BSTroM: A 28-nm Processing Unit with Sparse Dataflow & Backward-Sampling Support for Tractable Probabilistic Models

Anjunyi Fan[1], Hongxiao Zhao[1], Yihan Fu[1], Anji Liu[2], Yaojun Zhang[3] and Bonan Yan[1*]
[1]Institute for Artificial Intelligence, Peking University, Beijing, China
[2]School of Computing, National University of Singapore, Singapore
[3]Pimchip Technology Co., Ltd., Beijing, China.
*Email: bonanyan@pku.edu.cn

***Abstract*—Tractable probabilistic models (TPMs) are emerging in artificial intelligence, with probabilistic circuits (PCs) serving as a foundational architecture. Their internal connections are represented as directed acyclic graphs (DAGs) with sum and product nodes, ensuring expressive power for probabilistic inference. Executing TPMs remains challenging due to the irregular DAG structure. In this work, we propose the BSTroM, an efficient TPM processing unit that enables parallel execution through element-wise scheduling. It is hardware-efficient with datapath reuse for TPM forward inference and backward sampling. It implements local exponent normalization for underflow prevention and log-exp multiply-division for backward support. BSTroM reduces the TPM bandwidth requirement by 67.1%/60.8% in the forward/backward pass, and 13.7× more energy efficient compared to the SoTA DPU solution with negligible degradation in log-likelihood quality. Chip measurements show that it achieves 1.78 TFLOPS/W energy efficiency and 7.31 GFLOPS/mm^2 area efficiency.**

***Index Terms*—Tractable Probabilistic Models, AI Accelerator, Graph, Underflow Prevention, Divider.**

I. Introduction

Tractable probabilistic models (TPMs) constitute a broad class of probabilistic AI models that support efficient inference. Within this family, probabilistic circuits (PCs) provide a unified computational framework, with sum–product networks (SPNs) representing a well-known and constrained subclass [1]. TPMs are widely employed in probabilistic inference, density estimation, deep generative modeling, and uncertainty-aware machine learning [2], [3].

As shown in Fig. 1, TPMs are typically structured as sparsely connected directed acyclic graphs (DAGs). They consist of alternating sum and product nodes, which combine latent variables to represent complex probability distributions.

However, key challenges hinder the fast and efficient execution of TPMs on general-purpose CPUs/GPUs [2], [3] or state-of-the-art (SoTA) accelerator designs [4], [5]:

- **Memory-bound behavior**: For *performance*, over 80% of memory accesses during computation involve reusable structured data, causing performance to be limited by memory bandwidth rather than compute throughput.
- **Lack of backward sampling**: For *function*, practical TPM models require both forward pass (FW) and backward pass (BP). Previous accelerators only support FW. Supporting the BP introduces additional design complexity with division.

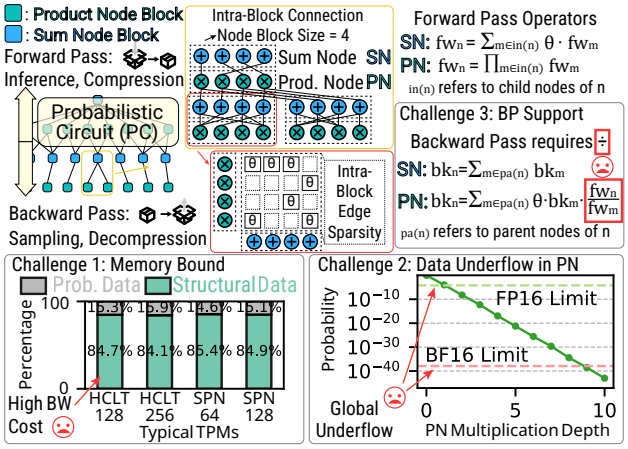


Fig. 1. TPM model structure & challenges for sparse DAG-based TPMs.

- **Underflow risk**: For *precision*, since probabilities are less than 1, repeated multiplication across product nodes can cause intermediate values to approach zero, leading to numerical underflow as the graph depth increases.

Prior works on accelerating SPNs have notable limitations. The DAG processing unit (DPU) [4] and its successor DPU-v2 [5] exploit intra-block edge sparsity supporting subgraph structures for FW acceleration, but do not support BP. Meanwhile, LSE-PE [6] introduces logarithmic computation to avoid underflow, yet lacks a hierarchical structure for DAGs.

To address these challenges, we propose **BSTroM**, a TPM accelerator built upon the element-wise dataflow of ES-TroM [7]. Beyond ESTroM, BSTroM introduces three key innovations:

- *Memory-reusable element computing with datapath reconfiguration:* BSTroM manages DAG execution through graph-element indexing, storing indices in on-chip structure memory to reduce external bandwidth. By reconfiguring datapaths between FW and BP, the same structure memory is reused, achieving high hardware utilization.
- *BP support via lossless log-exp multiply-division (LEMD) circuit:* BSTroM implements exponential and logarithmic conversions within a unified LEMD unit, eliminating costly division operations. Numerical precision is preserved

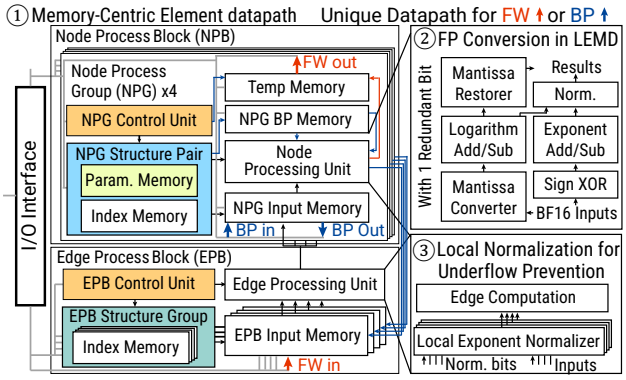


Fig. 2. Circuit scheme of BSTroM chip.

through redundant-bit representation, ensuring lossless computation during BP.

◇ *Underflow prevention through local normalization:* Local exponent normalizers and loggers are integrated to track and adjust probability exponents dynamically. This localized approach prevents global underflow without requiring additional data processing.

Together, these contributions enable BSTroM to perform both FW and BP efficiently while maintaining numerical stability across large-scale DAG-based probabilistic models.

II. BSTroM Design

A BSTroM core (Fig. 2) accelerates TPM with node block sparse subgraphs (stored in compressed adjacency matrix format) generated by PyJuice framework [2]. Model parameters and probability values are represented in BF16. Subgraph execution proceeds through element indexing and processing, achieving sparsity-aware computation through parallel execution along edges and nodes. This core consists of (a) Edge Process Block (EPB), (b) Node Process Block (NPB), and (c) I/O Interface. Sparsity handling and datapath reconfiguration are managed by element indexing inside the EPB and NPB.

Edge process block handles parameter-free operations: FW product nodes and BP sum nodes. It contains edge Processing Unit (PU), an edge Control Unit (CU), and 4×2 KB SRAMs as index memories for graph adjacency matrices and input memories for node probability. The edge PU uses local exponent normalizers to prevent underflow.

Node process block handles parameterized FW sum nodes and BP product nodes. Besides all components as in EPB, NPB's index memory is equipped with additional 2 KB parameter memory, temp memory, and BP memory, associating an independent node PU and CU, called a node process group (NPG). NPB comprises 4 NPGs to operate in parallel. Division is implemented using LEMD within node PUs.

I/O interface manages data transfers with 13-bit address (3-bit global address and 10-bit local address) and 64-bit data bus. The global address selects 4 blocks of same-function memories simultaneously. The local address 16-bit data from each selected memory.

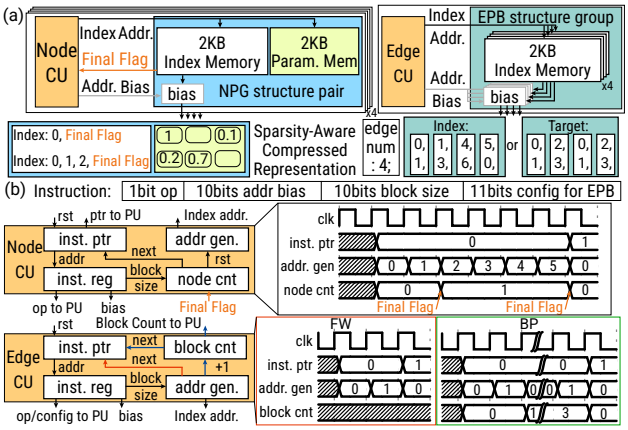


Fig. 3. (a) The structural memory encoding format. (b) Design and timing diagrams of CUs.

A. Feature 1: Element Indexing: FW & BP Datapath

EPB and NPB pipelines execute TPM computations through a coordinated element indexing and processing scheme. Fig. 3(a) illustrates the structure memories for NPG and EPB element indexing. In an NPG, the index memory stores the input indices for each node and a final flag marking the end. The parameter memory holds the corresponding parameter. The index and its parameter form pairs by being placed at the same address. The NPB contains 4 such pairs, enabling concurrent storage of different node connections within the same node block. Input probabilities reside in the input memory of its respective NPG for parallel data access.

Fig. 3(b) details the CU designs. Each node CU comprises an address generator, an instruction pointer, a set of instruction registers, and a node counter. The address generator starts at 0 and increments cyclically as the index address to read from the structure memories. 5×21 instruction registers coordinate execution. Each instruction contains the address bias, the block size, and an operation bit selecting the FW or BP. The node counter adds one when encountering a final flag. When the node counter equals the block size, the instruction pointer then advances to the next. Execution terminates when the pointer reaches the end. FW product nodes and BP sum nodes per block share the same input degree. The edge CU has a similar structure with following differences:

- Instructions are shared across EPB structure memories and include a 4-bit mask selection field and a 7-bit normalization exponent in an extra 11-bit config field.
- The index memory stores either input indices for FW or output targets for BP.
- The address generator is driven directly by the node counter, because each row in the index memory corresponds to one product node. A block counter tracks the number of traversed input memories for BP.

Fig. 4(a) illustrates the reconfigurable datapath of BSTroM. FW and BP reuse the same physical memories by reconfiguring the datapath. Structure memories supply addresses to the input memories. The operational sequence begins by loading

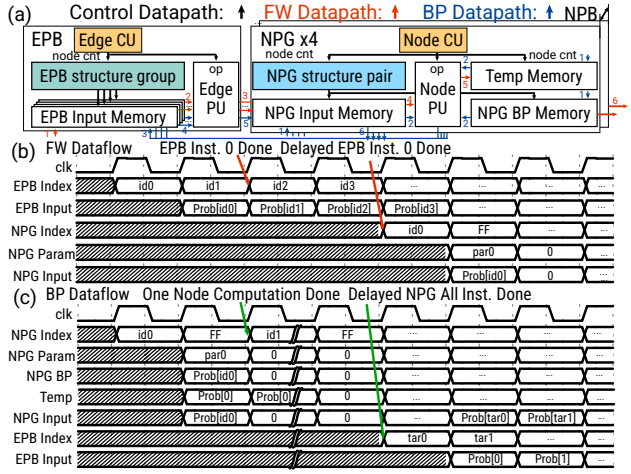


Fig. 4. (a) FW/BP Datapath. (b) Memory Timing Diagram for FW. (c) Memory Timing Diagram for BP.

the model structural data into the structure memories. When a new inference or sampling task is launched with different inputs, input memories and instruction registers are flushed.

Fig. 4(b) shows the timing diagram for FW. Input probabilities are distributed across four EPB memories and accessed in parallel via indices from structure memory. The edge PU processes the retrieved data and broadcasts results to all four NPG input memories. NPGs wait for the EPB instruction done signal, then process the data with their node PUs and write results to temp memory, completing a full FW.

Fig. 4(c) shows the timing diagram for BP. This process utilizes intermediate results from FW and engages all memory blocks. Node probabilities from FW are stored in temp memories, indexed by a node counter, while parent probabilities are held in BP memories, indexed by structure memory. BP begins from NPG input memories. Each NPG node PU generates a unique output, which is sent directly to the EPB input memory. The EPB synchronizes by waiting for all NPB instructions to finish, preventing address conflicts. Finally, the edge PU computes these values and writes results back into an NPG input memory, reusing the released addresses.

B. Feature 2: Precision-Control Element Processing PU

Fig. 5(a) details an edge PU circuit schematic. It comprises a multiplication tree and an adder, with 4 local exponent normalizers, an input mask/selector, a target mask, the accumulation selector, and a set of accumulation labels (ALs).

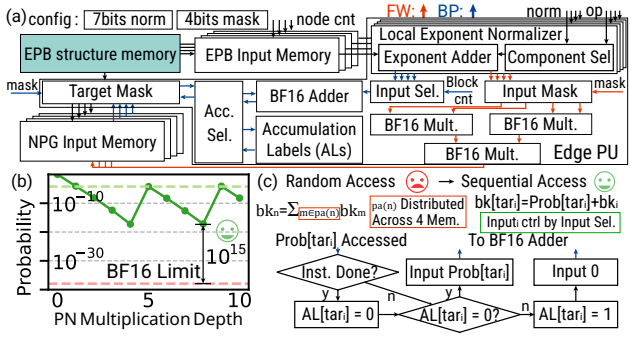


Fig. 5. (a) Edge PU design. (b) Underflow Prevention with Local Exponent Normalizer. (c) Accumulation Diagram for Sum Node BP.

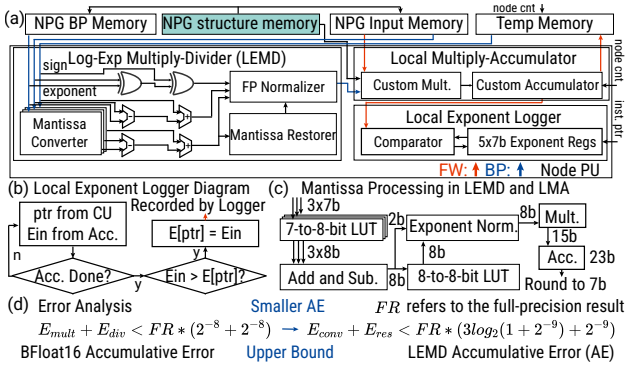


Fig. 6. (a) Node PU Design. (b) Local Exponent Logger Diagram. (c) Mantissa Conversion with LUTs. (d) Accumulative Error Analysis.

In *FW*, each normalizer adds a normalization exponent to the data fetched from the input memory. This pre-biasing step leaves 10^{15} margin, effectively preventing underflow in product nodes (Fig. 5(b)). After normalization, the input mask applies a 4-bit mask derived from the input degree to replace invalid inputs with 1. The masked data then proceeds through the multiplication tree to compute product nodes.

In *BP*, the normalizer subtracts the normalization exponent from the complement encoded data. To avoid random access contention in the input memories, accumulation occurs sequentially with ALs, as illustrated in Fig. 5(c). The input selector cycles through EPB input memories by the block counter, thereby covering all inputs. The target mask determines which NPG input memory receives the accumulated result with mask bits. Each target corresponds to a specific address in the NPG input memory, representing the running accumulation for that node.

The adder receives input and target operands from the accumulation selector. The ALs track if the target accumulation starts. All ALs are reset to 0 when the instruction begins. When an AL is 0, the accumulation selector supplies 0. When a target address is accessed, its AL is set to 1, and subsequent accesses use the target probability. Different product node blocks are processed by issuing separate instructions.

Fig. 6(a) depicts a node PU circuit, which comprises a local multiply-accumulator, an LEMD, and the local exponent logger. 4 node PUs in NPGs operate in parallel. In *FW*, the node PU performs multiply-accumulate cyclically in the local multiply-accumulator. The accumulator outputs the result to the logger and temp memory once one node computation completes. The logger records the max exponent for each instruction. Fig. 6(b) shows its mechanism. The logger has 5×7-bit registers. It utilizes the instruction pointer. Each register stores the offset of exponents relative to 0, representing the maximum output exponent.

BP extends FW computation by incorporating LEMD. The logger remains inactive during BP because the norm bits are from FW. The exponent and sign bits in LEMD are processed with integer circuits. Fig. 6(c) shows the mantissa processing in LEMD and the accumulator. The 7-bit mantis-

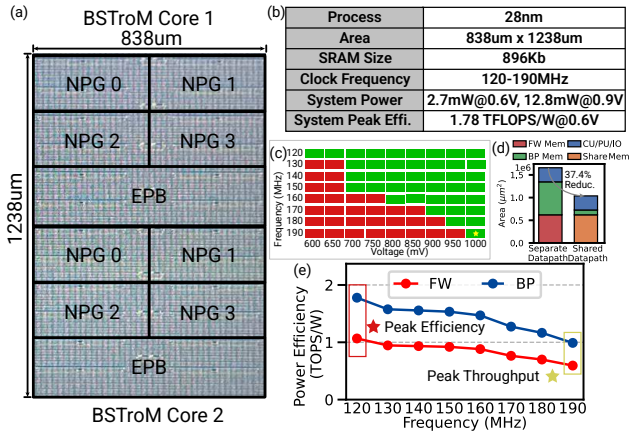
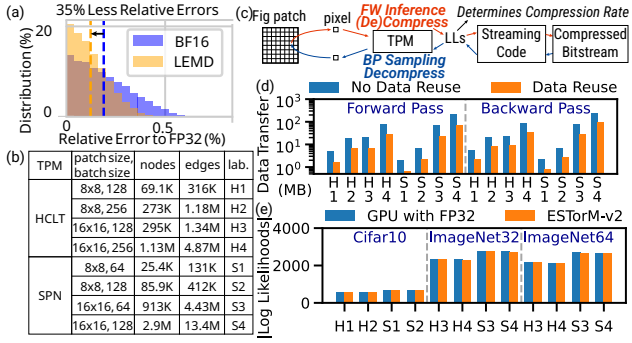


Fig. 7. (a) Die photo, (b) specifications, (c) shmoo plot, (d) area breakdown, (e) energy efficiency of BSTroM. Each mult./add/div. is one operation.



TPM	patch size, batch size	nodes	edges	lab.
HCLT	8x8, 128	69.1K	316K	H1
	8x8, 256	273K	1.18M	H2
	16x16, 128	295K	1.34M	H3
	16x16, 256	1.13M	4.87M	H4
SPN	8x8, 64	25.4K	131K	S1
	8x8, 128	85.9K	412K	S2
	16x16, 64	913K	4.43M	S3
	16x16, 128	2.9M	13.4M	S4

Fig. 8. (a) Error distribution (LEMD vs BF16). (b) Benchmark TPM Configs, (c) TPM-based Neural Lossless Compression Flow, (d) Bandwidth Requirement, (e) TPM Inference Results.

sas are first mapped to 8-bit logarithmic values using look-up tables (LUTs). This conversion replaces FP with simple integer addition and subtraction. Two overflow bits from the intermediate results control the normalization. The remaining 8 bits, including the redundant bit, are converted back and passed to the local multiply accumulator. The accumulator performs calculations internally with 23-bit mantissa precision, rounding only at the accumulation end. Fig. 6(d) details how LEMD avoids precision loss with less accumulative error.

III. Measured Results and Comparison

Fig. 7(a–b) show the die photo and specifications of two BSTroM cores in the system fabricated in a commercial 28nm CMOS process. Fig. 7(c) presents the shmoo plot. The following measurement is based on the system clock frequency with its corresponding lowest Vdd supplies. Fig. 7(d) shows the area breakdown of BSTroM. The datapath sharing reduces 37.4% area. Fig. 7(e) shows the measured results of energy efficiency versus frequencies, reaching 1.78 and 1.07 TFLOPS/W efficiency in BP and FW, respectively.

Fig. 8 summarizes the numerical behavior and system-level evaluation results on BSTroM. Fig. 8(a) details the accuracy of LEMD compared to BF16. It eliminates the relative error by 35% on average. Fig. 8(b) shows details of real-world TPMs for benchmark: HCLT [2] and SPN [3] on image

TABLE I
Comparison with SoTA TPM Accelerators.

Design	GPU RTX4090 [2]	DPU* [4]	DPU-v2 [5]	BSTroM (this work)
Tech. Node	7nm	28nm	28nm	28nm
Data Form.	FP32	POSIT8∼32	POSIT32	BF16
Underflow Prevention	Log-Exp	POSIT	POSIT	Exp Norm
Data Reuse	✘	✘	✘	✓
Frequency (MHz)	2100	278	300	190
Area Eff. (GOPS/mm^2)	N/A	4.68	0.86	7.31
Power Eff. (TOPS/W)	0.009	0.13	0.03	1.78
Support Operation	FW & BP	FW	FW	FW & BP

* Use POSIT32 for TPM comparison.

datasets with different block/patch sizes at intra-block sparsity 90%. Fig. 8(c) illustrates how TPM handles neural lossless compression. For each pixel, compression requires one FW, while decompression requires one FW and one BP. The inference log-likelihoods (LLs) determine the compression rate. Fig. 8(d) shows how the structural memory design benefits the bandwidth. BSTroM reduces bandwidth by 67.1% in FW and 60.8% in BP on average when the batch size is 4. Fig. 8(e) shows the TPM inference results. LLs have no evident degradation compared to FP32 with an average 0.6% variation. Tab. I is the comparison table with other works.

IV. Conclusion

This work presents BSTroM, a 28nm processing unit designed for TPMs with backward-sampling support. To the best of our knowledge, this work implements TPM BP acceleration for the first time. It achieves 1.78 TFLOPS/W peak energy efficiency and 7.31 GFLOPS/mm^2 area efficiency. BSTroM reduces the TPM bandwidth requirement by 67.1% and 60.8% in FW/BP, and is 13.7× more energy efficient than the SoTA DPU solution with negligible LLs degradation.

Acknowledgment

This work is supported by National Natural Science Foundation of China (T2350006, 92364102, 92264201), Beijing Nova Program, and the 111 Project under Grant B18001. This work is supported by High-performance Computing Platform of Peking University.

References

[1] M. Dang *et al.*, "Sparse probabilistic circuits via pruning and growing," *NeurIPS*, vol. 35, pp. 28 374–28 385, 2022.

[2] A. Liu *et al.*, "Scaling tractable probabilistic circuits: a systems perspective," in *ICML*, 2024, pp. 30 630–30 646.

[3] A. Molina *et al.*, "SPFlow: An easy and extensible library for deep probabilistic learning using sum-product networks," *arXiv preprint arXiv:1901.03704*, 2019.

[4] N. Shah *et al.*, "DPU: DAG processing unit for irregular graphs with precision-scalable posit arithmetic in 28 nm," *JSSC*, vol. 57, no. 8, pp. 2586–2596, 2021.

[5] N, Shah *et al.*, "DPU-v2: Energy-efficient execution of irregular directed acyclic graphs," in *MICRO*. IEEE, 2022, pp. 1288–1307.

[6] L. Yao *et al.*, "LogSumExp: Efficient approximate logarithm acceleration for embedded tractable probabilistic reasoning," *TCAS-I*, 2025.

[7] A. Fan *et al.*, "ESTroM: Element-flow architecture for processing sparse tractable probabilistic models," in *HPCA*. IEEE, 2026, pp. 1–15.