Speeding Up Probabilistic Circuits with Extended Einsum as an Intermediate Representation

Christoph Staudt[1] **Maurice Wenig**[1]

[1]Friedrich Schiller University Jena, Jena, Germany

Abstract

Probabilistic circuits are a powerful class of models for tractable probabilistic inference and expressive generative modeling. Recent systems have made these models practical at scale by tensorizing circuit evaluation and targeting accelerator-backed array frameworks. Yet this computation has structure that standard deep learning compilers often fail to exploit. Thus, we propose *extended einsum* as an intermediate representation for tensorized probabilistic circuits. This enables us to implement several optimizations from the growing ecosystem of tools around probabilistic circuits and einsum programs. The optimizations are applied at a backend-independent level before lowering to PyTorch, JAX, or specialized kernels. Translating circuits from the cirkit library to our intermediate representation, we achieve median speedups of $1.38\times$ for CP layers and $1.16\times$ for Tucker layers, with per-configuration medians ranging from $1.08\times$ to $1.49\times$. These results demonstrate that extended einsum is a viable IR for optimizing tensorized probabilistic circuits. Visit `https://extended.einsum.org` for more information and code.

1 INTRODUCTION

A central trend in recent probabilistic circuit (PC) research is the tensorization of circuit computation. Rather than evaluating individual sum and product units as a sparse symbolic graph, modern implementations group many units with identical or compatible scopes into dense tensor operations [Peharz et al., 2020]. This representation exposes parallelism, enables batching, and allows PCs to benefit from accelerator hardware such as GPUs and TPUs. Systems such as cirkit provide a symbolic interface for specifying circuits and already implement several important transformations and optimizations [Peharz et al., 2020, Loconte et al., 2025].

However, the current software landscape also reveals a limitation: many optimizations are expressed directly in terms of a specific backend or runtime representation. Folding, parameter normalization, tensor layout choices, and kernel selection are often implemented inside a particular library and are therefore difficult to reuse, compare, or compose across systems.

In this work, we argue that tensorized probabilistic circuits benefit from an explicit intermediate representation (IR). An IR provides a level of abstraction between the symbolic circuit and the final execution backend. At this level, one can reason about the computation as a collection of structured tensor operations, apply algebraic and numerical transformations once, and then lower the optimized program to PyTorch [Ansel et al., 2024], JAX [Bradbury et al., 2018], or specialized kernels. This separation is standard in compiler design, but remains underdeveloped for probabilistic circuits, where many transformations are currently implemented as library-specific rewrites.

We propose an extended einsum language as this intermediate representation. Standard einsum notation already offers a concise description of tensor contractions and has become a common interface for expressing multilinear computation. As a result, substantial work has gone into optimizing einsum computations, making their evaluation particularly efficient [Gray and Kourtis, 2021, Staudt et al., 2024, Orgler and Blacher, 2024, Breuer et al., 2025, Won et al., 2026, Deeds et al., 2025].

2 BACKGROUND

Probabilistic circuits. A probabilistic circuit is a rooted computational graph whose leaves are tractable input distributions and whose internal nodes are weighted sums and products [Vergari et al., 2021, Choi et al., 2020]. Product nodes combine factors, while sum nodes form weighted mix-

tures of their inputs. The focus of this paper is tensorized circuits in which many units with the same role are grouped into layers. Input layers produce vectors of local likelihoods or feature values, and inner layers repeatedly combine two child vectors into new parent vectors.

Such layers are often organized through a region graph [Dennis and Ventura, 2012]. Here we focus on the quad-tree region graph introduced by Loconte et al. [2025]. These circuits are built from image-like variables arranged on a grid. The grid is recursively partitioned into quadrants until small leaf regions are reached, and the circuit combines the corresponding child regions bottom-up. The related quad-graph region graph instead splits every region both horizontally and vertically and shares the resulting subregions between the overlapping partitions, turning the tree into a DAG [Loconte et al., 2025]. For a 2×2 grid this gives four leaf inputs $t_0, \ldots, t_3$, two intermediate layers combining pairs of leaves, and one root layer combining the intermediate results (Figure 1).

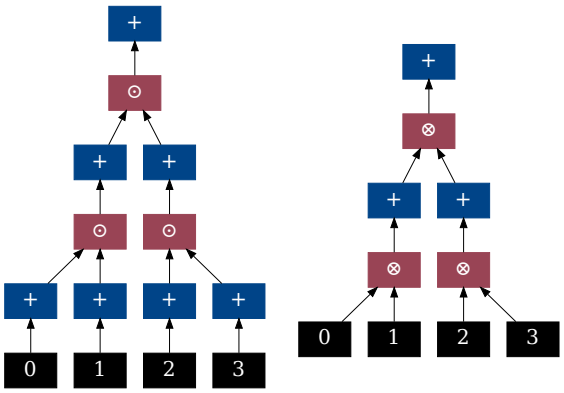

Figure 1: Quad-tree circuits generated with cirkit for CP (left) and Tucker (right) sum-product layers.

Each inner region applies a sum-product layer. Given two child vectors x and y, a Tucker layer first forms their outer product and then mixes the result with a full core tensor,

$$h_k = \sum_{i,j} x_i y_j W_{kij}.$$

A CP layer uses a factorized version of the same weight tensor,

$$h_k = \sum_{i,j} x_i y_j W^{(1)}_{ki} W^{(2)}_{kj},$$

which reduces the number of parameters and changes the resulting contraction structure. Both layers therefore implement the same circuit pattern, but expose different tensor programs to the backend.

Einsum. Einsum is a language for tensor contractions. As an example, batched matrix multiplication can be written as

$$C = \text{einsum}(bik, bkj \to bij;\ A, B),$$

which describes the same semantics as

$$\forall b, i, j : C_{bij} = \sum_k A_{bik} B_{bkj}.$$

We call "$bik, bkj \to bij$" the *format string* of the einsum operation. It generally consists of an *index string* (bik, bkj) for each argument tensor and an *output string* (bij) that describes the output format.

3 EXTENDED EINSUM

Pure einsum is insufficient to express probabilistic circuits on its own. We therefore extend einsum with element-wise nonlinearities $(\exp, \log)$, element-wise binary operations $(+, -, \times, /)$, operations that change the tensor layout (stack, take), convenience functions (softmax), and reuse of intermediate results.

The quad-tree circuit with CP layers shown in Figure 1 can be implemented for a batch of images as follows:

```
import extended_einsum as xe

def cp_layer(x_1, x_2, w_1, w_2):
    x_3 = xe.einsum("ab,cb->ac", x_1, w_1)
    x_4 = xe.einsum("ab,cb->ac", x_2, w_2)
    return xe.einsum("ab,ab->ab", x_3, x_4)

sw = map(xe.softmax, weights)
t_4 = cp_layer(t_0, t_1, sw[0], sw[1])
t_5 = cp_layer(t_2, t_3, sw[2], sw[3])
t_6 = cp_layer(t_4, t_5, sw[4], sw[5])
t_7 = xe.einsum("ab,cb->ac", t_6, sw[6])
```

where t_0, t_1, t_2, t_3 hold the four batches of input tensors and weights are parameter tensors in a backend library such as PyTorch or JAX. The corresponding extended-einsum DAG is shown in Figure 2, omitting the softmax for clarity.

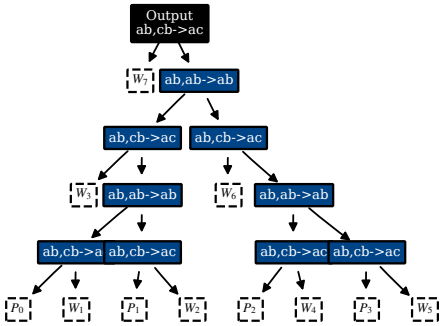

Figure 2: Extended-einsum DAG for the CP quad-tree circuit before compiler optimizations.

4 COMPILER OPTIMIZATIONS

A program written in our interface goes through the following optimization pipeline:

1. extract an internal representation of the extended einsum program
2. apply compiler optimizations to the extended einsum program
3. translate the optimized extended einsum program to backend-specific code

This pipeline is shown in Figure 3. The translation step also improves numerical stability: rather than working in log space, circuits are evaluated as a positive tensor with a separate broadcastable log scale (see Appendix F for the ablation and Appendix H for validation).

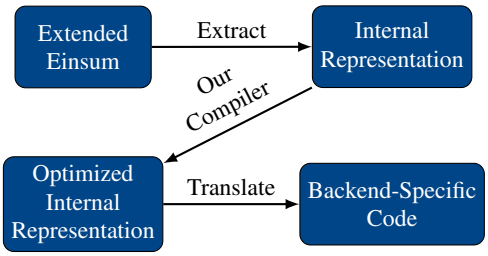

Figure 3: Compilation pipeline of a program written in our extended einsum interface.

Several optimizations are performed directly on the extended einsum IR.

4.1 CONTRACTION PATH

The contraction order induced by the layer definitions is not necessarily optimal for computational efficiency. Indeed, many layer-level optimizations admit a simple formulation as elementary rewrites of einsum expressions. As an illustrative example, adapted from Zhang et al. [2025], consider a sum layer receiving an input tensor $z \in \mathbb{R}^{m \times n}$, followed by a linear map whose parameter tensor $W \in \mathbb{R}^{k \times m \times n}$ is represented as an outer product of two parameter tensors $A \in \mathbb{R}^{k \times m}$ and $B \in \mathbb{R}^{k \times n}$. A direct implementation first materializes

$$W = \text{einsum}(ki, kj \to kij;\ A, B),$$

and then contracts the input with the resulting weights,

$$h = \text{einsum}(ij, kij \to k;\ z, W).$$

Instead, we merge these two einsum expressions into one before passing it to a contraction-path optimizer:

$$h = \text{einsum}(ij, ki, kj \to k;\ z, A, B).$$

To merge these expressions, we adapt the format string of h by replacing the input string of W (kij) with the index strings used to compute W (ki, kj). This kind of merge is possible for any nested einsum expression once index strings are made consistent across the contraction tree [Wenig et al., 2025]. From there, it can be lowered, for example, to the pairwise schedule

$$T = \text{einsum}(ij, ki \to kj;\ z, A),$$
$$h = \text{einsum}(kj, kj \to k;\ T, B).$$

This rewrite combines the construction of the parameters and the sum layer. The evaluation is still a sequence of pairwise contractions, but the full weight tensor W is no longer materialized. The optimizer merges as much of the expression DAG as possible into single einsum expressions and then applies a contraction-path optimizer such as `cgreedy` [Orgler and Blacher, 2024]. For the circuits benchmarked in Section 5, the optimizer finds no improvement over the contraction paths induced by the layer definitions and leaves them unchanged.

4.2 IR-LEVEL FOLDING

Folding is a common optimization for probabilistic circuits [Liu et al., 2024, Loconte et al., 2025, Peharz et al., 2020, The april Lab, 2024, de Campos et al., 2024]. It speeds up parallel contractions within the same layer when they share their operand shapes. The speedup comes from combining these parallel contractions into a single contraction with an additional batch index. For example, parallel matrix-matrix multiplications

$$C^{(i)} = \text{einsum}(ik, kj \to ij;\ A^{(i)}, B^{(i)})$$

with $A^{(i)} \in \mathbb{R}^{m \times r}, B^{(i)} \in \mathbb{R}^{r \times n}$ for $i \in \{1, \dots, p\}$ can be combined into a single batched matrix-matrix multiplication

$$Z = \text{einsum}(bik, bkj \to bij;\ X, Y)$$

where $X \in \mathbb{R}^{p \times m \times r}$ and $Y \in \mathbb{R}^{p \times r \times n}$ are the batched versions of the A and B matrices, respectively. By reducing the number of kernels, this optimization lowers kernel-launch overhead. To apply folding at the IR level, we automatically detect operations with matching shape information and batch them together. Figure 4 shows the resulting folded DAG for the CP quad-tree example.

4.3 MEMORY LAYOUT OPTIMIZATIONS

Folding introduces an additional axis whose entries correspond to the operations that were batched together. Consumers of a folded tensor therefore select subsets of this axis before applying the next contraction or element-wise operation. Scattered subsets force the backend to perform

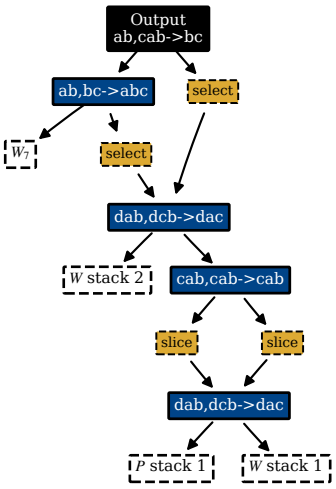

Figure 4: Extended-einsum DAG for the CP quad-tree circuit after IR-level folding.

irregular gathers or materialize copies, whereas contiguous subsets can be passed to the next kernel as slice views.

After folding, our compiler applies a layout pass that permutes stacked tensors so downstream operations read contiguous blocks whenever the dependency pattern permits it. For folded tensors that originate from input features, this permutation induces a fixed reordering of the data axes. The compiler emits this input permutation, allowing it to be applied once during data preprocessing rather than during every training step. In the 2×2 CP quad-tree example, the first folded layer consumes individual leaves and is insensitive to their order. The next layer requires elementwise products between pixel vectors 0 and 1, and between 2 and 3. Ordering the folded input axis as $0, 2, 1, 3$ makes the two operand blocks contiguous: the slice at positions 0:2 contains $(0, 2)$, while the slice at positions 2:4 contains $(1, 3)$. The folded Hadamard layer can then multiply these slices elementwise, computing both products without irregular indexing.

5 EXPERIMENTS

Experiments were run on a Linux workstation with two Intel Xeon Gold 6226R CPUs and 124 GB of host memory, using a single NVIDIA RTX A6000 GPU (48 GiB). The environment used Python 3.13.13, PyTorch 2.12.0, CUDA 13.0, NVIDIA driver 595.71.05, and cirkit 0.3.0 [The april Lab, 2024]. For both systems, we compile the complete forward-and-loss function with torch.compile in reduce-overhead mode and train on MNIST [LeCun et al., 2010] with fused Adam at learning rate 0.01. Every configuration runs in a fresh process for 30 warmup batches followed by three blocks of 30 measured batches, and we report the median per-batch time across these blocks. Each configuration has five independent paired runs that use the same minibatch permutations across systems. Error bands are 95% bootstrap intervals over these five measurements. Reported runtime is forward plus loss plus backward and excludes the optimizer update. See Appendix C for notes on PyJuice and other backends.

Training performance is evaluated on the quad-tree and quad-graph region graphs with CP sum-product layers by instantiating the same symbolic circuit in cirkit and translating it to our extended-einsum implementation (XE). We vary the number of units per layer and the batch size and report the speedup over the combined forward and backward pass.

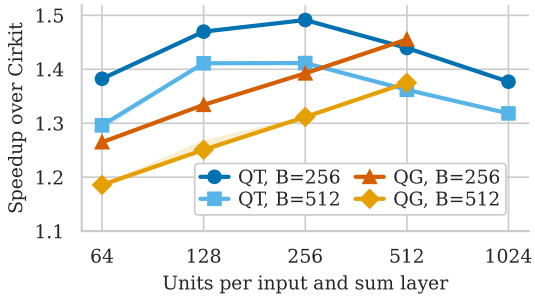

Figure 5: Extended Einsum CP-layer speedup over cirkit (higher is better). Quad-tree and quad-graph lines cover batches 256 and 512. 1024-unit quad graphs exceed GPU memory and are omitted. Shading shows 95% bootstrap intervals over five paired runs.

Figure 5 shows that XE is consistently faster across all configurations. The median configuration speedup is $1.38\times$, with per-configuration medians ranging from $1.19\times$ to $1.49\times$. The gains grow with width on the quad graph, where shared intermediate results create more opportunities for reuse-aware folding and layout. Tucker layers also improve consistently, but by a smaller median factor of $1.16\times$. Their dense multi-input contractions dominate more of the runtime, leaving a smaller fraction for the compiler to eliminate. This contrast supports the intended role of the IR: it is most valuable when dataflow and layout, rather than one dense contraction, determine performance. The complete Tucker results are reported in the supplementary material.

6 CONCLUSION AND FUTURE WORK

We presented extended einsum as an intermediate representation for tensorized probabilistic circuits. By lifting circuit computation to an explicit IR, we enable compiler optimizations (contraction path replanning, IR-level folding, and memory layout optimization) that are backend-independent and composable. Translating circuits from cirkit to our IR, we achieved consistent speedups without sacrificing backend portability. Future directions include slicing to reduce peak memory, distributed execution, and block-sparse tensor formats [Won et al., 2026].

Acknowledgements

This work was supported by the Carl-Zeiss-Stiftung within the project 'Interactive Inference'. We also thank Antonio Vergari, Charles Bricout, and Lorenzo Loconte for helpful discussions.

References

Jason Ansel, Edward Z. Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*, pages 929–947. ACM, April 2024. doi: 10.1145/3620665.3640366. URL https://docs.pytorch.org/assets/pytorch2-2.pdf.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Alexander Breuer, Mark Blacher, Max Engel, Joachim Giesen, Alexander Heinecke, Julien Klaus, and Stefan Remke. Einsum trees: An abstraction for optimizing the execution of tensor expressions. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2025. doi: 10.1145/3676641.3716254.

Siyuan Chen, Pratik Pramod Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. Ed-batch: Efficient automatic batching of dynamic neural networks via learned finite state machines. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 4514–4528. PMLR, 2023. URL https://proceedings.mlr.press/v202/chen23g.html.

YooJung Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic circuits: A unifying framework for tractable probabilistic models. Technical report, University of California, Los Angeles, October 2020.

Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the CIFAR datasets, 2017.

Cassio de Campos, Gennaro Gala, Erik Quaeghebeur, and Antonio Vergari. Scaling continuous latent variable models as probabilistic integral circuits. In *Advances in Neural Information Processing Systems 37*, NeurIPS 2024, pages 10959–10987. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2024. doi: 10.52202/079017-0350.

Kyle Deeds, Willow Ahrens, Magdalena Balazinska, and Dan Suciu. Galley: Modern query optimization for sparse tensor programs. *Proc. ACM Manag. Data*, 2025. doi: 10.1145/3725301.

Aaron Dennis and Dan Ventura. Learning the architecture of sum-product networks using clustering on variables. In F. Pereira, C.J. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 2021. doi: 10.22331/q-2021-03-15-410.

Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. MNIST handwritten digit database, 2010. URL http://yann.lecun.com/exdb/mnist.

Anji Liu, Kareem Ahmed, and Guy Van den Broeck. Scaling tractable probabilistic circuits: A systems perspective. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

Lorenzo Loconte, Antonio Mari, Gennaro Gala, Robert Peharz, Cassio de Campos, Erik Quaeghebeur, Gennaro Vessio, and Antonio Vergari. What is the relationship between tensor factorizations and circuits (and how can we exploit it)?, 2025.

Sheela Orgler and Mark Blacher. Optimizing tensor contraction paths: A greedy algorithm approach with improved cost functions, 2024.

Robert Peharz, Steven Lang, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Guy Van den Broeck, Kristian Kersting, and Zoubin Ghahramani. Einsum networks: Fast and scalable learning of tractable probabilistic circuits. In *International Conference on Machine Learning*, pages 7563–7574. PMLR, 2020.

Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proc. IEEE*, 77(2):257–286, 1989. doi: 10.1109/5.18626. URL https://doi.org/10.1109/5.18626.

Lukas Sommer, Cristian Axenie, and Andreas Koch. SPNC: an open-source mlir-based compiler for fast sum-product network inference on cpus and gpus. In Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, pages 1–11. IEEE, 2022. doi: 10.1109/CGO53902.2022.9741277. URL https://doi.org/10.1109/CGO53902.2022.9741277.

Christoph Staudt, Mark Blacher, Julien Klaus, Farin Lippmann, and Joachim Giesen. Improved Cut Strategy for Tensor Network Contraction Orders. In *22nd International Symposium on Experimental Algorithms (SEA)*, 2024. doi: 10.4230/LIPIcs.SEA.2024.27.

The april Lab. cirkit: a python framework to build, learn and reason about probabilistic circuits and tensor networks, October 2024. URL https://github.com/april-tools/cirkit.

Antonio Vergari, YooJung Choi, Anji Liu, Stefano Teso, and Guy Van den Broeck. A compositional atlas of tractable circuit operations for probabilistic inference. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 13189–13201. Curran Associates, Inc., 2021.

Maurice Wenig, Paul G. Rump, Mark Blacher, and Joachim Giesen. The syntax and semantics of einsum, 2025.

Jaeyeon Won, Willow Ahrens, Saman Amarasinghe, and Joel S. Emer. Insum: Sparse gpu kernels simplified and optimized with indirect einsums. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26. Association for Computing Machinery, 2026. doi: 10.1145/3779212.3790176.

Honghua Zhang, Meihua Dang, Benjie Wang, Stefano Ermon, Nanyun Peng, and Guy Van den Broeck. Scaling probabilistic circuits via monarch matrices. In *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25. JMLR.org, 2025.

Speeding Up Probabilistic Circuits with Extended Einsum as an Intermediate Representation (Supplementary Material)

Christoph Staudt[1] **Maurice Wenig**[1]

[1]Friedrich Schiller University Jena, Jena, Germany

A RELATED WORK

The related work cited in the main paper is spread across several sections rather than collected in the single dedicated section that some readers might expect. We therefore provide a consolidated related-work section here.

Probabilistic circuits and tensorization. Probabilistic circuits (PCs) organize tractable distributions as compositions of sums, products, and tractable input distributions. The circuit view unifies several model families and makes the structural conditions for exact inference explicit [Choi et al., 2020, Vergari et al., 2021]. Region graphs provide a common way to specify the recursive variable decompositions from which many such circuits are constructed [Dennis and Ventura, 2012]. Our work does not change these semantics or structural conditions. It starts after a tensorized circuit has been chosen and addresses how its computation is represented, optimized, stabilized, and executed.

Einsum Networks showed that many small circuit operations can be grouped into large tensor contractions, substantially improving accelerator utilization [Peharz et al., 2020]. More recent systems broaden this approach. cirkit provides a symbolic framework for constructing probabilistic circuits and tensor networks and lowers them to numerical backends [The april Lab, 2024]. PyJuice compiles PCs to a compact GPU-oriented representation designed for block-parallel execution, reduced data movement, and Tensor Core use [Liu et al., 2024]. Earlier compiler-oriented work, SPNC, defines SPN-specific MLIR dialects and lowers SPN inference to native CPU and GPU code [Sommer et al., 2022]. It operates at a different abstraction boundary from Extended Einsum: SPNC retains node-level SPN semantics and targets inference through native code generation, whereas Extended Einsum represents differentiable tensorized computations and delegates device-specific lowering to array compiler stacks.

These systems demonstrate that representation and compilation are central to PC scalability. Extended Einsum differs in where it places the abstraction boundary: it exposes contractions, explicit dataflow, layout operations, and the few required nonlinear operations in a backend-independent IR. Circuit frontends can therefore share whole-program transformations and numerical-stability passes, while execution remains delegated to backends such as PyTorch and JAX.

Tensor factorizations and scalable circuit parameterizations. The connection between tensor factorizations and circuit layers gives a systematic account of dense Tucker layers, factorized CP layers, their region graphs, and related parameterizations [Loconte et al., 2025]. This perspective motivates the layer families used in our evaluation, but it also creates a compiler problem: algebraically equivalent factorizations can induce very different contraction paths, intermediate sizes, and memory layouts. Structured parameter matrices provide a complementary route to scale. Monarch parameterizations reduce the cost of PC layers by replacing dense matrices with products of structured factors [Zhang et al., 2025], while probabilistic integral circuits extend the circuit framework to scalable continuous latent-variable models [de Campos et al., 2024]. Extended Einsum is not a new factorization or circuit family. It is intended to make such parameterizations easy to express and to optimize the resulting multi-stage tensor programs without adding a specialized backend implementation for each one.

Automatic batching and layout planning. Folding is a common optimization for probabilistic circuits [Peharz et al., 2020, Liu et al., 2024, The april Lab, 2024, Loconte et al., 2025]. It combines independent, shape-compatible operations into a single operator with an additional batch axis. More broadly, automatic-batching systems have observed that the benefits of batching depend on how operands and results are arranged in memory. Most closely related to our layout pass, ED-Batch couples dynamic batching with batching-aware memory planning so that batched operands can be accessed with less data movement [Chen et al., 2023].

Extended Einsum addresses a different, static layout problem. The order of members along a fold axis is semantically free, but determines whether later folded consumers can reuse a complete source or a contiguous slice, or instead require fragmented concatenations or indexed gathers. Our compiler therefore permutes fold axes, propagates those permutations across grouped producers and consumers, and uses an IR-level cost model with beam search to coordinate several dependent fold groups. Thus, both approaches recognize that batching and data arrangement should be optimized together, while differing in program model, optimization variables, and search procedure.

Einsum and tensor-expression optimization. Einsum is a compact notation for multilinear tensor computation; its syntax and semantics have recently been formalized independently of any particular array library [Wenig et al., 2025]. A large body of work then addresses the central execution problem of choosing a contraction order. Hyper-optimized contraction searches combine multiple heuristics and optimization strategies [Gray and Kourtis, 2021]; improved greedy cost models and graph-cut strategies target fast construction of high-quality orders [Orgler and Blacher, 2024, Staudt et al., 2024]. Einsum Trees make the chosen decomposition and execution schedule explicit, enabling architecture-aware optimization of tensor expressions [Breuer et al., 2025]. Extended Einsum uses this progress rather than replacing it: contraction-path selection is one compiler pass within a larger program representation.

Two recent systems extend the scope from dense contraction ordering toward program and kernel optimization. Galley applies database-style query optimization to sparse tensor programs, including optimization across multiple tensor operators [Deeds et al., 2025]. Insum represents indirect accesses inside einsum-like sparse GPU kernels and optimizes their implementation [Won et al., 2026]. These directions are complementary to Extended Einsum. Galley and Insum focus on sparse tensor execution, whereas our present work focuses on dense tensorized PCs and preserves circuit-wide dataflow so that it can fold compatible operations, reorder folded values according to their consumers, and insert stable arithmetic. Sparse lowering through these or similar systems is a natural future backend.

Numerical stability and scaled execution. Our scaled representation generalizes the classical rescaling used to prevent underflow in hidden Markov models [Rabiner, 1989] by automatically propagating normalized tensors and broadcastable scales through arbitrary Extended Einsum operations and tensor contractions.

Relation to general-purpose array compilation. PyTorch 2 captures and transforms tensor programs from dynamic Python, then lowers them through its compiler stack [Ansel et al., 2024]; JAX provides composable program transformations and accelerator compilation for NumPy-style programs [Bradbury et al., 2018]. Extended Einsum sits above these frameworks. Retaining contraction indices, parameter provenance, and circuit-wide dependencies before backend lowering enables transformations that would otherwise have to be rediscovered from a lower-level tensor graph. Conversely, the IR leaves device-specific fusion, code generation, and kernel selection to the mature backend. Its automatic log-space and scaled translations further address a PC-specific concern that contraction optimizers and general array compilers do not ordinarily encode. Thus the contribution is a domain-relevant intermediate layer between symbolic circuit libraries and general numerical compiler stacks, rather than a replacement for either.

B EXPERIMENTAL PROTOCOL

Reliable system comparisons depend on controlled and reproducible measurements, so this section specifies the workloads, hardware, timing procedure, and reported statistics. The MNIST experiments use 28×28 images with 256 categorical values per pixel. Both Extended Einsum and cirkit use fused Adam with learning rate 0.01. For both systems, the complete forward-and-loss function is compiled with torch.compile in reduce-overhead mode. Forward, loss, backward, and optimizer components are timed separately with CUDA events. Runtime figures and tables report forward and loss plus backward, excluding the optimizer update as this is identical in both systems.

Benchmarks run on a Linux workstation with two Intel Xeon Gold 6226R CPUs, 124 GB of host memory, and two NVIDIA RTX A6000 GPUs with 48 GiB of memory. Our experiments only use one GPU at a time. The environment uses Python 3.13.13, PyTorch 2.12.0, CUDA 13.0, NVIDIA driver 595.71.05, and cirkit 0.3.0.

Every backend and configuration is executed in a fresh Python process, preventing compiled graphs and CUDA allocator state from leaking between runs. After 30 complete warmup batches, we measure three consecutive blocks of 30 batches and store the median per-batch time across the blocks. Each configuration has five independent paired runs. Configuration blocks are shuffled, and backend order is reversed in alternating blocks. Error bars and confidence bands are 95% bootstrap intervals over the five paired measurements. Peak memory is the maximum allocated CUDA memory observed across construction, compilation, warmup, and measurement.

We define speedup as the runtime of the named baseline divided by the runtime of the compared system. Thus, values above one favor the compared system. In the main Extended Einsum comparisons, this is the cirkit runtime divided by the Extended Einsum runtime.

The full Extended Einsum pipeline combines scaled-max stability, detached reference shifts, input-depth folding, contraction-path optimization, and consumer-aware ordering. The cirkit baseline is unmodified and retains its native log-space evaluation. Configurations that do not fit on the 48 GiB RTX A6000 are excluded from the publication grid.

C BENCHMARK SELECTION

We compare primarily against cirkit because it can instantiate the same symbolic circuits and the same training objective. We omit PyJuice [Liu et al., 2024] from the CP comparison because it has no parameter-matched counterpart of our CP layers: PyJuice inserts an additional product layer after the inputs when they are passed directly to a sum layer, so the closest implementation carries more parameters and would make the comparison unfair. An exact match is possible for transposed-CP layers, where each region applies an element-wise product of its two children followed by a single dense sum. In separate experiments on that layer type with identical input, product, and sum counts, scopes, output widths, and logical parameter counts, XE is within 7% of PyJuice in the forward pass for the narrower configurations and $1.38\times$ faster than PyJuice at batch size 512 with 512 units. Even that comparison is not optimizer-equivalent, since native PyJuice backward computes positive EM parameter flows rather than log-likelihood gradients. Note also that PyJuice kernels are designed for block-sparse circuits, whereas the circuits evaluated here are dense, and that on the RTX A6000 PyJuice cannot use its compute-capability-9.0 kernels. Appendix J reports that parameter-matched comparison in full.

We omit the JAX and NumPy backends from the main comparison, since cirkit supports only PyTorch. In our experiments, JAX was between 0.3% and 22% slower than PyTorch on the CP configurations, but the optimized JAX program still ran $1.20\times$ to $1.37\times$ faster than cirkit. NumPy was much slower and not competitive for training. Appendix G repeats the ablations on the JAX/XLA stack.

D COMPLETE CP AND TUCKER RESULTS

Tables 1 and 2 report the median absolute runtime, paired speedup, run range, and peak-memory reduction for every benchmarked configuration. A reduction above one means Extended Einsum uses less peak allocated memory, which is the case in most configurations, but the difference is small. Extended Einsum is faster in all 26 configurations. Across configuration medians, the CP speedup is $1.376\times$, ranging from $1.186\times$ to $1.491\times$. The Tucker median is $1.158\times$, ranging from $1.081\times$ to $1.215\times$. Figure 6 and Figure 7 provide additional visualizations for these tables.

Table 1: Complete CP speedup results. Absolute times are medians over five independent runs. Speedup is the median paired cirkit/Extended Einsum ratio, followed by the minimum and maximum paired-run ratios in brackets. Memory reduction is the corresponding median ratio of peak allocated memory.

Layer	Topology	Batch	Units	Cirkit (ms)	Extended Einsum (ms)	Speedup	Memory red.
CP	Quad tree	256	64	4.5	3.3	$1.382\times$ [1.380, 1.383]	$1.013\times$
CP	Quad tree	256	128	10.0	6.8	$1.470\times$ [1.468, 1.472]	$1.012\times$
CP	Quad tree	256	256	23.4	15.7	$1.491\times$ [1.489, 1.494]	$1.010\times$
CP	Quad tree	256	512	62.7	43.6	$1.439\times$ [1.438, 1.441]	$0.999\times$
CP	Quad tree	256	1024	194.7	141.4	$1.377\times$ [1.376, 1.377]	$0.953\times$
CP	Quad tree	512	64	7.3	5.7	$1.296\times$ [1.286, 1.297]	$1.015\times$
CP	Quad tree	512	128	16.0	11.4	$1.411\times$ [1.408, 1.412]	$1.018\times$
CP	Quad tree	512	256	35.3	25.0	$1.412\times$ [1.410, 1.413]	$1.016\times$
CP	Quad tree	512	512	89.0	65.3	$1.362\times$ [1.361, 1.365]	$1.012\times$
CP	Quad tree	512	1024	272.9	207.1	$1.318\times$ [1.315, 1.319]	$1.000\times$
CP	Quad graph	256	64	8.9	7.1	$1.265\times$ [1.264, 1.268]	$1.051\times$
CP	Quad graph	256	128	19.9	14.9	$1.334\times$ [1.332, 1.335]	$1.053\times$
CP	Quad graph	256	256	47.9	34.5	$1.392\times$ [1.390, 1.397]	$1.054\times$
CP	Quad graph	256	512	132.3	90.9	$1.455\times$ [1.453, 1.456]	$1.000\times$
CP	Quad graph	512	64	15.3	12.9	$1.186\times$ [1.184, 1.187]	$1.062\times$
CP	Quad graph	512	128	33.2	26.5	$1.251\times$ [1.250, 1.268]	$1.067\times$
CP	Quad graph	512	256	74.7	57.0	$1.312\times$ [1.311, 1.313]	$1.065\times$
CP	Quad graph	512	512	193.4	140.9	$1.375\times$ [1.372, 1.376]	$1.059\times$

Table 2: Complete Tucker speedup results, using the same statistics as Table 1.

Layer	Topology	Batch	Units	Cirkit (ms)	Extended Einsum (ms)	Speedup	Memory red.
Tucker	Quad tree	256	32	11.4	10.0	$1.142\times$ [1.141, 1.149]	$1.028\times$
Tucker	Quad tree	256	64	49.0	43.6	$1.125\times$ [1.124, 1.126]	$1.014\times$
Tucker	Quad tree	512	32	19.8	17.9	$1.104\times$ [1.103, 1.105]	$1.037\times$
Tucker	Quad tree	512	64	80.3	74.3	$1.081\times$ [1.080, 1.083]	$1.018\times$
Tucker	Quad graph	256	32	21.9	18.0	$1.215\times$ [1.214, 1.215]	$1.040\times$
Tucker	Quad graph	256	64	96.4	80.2	$1.202\times$ [1.200, 1.203]	$1.018\times$
Tucker	Quad graph	512	32	38.8	32.5	$1.194\times$ [1.193, 1.194]	$1.050\times$
Tucker	Quad graph	512	64	159.4	135.7	$1.174\times$ [1.174, 1.176]	$1.023\times$

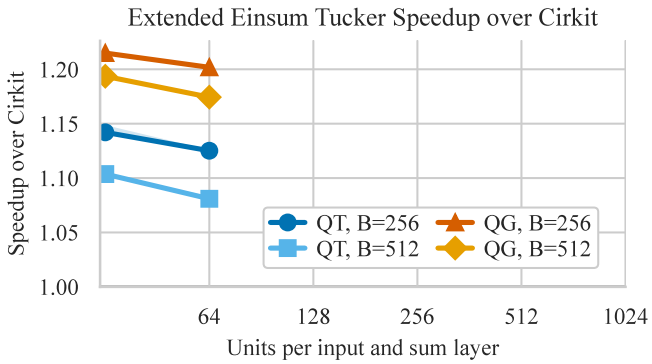

Figure 6: Extended Einsum speedup over unmodified cirkit. Lines distinguish region graph and batch size. Shaded regions are 95% bootstrap intervals over five independent paired runs.

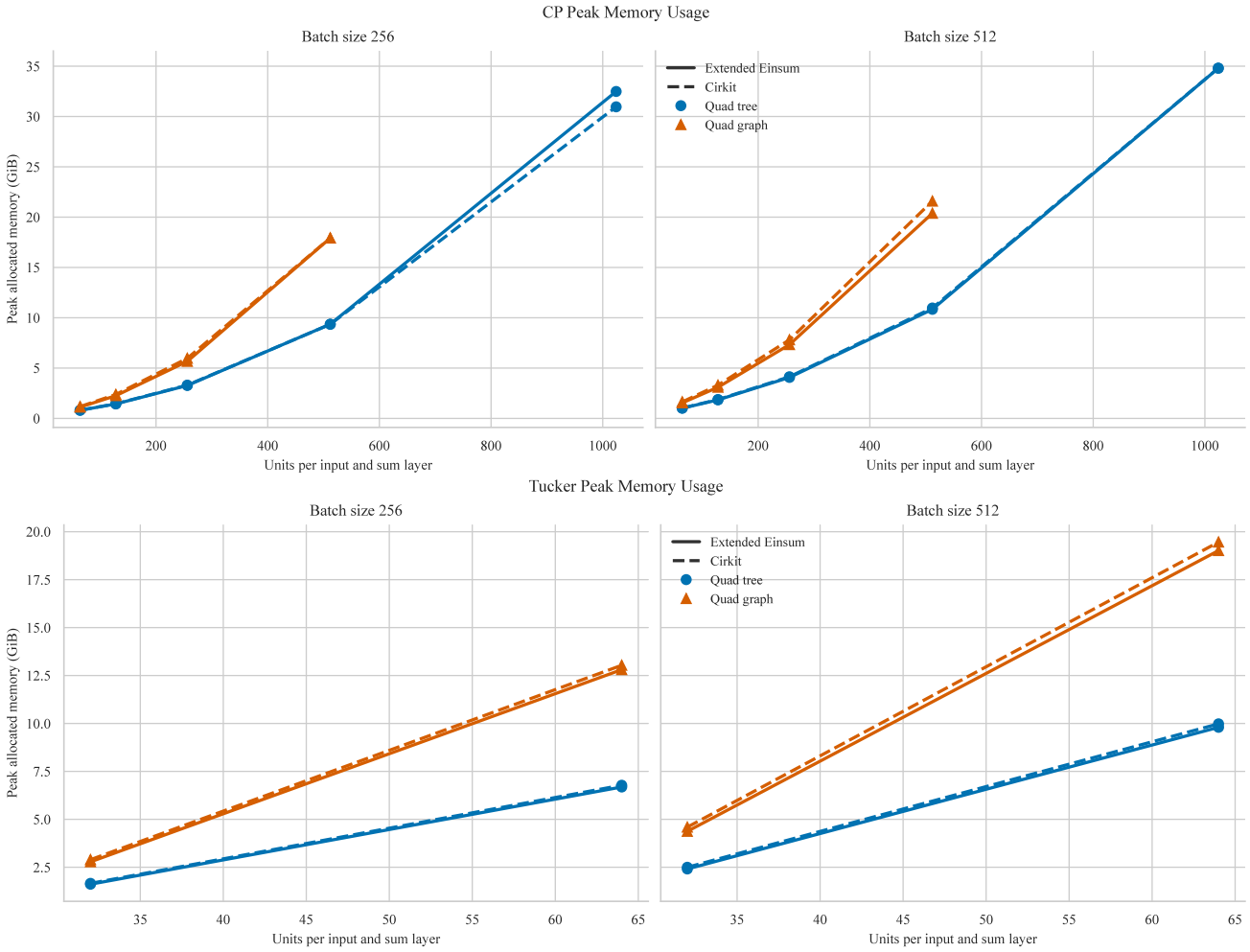

Figure 7: Absolute peak allocated GPU memory for CP (top) and Tucker (bottom). Panels within each plot separate batch sizes 256 and 512. Solid lines denote Extended Einsum and dashed lines denote cirkit. Shaded regions are 95% bootstrap intervals over five independent runs.

E COMPILATION TIME

We measure compilation separately from execution using the same publication configurations. Each measurement runs in a fresh process and is repeated five times. Model construction, data preparation, and device setup are excluded. For Extended Einsum, frontend time is the sum of circuit extraction, folding, contraction-path optimization, and stability-aware lowering to the PyTorch backend. This last stage performs the automatic numerical-stability translation and binds the remaining operations to their PyTorch implementations. For cirkit, frontend time is its native lowering step. In both cases, `torch.compile` is lazy, so its time includes the first complete forward and backward evaluation and a device synchronization. Total compile time is frontend time plus this `torch.compile` time.

Table 3 compares the complete pipelines on the eight MNIST ablation workloads and the four ImageNet64 dense/Monarch workloads. Extended Einsum compiles faster in all 12 matched configurations. The median cirkit/Extended Einsum ratio is $1.216\times$, with individual ratios from $1.004\times$ to $1.439\times$. The frontend alone is not uniformly faster: Extended Einsum performs additional optimization, but the program it emits generally takes less time in `torch.compile`.

Table 3: Matched end-to-end compilation time in seconds. Entries are medians over five process-isolated runs. "Frontend" denotes the native lowering pipeline of each system; total is frontend plus `torch.compile`. Dense and Monarch rows use 64×64 ImageNet64 inputs; CP and Tucker rows use 28×28 MNIST inputs.

				Extended Einsum			Cirkit		
Workload	Topology	B	H	Frontend	`torch.compile`	Total	Frontend	`torch.compile`	Total
CP	Quad graph	256	128	3.06	24.11	27.16	2.13	25.16	27.28
CP	Quad graph	512	512	3.05	27.73	30.77	6.99	27.14	34.14
CP	Quad tree	256	128	1.81	12.64	14.46	1.27	17.75	19.01
CP	Quad tree	512	512	1.80	14.72	16.51	3.91	19.82	23.75
Tucker	Quad graph	256	32	2.22	23.33	25.53	1.40	28.65	30.06
Tucker	Quad graph	512	64	2.22	27.30	29.51	3.51	33.55	37.03
Tucker	Quad tree	256	32	1.50	16.32	17.82	0.72	22.13	22.85
Tucker	Quad tree	512	64	1.50	18.35	19.80	1.81	24.85	26.70
Dense	Quad graph	256	64	12.69	20.31	33.02	10.05	27.72	37.51
Monarch	Quad graph	256	256	22.14	36.61	58.75	12.18	47.51	59.64
Dense	Quad tree	256	128	6.12	16.87	23.01	5.72	23.52	29.30
Monarch	Quad tree	256	512	11.11	33.83	44.94	6.95	43.46	50.40

Table 4 resolves the Extended Einsum frontend into its constituent stages. Across the 12 configurations, `torch.compile` accounts for 61.5%–92.7% of total compilation time (median 89.0%), so downstream PyTorch compilation is usually the dominant cost. Folding is the largest Extended Einsum frontend stage for the larger ImageNet64 programs: it takes 17.55 seconds for the quad-graph Monarch configuration and accounts for 79.3% of that frontend. Stability-aware lowering is comparatively stable at 1.13–1.60 seconds. Contraction-path optimization is below 0.01 seconds for the tree and Tucker configurations and reaches only 0.15 seconds in the measured quad-graph programs.

Table 4: Breakdown of Extended Einsum compilation time in seconds, reported as medians over five process-isolated runs. "Extract" translates the source circuit, "Fold" combines same-shaped operations, and "Path" selects contraction paths. "Stable lower" performs the automatic numerical-stability translation and emits the corresponding PyTorch backend program. Total is the sum of these four frontend stages and `torch.compile`.

Workload	Topology	B	H	Extract	Fold	Path	Stable	`torch.compile`	Total
CP	Quad graph	256	128	0.38	1.11	0.14	1.42	24.11	27.16
CP	Quad graph	512	512	0.40	1.10	0.14	1.41	27.73	30.77
CP	Quad tree	256	128	0.09	0.57	0.00	1.14	12.64	14.46
CP	Quad tree	512	512	0.10	0.56	0.00	1.13	14.72	16.51
Tucker	Quad graph	256	32	0.13	0.75	0.00	1.34	23.33	25.53
Tucker	Quad graph	512	64	0.13	0.75	0.00	1.33	27.30	29.51
Tucker	Quad tree	256	32	0.06	0.25	0.00	1.19	16.32	17.82
Tucker	Quad tree	512	64	0.06	0.24	0.00	1.19	18.35	19.80
Dense	Quad graph	256	64	2.01	9.29	0.00	1.39	20.31	33.02
Monarch	Quad graph	256	256	3.09	17.55	0.15	1.33	36.61	58.75
Dense	Quad tree	256	128	1.16	3.82	0.00	1.14	16.87	23.01
Monarch	Quad tree	256	512	1.60	7.75	0.15	1.60	33.83	44.94

F COMPLETE ABLATION

This section reports the complete ablation of our optimizations: all measured variants on all eight workloads, that is, CP and Tucker layers on both region graphs at two batch/width settings each. Besides the memory layout optimization, the ablation also varies how XE stabilizes the computation numerically. The optimized pipeline evaluates the circuit in a scaled representation, which keeps a positive tensor together with a broadcastable log scale, and detaches the reference shifts of that representation from the gradient computation. Detaching does not change the forward values; it only prevents automatic differentiation from propagating through the maximum reductions and the associated normalizations. The six variants are therefore:

- *XE optimized*: the full pipeline with contraction-path replanning, folding, memory layout optimization, scaled evaluation, and detached shifts. It is the reference for all reported changes.
- *Log-space*: conventional log-space arithmetic instead of scaled evaluation, with detached shifts retained.
- *Shift gradients*: scaled evaluation with differentiable shifts.
- *Log-space + shift gradients*: conventional log-space arithmetic with differentiable shifts, removing both numerical-stability optimizations at once.
- *No layout optimization*: only the layout pass is disabled; folding, input preordering, contraction paths, and stabilization are unchanged.
- *Cirkit*: the unmodified baseline with its native log-space evaluation.

Figure 8 and Table 5 give the CP results. The layout optimization is the largest isolated contribution across all four CP workloads: disabling it costs 18.5% to 23.9%, whereas replacing scaled evaluation by log-space arithmetic changes runtime by only -0.4% to $+2.8\%$ as long as the shifts stay detached. Its cost falls almost entirely on the backward pass, since the irregular gathers that the pass removes materialize additional copies whose gradients are expensive. Removing both stability optimizations, the conventional log-space combination with differentiable shifts, costs 10.1% to 15.9%. The scaled variant with differentiable shifts must be read as an interaction rather than as an independent effect: it retains the scaled representation but differentiates through the repeated maxima and normalizations that this representation introduces, which produces an especially expensive backward graph and a slowdown of 7.2% to 25.9%, the largest value on the wide quad graph. Unmodified cirkit is 33.6% to 47.0% slower than the optimized pipeline. Peak allocation follows the same pattern much more weakly, with at most 6.7% additional memory for any variant.

Figure 9 and Table 6 show that Tucker layers behave differently. Here disabling the layout optimization *improves* runtime by 1.6% to 4.6%. The pass does not change the Tucker einsum equations, operand shapes, or contraction paths; it changes how folded intermediates reach their consumers, replacing indexed gathers with contiguous slices. Slices are cheap in the forward pass, but their gradients introduce slice-backward materializations and separate reduction kernels, and in these arithmetic-intensive workloads that cost outweighs the routing benefit. The stability variants cost at most 8.8%, and XE still runs 8.1% to 21.5% faster than cirkit on every Tucker workload. Since the layout rewrite changes routing without changing the contractions themselves, this reversal indicates that the profitability of an IR-level layout depends on the downstream backend rather than on the IR alone. Appendix G confirms this by repeating the ablation on JAX/XLA, where the same ordering is favorable for Tucker.

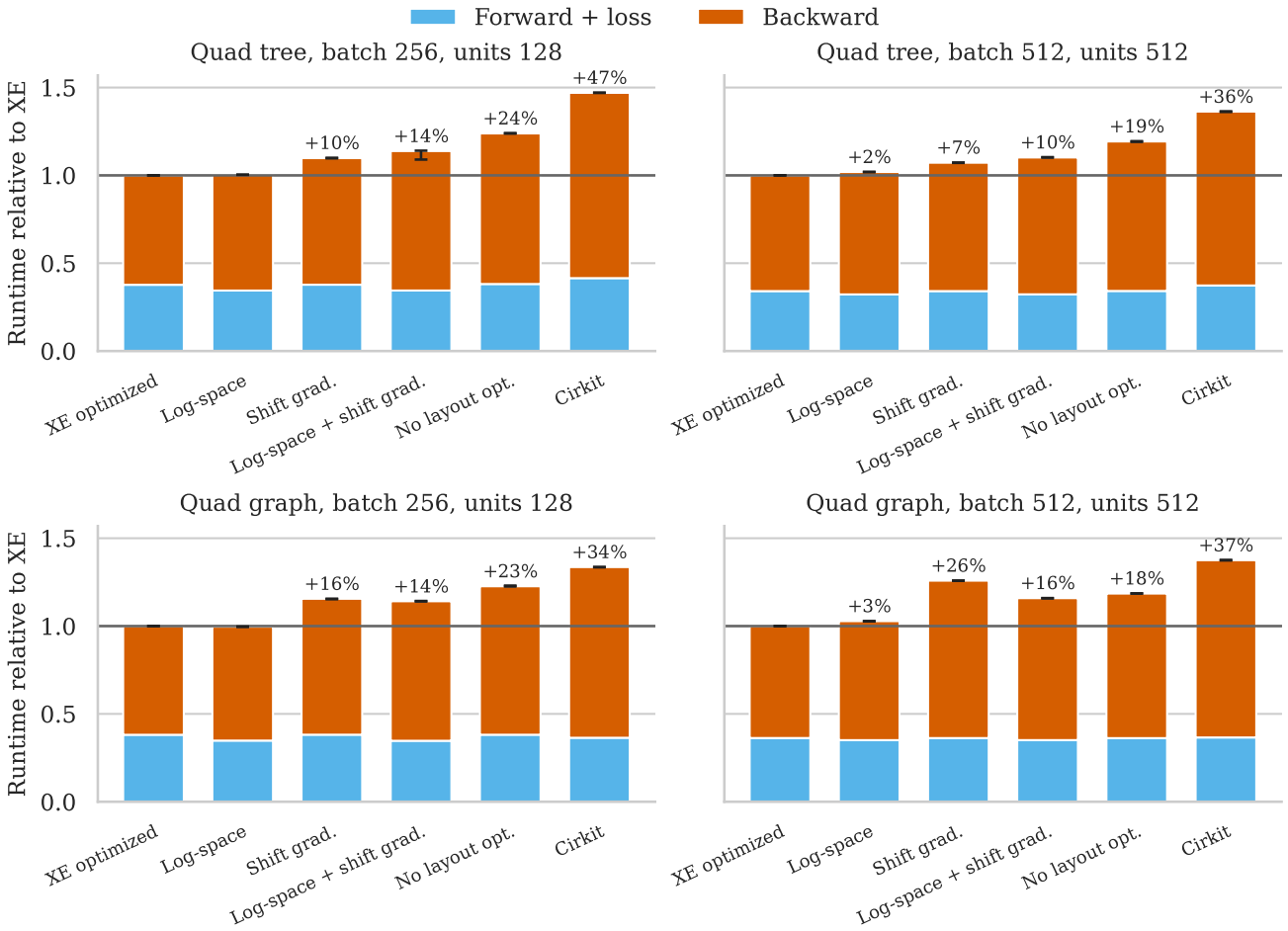

Figure 8: Complete CP ablation on both region graphs. Bars stack forward-plus-loss and backward time, normalized within each paired run to the optimized XE pipeline, and annotations report the median paired change in total time. Error bars are 95% bootstrap intervals for the total over five paired runs.

Table 5: Complete CP ablation. Runtime is the median forward-plus-loss and backward time per batch over five runs, and memory is the median CUDA peak allocation. Both changes are computed within each paired run relative to the optimized XE pipeline and then reported as the median.

Topology	Batch	Units	Variant	Runtime (ms)	Change	Memory (MiB)	Change
Quad tree	256	128	XE optimized	6.8	+0.0%	1473	+0.0%
Quad tree	256	128	Log-space	6.8	+0.4%	1465	-0.6%
Quad tree	256	128	Shift gradients	7.4	+9.9%	1480	+0.5%
Quad tree	256	128	Log-space + shift gradients	7.7	+13.8%	1466	-0.4%
Quad tree	256	128	No layout optimization	8.4	+23.9%	1503	+2.1%
Quad tree	256	128	Cirkit	10.0	+47.0%	1490	+1.2%
Quad tree	512	512	XE optimized	65.2	+0.0%	11101	+0.0%
Quad tree	512	512	Log-space	66.5	+1.9%	11745	+5.8%
Quad tree	512	512	Shift gradients	69.9	+7.2%	11751	+5.9%
Quad tree	512	512	Log-space + shift gradients	71.8	+10.1%	11356	+2.3%
Quad tree	512	512	No layout optimization	77.8	+19.3%	11749	+5.8%
Quad tree	512	512	Cirkit	88.9	+36.3%	11234	+1.2%
Quad graph	256	128	XE optimized	14.9	+0.0%	2313	+0.0%
Quad graph	256	128	Log-space	14.9	-0.4%	2402	+3.8%
Quad graph	256	128	Shift gradients	17.2	+15.5%	2348	+1.5%
Quad graph	256	128	Log-space + shift gradients	17.0	+14.1%	2405	+4.0%
Quad graph	256	128	No layout optimization	18.3	+22.7%	2439	+5.4%
Quad graph	256	128	Cirkit	19.9	+33.6%	2435	+5.3%
Quad graph	512	512	XE optimized	140.8	+0.0%	20870	+0.0%
Quad graph	512	512	Log-space	144.7	+2.8%	21614	+3.6%
Quad graph	512	512	Shift gradients	177.2	+25.9%	21120	+1.2%
Quad graph	512	512	Log-space + shift gradients	163.1	+15.9%	21869	+4.8%
Quad graph	512	512	No layout optimization	167.0	+18.5%	22275	+6.7%
Quad graph	512	512	Cirkit	193.6	+37.5%	22108	+5.9%

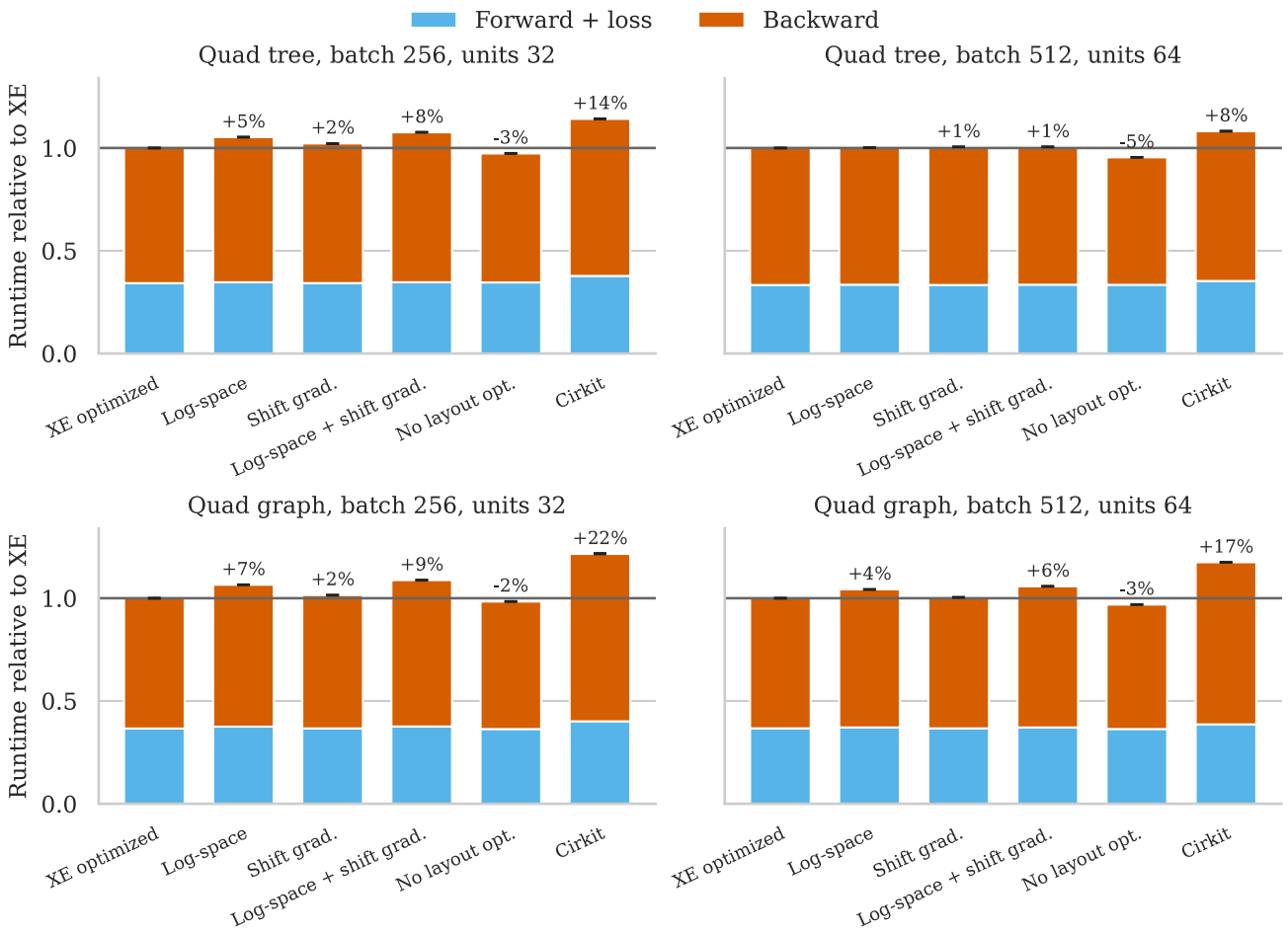

Figure 9: Complete Tucker ablation, using the same variants and normalization as Figure 8.

Table 6: Complete Tucker ablation, using the same statistics as Table 5.

Topology	Batch	Units	Variant	Runtime (ms)	Change	Memory (MiB)	Change
Quad tree	256	32	XE optimized	10.0	+0.0%	1653	+0.0%
Quad tree	256	32	Log-space	10.5	+5.3%	1651	-0.2%
Quad tree	256	32	Shift gradients	10.2	+2.2%	1651	-0.2%
Quad tree	256	32	Log-space + shift gradients	10.7	+7.6%	1651	-0.2%
Quad tree	256	32	No layout optimization	9.7	-2.7%	1650	-0.2%
Quad tree	256	32	Cirkit	11.4	+14.1%	1699	+2.8%
Quad tree	512	64	XE optimized	74.3	+0.0%	10034	+0.0%
Quad tree	512	64	Log-space	74.4	+0.1%	10022	-0.1%
Quad tree	512	64	Shift gradients	74.7	+0.5%	10022	-0.1%
Quad tree	512	64	Log-space + shift gradients	74.7	+0.5%	10022	-0.1%
Quad tree	512	64	No layout optimization	70.9	-4.6%	10019	-0.2%
Quad tree	512	64	Cirkit	80.3	+8.1%	10219	+1.8%
Quad graph	256	32	XE optimized	18.0	+0.0%	2850	+0.0%
Quad graph	256	32	Log-space	19.2	+6.6%	2881	+1.1%
Quad graph	256	32	Shift gradients	18.3	+1.5%	2864	+0.5%
Quad graph	256	32	Log-space + shift gradients	19.6	+8.8%	2881	+1.1%
Quad graph	256	32	No layout optimization	17.7	-1.6%	2846	-0.1%
Quad graph	256	32	Cirkit	21.9	+21.5%	2964	+4.0%
Quad graph	512	64	XE optimized	135.7	+0.0%	19472	+0.0%
Quad graph	512	64	Log-space	141.6	+4.3%	19596	+0.6%
Quad graph	512	64	Shift gradients	136.3	+0.4%	19529	+0.3%
Quad graph	512	64	Log-space + shift gradients	143.6	+5.8%	19597	+0.6%
Quad graph	512	64	No layout optimization	131.5	-3.1%	19459	-0.1%
Quad graph	512	64	Cirkit	159.5	+17.5%	19927	+2.3%

G JAX BACKEND ABLATIONS

To test whether the IR-level conclusions transfer across compiler stacks, this section repeats the ablations with JAX/XLA. The circuit and folded tensor programs, contraction paths, shapes, and workload grid are identical. Each JAX configuration is compiled before 30 warmup and 90 measured batches in a fresh process, and we report the directly measured combined forward-and-backward time over five runs. Compilation and optimizer time are excluded.

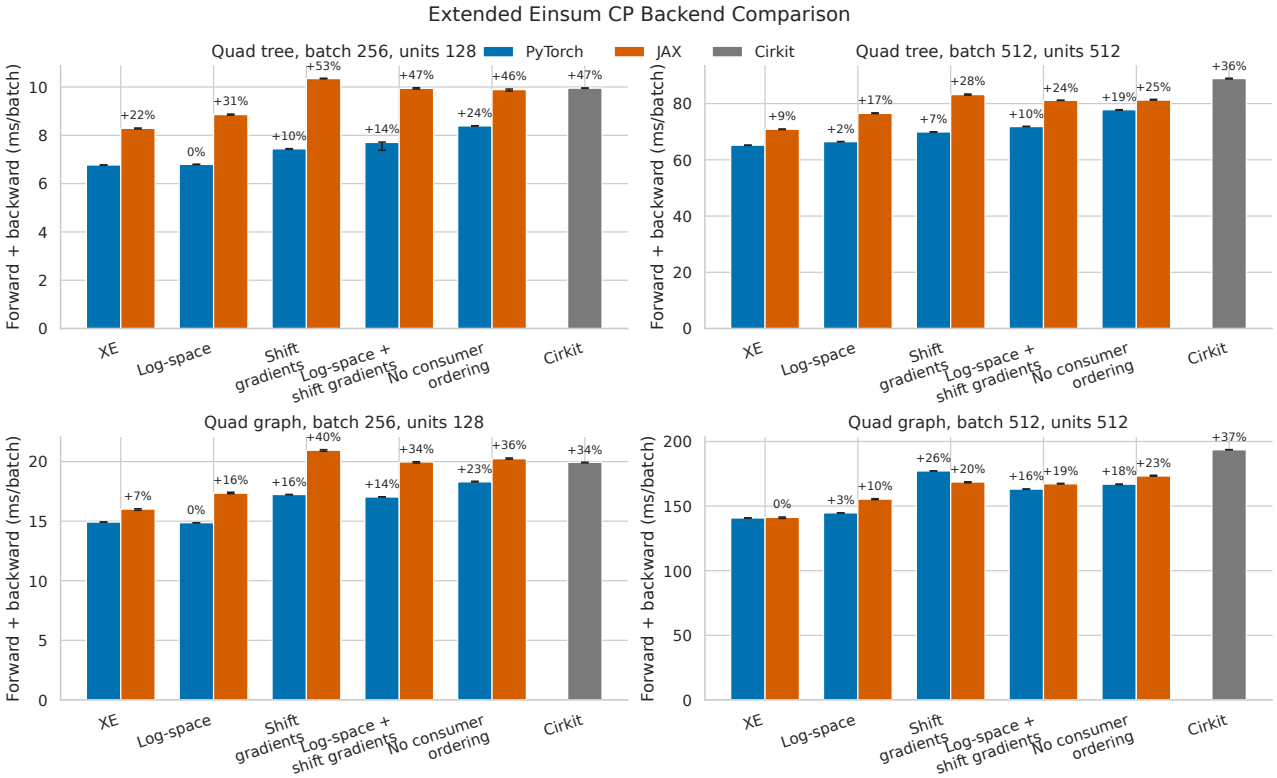

Figure 10: CP ablations under PyTorch and JAX. Bars report absolute forward-and-backward time; the cirkit bar is an external PyTorch reference. Percentages are relative to the full PyTorch Extended Einsum pipeline in each panel. Error bars are 95% bootstrap intervals over five runs.

For CP, the optimized JAX program is 1.20–1.37× faster than cirkit, although it is 0.3%–22% slower than the PyTorch program. The relative ablations agree across compilers: disabling ordering increases JAX runtime by 15%–26%, and differentiating through the reference shifts increases it by 17%–31%. The CP conclusions therefore transfer to XLA.

Tucker exposes stronger backend dependence. The optimized JAX program takes 1.39–1.87× the PyTorch time and is slower than cirkit in all four configurations. More importantly, disabling consumer ordering slows JAX by 6%–30%, whereas it makes PyTorch 2%–5% faster. Thus, the same IR ordering is favorable to XLA but unfavorable to PyTorch / Inductor for Tucker. The log-space variants show further reversals under XLA: the variant with log-space evaluation and shift gradients is actually faster than log-space without gradients. This again most likely reflects downstream fusion and contraction lowering rather than an algorithmic advantage of differentiable shifts.

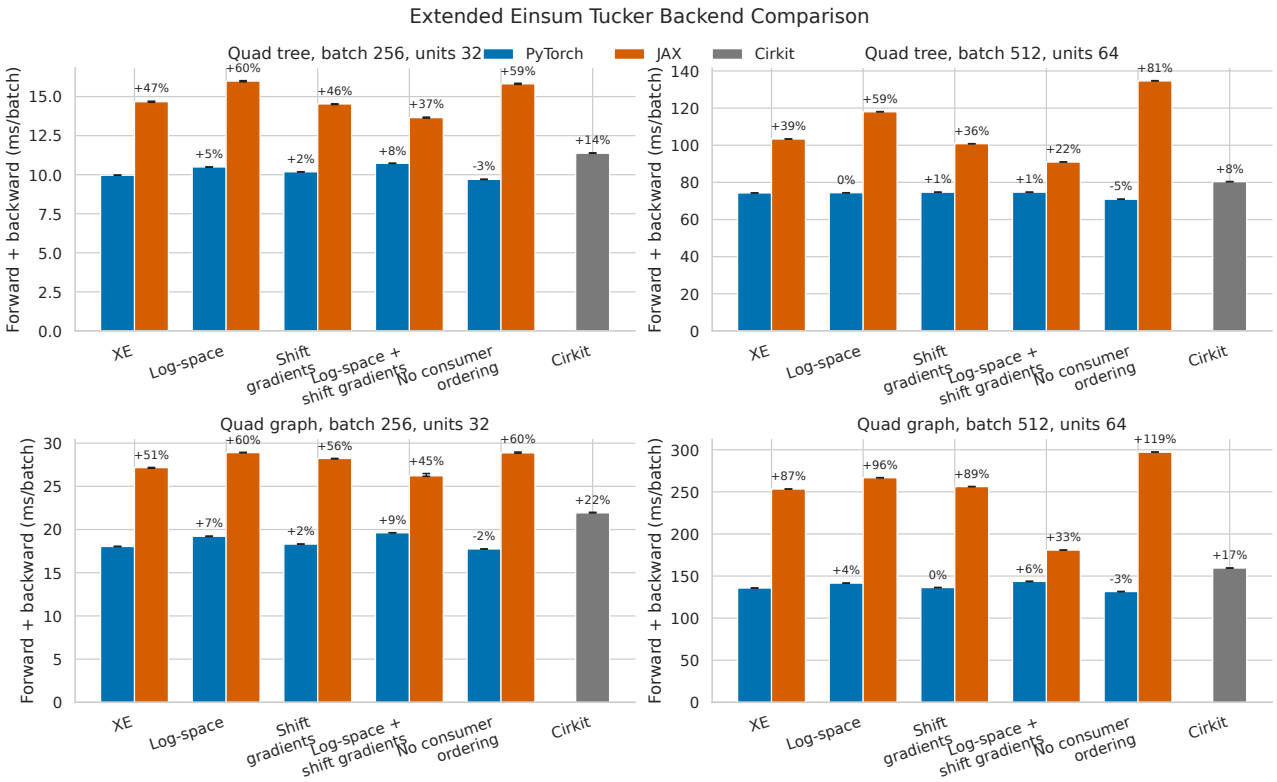

Figure 11: Tucker ablations under PyTorch and JAX, with the same timing and normalization as Figure 10.

H NUMERICAL ACCURACY AND UNDERFLOW

Because faster execution is useful only when circuit values and gradients remain reliable, this section details the stability translation and evaluates underflow, numerical agreement, and training convergence.

H.1 ALGEBRAIC BASIS OF THE STABILITY TRANSLATIONS

Probabilistic circuits repeatedly multiply and sum positive values, so even moderately small probabilities underflow after a few product layers. Because most deep learning models do not share this property, general-purpose deep learning compilers do not insert the log-space evaluations that PCs require, and users must implement them manually. Our compiler instead inserts these transformations automatically. We first reproduce the conventional log-space evaluation used by cirkit and introduced by Peharz et al. [2020], then detach its reference shifts from the gradient, and finally use a scaled representation that avoids converting complete result tensors between logarithmic and linear space. Because the contraction-path pass lowers every expression into a schedule of binary contractions, it is enough to state each transformation for a single binary einsum. Throughout, we write $\max(ab \to a;\ X)$ for the reduction that follows the format string $ab \to a$ but replaces the summation of an einsum by a maximum.

Log-space einsum. Consider the CP sum layer

$$Y = \text{einsum}(ab, bc \to ac;\ X, W),$$

where a is the batch index, b is contracted, and the softmax-normalized weights W sum to one over b. A direct log-sum-exp evaluation materializes a tensor with indices abc and cannot use a dense contraction kernel. Instead, for $\ell = \log X$, we shift each batch fiber before returning to linear space:

$$\begin{aligned} m &= \max(ab \to a;\ \ell), \\ \log Y &= \log \text{einsum}(ab, bc \to ac;\ \exp(\ell - m), W) + m. \end{aligned}$$

Because the same shift m_a is applied to every term summed over b, its exponential factors out of the contraction and can be restored exactly afterward. The shifted values are at most one, so the exponential cannot overflow. Keeping W in linear space makes the central operation an ordinary matrix multiplication that can use tensor-core kernels. In general, each log-space operand is shifted over its contracted indices, and the retained indices determine how the shift broadcasts over the output.

Einsums that contract no index need no shift at all. The Hadamard product "$ab, ab \to ab$" lowers to the sum $\ell^{(1)} + \ell^{(2)}$, and the Kronecker product "$ab, ac \to abc$" does the same up to broadcasting. The compiler recognizes such cases from the format string (every operand index also occurs in the output) and replaces the entire $\exp$–einsum–$\log$ sequence by broadcasted additions.

Detached reference shifts. Conventionally, the shift m remains part of the automatic-differentiation graph, although it only chooses an equivalent numerical representation. For a single contracted index,

$$\begin{aligned} f(\boldsymbol{\ell}, m) &= m + \log \text{einsum}(b \to;\ \exp(\boldsymbol{\ell} - m)), \\ \frac{\partial f}{\partial m} &= 0. \end{aligned}$$

The shift cancels algebraically, so its chain-rule contribution is zero even though m depends on $\boldsymbol{\ell}$. The same cancellation applies to every operand shift, since each is subtracted before the contraction and restored afterward. Extended Einsum therefore evaluates $m = \text{sg}(\max(ab \to a;\ \ell))$, where sg denotes stop-gradient. This removes the backward pass through the maximum reduction.

Scaled evaluation. The log-space translation still applies a full-tensor exponential before every contracting einsum and a full-tensor logarithm afterward. Scaled evaluation leaves the contraction and its linear-space parameters unchanged, but carries its normalized result to the next layer in linear space. It represents a positive tensor as $X = \widehat{X} \exp(s)$, where s is a broadcastable log scale. We use one scale per last-axis fiber and normalize that fiber by its maximum:

$$n = \text{sg}\left(\max(ab \to a;\ X)\right), \quad \widehat{X} = \frac{X}{n}, \quad s = \log n.$$

The stop-gradient in n is exact because the normalizer changes only the representation: $(X/n)\exp(\log n) = X$ for any positive n.

In the sum layer above, s is indexed by a alone, so contracting b leaves it untouched. The backend contracts $\widehat{X}$ with the same linear weights W, normalizes the output, and adds s to its new log scale. The next layer consumes the normalized result directly, while the root forms its log output as $\log \widehat{X} + s$. This removes one full-tensor exponential at every sum layer and applies the logarithm only to the small normalization tensor.

Scales that do vary along a contracted index have to be aligned first. In the Tucker layer $\text{einsum}(abc, dbc \to ad;\ T, W)$, for instance, the outer product T carries a scale indexed by ab while b is contracted. We then absorb only the deviation $s - \bar{s}$ from the reduced scale $\bar{s} = \text{sg}(\max(ab \to a;\ s))$ into $\widehat{T}$ and add $\bar{s}$ to the output scale. Exponentials are still needed when differently scaled fibers have to be aligned, but they then act on broadcastable scale tensors rather than on full intermediate tensors.

H.2 COMPILER-LEVEL STABILITY TRANSLATION

Appendix H.1 describes the algebraic basis of our log-space and scaled translations. We give the corresponding compiler procedure here. Numerical stabilization is applied during the stability-aware backend-lowering stage timed in Table 4, after folding and contraction-path selection. The translation operates on the program in static single-assignment (SSA) form and emits another straight-line program composed only of backend primitives. Consequently, PyTorch and JAX execute the same transformed tensor program. The backend-specific code is responsible only for implementing primitives such as `einsum`, reduction, exponentiation, and stop-gradient.

Lazy representations. For each SSA value, the translator records which numerical representations are already available and the backend-program position at which each was computed. The log-space translation stores either a raw value X, its logarithm $\ell = \log X$, or both. The scaled translation analogously stores a raw value or a pair

$$X = \widehat{X}\exp(s),$$

where s is broadcastable to $\widehat{X}$. A conversion is inserted only when demanded by a consumer and is memoized, so a value used by several instructions is converted at most once. The final SSA value is converted back to the raw representation because the backend runtime returns an ordinary tensor. This lazy construction is important: for example, $\log(\text{einsum}(\exp A, \exp B))$ can remain in the stable representation throughout and does not need to materialize the two unshifted exponentials.

The scaled pass uses one scale for each last-axis fiber. Thus, for a value of shape $(d_0, \ldots, d_{r-1})$, the initial scale has shape $(d_0, \ldots, d_{r-2}, 1)$; a scalar remains scalar. Given a positive fiber $X_{\boldsymbol{i},:}$, the compiler chooses a positive normalizer $n_{\boldsymbol{i}}$, detaches it from automatic differentiation, and emits

$$\widehat{X}_{\boldsymbol{i},:} = X_{\boldsymbol{i},:}/n_{\boldsymbol{i}}, \qquad s_{\boldsymbol{i}} = \log n_{\boldsymbol{i}}.$$

The publication configuration uses the maximum-based normalization of Appendix H.1. Detachment is exact because $(X/n)\exp(\log n) = X$ for every positive n.

Parameter and data operands. Before translating instructions, the compiler marks every SSA value whose entire dependency chain consists of parameters. Such values include softmax-normalized sum weights. They remain in ordinary linear space when used by an einsum, preserving dense matrix-multiplication lowering and avoiding unnecessary logarithms of model parameters. Non-parameter operands are translated to log space or to scaled pairs as required. This distinction is computed from the SSA dependency graph rather than from operator names, and therefore remains valid after folding and reassociation.

Contracting einsums. Let an operand have index string u, and let R be the axes whose labels do not occur in the output string. In the log-space translation, a non-parameter operand ℓ is reduced over exactly R:

$$m = \text{sg}\Big(\max_R \ell\Big), \qquad \widetilde{X} = \exp(\ell - m).$$

Axes retained by the output are retained in m, so the shift is as fine grained as the contraction permits. The backend evaluates the original einsum on the shifted data operands and the linear parameter operands, takes the logarithm of its result, reshapes

each retained shift to the output layout, and adds the shifts back. A scalar shift is used only when an operand has no retained label.

Scaled einsums apply the same principle to log scales. If an operand scale varies along a contracted axis, the compiler computes the detached reference

$$\bar{s} = \mathrm{sg}\Big(\max_{R} s\Big)$$

and replaces its normalized operand by $\widehat{X}\exp(s - \bar{s})$. The original einsum is then evaluated in linear space. Its output is normalized fiberwise, and the logarithm of the new normalizer and all retained operand scales are accumulated into the output scale. If a scale is already constant along R, this alignment is omitted. The compiler tracks scale shapes explicitly so that reductions, reshapes, and broadcasts are inserted only when required by the einsum index strings.

An einsum with no contracted label, such as a Hadamard or Kronecker product, is handled specially. Such an operation is only multiplication and broadcasting. The log-space pass replaces it by broadcasted addition of logarithms. The scaled pass applies the einsum only to the normalized tensors and adds their broadcasted log scales. Neither translation performs an exponential–einsum–log round trip or renormalizes a potentially large outer product.

Remaining operators and domain. Multiplication and division of scaled values multiply or divide normalized parts and add or subtract log scales. Addition first aligns both operands to their elementwise maximum log scale m, computes

$$\widehat{X}\exp(s_X - m) + \widehat{Y}\exp(s_Y - m),$$

and carries m as the result scale. Log-space addition uses the analogous shifted log-sum construction. Stacking likewise aligns its operands before stacking. Concatenation along the fiber axis requires a common scale, whereas concatenation across fibers preserves the independent scales. Slicing, selection, and gathering transform the normalized tensor and its scale together, but leave a scale untouched when it is constant along the indexed axis. Integer gather indices always remain raw. Explicit $\exp$ and $\log$ cross between representations, while softmax is evaluated raw because backend implementations already apply a stable maximum shift.

All maxima used only as reference shifts, as well as the normalizers that define scaled pairs, are wrapped in stop-gradient. Their cancellation is algebraic, so this changes neither the forward function nor its mathematical derivative, but removes reductions and normalization operations from the backward graph. Both stable translations assume that represented tensor values are strictly positive. Subtraction is therefore supported only when its result remains positive as arbitrary signed intermediates cannot be represented by a single logarithm or by a positive normalized tensor and log scale.

H.3 DEPTH UNDERFLOW STRESS TEST

To isolate underflow, we construct alternating product and sum circuits with factors scaled by 0.01 and increase their depth from 4 to 1024. The baseline at every depth is an unoptimized log-space FP64 evaluation of the same program with the same inputs and parameters. Every plotted forward error is computed against this baseline. Raw FP32 first produces non-finite outputs and gradients at depth 32, while raw FP64 fails at depth 256. The automatically generated scaled and log-space FP32 programs keep every output and gradient finite through depth 1024, and their median forward errors remain below 10^{-6} at every depth. The dotted line is the plotting floor at FP64 machine epsilon, $\epsilon_{\mathrm{FP64}} = 2^{-52} \approx 2.22 \times 10^{-16}$; values at the line may be smaller but are clipped to keep the logarithmic axis finite.

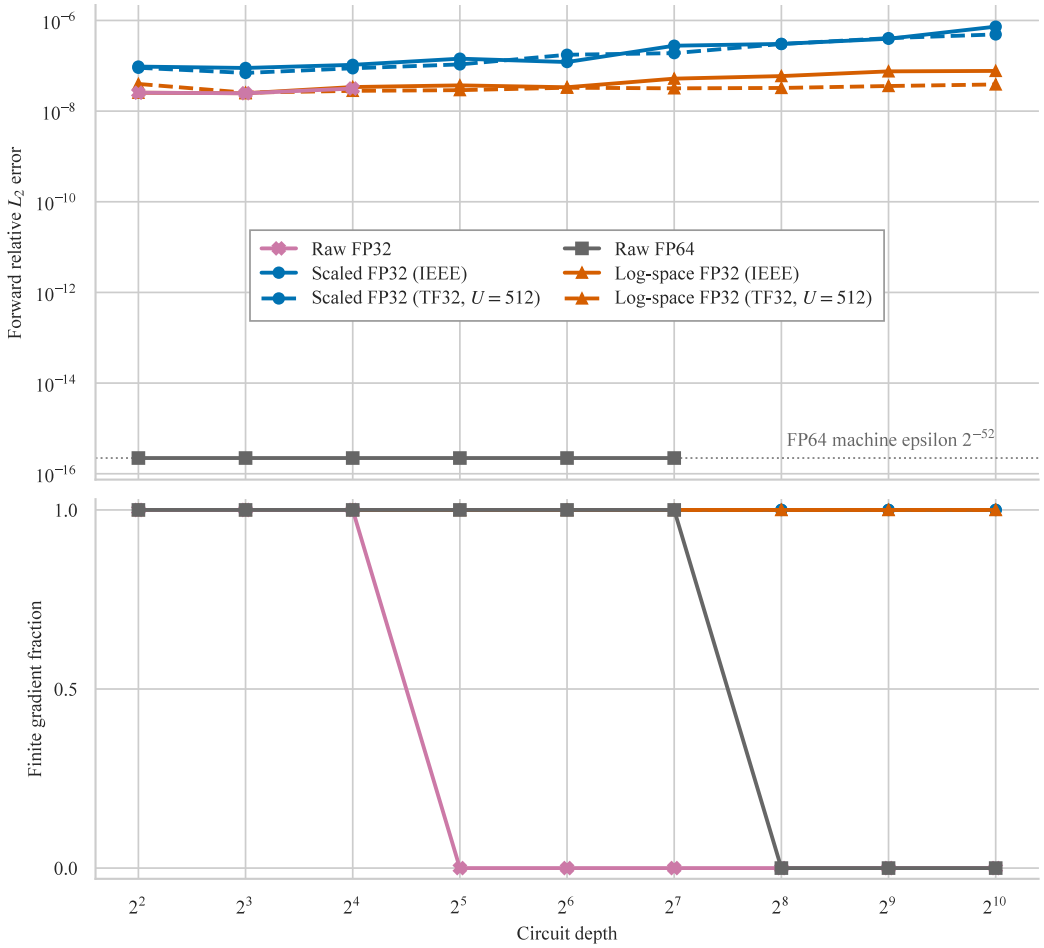

Figure 12: Numerical error relative to the unoptimized log-space FP64 baseline and gradient finiteness as circuit depth increases. Raw FP32 and FP64 fail at depths 32 and 256, respectively; both automatically stabilized FP32 translations remain finite through depth 1024. The dotted line marks the visualization floor at FP64 machine epsilon, 2^{-52}.

H.4 AGREEMENT WITH THE FP64 REFERENCE

We evaluate numerical correctness separately from runtime. The agreement suite covers CP and Tucker layers on both region graphs. Its baseline is the unoptimized log-space FP64 evaluation of each program. Each table row compares an optimized scaled or log-space FP32 evaluation with that baseline, using exactly the same inputs and parameter values. For a candidate tensor x and its FP64 baseline x_{ref}, we report

$$E_{\mathrm{rel}} = \frac{\|x - x_{\mathrm{ref}}\|_2}{\max(\|x_{\mathrm{ref}}\|_2, \tau_{\mathrm{FP64}})}, \qquad E_{\mathrm{abs}} = \max_i |x_i - x_{\mathrm{ref},i}|.$$

Here τ_{FP64} is the smallest positive normal FP64 value and only guards division by zero. The relative columns apply this definition to the forward log likelihoods, input gradients, and the concatenation of all parameter gradients. Because relative gradient errors are unreliable when the corresponding FP64 gradients are close to zero, Table 7 also reports the maximum elementwise absolute forward error and the maximum elementwise absolute error over all input and parameter gradients. Under IEEE FP32 matrix multiplication, forward and input-gradient relative errors remain below 1.4×10^{-7}, and aggregate parameter-gradient relative errors below 3.8×10^{-6}; the maximum absolute forward and gradient errors are 8.3×10^{-6} and 3.6×10^{-7}, respectively. Permitting TF32 leaves forward and input-gradient errors at the same scale but increases aggregate parameter-gradient relative error to at most 8.9×10^{-4}, while the maximum absolute gradient error remains 2.2×10^{-5}.

Table 7: Maximum errors of optimized FP32 evaluation against the corresponding unoptimized log-space FP64 baseline. Relative errors are normwise L_2 errors; "Forward abs." is the maximum elementwise log-likelihood error, and "Grad. abs." is the maximum elementwise error over input and parameter gradients. "IEEE" uses full-mantissa FP32 matrix multiplication; "TF32-permitted" uses the performance setting from the timing experiments.

Graph	Layer	FP32 math	Evaluation	Forward rel.	Forward abs.	Data grad rel.	Param. grad rel.	Grad. abs.
Quad tree	CP	IEEE	Scaled	8.6×10^{-8}	8.3×10^{-6}	1.4×10^{-7}	8.7×10^{-7}	4.1×10^{-8}
Quad tree	CP	IEEE	Log-space	4.7×10^{-8}	3.6×10^{-6}	1.4×10^{-7}	2.0×10^{-6}	3.2×10^{-7}
Quad tree	CP	TF32-permitted	Scaled	8.6×10^{-8}	8.3×10^{-6}	1.4×10^{-7}	8.0×10^{-4}	2.2×10^{-5}
Quad tree	CP	TF32-permitted	Log-space	4.7×10^{-8}	3.6×10^{-6}	1.4×10^{-7}	7.5×10^{-4}	2.2×10^{-5}
Quad tree	Tucker	IEEE	Scaled	6.7×10^{-8}	5.9×10^{-6}	1.1×10^{-7}	6.3×10^{-7}	2.1×10^{-8}
Quad tree	Tucker	IEEE	Log-space	4.6×10^{-8}	3.5×10^{-6}	1.3×10^{-7}	1.0×10^{-6}	5.1×10^{-8}
Quad tree	Tucker	TF32-permitted	Scaled	6.7×10^{-8}	5.9×10^{-6}	1.1×10^{-7}	6.2×10^{-4}	9.4×10^{-6}
Quad tree	Tucker	TF32-permitted	Log-space	4.6×10^{-8}	3.5×10^{-6}	1.3×10^{-7}	6.2×10^{-4}	9.4×10^{-6}
Quad graph	CP	IEEE	Scaled	9.4×10^{-8}	8.2×10^{-6}	1.3×10^{-7}	1.3×10^{-6}	7.9×10^{-8}
Quad graph	CP	IEEE	Log-space	6.7×10^{-8}	5.1×10^{-6}	1.4×10^{-7}	2.4×10^{-6}	3.6×10^{-7}
Quad graph	CP	TF32-permitted	Scaled	9.4×10^{-8}	8.2×10^{-6}	1.3×10^{-7}	8.9×10^{-4}	1.4×10^{-5}
Quad graph	CP	TF32-permitted	Log-space	6.7×10^{-8}	5.1×10^{-6}	1.4×10^{-7}	8.6×10^{-4}	1.4×10^{-5}
Quad graph	Tucker	IEEE	Scaled	4.8×10^{-8}	4.3×10^{-6}	1.3×10^{-7}	3.8×10^{-6}	2.2×10^{-7}
Quad graph	Tucker	IEEE	Log-space	5.4×10^{-8}	4.4×10^{-6}	1.3×10^{-7}	3.4×10^{-6}	2.0×10^{-7}
Quad graph	Tucker	TF32-permitted	Scaled	4.8×10^{-8}	4.3×10^{-6}	1.3×10^{-7}	6.4×10^{-4}	9.3×10^{-6}
Quad graph	Tucker	TF32-permitted	Log-space	5.4×10^{-8}	4.4×10^{-6}	1.3×10^{-7}	6.4×10^{-4}	9.3×10^{-6}

H.5 MNIST AGREEMENT AND END-TO-END TRAINING CONVERGENCE

Finally, on the complete MNIST programs used in the timing study, the baseline is the optimized log-space FP32 evaluation. The reported errors compare the optimized scaled FP32 evaluation against that baseline using the same MNIST batch and exactly the same parameter tensors. Table 8 reports the largest relative and absolute errors across the evaluated batches and all gradients remain finite.

Table 8: Error of optimized scaled FP32 evaluation against the corresponding optimized log-space FP32 baseline on complete MNIST programs. Both evaluations use the same batch and parameter tensors. Relative and absolute columns have the same definitions as Table 7.

Graph	Layer	Units	Forward rel.	Forward abs.	Data grad rel.	Param. grad rel.	Grad. abs.	Finite gradients
Quad tree	CP	512	7.9×10^{-8}	4.9×10^{-4}	2.9×10^{-7}	2.0×10^{-4}	5.4×10^{-7}	100.0%
Quad tree	Tucker	64	7.9×10^{-8}	9.8×10^{-4}	2.5×10^{-7}	3.0×10^{-5}	2.4×10^{-6}	100.0%
Quad graph	CP	512	3.5×10^{-7}	2.0×10^{-3}	6.3×10^{-5}	5.2×10^{-4}	2.2×10^{-5}	100.0%
Quad graph	Tucker	64	7.9×10^{-8}	9.8×10^{-4}	3.0×10^{-7}	4.7×10^{-4}	2.8×10^{-5}	100.0%

To check end-to-end convergence, we train a 512-unit CP quad-tree circuit on the MNIST training split for 20 epochs with batch size 512, fused Adam, and learning rate 0.01. Each epoch contains all 117 complete fixed-shape batches, or 59,904 examples; the remaining 96 examples are dropped. We compare unmodified cirkit log-space execution with Extended Einsum log-space and scaled-max execution. All implementations receive the same seeded minibatch permutation in every epoch. The two Extended Einsum variants also share identical initial parameters, whereas cirkit uses its own seeded initialization. The comparison therefore tests whether the implementations reach the same convergence regime, rather than whether their epoch-wise trajectories match pointwise.

Figure 13 shows that all three programs follow nearly identical convergence trajectories and remain numerically stable through epoch 20. Their final average training negative log likelihoods differ by at most 0.5014, less than 0.08% of the cirkit value. Both Extended Einsum variants finish slightly below cirkit: log-space reaches 637.8724, scaled-max reaches 637.9005, and cirkit reaches 638.3738.

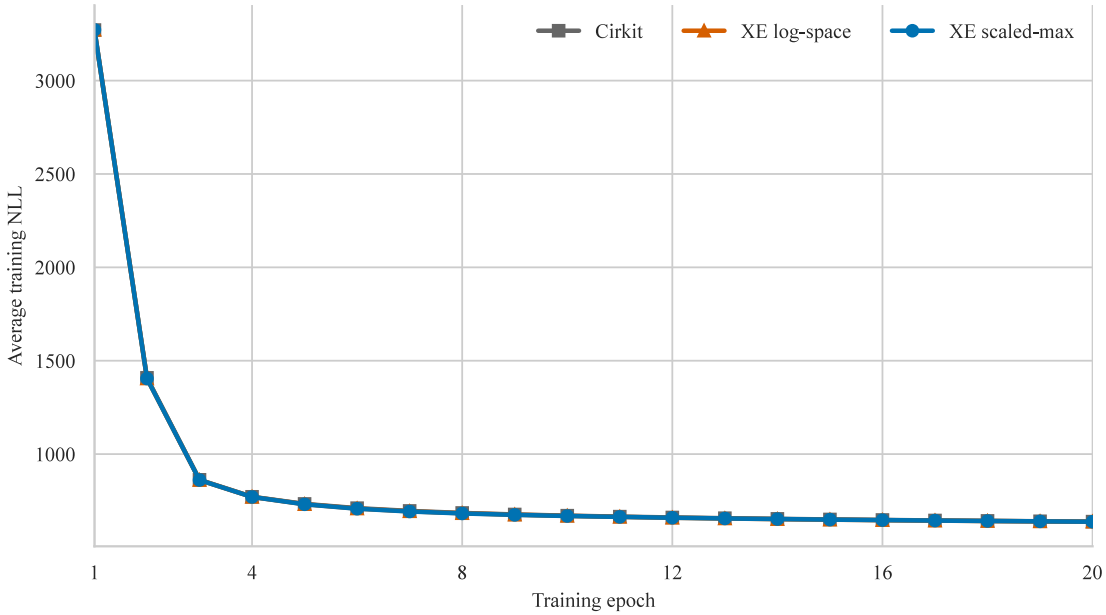

Figure 13: Average training negative log likelihood over 20 MNIST epochs for a 512-unit CP quad-tree circuit. The two Extended Einsum variants share an initialization; cirkit uses its own seeded initialization. The trajectories should therefore be compared as a convergence check rather than as pointwise matched optimization paths.

Table 9: MNIST training negative log likelihood. The final difference is the implementation's final NLL minus the final cirkit NLL, so negative values favor the listed implementation.

Implementation	Epoch 1 NLL	Final NLL	Best NLL	Final Δ vs. Cirkit
Cirkit	3270.8736	638.3738	638.3738	–
XE log-space	3271.8185	637.8724	637.8724	-0.5014
XE scaled-max	3271.8335	637.9005	637.9005	-0.4733

I IMAGENET64 AND MONARCH LAYER EXPERIMENTS

To determine whether Extended Einsum also benefits larger datasets and structured parameterizations, this section compares traditional and structured Monarch layers on ImageNet64 [Chrabaszcz et al., 2017].

The Monarch experiment uses full 64×64 grayscale ImageNet64 images and batch size 256. Images are read from the official downsampled ImageNet shards, converted deterministically to 8-bit luminance, and presented in the same order to each paired backend.

Here H is the number of units in each hidden region. A Monarch layer [Zhang et al., 2025] maps an input matrix $z \in \mathbb{R}^{p \times q}$ through a weight tensor $W \in \mathbb{R}^{r \times s \times p \times q}$ that is represented by two factor tensors $A \in \mathbb{R}^{r \times p \times q}$ and $B \in \mathbb{R}^{s \times q \times r}$. As in the contraction-path example of the main paper, the layer is evaluated without materializing W, using the pairwise schedule

$$T = \text{einsum}(ij, kij \to kj;\ z, A), \qquad h = \text{einsum}(kj, ljk \to kl;\ T, B).$$

We choose positive integer factors P and Q with $H = PQ$, and reshape each length-H hidden vector into a $P \times Q$ matrix. This specializes $p = r = P$ and $q = s = Q$. The two mathematical factor tensors therefore have shapes

$$A \in \mathbb{R}^{P \times P \times Q}, \qquad B \in \mathbb{R}^{Q \times Q \times P},$$

matching the contractions

$$T_{kj} = \sum_i z_{ij} A_{kij}, \qquad h_{k\ell} = \sum_j T_{kj} B_{\ell jk}.$$

The implementation stores the same entries in contraction-friendly layouts A_{jki} and $B_{k\ell j}$, whose raw-logit tensor shapes are (Q, P, P) and (P, Q, Q), respectively. Only the contracted input axis of each factor is normalized. Thus, each structured hidden $H \to H$ sum has $P^2Q + PQ^2 = H(P + Q)$ trainable logits. Categorical leaves and the $H \to 1$ root remain dense, and quad-graph partition mixing uses the same compact parameterization in both systems. The benchmark asserts equal unique trainable parameter counts and equal canonical initialization within every paired run.

Table 10 reports absolute times, parameter counts, and run ranges. Dense and Monarch rows use different widths as representative scalability points. Comparisons are made only within a row, where model parameters and inputs match exactly.

Table 10: Dense and Monarch forward-and-backward runtime on full ImageNet64 images. Times and speedups are medians over five independent paired runs.

Topology	Sum	H	$P \times Q$	Parameters	Cirkit (ms)	Extended Einsum (ms)	Speedup	Run range
Quad tree	Dense	128	—	268.4M	51.7	34.6	$1.495\times$	$[1.492, 1.500]\times$
Quad tree	Monarch	512	16×32	738.1M	360.3	198.1	$1.819\times$	$[1.817, 1.823]\times$
Quad graph	Dense	64	—	134.4M	45.2	35.5	$1.272\times$	$[1.265, 1.273]\times$
Quad graph	Monarch	256	16×16	403.3M	317.4	225.0	$1.411\times$	$[1.410, 1.413]\times$

J PARAMETER-MATCHED COMPARISON WITH PYJUICE

To contextualize performance against a specialized PC library, we additionally compare Extended Einsum and cirkit against PyJuice 2.6.1 [Liu et al., 2024] on CP-T circuits. PyJuice always places a product layer directly after an input layer, whereas a conventional CP circuit begins with a sum layer over the inputs. PyJuice therefore cannot express a parameter-matched CP circuit. Unlike a conventional CP layer, a CP-T layer first multiplies corresponding units from two child regions and then applies a dense sum. This operation maps directly onto a PyJuice product node followed by a sum node, which avoids the extra product layer that would otherwise leave the comparison parameter-mismatched.

All systems use the same quad tree over 784 categorical variables, with 783 internal CP-T patches. Before timing, the benchmark verifies that input, product, and sum counts match. It also verifies equal product and sum scopes, equal output widths, and the exact logical parameter count

$$784U \cdot 256 + 782U^2 + U.$$

We evaluate batch/width pairs $256/64$, $512/64$, $256/128$, and $512/512$, using five independent runs, 30 warmup batches, and 90 measured batches. Each system repeatedly receives the same run-specific synthetic categorical batch, removing data-loading variation.

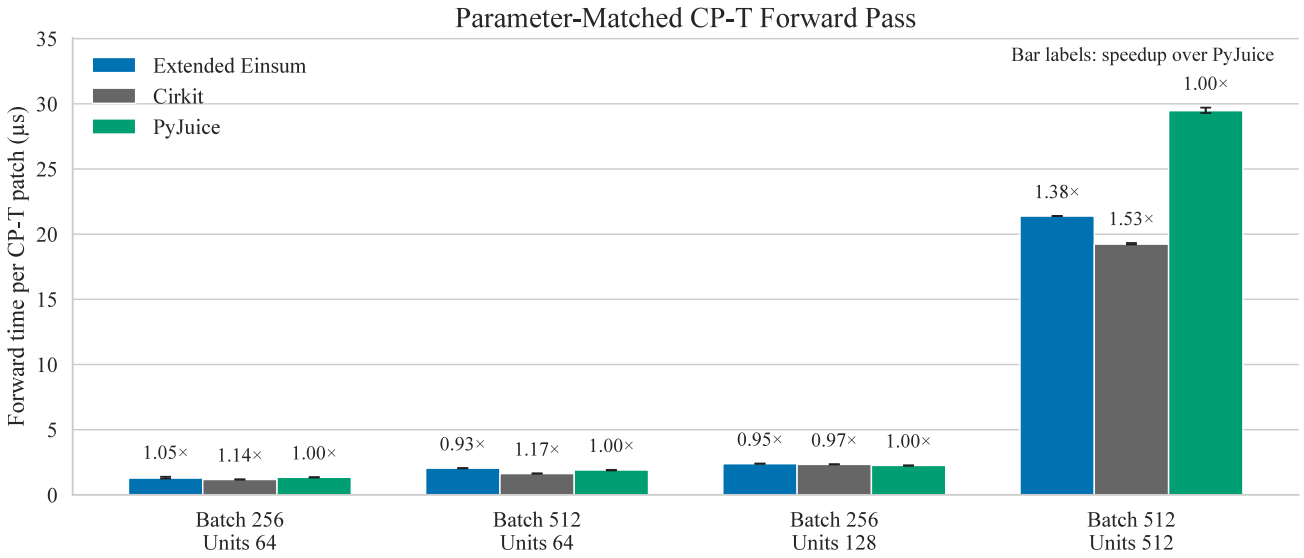

Figure 14: Forward time per internal CP-T patch for exactly parameter-matched quad trees. Labels report speedup over PyJuice. Values above one favor the indicated system. Error bars are 95% bootstrap intervals over five independent runs.

The forward pass is the cleanest cross-system comparison, because the circuits agree in structure, scopes, widths, and logical parameter counts, and because inputs, warmup, and measurement counts are identical across systems. Extended Einsum stays within 7% of PyJuice's forward runtime at the three lower-width settings and is $1.38\times$ faster at batch size 512 and width 512. Extended Einsum and cirkit run through compiled PyTorch, while PyJuice uses its native Triton and CUDA-graph path. The two designs also target different regimes. Extended Einsum and cirkit lower their contractions to dense matrix multiplication, whereas PyJuice's custom kernels are built to exploit block-sparse structure. The parameter-matched circuit used here is dense, so these results should not be extrapolated to circuits with exploitable structural sparsity.

The backward passes are not mathematically equivalent. PyJuice's native backward computes nonnegative EM parameter flows, whereas Extended Einsum and cirkit compute log-likelihood gradients with respect to the parameter logits. Figure 15 and Table 11 therefore report a systems-level timing comparison and not evidence that one system performs the same training update faster.

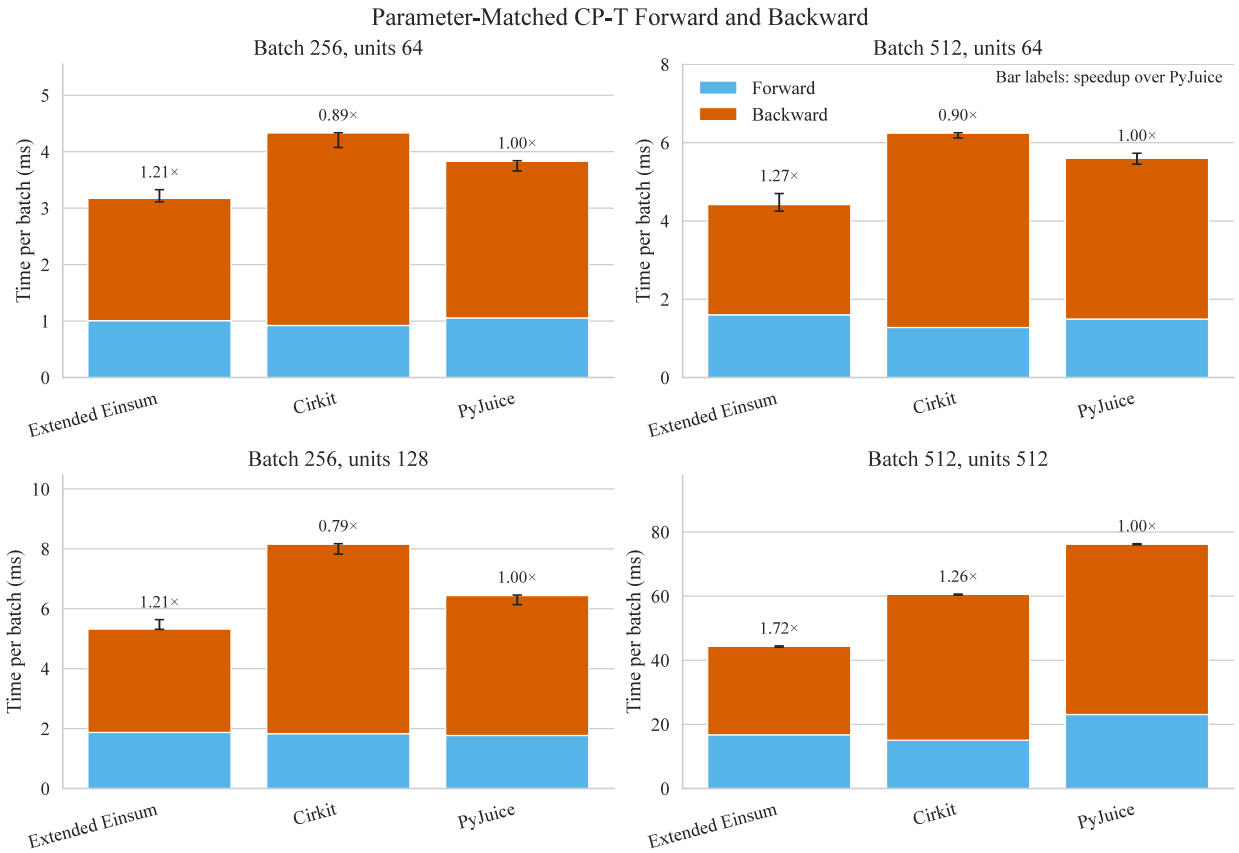

Figure 15: Forward and backward time for parameter-matched CP-T quad trees. Bars stack forward and backward time. The backward quantities differ across systems, so this is not a direct comparison.

Table 11: Median runtime in milliseconds over five independent runs, with median absolute deviation (MAD). Forward is structurally matched. Forward plus backward compares different backward quantities and must be interpreted with the qualification in the text.

Batch	Units	Forward runtime (ms)			Forward+backward runtime (ms)		
		PyJuice	Extended Einsum	Cirkit	PyJuice	Extended Einsum	Cirkit
256	64	1.0552 ± 0.0052	1.0071 ± 0.0050	0.9221 ± 0.0053	3.8354 ± 0.0053	3.1722 ± 0.0246	4.3243 ± 0.0052
256	128	1.7692 ± 0.0022	1.8716 ± 0.0002	1.8295 ± 0.0009	6.4416 ± 0.0159	5.3238 ± 0.0062	8.1539 ± 0.0181
512	64	1.4946 ± 0.0050	1.6025 ± 0.0030	1.2781 ± 0.0049	5.5971 ± 0.0056	4.4145 ± 0.1637	6.2347 ± 0.0192
512	512	23.0779 ± 0.0300	16.7474 ± 0.0057	15.0612 ± 0.0192	76.2681 ± 0.0803	44.3691 ± 0.0527	60.5771 ± 0.0602

K PREPROCESSING ALGORITHM AND PARAMETERS

We give implementation-level details for the three preprocessing steps summarized in the Compiler Optimizations section of the main paper. The input to every step is a static single-assignment (SSA) DAG whose values carry a shape and tensor-format annotation. The publication experiments use input-frontier folding with consumer ordering enabled, followed by contraction-path optimization. The passes can also be enabled separately.

Table 12 collects the parameters that change the preprocessing result. Parameters marked "automatic" are inferred from the SSA graph. The contraction attempts and beam width are fixed implementation constants rather than tuned per workload.

Table 12: Preprocessing parameters used in the implementation and publication experiments.

Parameter	Value	Meaning
Contraction-path attempts	8	Deterministic `cgreedy` trials with seeds $0, \ldots, 7$.
Path objective	`size`	Minimize the largest/materialized intermediate rather than only arithmetic count.
`max_repeats`, `skops_alpha`	128, 10	Search effort and contraction-tree balance penalty passed to `cgreedy`; `is_linear` is false.
Minimum fold-group size	2	A singleton is never rewritten as a batched operation.
Folding frontier	input depth	Publication setting; output-depth grouping is also implemented.
`split_by_routing`	automatic	Separates otherwise compatible folds whose operand fan-outs or downstream consumer kinds differ; automatic enables it for CP.
`order_by_input_access`	automatic	Uses input-access order for the same CP case; otherwise retains the group order before consumer optimization.
`optimize_group_order`	true	Enables consumer-aware fold-axis ordering.
Beam width	16	Maximum number of distinct complete layout states retained after each fold group.

K.1 CONTRACTION ORDERING

The contraction pass walks backward from each reachable einsum and grows a connected component by following its operand edges. It crosses an edge and merges the producer into the current component only if the producer is itself an einsum and its result has exactly one consumer. Every other operand becomes a boundary input of the merged expression. In particular, a non-einsum producer stops merging along that branch, as does an einsum result used by two or more consumers: absorbing the latter would duplicate a shared computation. The traversal may later discover another component beyond a non-einsum operation, but the two components remain separate. Components containing fewer than two einsums require no rewrite and are discarded. Within a component, index labels are allocated consistently before producer index strings are substituted into their consumers.

Numerical-stability semantics introduce one additional boundary rule. In any mode other than `unstable`, a cheap contraction-free product is normally kept outside the component. A contraction-free *outer* product may nevertheless be fused with its following reduction, because this avoids materializing the expanded tensor. A component consisting of exactly this outer-product/reduction pair is emitted as one multi-input einsum. Larger stable components containing such a pair are conservatively left unchanged, which avoids replacing a hierarchy of small Tucker kernels by one high-arity einsum.

For every other eligible component, the substituted expression and the shapes of its boundary operands are passed to the `sesum` implementation of `cgreedy`. The search uses

$$\texttt{minimize} = \texttt{size}, \qquad \texttt{is\_linear} = \texttt{false}, \qquad \texttt{max\_repeats} = 128, \qquad \texttt{skops\_alpha} = 10.$$

The `size` objective favors paths with smaller materialized intermediates. Setting `is_linear` to false permits general contraction trees rather than restricting the search to a linear sequence. `max_repeats` controls the search effort. The `skops_alpha` term penalizes unbalanced contraction trees: increasing it pushes the path finder more strongly toward balanced trees, which can perform better in practice even when a more unbalanced path has a comparable intermediate-size score. We use 10 for this tradeoff. Because the heuristic depends on its seed, the implementation tries eight deterministic seeds, $0, \ldots, 7$, and accepts the first valid plan. A plan is valid only if the longest SSA path to the component output is no

deeper than in the original program. This depth guard prevents an intermediate-size improvement from serializing otherwise parallel work. If a trial fails, has an invalid expression, or violates the depth guard, the next seed is tried; if all trials fail, the original component is retained.

The selected path is expanded into binary einsums. At each step an index is kept if it occurs in a not-yet-contracted operand or in the final output. Scaled min, max, and sum modes additionally place final-output indices first in intermediate format strings; this changes axis order but not the contraction. The old component is then replaced in topological order while all external SSA uses are remapped to the final planned result.

K.2 FOLDING

Folding finds independent, shape-compatible instructions and evaluates them with one operator carrying a new leading fold axis. The batchable operator set is

$$\{\sin, \cos, \tan, \exp, \log, \sqrt{\ }, (\cdot)^{-1}, +, -, \times, /, \mathrm{einsum}, \mathrm{stack}, \mathrm{slice}, \mathrm{select}, \mathrm{softmax}\}.$$

The default minimum group size is two. Members must have the same canonical operator, canonical operand shapes, operand tensor formats, and grouping depth, and no member may depend on another member in the group. Addition and multiplication operands are sorted canonically by shape and tensor format. Einsums are canonicalized by renaming index symbols from the output outward and sorting indistinguishable operands. Thus, superficial differences in operand order or index names do not prevent a legal fold.

The implementation supports two notions of grouping depth. Output-depth folding assigns a live SSA value its shortest distance to the program output. It additionally separates groups whose corresponding operands come from incompatible source kinds, and iteratively refines them at stack boundaries and producer-group boundaries. This conservative mode guarantees that the batched operands can be materialized uniformly.

The publication path instead uses *input-frontier depth*: the longest SSA distance from a program input. A transform depending only on parameters would otherwise appear near the input even when its first use is much later, so such a value inherits the earliest live frontier of its consumers. Repeated use of the same operand across group members is allowed in this mode because the rewrite can route or repeat the corresponding element.

Matching operator, shape, format, and input depth does not necessarily imply matching dataflow. For example, one candidate operation may read an operand that is used only once and feed an einsum, while another may read a shared operand and feed a pointwise product. Placing both in one fold group forces them to share a single fold-axis order even though their surrounding branches have different routing requirements. With `split_by_routing`, each candidate member is therefore assigned a routing signature consisting of the fan-out of every canonical operand and the multiset of operator kinds that consume its result. Only members with the same signature remain in the same fold group. The option defaults to true for CP programs with a same-index product contraction, where input-depth grouping would otherwise combine spatially separate branches. It can be forced on or off, and any split part smaller than two is discarded.

Input-only pointwise operations are handled specially. They are grouped across ordinary depth boundaries when their operator and operand signatures match, while still requiring at least two independent members. After all groups have been chosen, grouped and ungrouped instructions become atomic events and are topologically sorted. Rewriting a group stacks each canonical operand position, adds a fresh leading label to an einsum (or the analogous batch axis to another operator), emits one batched instruction, and records the physical position of every original result.

K.3 MEMORY-LAYOUT OPTIMIZATION (CONSUMER ORDERING)

The member order within a fold group is semantically free but determines the physical order of the fold axis. The implementation first constructs a small candidate set for each group. For a group with m canonical operands, it contains at most $2m + 4$ orders before duplicate removal:

1. the original operation order;
2. the order induced by the earliest future consumer of each member;
3. for every operand position, two source-based orders, one sorting by source tensor and source position and one prioritizing consumer order within a common source; and

4. two all-operand lexicographic orders, traversing operand positions forward and backward.

Original operation indices break all ordering ties. Before the global search, groups are visited in reverse topological order and the locally best candidate is selected using, in order, the number of fragmented future-consumer runs, the number of source runs needed to build the group's own operands, the future consumer keys, and original operation indices. This produces a strong, deterministic initial layout.

Beam-search state and transition. Let the folded events be $G_1, \ldots, G_g$. A search state stores a complete member order for every group, a map from every logical result to its $(\text{group}, \text{position})$, any induced axis-0 input permutations, the estimated cost of every grouped event, and their sum. The state key is the tuple of original operation indices in each member order. Starting with the reverse-topological layout, the search visits groups in topological event order. For every retained state it regenerates the current group's candidate orders, substitutes each candidate, updates the position map, and scores the resulting complete layout.

Candidates depend only on the current group, its grouped producers, and its grouped consumers. The implementation therefore caches candidate sets by the orders of precisely these dependency groups. Ordinarily a transition changes only the cost of the reordered group and its grouped consumers. If the group directly batches axis-0 selections from an input, its order can induce a new input permutation, in which case all group costs are recomputed.

After each group, states with identical order keys are deduplicated and sorted by

$$\big(C_{\text{total}},\ \text{state key}\big).$$

Only the best 16 states are retained. The final state is selected by the same ordering. The width 16 is fixed for every experiment; there is no workload-specific tuning, stochastic sampling, or enumeration of the $p!$ permutations of a p-member group. Keeping multiple complete layouts is important because changing a producer alone can appear worse until the consuming group is changed to the corresponding order.

Cost model and emitted layout. For each canonical operand of each folded event, the requested elements are represented as pairs (S, i), where S identifies an original input, a direct axis-0 input view, an earlier producer group, or an ungrouped SSA value, and i is its physical position. Consecutive pairs $(S, i), (S, i+1), \ldots$ form one maximal run. For runs $\mathcal{R} = (R_1, \ldots, R_k)$, the estimated number of layout instructions is

$$C(\mathcal{R}) = \sum_{j=1}^{k} \mathbb{I}[R_j \text{ is a strict slice of its source}] + \mathbb{I}[k > 1].$$

A run covering a complete input view or complete producer group is free, a strict slice costs one instruction, and two or more runs add one concatenation instruction. Costs are summed over canonical operands and fold groups. This is an IR instruction-count objective, not a hardware runtime model; the backend remains free to lower a fragmented batch as a gather.

The final rewrite propagates the selected order to every use. Parameter inputs used together are exposed as stack orders and packed once. For distinct axis-0 selections from a non-parameter input, nonoverlapping runs are chosen by decreasing length and earlier-event priority, completed with unused indices, and applied once as an input permutation. Complete sources are reused, strict runs become slices, and fragmented sequences become concatenations or explicit gathers. Returned metadata records batched-result and stack orders, input permutations, concatenation orders, and gather-index orders, so the runtime boundary realizes the layout without changing the logical program.

L EXTENDED EINSUM LANGUAGE AND INTERFACE

To make the compiler abstraction precise and reproducible, this section defines the language's values, operators, shape rules, and Python interface.

Extended Einsum is a purely functional tensor-expression language embedded in Python. Its values are tensors with statically known shapes, and its expressions form directed acyclic graphs. Constructing an expression is lazy: an operator returns a tensor expression without immediately evaluating it. Evaluation is requested explicitly through the `materialize` method.

Let x denote an input tensor, a an axis, A a nonempty tuple of distinct axes, i an integer index, s and t slice bounds, and σ an Einstein-summation specification, defined below. The expression language is

$$\begin{aligned} e ::= {} & x \mid f(e) \mid e \circ e \mid \mathrm{einsum}_{\sigma}(e_1, \ldots, e_k) \\ & \mid \mathrm{stack}_a(e_1, \ldots, e_k) \mid \mathrm{take}_a(e, j) \mid \mathrm{slice}_{a,s:t}(e) \\ & \mid \mathrm{select}_{a,i}(e) \mid \mathrm{softmax}_A(e), \end{aligned} \tag{1}$$

where

$$f \in \left\{\sin, \cos, \tan, \exp, \log, \sqrt{\ }, (\cdot)^{-1}\right\}, \qquad \circ \in \{+, -, \times, /\}.$$

All operators are side-effect free and produce a single tensor-valued result. There is no mutation, control flow, or implicit data-dependent change of shape.

Inputs and expressions. A backend tensor X is introduced using

`xe.array(X)`.

The resulting object records the shape of X and may be used as a leaf in subsequent expressions. All operands of one expression must use the same numerical backend. Scalar inputs are represented by rank-zero backend tensors. Every tensor expression exposes its statically inferred `shape`. If

$$\mathrm{shape}(X) = (d_0, \ldots, d_{r-1}),$$

then r is the rank of X. Negative axes accepted by the Python interface are interpreted relative to the rank in the usual way.

Elementwise operators. The unary functions are applied elementwise and preserve shape:

$$\mathrm{shape}(f(X)) = \mathrm{shape}(X).$$

They are constructed by the functions

`xe.sin`, `xe.cos`, `xe.tan`, `xe.exp`, `xe.log`, `xe.sqrt`, and `xe.inverse`.

The inverse operation denotes elementwise reciprocal,

$$\mathrm{inverse}(X)_{\boldsymbol{i}} = \frac{1}{X_{\boldsymbol{i}}}.$$

Addition, subtraction, multiplication, and division are written using the corresponding Python operators:

$$E_1 + E_2, \qquad E_1 - E_2, \qquad E_1 * E_2, \qquad E_1 / E_2.$$

Their operands follow the usual tensor-broadcasting rules. Consequently,

$$\mathrm{shape}(E_1 \circ E_2) = \mathrm{broadcast}\big(\mathrm{shape}(E_1), \mathrm{shape}(E_2)\big).$$

Einstein summation. An Einstein summation is constructed as

$$\texttt{xe.einsum}(\sigma, E_1, \ldots, E_k),$$

where

$$\sigma = (s_1, \ldots, s_k \to s_{\mathrm{out}})$$

consists of one index string s_ℓ for each operand and an output index string s_{out}. Each symbol names an axis. The number of input strings must equal the number of operands, the rank of E_ℓ must equal the length of s_ℓ, and all axes carrying the same symbol must have the same size. Every output symbol must occur in an input string and may occur only once in the output.

Writing I for the set of all index symbols in the input strings, I_{out} for the symbols in s_{out}, and $R = I \setminus I_{\mathrm{out}}$ for the reduction indices, the semantics are

$$\mathrm{einsum}_\sigma(E_1, \ldots, E_k)_{\boldsymbol{i}_{s_{\mathrm{out}}}} = \sum_{\boldsymbol{i}_R} \prod_{\ell=1}^{k} (E_\ell)_{\boldsymbol{i}_{s_\ell}}.$$

Thus, indices absent from the output are summed over, while the output axes appear in the order given by s_{out}. For example,

$$\mathrm{einsum}_{ab,bc\to ac}(X, W)_{ac} = \sum_b X_{ab} W_{bc}.$$

The output shape is the tuple of sizes of the output indices in their specified order. Einstein summation therefore subsumes contractions, reductions, transpositions, outer products, Hadamard products, and matrix multiplication.

Stacking. Given $k \geq 1$ tensors of equal shape

$$\mathrm{shape}(E_\ell) = (d_0, \ldots, d_{r-1}),$$

the expression

$$\texttt{xe.stack}([E_1, \ldots, E_k],\ \ \texttt{axis} = a)$$

introduces a new axis of size k:

$$\mathrm{shape}\big(\mathrm{stack}_a(E_1, \ldots, E_k)\big) = (d_0, \ldots, d_{a-1}, k, d_a, \ldots, d_{r-1}).$$

Stacking differs from broadcasting: it preserves every operand as a distinct slice along the newly introduced axis.

Gathering. The expression

$$\texttt{xe.take}(E, J,\ \ \texttt{axis} = a)$$

gathers elements of E along axis a, using the entries of the rank-one index tensor J. If

$$\mathrm{shape}(E) = (d_0, \ldots, d_{r-1}), \qquad \mathrm{shape}(J) = (m),$$

then

$$\mathrm{shape}\big(\mathrm{take}_a(E, J)\big) = (d_0, \ldots, d_{a-1}, m, d_{a+1}, \ldots, d_{r-1}).$$

In components,

$$\big(\mathrm{take}_a(E, J)\big)_{i_0,\ldots,i_{a-1},q,i_{a+1},\ldots,i_{r-1}} = E_{i_0,\ldots,i_{a-1},J_q,i_{a+1},\ldots,i_{r-1}}.$$

Slicing. A contiguous half-open interval is selected using

$$\texttt{xe.slice}(E, s, t,\ \ \texttt{axis} = a).$$

For $0 \leq s \leq t \leq d_a$, its shape is

$$\mathrm{shape}\big(\mathrm{slice}_{a,s:t}(E)\big) = (d_0, \ldots, d_{a-1}, t - s, d_{a+1}, \ldots, d_{r-1}),$$

and its entries satisfy

$$\big(\mathrm{slice}_{a,s:t}(E)\big)_{i_0,\ldots,i_{a-1},q,i_{a+1},\ldots,i_{r-1}} = E_{i_0,\ldots,i_{a-1},s+q,i_{a+1},\ldots,i_{r-1}}.$$

Table 13: Public Extended Einsum expression interface.

Operation	Python interface	Shape effect
Input	`xe.array(X)`	shape of X
Unary function	`xe.exp(E)`, `xe.log(E)`, etc.	unchanged
Binary arithmetic	`E1 + E2`, `E1 * E2`, etc.	broadcasting
Einstein summation	`xe.einsum(`σ`, E1, ..., Ek)`	output indices of σ
Stack	`xe.stack([E1, ..., Ek], axis=a)`	inserts an axis
Take	`xe.take(E, J, axis=a)`	replaces one axis
Slice	`xe.slice(E, s, t, axis=a)`	shortens one axis
Select	`xe.select(E, i, axis=a)`	removes one axis
Softmax	`xe.softmax(E, axis=A)`	unchanged
Materialization	`E.materialize()`	unchanged

Selection. A single index is selected using

$$\texttt{xe.select}(E, i,\ \texttt{axis} = a).$$

Unlike `take`, selection removes the indexed axis. For $0 \le i < d_a$,

$$\operatorname{shape}\big(\operatorname{select}_{a,i}(E)\big) = (d_0, \ldots, d_{a-1}, d_{a+1}, \ldots, d_{r-1}),$$

with

$$\big(\operatorname{select}_{a,i}(E)\big)_{i_0,\ldots,i_{a-1},i_{a+1},\ldots,i_{r-1}} = E_{i_0,\ldots,i_{a-1},i,i_{a+1},\ldots,i_{r-1}}.$$

Softmax. Softmax is constructed as

$$\texttt{xe.softmax}(E,\ \texttt{axis} = A),$$

where A may be a single axis or a nonempty tuple of distinct axes. It preserves the input shape. Writing $\overline{A}$ for the complementary axes, the operation normalizes jointly over all indices belonging to A:

$$\operatorname{softmax}_A(E)_{\boldsymbol{i}_{\overline{A}},\boldsymbol{i}_A} = \frac{\exp\left(E_{\boldsymbol{i}_{\overline{A}},\boldsymbol{i}_A}\right)}{\sum_{\boldsymbol{j}_A} \exp\left(E_{\boldsymbol{i}_{\overline{A}},\boldsymbol{j}_A}\right)}.$$

A tuple of axes therefore denotes one joint normalization, rather than a sequence of independent one-axis softmax operations.

Materialization. Given a tensor expression E, evaluation is requested explicitly by

`E.materialize()`.

The result is again an Extended Einsum array and can either participate in a new expression or be accessed through its underlying backend tensor. Expression construction and materialization are thus separated: the interface first describes the complete tensor computation and only then requests its numerical evaluation.

M BACKEND INTERFACE

To add a numerical backend, its array type must expose a `shape` property and the backend must implement the operations in Table 14. The first group directly realizes the language defined in Appendix L. Exponentiation, logarithm, and several arithmetic operations also appear in the stable translations, but are language operations in their own right. Concatenation is not public syntax, but preprocessing introduces it when folded intermediates are joined. The final group is required only by the scaled and log-space translations: reductions choose normalization factors or reference shifts, `maximum` combines shifts, `reshape` and `broadcast_to` align their shapes, and `stop_gradient` prevents differentiation through these numerically arbitrary references. Thus, an unstable interpreter for the language does not need this final group.

Table 14: Functions required from a numerical backend.

Purpose	Backend functions
Public language	`sin`, `cos`, `tan`, `exp`, `log`, `sqrt`, `inverse`, `stack`, `take`, `select`, `slice`, `softmax`, `einsum`, `add`, `subtract`, `multiply`, and `divide`
Internal preprocessing	`concat`
Numerical stability	`stop_gradient`, `sum`, `max`, `min`, `maximum`, `reshape`, and `broadcast_to`

The backend compiler is separate from these primitive operations: its `compile` method maps the resulting straight-line backend program and its example inputs to an executable callable. The PyTorch backend implements this hook with `torch.compile`.