n$ in the nodes of $C$, children before parents do
    if $n$ is a leaf node then
        height[$n$] ← 0 ;
        hashes[$n$] ← $hash(n)$;
    else
        height[$n$] ← $1 + \max_{c \in \text{children}(n)}$ height[$c$];
        /* Bring children to the prior layer */
        for child node $c$ in children($n$) do
            while height[c]+1 $\neq$ height[n] do
                $c$ ← new unary node with child $c$;
                height[$c$] ← height[child($c$)] + 1;
                hashes[$c$] ← $hash$(hashes[child($c$)]);
                layers[height[$c$]][hashes[$c$]] ← $c$;
            end
        end
        hashes[$n$] ← $\bigoplus_{c \in \text{children}(n)}$ $hash$(hashes[$c$])
    end
    /* Add node to its layer */
    layers[height[$n$]][hashes[$n$]] ← $n$;
end
return layers;
```

Algorithm 3: KLay Tensorization

```
Input: layers.
/* We assume the nodes in each layer are ordered,
   such that index(n) is the index of node n in its layer.
   */
for layer $i$ in layers do
    $\mathbf{S}_i$ ← [ ];
    $\mathbf{R}_i$ ← [ ];
    for node $n$ in layer $i$ do
        for child $c$ of node $n$ do
            $\mathbf{R}_i$.push(index($n$));
            $\mathbf{S}_i$.push(index($c$));
        end
    end
end
return $\{\mathbf{S}_1, \dots, \mathbf{S}_L\}, \{\mathbf{R}_1, \dots, \mathbf{R}_L\}$;
```

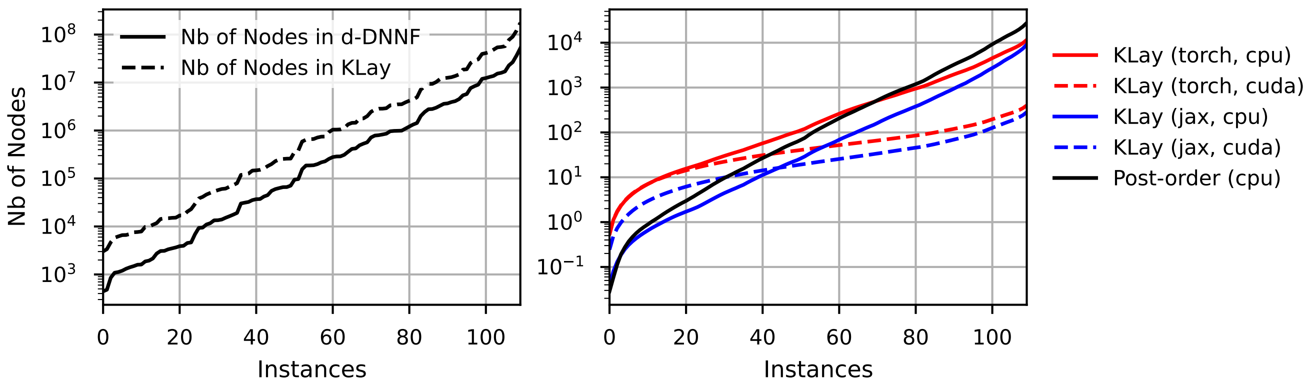

Figure 4: (Left) Number of nodes in the original d-DNNF, compared to KLAY's layerized circuit. (Right) Cumulative runtime in milliseconds for the forward and backward pass on a d-DNNF circuit in the log semiring. Timings for each individual circuit are averaged over 10 runs. Each instance is a randomly generated logical formula in 3-CNF, compiled into a d-DNNF circuit using D4. The number of variables and clauses in the CNF was varied to achieve different levels of difficulty.