Scaling Tractable Probabilistic Circuits: A Systems Perspective

Anji Liu [1] **Kareem Ahmed** [1] **Guy Van den Broeck** [1]

Abstract

Probabilistic Circuits (PCs) are a general framework for tractable deep generative models, which support exact and efficient probabilistic inference on their learned distributions. Recent modeling and training advancements have enabled their application to complex real-world tasks. However, the time and memory inefficiency of existing PC implementations hinders further scaling up. This paper proposes PyJuice, a general GPU implementation design for PCs that improves prior art in several regards. Specifically, PyJuice is 1-2 orders of magnitude faster than existing systems (including very recent ones) at training large-scale PCs. Moreover, PyJuice consumes 2-5x less GPU memory, which enables us to train larger models. At the core of our system is a compilation process that converts a PC into a compact representation amenable to efficient block-based parallelization, which significantly reduces IO and makes it possible to leverage Tensor Cores available in modern GPUs. Empirically, PyJuice can be used to improve state-of-the-art PCs trained on image (e.g., ImageNet32) and language (e.g., WikiText, CommonGen) datasets. We further establish a new set of baselines on natural image and language datasets by benchmarking existing PC structures but with much larger sizes and more training epochs, with the hope of incentivizing future research. Code is available at `https://github.com/Tractables/pyjuice`.

1. Introduction

Many tasks require not only precise modeling of intricate, high-dimensional data distributions but also the efficient execution of probabilistic inference on the learned model. To satisfy inference-side demands, tractable deep generative models are designed to support efficient computation of various probabilistic queries. Probabilistic Circuits (PCs) (Choi et al., 2020; Vergari et al., 2020) are a unified framework that abstracts a myriad of tractable model families. PCs have been applied to many domains such as explainability and causality (Correia et al., 2020; Wang & Kwiatkowska, 2023), graph link prediction (Loconte et al., 2023), and neuro-symbolic AI (Xu et al., 2018; Manhaeve et al., 2018; Ahmed et al., 2022a). In particular, there is a trend of using PCs' tractability to control expressive deep generative models, including (large) language models (Zhang et al., 2023), image diffusion models (Liu et al., 2024), and reinforcement learning models (Liu et al., 2023b).

The backbone of the application-side advancements is the recent breakthroughs on the modeling and learning side of PCs, which include designing better PC structures (Peharz et al., 2020b; Correia et al., 2023; Mathur et al., 2023; Loconte et al., 2024; Gala et al., 2024), effective structure learning algorithms (Gens & Pedro, 2013; Dang et al., 2020; 2022; Yang et al., 2023), and distilling from expressive deep generative models (Liu et al., 2023a). Despite such algorithmic innovations, a fundamental obstacle to further scaling up PC learning and inference is the time and memory inefficiency of existing implementations, hindering the training of large PC models and their application to large-scale datasets.

In this work, we develop an efficient and flexible system called PyJuice that addresses various training and inference tasks for PCs. As shown in Table 1, PyJuice is orders of magnitude faster than previous implementations for PCs (e.g., SPFlow (Molina et al., 2019), EiNet (Peharz et al., 2020a), and Juice.jl (Dang et al., 2021)) as well as Hidden Markov Models[1] (e.g., Dynamax (Murphy et al., 2023)). Additionally, as we shall demonstrate in the experiments, PyJuice is more memory efficient than the baselines, enabling us to train larger PCs with a fixed memory quota.

Unlike other deep generative models based on neural network layers that are readily amenable to efficient systems (e.g., a fully connected layer can be emulated by a single matrix multiplication and addition kernel plus an element-wise activation kernel), PCs cannot be *efficiently* computed using well-established operands due to (i) the unique connection

[1]Department of Computer Science, University of California, Los Angeles, USA. Correspondence to: Anji Liu <liuanji@cs.ucla.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

[1]Every HMM has an equivalent PC representation.

Table 1. **Average ($\pm$ standard deviation of 5 runs) runtime (in seconds) per training epoch** of 60K samples for PyJuice and the baselines SPFlow (Molina et al., 2019), EiNet (Peharz et al., 2020a), Juice.jl (Dang et al., 2021), and Dynamax (Murphy et al., 2023). We adopted four PC structures: PD, RAT-SPN, HCLT, and HMM. All experiments were carried out on an RTX 4090 GPU with 24GB memory. To maximize parallelism, we always use the maximum possible batch size. "OOM" denotes out-of-memory with batch size 2. The best numbers are in boldface.

	PD (Poon & Domingos, 2011)				
# nodes	172K	344K	688K	1.38M	2.06M
# edges	15.6M	56.3M	213M	829M	2.03B
SPFlow	>25000	>25000	>25000	>25000	>25000
EiNet	34.2±0.0	88.7±0.2	456.1±2.3	1534.7±0.5	OOM
Juice.jl	12.6±0.5	37.0±1.7	141.7±6.9	OOM	OOM
PyJuice	**2.0**±0.0	**5.3**±0.0	**15.4**±0.0	**57.1**±0.2	**203.7**±0.1
	RAT-SPN (Peharz et al., 2020b)				
# nodes	58K	116K	232K	465K	930K
# edges	616K	2.2M	8.6M	33.4M	132M
SPFlow	6372.1±4.2	>25000	>25000	>25000	>25000
EiNets	38.5±0.0	83.5±0.0	193.5±0.1	500.6±0.2	2445.1±2.6
Juice.jl	6.0±0.3	9.4±0.3	25.5±2.4	84.0±4.0	375.1±3.4
PyJuice	**0.6**±0.0	**0.9**±0.1	**1.6**±0.0	**5.8**±0.1	**13.8**±0.0
	HCLT (Liu & Van den Broeck, 2021)				
# nodes	89K	178K	355K	710K	1.42M
# edges	2.56M	10.1M	39.9M	159M	633M
SPFlow	22955.6±18.4	>25000	>25000	>25000	>25000
EiNet	52.5±0.3	77.4±0.4	233.5±2.8	1170.7±8.9	5654.3±17.4
Juice.jl	4.7±0.2	6.4±0.5	12.4±1.3	41.1±0.1	143.2±5.1
PyJuice	**0.8**±0.0	**1.3**±0.0	**2.6**±0.0	**8.8**±0.0	**24.9**±0.1
	HMM (Rabiner & Juang, 1986)				
# nodes	33K	66K	130K	259K	388K
# edges	8.16M	32.6M	130M	520M	1.17B
Dynamax	111.3±0.4	441.2±3.9	934.7±6.3	2130.5±19.5	4039.8±38.3
Juice.jl	4.6±0.1	18.8±0.1	91.6±0.1	OOM	OOM
PyJuice	**0.6**±0.0	**1.0**±0.0	**2.9**±0.1	**10.1**±0.2	**39.9**±0.1

patterns of their computation graph,[2] and (ii) the existence of log probabilities at drastically different scales in the models, which requires to properly handle numerical underflow problems. To parallelize PCs at scale, we propose a compilation phase that converts a PC into a compact data structure amenable to block-based parallelization on modern GPUs. Further, we improve the backpropagation process by indirectly computing the parameter updates by backpropagating a quantity called PC flow (Choi et al., 2021) that is more numerically convenient yet mathematically equivalent.

In the following, we first formally define PCs and discuss common ways to parallelize their computation in Section 2. Section 3 examines the key bottlenecks in PC parallelization. Section 4 and 5 explains our design in details.

[2]Commonly used neural network layers mainly employ "regular" tensor operations such as matrix multiplications and tensor inner-/outer-products. In contrast, PC layers can contain nodes that are sparsely connected.

2. Preliminaries and Related Work

Many probabilistic inference tasks can be cast into computing sums of products. By viewing them from a computation graph standpoint, PCs provide a unified perspective on many bespoke representations of tractable probability distributions, including Arithmetic Circuits (Darwiche, 2002; 2003), Sum-Product Networks (Poon & Domingos, 2011), Cutset Networks (Rahman et al., 2014), and Hidden Markov Models (Rabiner & Juang, 1986). Specifically, PCs define distributions with computation graphs consisting of sum and product operations, as elaborated below.

Definition 1 (Probabilistic Circuit)**.** A PC defined over variables $\mathbf{X}$ is represented by a parameterized Directed Acyclic Graph (DAG) with a single root node n_{r}. Every leaf node in the DAG represents an input node that defines a primitive distribution over some variable $X \in \mathbf{X}$. Every inner node n is either a sum node or a product node, which merges the distributions encoded by its children, denoted $\mathsf{ch}(n)$, to construct more complex distributions. The distribution represented by every node is defined recursively as:

$$p_n(\boldsymbol{x}) := \begin{cases} f_n(\boldsymbol{x}) & n \text{ is an input node}, \\ \prod_{c \in \mathsf{ch}(n)} p_c(\boldsymbol{x}) & n \text{ is a product node}, \\ \sum_{c \in \mathsf{ch}(n)} \theta_{n,c} \cdot p_c(\boldsymbol{x}) & n \text{ is a sum node}, \end{cases} \quad (1)$$

where $f_n(\boldsymbol{x})$ is an univariate input distribution (e.g., Gaussian, Categorical), and $\theta_{n,c}$ denotes the parameter corresponding to edge (n, c). Intuitively, sum nodes model mixtures of their input distributions, which require the mixture weights to be in the probability simplex: $\sum_{c \in \mathsf{ch}(n)} \theta_{n,c} = 1$ and $\forall c \in \mathsf{ch}(n), \theta_{n,c} \geq 0$. And product nodes build factorized distributions over their inputs. The size of a PC, denoted $|p|$, is the number of edges in its DAG.

The key to guaranteeing exact and efficient computation of various probabilistic queries is to impose proper structural constraints on the DAG of the PC. As an example, with smoothness and decomposability (Poon & Domingos, 2011), computing any marginal probability amounts to a forward pass (children before parents) following Equation (1), with the only exception that we set the value of input nodes defined on marginalized variables to be 1. Please refer to Choi et al. (2020) for a comprehensive overview of different structural constraints and what queries they enable.

Although different algorithms are used for different training and inference tasks, they are mostly based on (variants of) the following subroutines: a feedforward pass (Eq. (1)) that computes $\log p_{n_{\mathrm{r}}}(\boldsymbol{x})$, and a backward pass computing

$$\forall n, \frac{\partial \log p_{n_{\mathrm{r}}}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})} \text{ and } \forall \theta_{n,c}, \frac{\partial \log p_{n_{\mathrm{r}}}(\boldsymbol{x})}{\partial \theta_{n,c}}. \quad (2)$$

For example, Peharz et al. (2020a) demonstrate how the above parameter gradients can be used to apply Expectation-Maximization (EM) updates, and Vergari et al. (2021) elab-

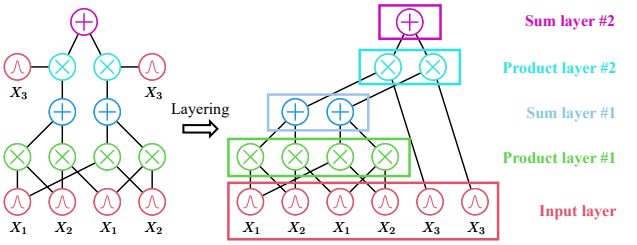
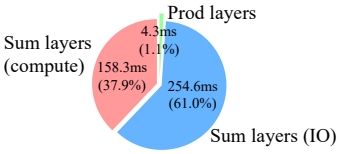


Figure 1. Layering a PC by grouping nodes with the same topological depth (as indicated by the colors) into disjoint subsets. Both the forward and the backward computation can be carried out independently on nodes within the same layer.



Summary of the PC structure

	Sum layers	Prod layers
# nodes	200K	815K
# edges	154M	1.81M
# params	154M	-

Figure 2. Runtime breakdown of the feedforward pass of a PC with ∼150M edges. Both the IO and the computation overhead of the sum layers are significantly larger than the total runtime of product layers. Detailed configurations of the PC are shown in the table.

orates how the forward pass can be used to compute various probabilistic and information-theoretic queries when coupled with PC structure transformation algorithms. Therefore, the speed and memory efficiency of these two procedures largely determine the overall efficiency of PCs.

Related work on accelerating PCs. There has been a great amount of effort put into speeding up training and inference for PCs. One of the initial attempts performs node-based computations on both CPUs (Lowd & Rooshenas, 2015) and GPUs (Pronobis et al., 2017; Molina et al., 2019), i.e., by computing the outputs for a mini-batch of inputs (data) recursively for every node. Despite its simplicity, it fails to fully exploit the parallel computation capability possessed by modern GPUs since it can only parallelize over a batch of samples. This problem is mitigated by also parallelizing topologically independent nodes (Peharz et al., 2020a; Dang et al., 2021). Specifically, a PC is chunked into topological layers, where nodes in the same layer can be computed in parallel. This leads to 1-2 orders of magnitude speedup compared to node-based computation.

The regularity of edge connection patterns is another key factor influencing the design choices. Specifically, EiNets (Peharz et al., 2020a) leverage off-the-shelf Einsum operations to parallelize dense PCs where every layer contains groups of densely connected sum and product/input nodes. Mari et al. (2023) generalize the notion of dense PCs to tensorized PCs, which greatly expands the scope of EiNets. Dang et al. (2021) instead focus on speeding up sparse PCs, where different nodes could have drastically different numbers of edges. They use custom CUDA kernels to balance the workload of different GPU threads and achieve decent speedup on both sparse and dense PCs.

Another thread of work focuses on designing computation hardware that is more suitable for PCs. Specifically, Shah et al. (2021) propose DAG Processing Units (DPUs) that can efficiently traverse sparse PCs, Dadu et al. (2019) introduce an indirect read reorder-buffer to improve the efficiency of data-dependent memory accesses in PCs, and Yao et al. (2023) use addition-as-int multiplications to significantly improve the energy efficiency of PC inference algorithms.

Applications of PCs. PCs have been applied to many domains such as explainability and causality (Correia et al., 2020; Wang & Kwiatkowska, 2023), graph link prediction (Loconte et al., 2023), lossless data compression (Liu et al., 2022), neuro-symbolic AI (Xu et al., 2018; Manhaeve et al., 2018; Ahmed et al., 2022a;b), gradient estimation (Ahmed et al., 2023b), graph neural networks rewiring (Qian et al., 2023), and even large language model detoxification (Ahmed et al., 2023a).

3. Key Bottlenecks in PC Parallelization

This section aims to lay out the key bottlenecks to efficient PC implementations. For ease of illustration, we focus solely on the forward pass, and leave the unique challenges posed by the backward pass and their solution to Section 5.

We start by illustrating the layering procedure deployed for PCs. Starting from the input nodes, we perform a topological sort of all nodes, clustering nodes with the same topological depth into a layer. For example, in Figure 1, the PC on the left side is transformed into an equivalent layered representation on the right, where nodes of the same color belong to the same layer. The forward pass proceeds by sequentially processing each layer, and finally returns the root node's output. To avoid underflow, all probabilities are stored in the logarithm space. Therefore, product layers just need to sum up the corresponding input log-probabilities, while sum layers compute weighted sums of input log-probabilities utilizing the logsumexp trick.

Assume for now that all nodes in every layer have the same number of children. A straightforward strategy is to parallelize over every node and every sample. Specifically, given a layer of size M and batch size B, we need to compute in total $M \times B$ output values, which are evenly distributed to all processors (e.g., thread-blocks in GPUs). We apply this idea to a PC with the PD structure (Poon & Domingos, 2011). The PC has ∼1M nodes and ∼150M edges. Additionally, all nodes within a layer have the same number of children, making it an ideal testbed for the aforementioned algorithm.

Figure 2 illustrates the runtime breakdown of the forward pass (with batch size 512). As shown in the pie chart, both the IO and the computation overhead of the sum layers are

much larger than that of the product layers. We would expect sum layers to exhibit a higher computation overhead due to (i) the number of sum edges being $\sim$85x more than the product edges (see the table in Fig. 2), and (ii) sum edges requiring more compute compared to product edges. However, we would not expect the gap in IO overhead to be as pronounced as indicated in the pie chart. Specifically, with batch size 512, the ideal memory read count of product layers should be roughly [batch size] $\times$ [#sum nodes] $\approx$ 102M since all children of product nodes are sum or input nodes (the number of input nodes is an order of magnitude smaller and is omitted). Similarly, the number of memory reads required by the sum layers is approximately [batch size]$\times$[#prod nodes]+[#parameters] $\approx$ 571M, which is only 5.6x compared to the product layers. The ideal memory write count of product layers should be larger since there are about 4x more product nodes compared to sum nodes.

While the ideal IO overhead of the sum layers is not much larger than that of the product layers, the drastic difference in runtime (over 50x) can be explained by the significant amount of reloads of child nodes' probabilities in the sum layers. Specifically, in the adopted PD structure, every sum node has no more than 12 parents, while most product nodes have 256 parents.[3] Recall that the parents of product nodes are sum nodes and vice versa. As a result, each sum layer needs to reload the output of every product node multiple times. Although this does not lead to 256x loads from the GPU's High-Bandwidth Memory (HBM) thanks to its caching mechanism, such excessive IO access still significantly slows down the algorithm.

The fundamental principle guiding our design is to *properly group, or allocate, sum edges to different processors to minimize the reloading of product nodes' outputs*. As an added benefit, this allows us to interpret part of the core computation as matrix multiplications, allowing us to harness Tensor Cores available in modern GPUs and resulting in a significant reduction in sum layers' computational overhead.

4. Harnessing Block-Based PC Parallelization

This section takes gradual steps toward demonstrating how we can reduce both the IO and computation overhead using block-based parallelization. Specifically, we first utilize a fully connected sum layer to sketch the high-level idea (Sec. 4.1). Consequently, we move on to the general case, providing further details of the algorithm (Secs. 4.2, 4.3).

4.1. Fully Connected Sum Layers

Consider a fully connected sum layer comprised of M sum nodes, each connected to the same set of N product nodes as inputs. Under the parallelization strategy mentioned in Section 3, with a single sample, we have M processors each computing the output of a sum node. Since the layer is fully connected, every processor loads all N input log-probabilities, which results in M reloads of every input.

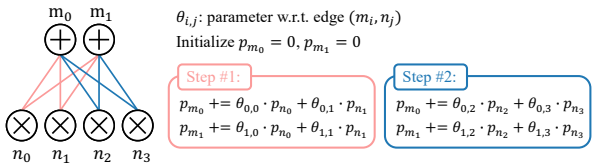


Figure 3. Illustration of block-based parallelization. A processor computes the output of 2 sum nodes, by iterating through blocks of 2 input product nodes and accumulating partial results.

The key to reducing excessive IO overhead is by parallelizing over blocks of nodes/edges. Specifically, we divide the M sum nodes into blocks of K_M nodes and the N product nodes into blocks of K_N nodes. We assume without loss of generality that M and N are divisible by K_M and K_N, respectively.[4] Instead of independently computing the output of every sum node, we calculate the K_M outputs of a sum node block in a single processor. To achieve this, we iterate through every product node block to compute and accumulate the partial results from the $K_M \times K_N$ edges between the corresponding sum node block and product node block.

In every step, the processor loads a block of $\boldsymbol{\theta} \in \mathbb{R}^{K_M \times K_N}$ parameters and a vector of $\boldsymbol{p}_{\text{prod}} \in \mathbb{R}^{K_N}$ input probabilities, where we (temporarily) omit the fact that all probabilities are stored in the logarithm space. The partial outputs $\boldsymbol{p}_{\text{sum}} \in \mathbb{R}^{K_M}$ are computed via a matrix-vector multiplication between $\boldsymbol{\theta}$ and $\boldsymbol{p}_{\text{prod}}$. Note that if we add a second "batch" dimension to $\boldsymbol{p}_{\text{prod}}$ and $\boldsymbol{p}_{\text{sum}}$, the computation immediately becomes a matrix-matrix multiplication, which can be computed efficiently using GPU Tensor Cores.

For example, in Figure 3, define $K_M = K_N = 2$, we compute the output of m_0 and m_1 by first calculating the weighted sum w.r.t. the input probability of n_0 and n_1 in step #1, and then accumulate the probabilities coming from n_2 and n_3 in step #2. With the new parallelization strategy, every processor that computes K_M output values needs to load every input probability only once, and the number of reloads is reduced from M to M/K_M.

4.2. Generalizing To Practical Sum Layers

Many sum layers in practical PCs are not fully connected (e.g., in Dang et al. (2022); Liu et al. (2023a)). However, as

[3] Only the children of the root sum node have 1 parent.

[4] When the number of product and sum nodes are not divisible by the respective block size, we can add at most $K_M - 1$ (or $K_N - 1$) placeholder nodes to make them divisible by the block size. The incurred additional computation overhead can be small since we can achieve good efficiency with relatively small block sizes (e.g., 32 or 64) given that the number of nodes in a layer is typically greater than a few thousand.

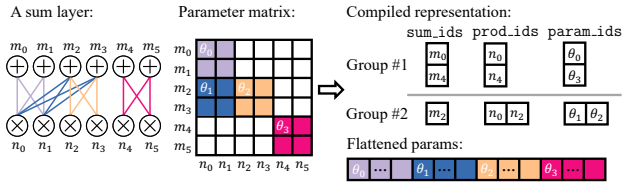


Figure 4. A sum layer (left) with a block-sparse parameter matrix (middle) is compiled into two kernels (right) each with a balanced workload. During execution, each kernel uses the compiled sum/prod/param indices to compute the outputs of $m_0, \ldots, m_5$.

we shall demonstrate, they can still harness the advantages of block-based parallelization. Specifically, consider a sum layer with M sum nodes and N product nodes as inputs. Following Section 4.1, we partition the sum and the product nodes into blocks of K_M and K_N nodes, respectively. For every pair of sum and product node blocks, if it is either fully connected (i.e., featuring $K_M \times K_N$ edges) or unconnected (i.e., no edge between them), we call the layer block-sparse. In the following, we focus on efficiently parallelizing block-sparse PCs (whose sum layers all exhibit block-sparsity). We show in Appendix B.1 that many widely-adopted PCs are indeed block sparse w.r.t. large block sizes. In Section 4.4, we describe how our implementation can speed up sparse PCs. We also show in Section 6.1 that PyJuice speeds up sparse PCs.

As an example, the layer illustrated in Figure 4 (left) exhibits block sparsity with block sizes $K_M = K_N = 2$. This is evident as each pair of sum and product node blocks is either fully connected (e.g., $\{m_2, m_3\}$ and $\{n_0, n_1\}$) or disjoint (e.g., $\{m_4, m_5\}$ and $\{n_2, n_3\}$). In Figure 4 (middle), this pattern is more discernible in the parameter matrix, where *aligned* 2×2 blocks display either all non-zero parameters (indicated by the colors) or all zero parameters.

Similar to the procedure outlined in Section 4.1, computing the outputs of a block of K_M sum nodes involves iterating through all its connected product node blocks. This introduces two additional problems: (i) how to efficiently index the set of connected product node blocks, which may vary for each sum node block; (ii) different sum node blocks could connect to different numbers of product node blocks, which causes an imbalanced workload among processors. For instance, consider the layer in Figure 4. The first issue is exemplified by the two sum node blocks $\{m_0, m_1\}$ and $\{m_4, m_5\}$, both of which possess a single child node block, albeit different ones. The second issue is illustrated by the node block $\{m_2, m_3\}$, which connects to two child node blocks, while the others connect to only one.

4.3. Efficient Implementations by Compiling PC Layers

We address both problems through a compilation process, where we assign every node an index, and precompute index tensors that enable efficient block-based parallelization. The first step is to partition the sum node blocks into groups, such that every node block within a group has a similar number of connected child node blocks. We then pad the children with pseudo-product node blocks with probability 0 such that all sum node blocks in a group have the same number of children. The partition is generated by a dynamic programming algorithm that aims to divide the layer into the smallest possible number of groups while ensuring that the fraction of added pseudo-node blocks does not exceed a predefined threshold. Due to space constraints, we elaborate the node block partitioning algorithm in Appendix A.1. We also discuss its optimality and time/memory efficiency.

We move on to construct the index tensors for each group. In addition to assigning every node an index, we create a vector $\boldsymbol{\theta}_{\text{flat}}$, a concatentation of all the PC parameters. For every sum node block in a group with C_N child node blocks, we record (i) the starting index of the sum node block, (ii) the set of initial indices of its C_N child node blocks, and (iii) the corresponding set of C_N parameter indices (that point to the first parameter in the respective block of parameters in $\boldsymbol{\theta}_{\text{flat}}$). These parameter indices each denote the starting point for the $K_M \times K_N$ parameters of the corresponding pair of sum and product node blocks. Let C_M represent the total number of node blocks in the group. Following the indices described above, we record the following tensors: $\texttt{sum\_ids} \in \mathbb{Z}^{C_M}$ containing indices of all sum node blocks; $\texttt{prod\_ids}, \texttt{param\_ids} \in \mathbb{Z}^{C_M \times C_N}$, whose ith row represent the child indices and parameter indices of the ith sum node block (i.e., the node block with the start index $\texttt{sum\_ids}[i]$), respectively.

Figure 4 (right) illustrates the compiled index tensors of the sum layer shown on the left. Recall that we use the block sizes $K_M = K_N = 2$. The layer is then divided into two groups: the first group including two sum node blocks, $\{m_0, m_1\}$ and $\{m_4, m_5\}$, each having one child node block, and the second group including one sum node block, $\{m_2, m_3\}$, which has two child node blocks. Take, for instance, the first group. `sum_ids` stores the start indices (i.e., m_0 and m_4) of the two sum node blocks. `prod_ids` stores the initial indices of the child node blocks (i.e., n_0 and n_4) of the two sum node blocks, respectively. `param_ids` encodes the corresponding initial parameter indices θ_0 and θ_2.

Partitioning a layer into groups with the same number of children allows us to use different kernel launching hyperparameters according to the specific setup of every node group (e.g., number of nodes) to achieve better performance.

For every group in a sum layer, the three index tensors serve as inputs to a CUDA kernel computing the log-probabilities of the sum nodes in the group. Define $\boldsymbol{l}_{\text{prod}} \in \mathbb{R}^{N \times B}$ and $\boldsymbol{l}_{\text{sum}} \in \mathbb{R}^{M \times B}$ (B is the batch size) as the set of input and output log-probabilities, respectively. Consider a group with C_M sum node blocks and C_N child node blocks per sum

Algorithm 1 Forward pass of a sum layer group

1: **Inputs:** log-probs of product nodes $\boldsymbol{l}_{\text{prod}}$, flattened parameter vector $\boldsymbol{\theta}_{\text{flat}}$, `sum_ids`, `prod_ids`, `param_ids`
2: **Inputs:** # sum nodes: M, # product nodes: N, batch size: B
3: **Inputs:** block sizes K_M, K_N, K_B for the sum node, product node, and batch dimensions, respectively
4: **Inputs:** number of sum node blocks C_M; number of product node blocks C_N; number of batch blocks C_B
5: **Outputs:** log-probs of sum nodes $\boldsymbol{l}_{\text{sum}}$
6: **Kernel launch:** schedule to launch $C_M \times C_B$ thread-blocks with $\texttt{m}=0,\dots,C_M-1$ and $\texttt{b}=0,\dots,C_B-1$
7: $\texttt{cum} \leftarrow (-\infty)_{K_M \times K_B} \in \mathbb{R}^{K_M \times K_B}$ ▷ Scratch space on SRAM
8: $\texttt{bs}, \texttt{be} \leftarrow \texttt{b} \cdot K_B, (\texttt{b}+1) \cdot K_B$ ▷ Start and end batch index
9: **for** $\texttt{n} = 0$ **to** $C_N - 1$ **do**
10: $\quad \texttt{ps}, \texttt{ns} \leftarrow \texttt{param\_ids}[\texttt{m}, \texttt{n}], \texttt{prod\_ids}[\texttt{n}, \texttt{b}]$
11: $\quad$ Load $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_{\text{flat}}[\texttt{ps}:\texttt{ps}+K_M K_N].\texttt{view}(K_M, K_N)$ to SRAM
12: $\quad$ Load $\boldsymbol{l} \leftarrow \boldsymbol{l}_{\text{prod}}[\texttt{ns}:\texttt{ns}+K_N, \texttt{bs}:\texttt{be}] \in \mathbb{R}^{K_N \times K_B}$ to SRAM
13: $\quad \boldsymbol{l}_{\max} \leftarrow \max(\boldsymbol{l}, \texttt{dim}=0) \in \mathbb{R}^{1 \times K_B}$ ▷ Compute on chip
14: $\quad \boldsymbol{p}_{\text{p}} \leftarrow \exp(\boldsymbol{l} - \boldsymbol{l}_{\max}) \in \mathbb{R}^{K_N \times K_B}$
15: $\quad \boldsymbol{p}_{\text{s}} \leftarrow \texttt{matmul}(\boldsymbol{\theta}, \boldsymbol{p}_{\text{p}}) \in \mathbb{R}^{K_M \times K_B}$ ▷ With Tensor Cores
16: $\quad \texttt{cum} \leftarrow \texttt{where}(\boldsymbol{l}_{\max} > \texttt{cum},$
$\qquad\qquad \log(\boldsymbol{p}_{\text{s}} + \exp(\texttt{cum} - \boldsymbol{l}_{\max}) + \boldsymbol{l}_{\max},$
$\qquad\qquad \log(\exp(\boldsymbol{l}_{\max} - \texttt{cum}) \cdot \boldsymbol{p}_{\text{s}} + 1) + \texttt{cum})$
17: $\boldsymbol{l}_{\text{sum}}[\texttt{ms}:\texttt{ms}+K_M, \texttt{bs}:\texttt{be}] \leftarrow \texttt{acc}$ (where $\texttt{ms} \leftarrow \texttt{sum\_ids}[\texttt{m}]$)

node block. Algorithm 1 computes the log-probabilities of the C_M sum node blocks and stores the results in the proper locations in $\boldsymbol{l}_{\text{sum}}$. Specifically, we also divide the B samples into blocks of size K_B, leading to $C_B := B/K_B$ blocks (assume w.l.o.g. that B is divisible by K_B). Algorithm 1 schedules to launch $C_M \times C_B$ thread-blocks, each responsible for computing $K_M \times K_B$ outputs (line 6). The main loop in line 9 iterates over all C_N child node blocks. In every step, we first load the corresponding parameter matrix $\boldsymbol{\theta} \in \mathbb{R}^{K_M \times K_N}$ (line 11) and input matrix $\boldsymbol{l} \in \mathbb{R}^{K_N \times K_B}$ (line 12). Since $\boldsymbol{l}$ contains log-probabilities, we apply a variant of the logsumexp trick: we first convert $\boldsymbol{l}$ to the arithmetic space by subtracting the per-sample maximum log-probability (lines 13-14), then compute the (partial) output probabilities from the current set of $K_M \times K_N$ edges via matrix multiplication (line 15), and in line 16 aggregate the results back to the accumulator `cum` defined in line 7. Finally, we store the log-probabilities to the target locations in $\boldsymbol{l}_{\text{sum}}$ (line 17).

4.4. Analysis: IO and Computation Overhead

We analyze the efficiency and IO complexity of our block-based parallelization strategy. Specifically, we benchmark on the largest sum layer in the PD structure adopted in Section 3. The layer consists of 29K nodes and 30M edges. In addition to the computation time, we record two types of IO overhead: (i) the IO between the L1/texture cache and the L2 cache, and (ii) the reads/writes between the L2 cache and the GPU High-Bandwidth Memory (HBM). We vary the block sizes K_M and K_N exponentially from 1 to 64. To ensure a fair comparison, we implement a dedicated kernel for $K_M = K_N = 1$, which directly parallelizes over sum node/sample pairs, allowing for better workload allocation. For other block sizes, we adjust K_B and other kernel launching hyperparameters (e.g., warps per block) and report the best runtime for every case. Results of the backward pass (w.r.t. inputs) are also reported for completeness.

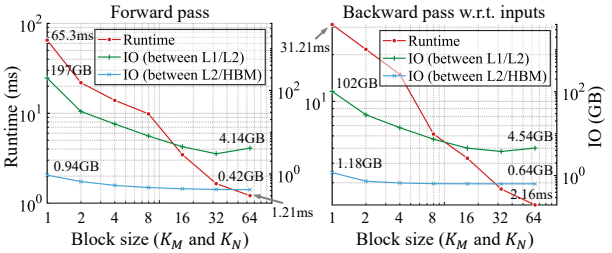


Figure 5. Runtime and IO overhead of a sum layer from the PD structure (with 29K nodes and 30M edges). The results demonstrate significant performance gains from our block-based parallelization, even with small block sizes.

Results are shown in Figure 5. As the block size increases, both the forward and the backward pass become significantly faster. Notably, this is accompanied by a significant drop in IO overhead. Specifically, with a large block size, the kernel consumes 2x fewer reads/writes between the L2 cache and the HBM, and 25-50x fewer IO between the L1 and L2 cache. This corroborates the hypothesis stated in Section 3 that the extensive value reloads significantly slow down the computation.

Additionally, we note that even with small block sizes (e.g., 2 or 4), the speedup is quite significant compared to the baseline case ($K_M = K_N = 1$), which allows us to speed up *sparse* PCs. Specifically, with the observation that every sparse PC can be viewed as a block-sparse PC with block size 1, we can transform a sparse PC into a block-sparse one, and pad zero parameters to edges belonging to the block-sparse PC but not the sparse PC. For PCs with relatively regular sparsity patterns, increasing the block sizes to even small values like 2 or 4 can lead to significant speedup even though a relatively large number of pseudo edges need to be padded.

the speedup obtained by having a larger block size outpaces the overhead caused by padded edges with zero parameters, which leads to speed-ups.

5. Optimizing Backpropagation with PC Flows

The previous section focuses on speeding up sum layers by reducing excessive memory reloads and leveraging Tensor Cores. However, when it comes to backpropagation, directly adapting Algorithm 1 by differentiating lines 13-16 would

lead to poor performance due to the following. First, we need to either store some intermediate values (e.g., $\boldsymbol{l}_{\max}$ and $\boldsymbol{p}_{\mathrm{p}}$) in the forward pass or recompute them in the backward pass. Next, since different thread-blocks could access the same product node log-probabilities in line 12, they both need to write (partial) gradients of it, which introduces inter-thread-block barriers that slow down the execution.

We overcome the problems by leveraging PC flows (Choi et al., 2021), which is only a factor of $\theta_{n,c}$ away from the desired gradients (Eq. 2). PC flows exhibit a straightforward recursive definition, facilitating a seamless transformation into an efficient implementation for the backward pass.

Definition 2 (PC flows)**.** For a PC $p_{n_r}(\mathbf{X})$ rooted at node n_r and a sample $\boldsymbol{x}$, the flow $\mathrm{F}_n(\boldsymbol{x})$ of every node n is defined recursively as follows (assume that no consecutive sum nodes or product nodes exist in the PC):[5]

$$\mathrm{F}_n(\boldsymbol{x}) := \begin{cases} 1 & n \text{ is the root node,} \\ \sum_{m \in \mathsf{pa}(n)} \mathrm{F}_m(\boldsymbol{x}) & n \text{ is input or sum,} \\ \sum_{m \in \mathsf{pa}(n)} \frac{\theta_{m,n} \cdot p_n(\boldsymbol{x})}{p_m(\boldsymbol{x})} \cdot \mathrm{F}_m(\boldsymbol{x}) & n \text{ is a product node,} \end{cases}$$

where $\mathsf{pa}(n)$ is the set of parents of n. Similarly, the edge flow $\mathrm{F}_{n,c}(\boldsymbol{x})$ w.r.t. the sample $\boldsymbol{x}$ ($c \in \mathsf{ch}(n)$) is defined as

$$\mathrm{F}_{n,c}(\boldsymbol{x}) := \theta_{n,c} \cdot p_c(\boldsymbol{x}) / p_n(\boldsymbol{x}) \cdot \mathrm{F}_n(\boldsymbol{x}).$$

While similar results have been established in a slightly different context (Peharz et al., 2020a), we prove the following equations in Appendix B.2 for completeness:

$$\mathrm{F}_n(\boldsymbol{x}) = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})} \text{ and } \mathrm{F}_{n,c}(\boldsymbol{x}) = \theta_{n,c} \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \theta_{n,c}}.$$

Following Definition 2, we can compute $F_n(\boldsymbol{x})$ for every node n utilizing the same set of layers created for the feed-forward pass. Specifically, we first set the flow of the root node to 1 following its definition. We then iterate through the layers in reverse order (i.e., parent layers before child layers). While processing a layer, all flows of the nodes in the layer are computed by the preceding layers. And our goal is to compute the (partial) flows of the child nodes of the layer. Similar to the forward pass, we compile every layer by grouping child node blocks with a similar number of parents, and use block-based parallelization to reduce reloads of parent log-probabilities. We provide the full details of the backpropagation algorithm in Appendix A.2.

Another important design choice that leads to a significant reduction in memory footprint is to recompute the product nodes' probabilities in the backward pass instead of storing them all in the GPU memory during the forward pass. Specifically, we maintain a scratch space on GPU HBM that can hold the results of the largest product layer. All product layers write their outputs to this same scratch space, and the required product node probabilities are re-computed when requested by a sum layer during backpropagation. Since product layers are extremely fast to evaluate compared to the sum layers (e.g., see the runtime breakdown in Fig. 2), this leads to significant memory savings at the cost of slightly increased computation time.

[5] If such nodes exist, we can always collapse them into a single sum or product node.

6. Experiments

We evaluate the impact of using PyJuice to train PC models. In Section 6.1, we compare PyJuice against existing implementations regarding time and memory efficiency. To demonstrate its generality and flexibility, we evaluate PyJuice on four commonly used dense PC structures as well as highly unstructured and sparse PCs. Next, we demonstrate that PyJuice can be readily used to scale up PCs for various downstream applications in Section 6.2. Finally, in Section 6.3, we benchmark existing PCs on high-resolution image datasets, hoping to incentivize future research to develop better PC structures as well as learning algorithms.

6.1. Faster Models with PyJuice

We first benchmark the runtime of PyJuice on four commonly used PC structures: PD (Poon & Domingos, 2011), RAT-SPN (Peharz et al., 2020b), HCLT (Liu & Van den Broeck, 2021), and HMM (Rabiner & Juang, 1986). For all models, we record the runtime to process 60,000 samples (including the forward pass, the backward pass, and mini-batch EM updates). We vary their structural hyperparameters and create five PCs for every structure with sizes (i.e., number of edges) ranging from 500K to 2B. We compare against four baselines: SPFlow (Molina et al., 2019), EiNet (Peharz et al., 2020a), Juice.jl (Dang et al., 2021), and Dynamax (Murphy et al., 2023). Dynamax is dedicated to State Space Models so it is only used to run HMMs; SPFlow and EiNet are excluded in the HMM results because we are unable to construct homogeneous HMMs with their frameworks due to the need to share the transition and emission parameters at different time steps. We describe how PyJuice implements PCs with tied parameters in Appendix A.3. All experiments in this subsection are carried out on an RTX 4090 GPU with 24GB memory.

Table 1 reports the runtime in seconds per epoch with mini-batch EMs. PyJuice is orders of magnitude faster than all baselines in both small and large PCs. Further, we observe that most baselines exhaust 24GB of memory for larger PCs (indicated by "OOM" in the table), while PyJuice can still efficiently train these models. Additionally, in Appendix D.1,

[7] In the adopted HMM, running Dynamax with batch size ≥ 128 leads to internal errors, and thus the results are not reported.

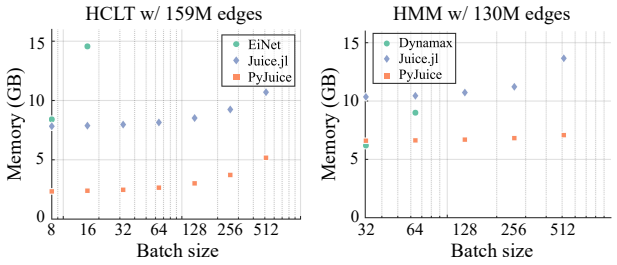


Figure 6. Comparison on memory efficiency. We take two PCs (i.e., an HCLT w/ 159M edges and an HMM w/ 130M edges) and record GPU memory usage under different block sizes.[7]

we show the efficiency of the compilation process. For example, it takes only $\sim$8.7s to compile an HCLT with 159M edges. Note that we only compile the PC once and then reuse the compiled structure for training and inference.

In Figure 6, we take two PCs to show the GPU memory consumption with different batch sizes. The results demonstrate that PyJuice is more memory efficient than the baselines, especially in the case of large batch sizes (note that we always need a constant-size space to store the parameters).

We move on to benchmark PyJuice on block-sparse PCs. We create a sum layer with 209M edges (see Appx. C.1 for details). We partition the sum and input product nodes in the layer into blocks of 32 nodes respectively. We randomly discard blocks of 32×32 edges, resulting in block-sparse layers. As shown in Figure 7, as the fraction of removed edge blocks increases, the runtime of both the forward and the backward pass decreases significantly. This motivates future work on PC modeling to focus on designing effective block-sparse PCs.

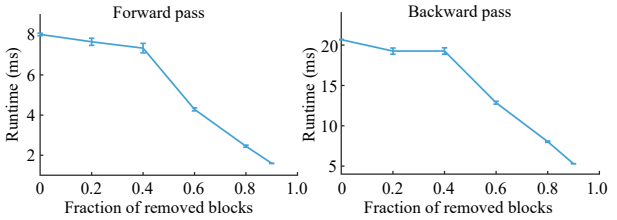


Figure 7. Runtime of a block-sparse sum layer as the function of the fraction of kept (non-dropped) edge blocks. The error bars represent standard deviations over 5 runs.

Finally, we proceed to evaluate the runtime of sparse PCs. We adopt the PC pruning algorithm proposed by Dang et al. (2022) to prune two HCLTs with 10M and 40M edges, respectively. We only compare against Juice.jl since all other implementations do not support sparse PCs. As shown in Figure 8, PyJuice is consistently faster than Juice.jl, despite the diminishing gap when over 90% edges are pruned. Note that with sparse PCs, PyJuice cannot fully benefit from the block-based parallelization strategy described in Section 4, yet it can still outperform the baseline.

Table 2. Perplexity of HMM language models trained on the CommonGen benchmark (Lin et al., 2020).

	Zhang et al. (2023)	PyJuice	
# hidden states	4096	4096	8192
Perplexity	9.78	8.81	**8.65**

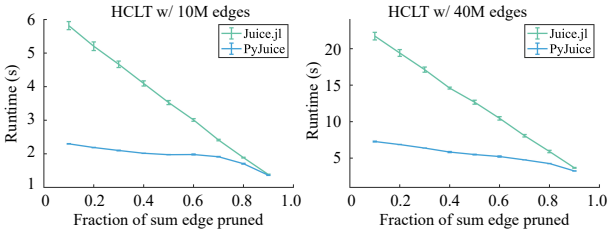


Figure 8. Runtime per epoch (with 60K samples) of two sparse HCLTs with different fractions of pruned edges. The error bars represent standard deviations over 5 runs.

6.2. Better PCs At Scale

This section demonstrates the ability of PyJuice to improve the state of the art by simply using larger PCs and training for more epochs thanks to its speed and memory efficiency. Specifically, we take the HMM language model proposed by Zhang et al. (2023) and the image model introduced by Liu et al. (2023c) as two examples.

HMM language models. Zhang et al. (2023) use the Latent Variable Distillation (LVD) (Liu et al., 2023a) technique to train an HMM with 4096 hidden states on sequences of 32 word tokens. Specifically, LVD is used to obtain a set of "good" initial parameters for the HMM from deep generative models. The HMM language model is then fine-tuned on the CommonGen dataset (Lin et al., 2020), and is subsequently used to control the generation process of (large) language models for constrained generation tasks. Following the same procedure, we use PyJuice to fine-tune two HMMs with hidden sizes 4096 and 8192, respectively.

As shown in Table 2, by using the same HMM with 4096 hidden states, PyJuice improved the perplexity by $\sim$1.0 by running many more epochs in less time compared to the original model. We also train a larger HMM with 8192 hidden states and further improved the perplexity by a further 0.16. We refer the reader to Appendix C.2 for more details.

Sparse Image Models. Liu et al. (2023c) design a PC learning algorithm that targets image data by separately training two sets of PCs: a set of sparse patch-level PCs (e.g., 4×4 patches) and a top-level PC that aggregates outputs of the patch-level PC. In the final training step, the PCs are supposed to be assembled and jointly fine-tuned. However, due to the huge memory consumption of the PC (with over 10M nodes), only the top-level model is fine-tuned in the original paper. With PyJuice, we can fit the entire model in 24GB of memory and fine-tune the entire model. For the PC trained on the ImageNet32 dataset (Deng et al., 2009), this

Table 3. Density estimation performance of PCs on three natural image datasets. Reported numbers are test set bits-per-dimension.

Dataset	PD-mid	PD-large	HCLT-mid	HCLT-large
ImageNet32	5.22	5.20	4.36	4.33
ImageNet	4.98	4.95	3.57	3.53
CelebA-HQ	4.35	4.29	2.43	2.38

fine-tuning step leads to an improvement from 4.06 to 4.04 bits-per-dimension. See Appendix C.3 for more details.

6.3. Benchmarking Existing PCs

We use PyJuice to benchmark the performance of the PD and the HCLT structure on three natural image datasets: ImageNet (Deng et al., 2009) and its down-sampled version ImageNet32, and CelebA-HQ (Liu et al., 2015). For all three datasets, we train the PCs on randomly sampled 16×16 patches, which results in a total of $16\times 16\times 3 = 768$ categorical variables each with $2^8 = 256$ possible values. As a preprocessing step, the image patches are converted into the YCoCg color space since it is observed that such color space transformations lead to improved density estimation performance. Note that due to the lossy transformation between the RGB space and the YCoCg space, our results are not directly comparable to the results obtained from RGB images.

We adopt two PD structures (i.e., PD-mid with 107M edges and PD-large with 405M edges) as well as two HCLT structures (i.e., HCLT-mid with 40M edges and HCLT-large with 174M edges). Details of the adopted models are described in Appendix C.4. We experiment with different optimization strategies and adopt full-batch EM as it yields consistently better performance across models and datasets. Specifically, the computed PC flows are accumulated across all samples in the training set before doing one EM step.

Results are shown in Table 3. Notably, we achieve *better* results compared to previous papers. For example, Liu et al. (2023a) reports 4.82 bits-per-dimension (bpd) for HCLT on ImageNet32, while we achieved 4.33 bpd. The performance improvements stem from more training epochs and the ability to do more hyperparameter search thanks to the speedup. We highlight that the goal of this section is not to set new records for tractable deep generative models, but to establish a set of baselines that can be easily reproduced to track the progress of developments in PC modeling and learning. In Appendix C.4, we include additional benchmark results on the WikiText dataset (Merity et al., 2016).

7. Conclusion

We proposed PyJuice, a novel system that supports training and inference of probabilistic circuits. PyJuice is orders of magnitude faster and much more memory efficient than even very recent baselines. We hope PyJuice can boost future research on tractable deep generative models by allowing for efficient training of large-scale architectures.

Acknowledgements

This work was funded in part by the DARPA PTG Program under award HR00112220005, the DARPA ANSR program under award FA8750-23-2-0004, and the NSF grant #IIS-1943641. We thank Honghua Zhang, Pasha Khosravi, and Poorva Garg for providing valuable feedback during the development of PyJuice.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Ahmed, K., Teso, S., Chang, K.-W., Van den Broeck, G., and Vergari, A. Semantic probabilistic layers for neuro-symbolic learning. In *Advances in Neural Information Processing Systems 35 (NeurIPS)*, 2022a.

Ahmed, K., Wang, E., Chang, K.-W., and Van den Broeck, G. Neuro-symbolic entropy regularization. In *Proceedings of the 38th Conference on Uncertainty in Artificial Intelligence (UAI)*, 2022b.

Ahmed, K., Chang, K.-W., and Van den Broeck, G. A pseudo-semantic loss for deep autoregressive models with logical constraints. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, 2023a.

Ahmed, K., Zeng, Z., Niepert, M., and Van den Broeck, G. Simple: A gradient estimator for k-subset sampling. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023b.

Choi, Y., Vergari, A., and Van den Broeck, G. Probabilistic circuits: A unifying framework for tractable probabilistic models. *techreport*, 2020. URL http://starai.cs.ucla.edu/papers/ProbCirc20.pdf.

Choi, Y., Dang, M., and Van den Broeck, G. Group fairness by probabilistic modeling with latent fair decisions. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, 2021.

Correia, A., Peharz, R., and de Campos, C. P. Joints in random forests. *Advances in Neural Information Processing Systems*, 33:11404–11415, 2020.

Correia, A. H., Gala, G., Quaeghebeur, E., de Campos, C., and Peharz, R. Continuous mixtures of tractable probabilistic models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 7244–7252, 2023.

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

Dang, M., Vergari, A., and Van den Broeck, G. Strudel: Learning structured-decomposable probabilistic circuits. In *International Conference on Probabilistic Graphical Models*, pp. 137–148. PMLR, 2020.

Dang, M., Khosravi, P., Liang, Y., Vergari, A., and Van den Broeck, G. Juice: A julia package for logic and probabilistic circuits. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 16020–16023, 2021.

Dang, M., Liu, A., and Van den Broeck, G. Sparse probabilistic circuits via pruning and growing. *Advances in Neural Information Processing Systems*, 35:28374–28385, 2022.

Darwiche, A. A logical approach to factoring belief networks. *KR*, 2:409–420, 2002.

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

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Gala, G., de Campos, C., Peharz, R., Vergari, A., and Quaeghebeur, E. Probabilistic integral circuits. In *International Conference on Artificial Intelligence and Statistics*, pp. 2143–2151. PMLR, 2024.

Gens, R. and Pedro, D. Learning the structure of sum-product networks. In *International conference on machine learning*, pp. 873–880. PMLR, 2013.

Lin, B. Y., Zhou, W., Shen, M., Zhou, P., Bhagavatula, C., Choi, Y., and Ren, X. Commongen: A constrained text generation challenge for generative commonsense reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1823–1840, 2020.

Liu, A. and Van den Broeck, G. Tractable regularization of probabilistic circuits. *Advances in Neural Information Processing Systems*, 34:3558–3570, 2021.

Liu, A., Mandt, S., and Van den Broeck, G. Lossless compression with probabilistic circuits. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2022.

Liu, A., Zhang, H., and Van den Broeck, G. Scaling up probabilistic circuits by latent variable distillation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023a.

Liu, A., Niepert, M., and Van den Broeck, G. Image inpainting via tractable steering of diffusion models. 2024.

Liu, X., Liu, A., Van den Broeck, G., and Liang, Y. Expressive modeling is insufficient for offline rl: A tractable inference perspective. *arXiv preprint arXiv:2311.00094*, 2023b.

Liu, X., Liu, A., Van den Broeck, G., and Liang, Y. Understanding the distillation process from deep generative models to tractable probabilistic circuits. In *International Conference on Machine Learning*, pp. 21825–21838. PMLR, 2023c.

Liu, Z., Luo, P., Wang, X., and Tang, X. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, 2015.

Loconte, L., Di Mauro, N., Peharz, R., and Vergari, A. How to turn your knowledge graph embeddings into generative models. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Loconte, L., Sladek, A. M., Mengel, S., Trapp, M., Solin, A., Gillis, N., and Vergari, A. Subtractive mixture models via squaring: Representation and learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

Lowd, D. and Rooshenas, A. The libra toolkit for probabilistic models. *Journal of Machine Learning Research*, 16:2459–2463, 2015.

Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., and Raedt, L. D. Deepproblog: neural probabilistic logic programming. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, 2018.

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

Mathur, S., Gogate, V., and Natarajan, S. Knowledge intensive learning of cutset networks. In *Uncertainty in Artificial Intelligence*, pp. 1380–1389. PMLR, 2023.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Molina, A., Vergari, A., Stelzner, K., Peharz, R., Subramani, P., Di Mauro, N., Poupart, P., and Kersting, K. Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks. *arXiv preprint arXiv:1901.03704*, 2019.

Murphy, K., Linderman, S., Chang, P. G., Li, X., Kara, A., Harper-Donnelly, G., and Duran-Martin, G. Dynamax, 2023. URL https://github.com/probml/dynamax.

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

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

Poon, H. and Domingos, P. Sum-product networks: A new deep architecture. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pp. 689–690. IEEE, 2011.

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

Qian, C., Manolache, A., Ahmed, K., Zeng, Z., Van den Broeck, G., Niepert, M., and Morris, C. Probabilistic task-adaptive graph rewiring. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

Rabiner, L. and Juang, B. An introduction to hidden markov models. *ieee assp magazine*, 3(1):4–16, 1986.

Rahman, T., Kothalkar, P., and Gogate, V. Cutset networks: A simple, tractable, and scalable approach for improving the accuracy of chow-liu trees. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2014, Nancy, France, September 15-19, 2014. Proceedings, Part II 14*, pp. 630–645. Springer, 2014.

Shah, N., Olascoaga, L. I. G., Zhao, S., Meert, W., and Verhelst, M. 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, 2021.

Vergari, A., Choi, Y., Peharz, R., and Van den Broeck, G. Probabilistic circuits: Representations, inference, learning and applications. *AAAI Tutorial*, 2020.

Vergari, A., Choi, Y., Liu, A., Teso, S., and Van den Broeck, G. A compositional atlas of tractable circuit operations for probabilistic inference. *Advances in Neural Information Processing Systems*, 34:13189–13201, 2021.

Wang, B. and Kwiatkowska, M. Compositional probabilistic and causal inference using tractable circuit models. In *International Conference on Artificial Intelligence and Statistics*, pp. 9488–9498. PMLR, 2023.

Xu, J., Zhang, Z., Friedman, T., Liang, Y., and Van den Broeck, G. A semantic loss function for deep learning with symbolic knowledge. In *Proceedings of the 35th International Conference on Machine Learning*, 2018.

Yang, Y., Gala, G., and Peharz, R. Bayesian structure scores for probabilistic circuits. In *International Conference on Artificial Intelligence and Statistics*, pp. 563–575. PMLR, 2023.

Yao, L., Trapp, M., Periasamy, K., Leslin, J., Singh, G., and Andraud, M. Logarithm-approximate floating-point multiplier for hardware-efficient inference in probabilistic circuits. In *The 6th Workshop on Tractable Probabilistic Modeling*, 2023.

Zhang, H., Dang, M., Peng, N., and Van den Broeck, G. Tractable control for autoregressive language generation. In *International Conference on Machine Learning*, pp. 40932–40945. PMLR, 2023.

A. Algorithm Details

In this section, we provide additional details of the design of PyJuice. Specifically, we introduce the layer partitioning algorithm that divides a layer into groups of node blocks with a similar number of children in Appendix A.1, and describe the details of the backpropagation algorithm in Appendix A.2.

A.1. The Layer Partitioning Algorithm

The layer partitioning algorithm receives as input a vector of integers `nchs` where each number denotes the number of child node blocks connected to a node block in the layer. It also receives as input the maximum number of groups to be considered (denoted `G`) and a sparsity tolerance threshold $\texttt{tol} \in (0, 1]$. Our goal is to search for a set of n (at most `G`) groups with capacities $g_1, \ldots, g_n$, respectively. Every number in `nchs` is then placed into the group with the smallest capacity it can fit in. Every number in `nchs` must fit in a group. Assume there are k_i numbers assigned to group i, the overhead/cost w.r.t. a partitioning $\{g_1, \ldots, g_n\}$ is defined as $\sum_{i \in [n]} k_i \cdot g_i$. Our goal is to find a partitioning with overhead smaller than $\texttt{sum}(\texttt{nchs}) \cdot (1 + \texttt{tol})$.

Algorithm 2 Partition a layer into groups

Inputs: a list of child node (block) counts of the current layer $\texttt{nchs} \in \mathbb{Z}^N$ (N is the number of node blocks in the layer)
Inputs: the maximum number of groups `G`, the sparsity tolerance threshold $\texttt{tol} \in (0, 1]$
$\texttt{uni\_nchs}, \texttt{counts} \leftarrow \texttt{unique}(\texttt{nchs}, \texttt{sorted} = \texttt{True})$ (get the unique values and their appearance counts; we require the numbers in `uni_nchs` to be sorted in ascending order)
$\texttt{L} \leftarrow \texttt{length}(\texttt{uni\_nchs})$
$\texttt{target\_overhead} \leftarrow \lceil \texttt{sum}(\texttt{uni\_nchs} * \texttt{counts}) * (1.0 + \texttt{tol}) \rceil$ (get the target overhead)
$\texttt{cum\_counts} \leftarrow \texttt{cumsum}(\texttt{counts})$
$\texttt{dp}, \texttt{backtrace} \leftarrow (0)_{\texttt{L} \times \texttt{G}+1} \in \mathbb{R}^{\texttt{L} \times \texttt{G}+1}, (0)_{\texttt{L} \times \texttt{G}+1} \in \mathbb{Z}^{\texttt{L} \times \texttt{G}+1}$
for $i = 0$ **to** $\texttt{L} - 1$ **do**
  $\texttt{dp}[i, 1] \leftarrow \texttt{uni\_nchs}[i] * \texttt{cum\_counts}[i]$
Main DP algorithm
$\texttt{target\_n\_group} \leftarrow \texttt{G}$
for $\texttt{n\_group} = 2$ **to** `G` **do**
  $\texttt{dp}[0, \texttt{n\_group}] \leftarrow \texttt{uni\_nchs}[0] * \texttt{cum\_counts}[0]$
  $\texttt{backtrace}[0, \texttt{n\_group}] \leftarrow 0$
  for $\texttt{i} = 1$ **to** $\texttt{L} - 1$ **do**
    $\texttt{min\_overhead}, \texttt{best\_idx} \leftarrow \texttt{inf}, -1$
    for $\texttt{j} = 0$ **to** $\texttt{i} - 1$ **do**
      $\texttt{curr\_overhead} \leftarrow \texttt{dp}[\texttt{j}, \texttt{n\_group} - 1] + \texttt{uni\_nchs}[\texttt{i}] * (\texttt{cum\_counts}[\texttt{i}] - \texttt{cum\_counts}[\texttt{j}])$
      if $\texttt{curr\_overhead} < \texttt{min\_overhead}$ **then**
        $\texttt{min\_overhead}, \texttt{best\_idx} \leftarrow \texttt{curr\_overhead}, \texttt{j}$
    $\texttt{dp}[\texttt{i}, \texttt{n\_group}], \texttt{backtrace}[\texttt{i}, \texttt{n\_group}] \leftarrow \texttt{min\_overhead}, \texttt{best\_idx}$
  if $\texttt{dp}[-1, \texttt{n\_group}] <= \texttt{target\_overhead}$ **then**
    $\texttt{target\_n\_group} \leftarrow \texttt{n\_group}$
Backtrace
$\texttt{group\_sizes} \leftarrow (0)_{\texttt{target\_n\_group}} \in \mathbb{Z}^{\texttt{target\_n\_group}}$
$\texttt{i} \leftarrow \texttt{L} - 1$
for $n = \texttt{target\_n\_group}$ **to** 1 **do**
  $\texttt{group\_sizes}[n - 1] \leftarrow \texttt{i}$
  $\texttt{i} \leftarrow \texttt{backtrace}[\texttt{i}, \texttt{target\_n\_group}]$
return `group_sizes`

We use a dynamic programming algorithm that is based on the following main idea. We first sort the numbers in `nchs` in ascending order. Denote `L` as the size of `nchs`, we maintain a scratch table of size $\texttt{L} \times \texttt{G}$ whose ith row and jth column indicates the best possible overhead achieved by the first i numbers in `nchs` when having in total at most j partitions. The update formula of the DP table is

$$\texttt{dp}[i, j] \leftarrow \min_{k \in [i-1]} \texttt{dp}[k, j - 1] + \texttt{nchs}[i] \cdot (i - k), \tag{3}$$

where we try to find the best place (k) to put a new group/partition. By simultaneously maintaining a matrix for backtracking, we can retrieve the best partition found by the algorithm.

The algorithm is shown in Algorithm 2. A practical trick to speed it up is to coalesce the identical values in `nchs` as done in line 3. Lines 7-9 initialize the buffers, and lines 11-23 are the main loop of the DP algorithm. Finally, the result partitioning is retrieved using lines 25-29.

Theoretical guarantee. Algorithm 2 is guaranteed to find an optimal grouping given a pre-specified number of groups, and is fairly efficient in practice. We formally state the problem in the following and provide the proof and analysis as follows.

As described in Appendix A.1, the grouping algorithm essentially takes as input a list of "# child node blocks" for each parent node block in a layer, and the goal is to partition all parent node blocks into K groups such that we minimize the following cost: the sum of the cost of each group, where the cost of a group is the maximum "# child node blocks" in the group times the number of parent node blocks in the group. In the following, we first demonstrate that the proposed dynamic programming (DP) algorithm (Algorithm 2) can retain the optimal cost for every K. We then proceed to analyze the time and space complexity of the algorithm.

To simplify notations, we assume the input is a vector of integers $[n_1, \ldots, n_N]$. We assume without loss of generality that the numbers are sorted because if not, we can apply any sorting algorithm. The main idea of the DP algorithm is to maintain a table termed dp of size N times K, where $\texttt{dp}[i, j]$ indicates the optimal cost when partitioning the first i integers into j groups. For the base cases, we can set $\texttt{dp}[i, 1] = n_i(\forall i)$ and $\texttt{dp}[1, j] = n_1(\forall j)$. For the inductive case, we have Equation (3). It is straightforward to verify that when $\texttt{dp}[k, j-1](\forall k \in [1, i-1])$ are optimal, $\texttt{dp}[i, j]$ is also optimal. Therefore, for any K, Algorithm 2 computes the optimal grouping strategy for K groups.

Efficiency. We then focus on the runtime. Given N and K, Algorithm 2 requires $\mathcal{O}(KN^2)$ runtime and $\mathcal{O}(KN)$ memory, which is undesired for large N (in practice, we set K to be smaller than 10). However, as demonstrated in Algorithm 2 (line 3), we only need to enumerate through the unique values in $[n_1, \ldots, n_N]$, which could potentially lower the computation cost significantly. Even when we are dealing with highly non-structured PCs, we can always round the numbers up to a minimum integer that is divisible by a small integer such as 10. This allows us to achieve a decent approximated solution with much less computation time.

A.2. Details of the Backpropagation Algorithm for Sum Layers

Algorithm 3 Backward pass of a sum layer group w.r.t. parameters

Inputs: log-probs of product nodes $\boldsymbol{l}_{\text{prod}}$, log-probs of sum nodes $\boldsymbol{l}_{\text{sum}}$, flows of sum nodes $\texttt{f}_{\text{sum}}$, flattened parameter vector $\boldsymbol{\theta}_{\text{flat}}$, `sum_ids`, `prod_ids`, `param_ids`
Inputs: # sum nodes: M, # product nodes: N, batch size: B
Inputs: block sizes K_M, K_N, K_B for the sum node, product node, and batch dimensions, respectively
Inputs: number of sum node blocks C_M; number of product node blocks C_N; number of batch blocks C_B
Outputs: flows of params $\texttt{f}_{\text{params}}$
Kernel launch: schedule to launch $C_M \times C_N$ thread-blocks with $\texttt{m}=0, \ldots, C_M-1$ and $\texttt{n}=0, \ldots, C_N-1$
$\texttt{cum} \leftarrow (0)_{K_M \times K_N} \in \mathbb{R}^{K_M \times K_N}$ ▷ Scratch space on SRAM
$\texttt{ms}, \texttt{me} \leftarrow \texttt{sum\_ids}[\texttt{m}], \texttt{sum\_ids}[\texttt{m}] + K_M$
$\texttt{ns}, \texttt{ne} \leftarrow \texttt{prod\_ids}[\texttt{m}, \texttt{n}], \texttt{prod\_ids}[\texttt{m}, \texttt{n}] + K_N$
for $\texttt{b} = 0$ **to** $C_B - 1$ **do**
  $\texttt{bs}, \texttt{be} \leftarrow \texttt{b} \cdot K_B, (\texttt{b}+1) \cdot K_B$ ▷ Start and end batch index
  Load $\boldsymbol{f}_{\text{s}} \leftarrow \texttt{f}_{\text{sum}}[\texttt{ms:me}, \texttt{bs:be}] \in \mathbb{R}^{K_M \times K_B}$ and $\boldsymbol{l}_{\text{s}} \leftarrow \boldsymbol{l}_{\text{sum}}[\texttt{ms:me}, \texttt{bs:be}] \in \mathbb{R}^{K_M \times K_B}$ to SRAM
  Load $\boldsymbol{l}_{\text{p}} \leftarrow \boldsymbol{l}_{\text{prod}}[\texttt{ns:ne}, \texttt{bs:be}] \in \mathbb{R}^{K_N \times K_B}$ to SRAM
  $\texttt{log\_nf} \leftarrow \log(\boldsymbol{f}_{\text{s}}) - \boldsymbol{l}_{\text{s}}$
  $\texttt{log\_nf\_max} \leftarrow \texttt{max}(\texttt{log\_nf}, \texttt{dim}=0) \in \mathbb{R}^{1 \times K_B}$ ▷ Compute on chip
  $\texttt{log\_nf\_sub} \leftarrow \texttt{exp}(\texttt{log\_nf} - \texttt{log\_nf\_max}) \in \mathbb{R}^{K_M \times K_B}$
  $\texttt{scaled\_emars} \leftarrow \texttt{transpose}(\texttt{exp}(\boldsymbol{p}_{\text{p}} + \texttt{log\_nf\_max})) \in \mathbb{R}^{K_B \times K_N}$
  $\texttt{partial\_flows} \leftarrow \texttt{matmul}(\texttt{log\_nf\_sub}, \texttt{scaled\_emars}) \in \mathbb{R}^{K_M \times K_N}$ ▷ With Tensor Cores
  $\texttt{cum} \leftarrow \texttt{cum} + \texttt{partial\_flows}$
$\texttt{ps}, \texttt{pe} \leftarrow \texttt{param\_ids}[\texttt{m}, \texttt{n}], \texttt{param\_ids}[\texttt{m}, \texttt{n}] + K_M \cdot K_N$
$\texttt{f}_{\text{params}}[\texttt{ps:pe}] \leftarrow \texttt{f}_{\text{params}}[\texttt{ps:pe}] + \boldsymbol{\theta}_{\text{flat}}[\texttt{ps:pe}] * \texttt{cum.view}(K_M * K_N)$

We compute the backward pass with respect to the inputs and the parameters of the sum layer in two different kernels as we need two different layer partitioning strategies to improve efficiency. In the following, we first introduce the backpropagation algorithm for the parameters since it reuses the index tensors compiled for the forward pass (i.e., `sum_ids`, `prod_ids`, and

`param_ids`).

The algorithm is shown in Algorithm 3. In addition to the log-probabilities of the product nodes (i.e., $\boldsymbol{l}_{\text{prod}}$), the log-probabilities of the sum nodes (i.e., $\boldsymbol{l}_{\text{sum}}$), and the flattened parameters (i.e., $\boldsymbol{\theta}_{\text{flat}}$), the algorithm takes as input the flows $\mathtt{f}_{\text{sum}}$ computed for the sum nodes. Following Definition 2, we can compute the flow w.r.t. the sum parameters as

$$\mathrm{F}_{n,c}(\boldsymbol{x}) := \theta_{n,c} \cdot p_c(\boldsymbol{x})/p_n(\boldsymbol{x}) \cdot \mathrm{F}_n(\boldsymbol{x}).$$

Similar to Algorithm 1, we partition the sum nodes, product nodes, and samples into blocks of size K_M, K_N, and K_B, respectively. We schedule to launch $C_M \times C_N$ thread-blocks, each responsible for computing the parameter flows for a block of $K_M \times K_N$ parameter flows. The main loop (line 10) iterates through blocks of K_B samples. In every iteration, we first load the log-probabilities (i.e., $\boldsymbol{l}_{\text{s}}$ and $\boldsymbol{l}_{\text{p}}$) and the sum node flows (i.e., $\boldsymbol{f}_{\text{s}}$) to compute the partial flow $p_c(\boldsymbol{x})/p_n(\boldsymbol{x}) \cdot \mathrm{F}_n(\boldsymbol{x})$ for the block of samples (note that this equals $\mathrm{F}_{n,c}(\boldsymbol{x})/\theta_{n,c}$. The partial flows are accumulated in the matrix `cum` initialized in line 7. After processing all blocks of samples, we add back the parameter flows by accumulating `cum` $*$ [the corresponding parameters] in line 21.

As elaborated in Section 5, if we use the same set of index tensors used in the forward pass, we have the problem of different thread-blocks needing to write (partial) flows to the same input product node blocks. Therefore, we do a separate compilation step for the backward pass. Consider a sum layer with sum node blocks of size K_M and child product node blocks of size K_N. We first partition the C_N children into groups such that every child node block in a group has a similar number of parents. This is done by the dynamic programming algorithm described in Appendix A.1.

Similar to the compilation procedure of the forward pass, for a group with C_N child node blocks (assume every block has C_M blocks of parents), we generate three index tensors: $\mathtt{ch\_ids} \in \mathbb{Z}^{C_N}$ and $\mathtt{par\_ids}, \mathtt{par\_param\_ids} \in \mathbb{Z}^{C_N \times C_M}$. `ch_ids` contains the initial index of all C_N child node blocks belonging to the group. For the ith node block in the group (i.e., the product node block with the initial index $\mathtt{ch\_ids}[i]$), $\mathtt{par\_ids}[i, :]$ encode the start indices of its parent sum node blocks, and $\mathtt{par\_param\_ids}[i, :]$ represent the corresponding initial parameter indices.

The main algorithmic procedure is very similar to *Algorithm 1*. Specifically, the kernel schedules to launch $C_N \times C_B$ thread-blocks each computing a block of $K_N \times K_B$ product node flows. In the main loop (line 9), we iterate through all C_M parent node blocks. In lines 13-16, we are essentially computing $\theta_{n,c}/p_n(\boldsymbol{x}) \cdot \mathrm{F}_n(\boldsymbol{x})$ (notations inherited from Definition 2) for the block of $K_N \times K_B$ values using the logsumexp trick. Finally, we store the results back to $\mathtt{f}_{\text{prod}}$.

A.3. PCs with Tied Parameters

Formally, PCs with tied parameters are PCs containing same sub-structures in different parts of its DAG. Although the nodes in these sub-structures could have different semantics, they can have shared/tied parameters. For example, in homogeneous HMMs, although the transition probabilities between different pairs of consecutive latent variables are represented by different sets of nodes and edges in the PC, they all have the *same* set of probability parameters.

PyJuice can be readily adapted to PCs with tied parameters. For the forward pass, we just need the compiler to assign the same parameter indices in `param_ids`. Similarly, we only need to slightly change the compilation procedure of `par_param_ids`. One notable difference is that in the backward pass w.r.t. the parameters, multiple thread-blocks would need to write partial flows to the same memory addresses, which leads to inter-thread-block barriers. We implemented a memory-IO tradeoff by letting the compiler create new sets of memory addresses to store the parameter flows when the number of thread-blocks writing to the same address is greater than a predefined threshold (by default set to 4).

B. Additional Technical Details

B.1. Block-Sparsity of Common PC Structures

Most commonly-adopted PC structures such as PD (Poon & Domingos, 2011), RAT-SPN (Peharz et al., 2020b), and HCLT (Liu & Van den Broeck, 2021) have block-sparse sum layers because one of the key building blocks of the structure is a set of sum nodes fully connected to their inputs. Therefore, every sum layer must contain multiple fully-connected blocks of sum and product nodes, and hence they are block sparse.

Algorithm 4 Backward pass of a sum layer group w.r.t. inputs

Inputs: log-probs of product nodes $\boldsymbol{l}_{\text{prod}}$, log-probs of sum nodes $\boldsymbol{l}_{\text{sum}}$, flows of sum nodes $\texttt{f}_{\text{sum}}$, flattened parameter vector $\boldsymbol{\theta}_{\text{flat}}$, $\texttt{ch\_ids}, \texttt{par\_ids}, \texttt{par\_param\_ids}$
Inputs: # sum nodes: M, # product nodes: N, batch size: B
Inputs: block sizes K_M, K_N, K_B for the sum node, product node, and batch dimensions, respectively
Inputs: number of sum node blocks C_M; number of product node blocks C_N; number of batch blocks C_B
Outputs: flows of inputs $\texttt{f}_{\text{prod}}$
Kernel launch: schedule to launch $C_N \times C_B$ thread-blocks with $\texttt{n}=0,\dots,C_N-1$ and $\texttt{b}=0,\dots,C_B-1$
$\texttt{cum} \leftarrow (-\infty)_{K_N \times K_B} \in \mathbb{R}^{K_N \times K_B}$ ▷ Scratch space on SRAM
$\texttt{bs}, \texttt{be} \leftarrow \texttt{b} \cdot K_B, (\texttt{b}+1) \cdot K_B$
for $\texttt{m} = 0$ **to** $C_M - 1$ **do**
  $\texttt{ps}, \texttt{pe} \leftarrow \texttt{par\_param\_ids}[\texttt{n}, \texttt{m}]$
  Load $\boldsymbol{f}_{\text{s}} \leftarrow \texttt{f}_{\text{sum}}[\texttt{ms:me}, \texttt{bs:be}] \in \mathbb{R}^{K_M \times K_B}$ and $\boldsymbol{l}_{\text{s}} \leftarrow \boldsymbol{l}_{\text{sum}}[\texttt{ms:me}, \texttt{bs:be}] \in \mathbb{R}^{K_M \times K_B}$ to SRAM
  Load $\boldsymbol{\theta} \leftarrow \texttt{transpose}(\boldsymbol{\theta}_{\text{flat}}[\texttt{ps:pe}].\texttt{view}(K_M, K_N)) \in \mathbb{R}^{K_N \times K_M}$ to SRAM
  $\texttt{log\_nf} \leftarrow \log(\boldsymbol{f}_{\text{s}}) - \boldsymbol{l}_{\text{s}}$
  $\texttt{log\_nf\_max} \leftarrow \texttt{max}(\texttt{log\_nf}, \texttt{dim}=0) \in \mathbb{R}^{1 \times K_B}$ ▷ Compute on chip
  $\texttt{log\_nf\_sub} \leftarrow \texttt{exp}(\texttt{log\_nf} - \texttt{log\_nf\_max}) \in \mathbb{R}^{K_M \times K_B}$
  $\texttt{partial\_flows} \leftarrow \texttt{matmul}(\boldsymbol{\theta}, \texttt{log\_nf\_sub}) \in \mathbb{R}^{K_M \times K_N}$ ▷ With Tensor Cores
  $\texttt{cum} \leftarrow \texttt{where}(\texttt{log\_nf\_max} > \texttt{cum},$
    $\texttt{log}(\texttt{partial\_flows} + \texttt{exp}(\texttt{cum} - \texttt{log\_nf\_max}) + \texttt{log\_nf\_max},$
    $\texttt{log}(\texttt{exp}(\texttt{log\_nf\_max} - \texttt{cum}) \cdot \texttt{partial\_flows} + 1) + \texttt{cum})$
$\texttt{ns}, \texttt{ne} \leftarrow \texttt{ch\_ids}[\texttt{n}], \texttt{ch\_ids}[\texttt{n}] + K_N$
$\texttt{f}_{\text{prod}}[\texttt{ns:ne}, \texttt{bs:be}] \leftarrow \exp(\texttt{cum} + \boldsymbol{l}_{\text{prod}}[\texttt{ns:ne}, \texttt{bs:be}])$

B.2. Relation Between PC Flows and Gradients

We first show the equality for the node flows:

$$\mathrm{F}_n(\boldsymbol{x}) = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})}. \tag{4}$$

We do the proof by induction. As a base case, we have by definition that $\mathrm{F}_{n_r}(\boldsymbol{x}) = \partial \log p_{n_r}(\boldsymbol{x}) / \partial \log p_{n_r}(\boldsymbol{x}) = 1$.

Next, suppose n is a sum or an input node, and for all its parents m, we have Equation (4) is satisfied by induction. Since all parents of n are product nodes, we have

$$\mathrm{F}_n(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} \mathrm{F}_m(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_m(\boldsymbol{x})} = \sum_{m \in \mathsf{pa}(n)} \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_{n \to m}(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})},$$

where $p_{n \to m}(\boldsymbol{x})$ denotes the probability carried by the edge from n to m.

Finally, suppose n is a product node and thus all its parents are sum nodes. We have

$$\mathrm{F}_n(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} \frac{\theta_{m,n} \cdot p_n(\boldsymbol{x})}{p_m(\boldsymbol{x})} \cdot \mathrm{F}_m(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} \frac{\theta_{m,n} \cdot p_n(\boldsymbol{x})}{p_m(\boldsymbol{x})} \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_m(\boldsymbol{x})}, \tag{5}$$

$$= \sum_{m \in \mathsf{pa}(n)} \theta_{m,n} \cdot p_n(\boldsymbol{x}) \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_m(\boldsymbol{x})}. \tag{6}$$

Denote $p_{n \to m}(\boldsymbol{x}) = \theta_{m,n} \cdot p_n(\boldsymbol{x})$ as the probability carried on the edge (m, n). Since $p_m(\boldsymbol{x}) = \sum_{n' \in \mathsf{ch}(m)} p_{n' \to m}(\boldsymbol{x})$, we have

$$\forall n \in \mathsf{ch}(m), \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_m(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_{n \to m}(\boldsymbol{x})}.$$

Plug in the above equation on $\mathrm{F}_n(\boldsymbol{x})$, this results in

$$\mathrm{F}_n(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} p_{n \to m}(\boldsymbol{x}) \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_{n \to m}(\boldsymbol{x})} = \sum_{m \in \mathsf{pa}(n)} \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_{n \to m}(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})}. \tag{7}$$

We move on to demonstrate the following relation:

$$\mathrm{F}_{n,c}(\boldsymbol{x}) = \theta_{n,c} \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \theta_{n,c}} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log \theta_{n,c}},$$

where n is a sum node and c is one of its children. We reuse the results derived in Equations (6) and (7), where we replace n with c and m with n:

$$\mathrm{F}_{n,c}(\boldsymbol{x}) = \frac{\theta_{n,c} \cdot p_c(\boldsymbol{x})}{p_n(\boldsymbol{x})} \cdot \mathrm{F}_n(\boldsymbol{x}) = \theta_{n,c} \cdot p_c(\boldsymbol{x}) \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_n(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_{c \to n}(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log \theta_{n,c}}.$$

C. Experimental Details

C.1. The Adopted Block-Sparse PC Layer

The PC layer contains 200 independent fully-connected sets of nodes. Every connected subset consists of 1024 sum nodes and 1024 product nodes. When compiling the layer, we divide the layer into blocks of size 32. When dropping 32×32 edge blocks from the layer, we ensure that every sum node has at least one child.

C.2. Details of Training the HMM Language Model

Following Zhang et al. (2023), we first fine-tune a GPT-2 model with the CommonGen dataset. We then sample 8M sequences of length 32 from the fine-tuned GPT-2. After initializing the HMM parameters with latent variable distillation, we fine-tune the HMM with the sampled data. Specifically, following Zhang et al. (2023), we divide the 8M samples into 40 equally-sized subsets, and run full-batch EM on the 40 subsets repeatedly. Another set of 800K samples is drawn from the fine-tuned GPT as the validation set.

C.3. Details of Training the Sparse Image Model

Following Liu et al. (2023c), we fine-tune the model with an equivalent batch size of 6400 and a step size of 0.01 in the mini-batch EM algorithm. Specifically, suppose $\boldsymbol{\theta}$ are the current parameters, $\boldsymbol{\theta}^{\text{new}}$ are the new set of parameters computed by the EM update. Given step size α, the update formula is $\boldsymbol{\theta} \leftarrow (1-\alpha)\boldsymbol{\theta} + \alpha\boldsymbol{\theta}^{\text{new}}$.

C.4. Additional Benchmark Results

Hyperparameters of the adopted HCLTs. We adopt two HCLTs (Liu & Van den Broeck, 2021) with hidden sizes 256 and 512, respectively. The backbone CLT structure is constructed using 20,000 randomly selected training samples.

Hyperparameters of the adopted PDs. Starting from the set of all random variables, the PD structure recursively splits the subset with product nodes. Specifically, consider an image represented as a $H \times W \times C$ (H is the hight; W is the width; C is the number of channels), the PD structure recursively splits over both the height and the width coordinates, where every coordinate has a set of pre-defined split points. For both the height and the width coordinates, we add split points with interval 2. PD-mid has a hidden dimension of 128 and PD-large has 256.

Benchmark results on WikiText-103. Table 4 illustrates results on WikiText-103. We train the model on sequences with 64 tokens. We adopt two (homogeneous) HMM models, HMM-mid and HMM-large with hidden sizes 2048 and 4096, respectively.

Table 4. Density estimation performance of PCs on the WikiText-103 dataset. Reported numbers are test set perplexity.

Dataset	HMM-mid	HMM-large
WikiText-103	146.59	167.65

D. Additional Experiments

D.1. Speed of the Compilation Process

In Table 5, we show the compilation speed of PCs with different structures and different sizes. Experiments are conducted on a server with an AMD EPYC 7763 64-Core Processor and 8 RTX 4090 GPUs (we only use one GPU). The results demonstrate the efficiency of the compilation process, where even the PD model with close to 1B parameters can be compiled in around 30 seconds.

Table 5. Average ($\pm$ standard deviation of 3 runs) runtime (in seconds) of the compilation process of four PCs.

Structure	HMM	PD	HCLT	RAT-SPN
# nodes	130K	1.38M	710K	465K
# edges	130M	829M	159M	33.4M
Compilation time (s)	1.50$\pm$0.02	30.57$\pm$0.86	8.70$\pm$0.32	4.72$\pm$0.16

D.2. Runtime on Different GPUs

In addition to the RTX 4090 GPU adopted in the experiments in Table 1, we compare the runtime of PyJuice with the baselines on an NVIDIA A40 GPU. As shown in the following table, PyJuice is still significantly faster than all baselines for PCs of different sizes.

Table 6. **Average ($\pm$ standard deviation of 5 runs) runtime (in seconds) per training epoch** of 60K samples for PyJuice and the baselines on five RAT-SPNs (Peharz et al., 2020b) with different sizes. All other settings are the same as described in Section 6.1.

# nodes	58K	116K	232K	465K	930K
# edges	616K	2.2M	8.6M	33.4M	132M
EiNet	60.29$\pm$0.30	136.85$\pm$0.13	282.58$\pm$0.27	690.73$\pm$0.08	1936.28$\pm$0.26
Juice.jl	4.41$\pm$0.21	11.57$\pm$0.07	32.74$\pm$1.86	121.25$\pm$0.43	331.98$\pm$2.87
PyJuice	**1.53**$\pm$0.07	**3.11**$\pm$0.07	**6.47**$\pm$0.08	**13.62**$\pm$0.37	**30.69**$\pm$0.19

D.3. Runtime on Different Batch Sizes

As a supplement to Table 1, we report the runtime for a RAT-SPN (Peharz et al., 2020b) with $465K$ nodes and $33.4M$ edges using batch sizes $\{8, 16, 32, 64, 128, 256, 512\}$. To minimize distractions, we only record the time to compute the forward and backward process, but not the time used for EM updates. Results are shown in the table below.

Table 7. **Average ($\pm$ standard deviation of 5 runs) runtime (in seconds) per training epoch (excluding EM updates)** of 60K samples for PyJuice and the baselines on a RAT-SPNs (Peharz et al., 2020b) with $465K$ nodes and $33.4M$ edges. All other settings are the same as described in Section 6.1. OOM denotes out-of-memory.

Batch size	8	16	32	64	128	256	512
EiNet	332.87$\pm$0.21	OOM	OOM	OOM	OOM	OOM	OOM
Juice.jl	1045.04$\pm$0.06	853.15$\pm$0.03	775.87$\pm$0.02	642.54$\pm$0.04	324.23$\pm$0.02	163.68$\pm$0.02	80.57$\pm$0.01
PyJuice	**43.09**$\pm$0.04	**18.63**$\pm$0.02	**7.38**$\pm$0.01	**4.58**$\pm$0.01	**3.50**$\pm$0.01	**3.04**$\pm$0.01	**2.76**$\pm$0.03