FASTOMNITMAE: PARALLEL CLAUSE LEARNING FOR SCALABLE AND HARDWARE-EFFICIENT TSETLIN MACHINE EMBEDDINGS

Anonymous authors
Paper under double-blind review

ABSTRACT

Embedding models in natural language processing (NLP) have achieved remarkable success, but their representations often lack interpretability. The Tsetlin Machine (TM) provides an interpretable, logic-based alternative. Omni TM Autoencoder (Omni TM-AE) applies this paradigm to static embeddings, but its training remains slow. In this work, we propose FastOmniTMAE, which replaces sequential training dependencies with a two-stage parallel process of evaluation and update. Across classification, similarity, and clustering tasks, FastOmniTMAE achieves up to $5\times$ faster training while maintaining comparable embedding quality. We further implement FastOmniTMAE as a reusable accelerator on SoC-FPGA platforms, achieving similarity scores of 0.669 on a resource-constrained FPGA and 0.696 on an UltraScale+ SoC. These results demonstrate an efficient and interpretable approach to logic-based embedding training with a small hardware footprint.

1 INTRODUCTION

Artificial Intelligence (AI) models are rapidly evolving and becoming increasingly complex to meet the demand for higher accuracy and broader capabilities. Among the most prominent examples are large language models such as ChatGPT and Gemini, which are primarily based on transformer architectures. However, these models have grown to a scale where understanding and verifying their decision-making processes has become challenging. This limitation arises from their reliance on neural networks, where knowledge is represented through continuous numerical values, making reasoning difficult to interpret. Consequently, there is a growing need for alternative approaches that provide transparency and explainability, particularly in applications where understanding the reasoning process is essential.

The Tsetlin Machine (TM) offers one such alternative for building transparent AI models (Granmo, 2018). It is inherently interpretable in both structure and reasoning. Structurally, TM is composed of logical operations governed by Tsetlin automata (TAs), which follow a well-defined learning mechanism (Omslandseter et al., 2024). From a reasoning perspective, decisions are formed through clauses, which represent logical expressions capturing relevant patterns in the data Abeyrathna et al. (2023). These clauses are directly traceable and interpretable, as demonstrated in prior works (Jeeru et al., 2025; Saha et al., 2023). Another key advantage of TM is its hardware-friendly nature, stemming from its reliance on simple logical operations. Previous studies Abeyrathna et al. (2021) and Tunheim et al. (2026) have explored hardware implementations of TM, primarily focusing on inference for classification tasks, with limited work addressing training on hardware (Maheshwari et al., 2023; Tunheim et al., 2025). This design enables the development of AI systems with a small computational footprint, without requiring specialized architectures typically used for neural network acceleration (Rahman et al., 2022; Kishore et al., 2023).

Previous work on Tsetlin Machine Autoencoder (TM-AE) (Bhattarai et al., 2024; Kadhim et al., 2025c; 2024; 2025a), including the Omni Tsetlin Machine Autoencoder (Omni TM-AE) model (Kadhim et al., 2025b), has primarily focused on achieving high embedding quality in comparison with models such as Word2Vec, GloVe, BERT, and FastText (Mikolov et al., 2013; Pennington et al., 2014; Devlin et al., 2018; Bojanowski et al., 2017). However, training efficiency has received limited attention, despite being a significant limitation. Reported training times can extend to several

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

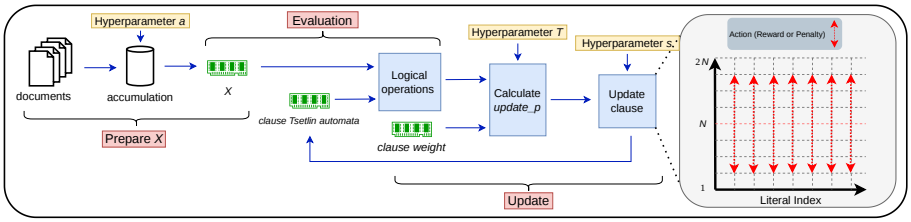

Figure 1: Overview of the TM-based embedding training flow, showing input accumulation, clause evaluation, update-probability calculation, and automaton-state updates.

months Kadhim et al. (2025c), while Omni TM-AE reduces this to several days by eliminating multi-layer structures Kadhim et al. (2025b). Additional related work is discussed in Appendix A.1. In this work, we analyze the TM architecture with a focus on training efficiency across different hardware platforms, highlighting its limitations, particularly on Graphics Processing Units (GPUs). We then introduce a redesigned version of the Omni TM-AE model, termed Fast Omni Tsetlin Machine Autoencoder (FastOmniTMAE), which improves training speed while preserving embedding quality. To evaluate embedding performance, we develop a dedicated benchmark that systematically assesses embedding quality in Natural Language Processing (NLP) tasks. This model also serves as the foundation for exploring training performance across heterogeneous hardware platforms.

The main contributions of this work are summarized as follows:

1. Developing a new Omni TM-AE architecture, FastOmniTMAE, that improves training speed while maintaining comparable embedding quality.
2. Introducing a benchmark (Single-Run Multi-Environment Benchmark) for evaluating embedding quality in NLP applications.
3. Analyzing the effectiveness of GPU-based training for TM and identifying associated inefficiencies.
4. Demonstrating the first full-scale training of a TM-based embedding model on Field-Programmable Gate Array (FPGA) hardware for NLP.
5. Proposing a benchmark (Multi-Hardware Benchmark) for comprehensive comparison across Central Processing Units (CPUs), GPUs, and FPGA-based systems.

2 Methodology

In this section, we first describe the Omni TM-AE approach in detail to highlight the weaknesses of the algorithm in terms of computation complexity. We then present the proposed FastOmniTMAE method. Finally, we discuss the main challenges of implementing TM structures on hardware and how the proposed design addresses these challenges.

2.1 The Tsetlin Machine and Embedding Architecture

The operation of the TM is primarily based on basic learning units TAs. In NLP applications, each automaton corresponds to a feature in the vocabulary. For example, given a vocabulary (x_1, x_2, x_3), each feature is assigned an automaton that captures its learned state through training. Automata are combined into logical expressions called clauses, which represent class-specific patterns. Multiple clauses may represent the same class, and clauses may also be shared across classes, as in the Coalesced Tsetlin Machine (CoTM) structure (Glimsdal & Granmo, 2021). Figure 1 summarizes this flow, from document accumulation and clause evaluation to update-probability calculation and automaton-state updates.

When feature negation is used, the clause space is extended from features to literals, e.g., $(x_1, x_2, x_3, \neg x_1, \neg x_2, \neg x_3)$. Each automaton operates in a discrete state space controlled by b state bits, giving 2^b possible states. In compact binary storage, each automaton requires b bits. Thus, for C clauses and L literals, the automata state memory is $M_{\text{TA}} = C \times L \times b$ bits. For example, when $b = 8$, each automaton has 256 possible states, ranging from 0 to 255. An intermediate threshold

N determines whether a literal is included in clause evaluation. Literals with states above N are included, while those below N are excluded. For a single clause C_j, the clause output c_j that is used for evaluation is therefore the logical AND of its included literals l:

$$c_j = \bigwedge_{l \in L,\ \text{state}(l) > N} l.$$

For example, a clause representing the target word "space" may include literals such as "vehicle", "vacuum", and "light" during evaluation, while still retaining states for all literals.

The first stage of clause training is evaluation, where logical operations such as XOR and AND compare the input example X with the clause structure. A supportive clause evaluation reinforces selected automata, while a non-supportive evaluation penalizes them. State updates are performed through increment or decrement operations. During training, each clause undergoes three main update types: (i) memorizing operation, which reinforces true literals in X when the clause agrees with the input and the output is 1, while false literals may undergo probabilistic forgetting controlled by the hyperparameter s; (ii) forgetting operation, which decreases the influence of true literals when the clause does not agree with the input, with probability governed by s; and (iii) invalidation operation, which reinforces false literals to modify the clause and reject incorrect patterns.

In addition to the automata memory state matrix, the CoTM structure includes a weights matrix. This matrix assigns integer weights to clauses, allowing them to be promoted or suppressed during voting.

The Omni TM-AE model is built on the CoTM structure combined with an autoencoder framework (Kadhim et al., 2025b). Unlike traditional autoencoders, Omni TM-AE is trained separately for each target token, enabling the reuse of generated embedding vectors. It also uses the full automaton state space, including excluded literals below the threshold N. This allows the model to extract embeddings directly from the clause states without requiring an additional embedding extraction stage, producing static embeddings similar to Word2Vec and GloVe while retaining interpretable vocabulary-aligned dimensions. A concrete interpretability trace is provided in Appendix A.2.

Despite its effectiveness, Omni TM-AE suffers from long training times, reaching several minutes per token on large datasets, which limits its practicality for large-scale applications. For each target token, the Omni TM-AE training procedure consists of the following stages:

1. Output preparation: Define a binary label ($output$), where 1 indicates the presence of the target token, and 0 indicates its absence from documents in the dataset.
2. Input preparation: Construct X by randomly accumulating a documents according to the target token label.
3. Evaluation: Evaluate each clause output c_j against the input X to produce a binary clause output o_j, indicating whether the included literals of clauses are satisfied.
4. Class sum calculation: Aggregate the weighted clause outputs, clip the result by the hyperparameter T, and compute $update_p$, which defines the probability of updating clauses:

$$class_sum = \text{clip}\left(\sum_{j=1}^{C} w_j o_j,\ -T,\ T\right), \qquad update_p = \begin{cases} \dfrac{T - class_sum}{2T}, & output = 1, \\ \dfrac{T + class_sum}{2T}, & output = 0. \end{cases}$$

   Here, j indexes each clause in C, and w_j is the weight of clause C_j.
5. Clause update: Update clauses according to the output and weights.

2.1.1 Class Sum Calculation Analysis

One of the key components of the TM training process is the $class_sum$ calculation, which is typically used to determine the voting outcome for the outputs. In the case of Omni TM-AE, where only two outputs are considered (0 and 1), the primary role of $class_sum$ is to compute the update rate $update_p$ and guide the convergence of the model.

Specifically, $class_sum$, and ultimately $update_p$, controls the proportion of clauses selected for updating. This mechanism plays an important role in driving the model toward a stable state, where clauses are sufficiently trained to represent the underlying patterns and no longer require significant updates. At this stage, training can be terminated.

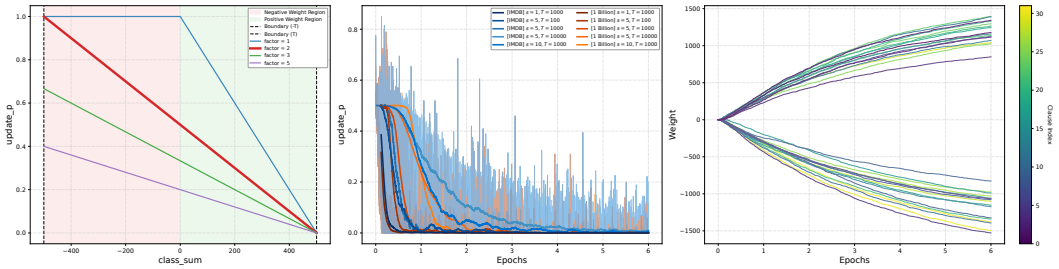

Figure 2: (a on the left) Relationship between the $class_sum$ and the update probability $update_p$. (b in the middle) Update probability $update_p$ over epochs for different datasets and hyperparameters. (c on the right) Clause weights variation over epochs.

Figure 2(a) illustrates the relationship between $class_sum$ and $update_p$. The vertical axis represents the $update_p$, while the horizontal axis corresponds to $class_sum$. The figure includes multiple curves reflecting different scaling factors in the update probability equation (e.g., division by 2 in the formulation above). In practice, a factor of 2 is commonly used, as it provides a balanced adjustment between negatively weighted clauses (red region) and positively weighted clauses (blue region).

Since $class_sum$ is bounded by T, the factor-2 curve shows a neutral update rate of approximately 50% when neither positive nor negative clauses dominate. Activation of many negative clauses increases the update rate above 50%, while activation of many positive clauses reduces it below 50%, indicating that the clauses are becoming stable and require fewer updates.

2.2 FastOmniTMAE

In this work, the Omni TM-AE model is redesigned to improve training speed while maintaining embedding quality. Two main limitations in the original training procedure were identified.

The first limitation is the $class_sum$ calculation. This step requires all clauses to be evaluated before any update can be performed, creating a global dependency in the training process. As discussed previously, this mechanism is intended to guide convergence (Zhang et al., 2026). While this may be suitable for small-scale applications, it becomes inefficient in NLP tasks with large vocabularies and many training documents.

Figure 2(b) illustrates this behavior on two datasets: IMDb Maas et al. (2011), with fewer than 2,000 tokens, and the One Billion Word dataset Chelba et al. (2014), with 40,000 tokens. The curves are generated using different values of $s \in \{1, 5, 10\}$ and $T \in \{100, 1000, 10000\}$, where each epoch contains 2,000 training examples. At the beginning of training, the update rate is approximately 50%, since no clauses have yet adapted to the input patterns. As training progresses, $class_sum$ increases and the update rate gradually decreases toward zero.

This convergence behavior does not necessarily imply sufficient training for high-quality embeddings. For example, IMDb provides limited contextual information, leading to weak embeddings regardless of convergence. In contrast, large-scale datasets provide richer contextual diversity, allowing stronger embeddings to be obtained within a small number of epochs. Thus, embedding quality in TM-based NLP models is more strongly influenced by data richness than by convergence behavior alone.

The second limitation concerns the weight matrix. In general CoTM formulations (Glimsdal & Granmo, 2021), weights support voting across multiple classes. However, in Omni TM-AE, where only binary outputs are used, weights mainly influence $class_sum$ and therefore act as a convergence-scaling mechanism. As shown in Figure 2(c), weights follow a monotonic pattern during training, with positive and negative weights preserving their signs. This indicates that their main role is to reduce the update rate rather than alter the learned decision structure.

Based on these observations, explicit $class_sum$ calculation is not required for embedding quality. Instead, training for a sufficient number of epochs is adequate. Removing $class_sum$ eliminates global synchronization across clauses and reduces evaluation waiting time.

To address this bottleneck, we redesign the training process by shifting from global to local decision-making. Clause evaluation and updating are decoupled and performed independently, with the update probability computed locally from each clause's evaluation outcome. This enables parallel execution of evaluation and update operations and forms the foundation of the FastOmniTMAE model. The classification and similarity results in the Single-Run Multi-Environment Benchmark show that removing the global $class_sum$ dependency does not degrade embedding quality compared with Omni TM-AE, but primarily removes a training bottleneck. A detailed description of the FastOmniTMAE training procedure is provided in Appendix A.3.

2.2.1 CONVERGENCE INTERPRETATION

Through extensive experimentation and the developed benchmarks used to rebuild the model while maintaining high embedding quality, it became necessary to identify reliable indicators of convergence and determine when further training is no longer required. A potential approach is to analyze the distribution of literals within the automaton state space.

As discussed in Kadhim et al. (2025b; 2024), as the vocabulary size and training data increase, obtaining a clear distribution becomes more difficult. With a small number of epochs, the model tends to push irrelevant literals toward lower states, while important literals involved in clause formation are elevated toward the upper threshold (N). This process typically relies on feature negation to create a distribution within the negated literal space, which is then used in embedding computation, as in the Omni TM-AE model.

Although this distribution may appear to indicate sufficient training, it does not necessarily correlate with embedding quality. At present, there is no established relationship between the state distribution and embedding performance. Identifying such a relationship could enable early stopping criteria and the formulation of a convergence measure tailored to TM-based embedding models. The observed state distributions are provided in Appendix A.4.

2.2.2 TM ON HARDWARE

The proposed FastOmniTMAE approach demonstrates efficient performance on CPUs due to its parallel structure. An implementation was also developed using Compute Unified Device Architecture (CUDA) for GPU execution. However, the observed performance was limited compared to deep learning models, even on high-end systems such as the NVIDIA DGX H100.

Modern GPUs, such as those based on the Hopper H100 architecture, are designed to accelerate floating-point computations. Each GPU consists of multiple Streaming Multiprocessors (SMs), each containing CUDA cores, Tensor cores, and additional specialized units. These components are optimized for operations such as fused multiply-add (FMA), which are fundamental to neural network computations (NVIDIA, 2022).

In contrast, the TM primarily relies on simple logical operations, including AND, OR, NOT, $Shift$ operations, and XOR. While these operations are computationally lightweight, they are not efficiently supported by GPU architectures optimized for dense floating-point arithmetic. A significant portion of GPU hardware resources is dedicated to FMA-based computation, while only a fraction is allocated to integer-based operations. Consequently, much of the GPU architecture remains underutilized when executing TM-based workloads.

Additionally, specialized units such as Tensor Cores and Ray Tracing Cores are not applicable to TM operations, leading to further inefficiencies. These observations highlight the mismatch between TM computational requirements and GPU design, motivating the need for dedicated hardware solutions to efficiently accelerate TM training. Detailed CUDA profiling results are provided in Appendix A.5.

2.2.3 THE NEW HARDWARE DESIGN

The proposed design shifts the computational burden of FastOmniTMAE training from general-purpose processors to a dedicated accelerator on FPGA hardware. This approach addresses the limitations of sequential execution and leverages the logical nature of the TM architecture.

The accelerator is implemented as a reusable Intellectual Property (IP) core, with variants targeting both resource-constrained System on Chip (SoC) platforms, such as Zynq-7000, and higher-

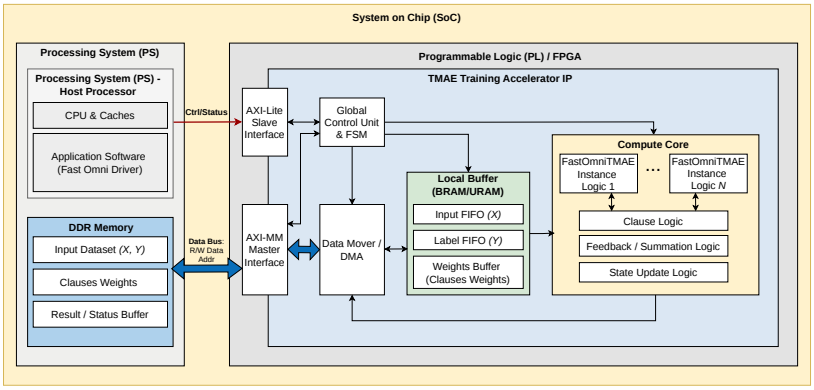

Figure 3: SoC-based architecture of the FastOmniTMAE accelerator with PS–PL communication over AXI interfaces for efficient parallel training.

performance SoC-FPGA platforms, such as Zynq UltraScale+ MPSoC. The design adopts a batch-processing strategy that enables autonomous and continuous training while minimizing dependence on the host processor.

Figure3 illustrates the system architecture. The SoC platform consists of a Processing System (PS) and a Programmable Logic (PL) component (FPGA). Communication between these components is via standard Advanced eXtensible Interface (AXI) protocols. Two primary interfaces are used:

- AXI-Lite: A control interface that allows the processor to configure training parameters, including memory addresses, batch size, and control signals such as start and reset.
- AXI-MM (Memory-Mapped): A high-throughput interface that enables direct memory access for reading training data and updating automata states stored in external DDR memory, minimizing processor involvement and reducing latency.

The design adopts a two-stage parallel training mechanism (evaluation and update), ensuring both efficiency and correctness. Training data is streamed from external memory into on-chip buffers (BRAM) to reduce communication overhead.

The accelerator is controlled by a finite state machine (FSM) that manages the training workflow:

1. Data Fetching: Loading input features X and labels Y in batches.
2. Logical Computation: Executing TM operations directly using logic gates, resulting in lower power consumption compared to floating-point operations on GPUs.
3. State Update: Updating automata states and writing results back to external memory for subsequent iterations.

The design is scalable and adaptable to different hardware constraints. A Zybo board implementation targets resource-constrained SoC platforms (Zynq-7000), prioritizing minimal logic utilization and efficient memory access, while a ZCU104 board implementation exploits the higher bandwidth and larger programmable-logic resources of the UltraScale+ MPSoC to support greater parallelism and larger-scale training. Additional implementation details are provided in Appendix A.6.

3 Evaluation

3.1 Single-Run Multi-Environment Benchmark

The purpose of the first benchmark was to validate the development stages of the proposed FastOmniTMAE model through continuous comparison with the original Omni TM-AE model. The experiments were conducted on an NVIDIA DGX H100 server with 8 GPUs and 2.0 TiB of RAM. DevContainer was used to create two isolated environments for FastOmniTMAE and Omni TM-AE

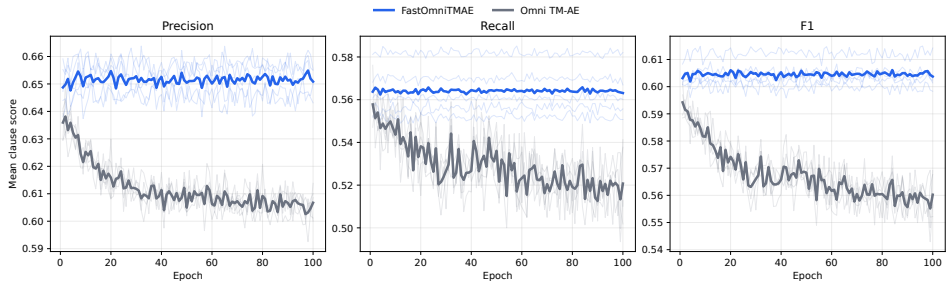

Figure 4: Classification performance of FastOmniTMAE and Omni TM-AE over 100 epochs using precision, recall, and F1-score.

Table 1: Word-similarity benchmark results for embeddings. Entries report the mean and standard deviation over five runs; ρ denotes Spearman correlation and τ denotes Kendall correlation.

Model	**Metric**	**RG-65**	**WS-353-Sim**	**MTurk-287**	**MTurk-771**	**MEN**	**Avg.**
GloVe	ρ	0.528 $\pm$ 0.013	0.514 $\pm$ 0.005	0.521 $\pm$ 0.004	0.461 $\pm$ 0.001	0.564 $\pm$ 0.001	0.518 $\pm$ 0.004
	τ	0.365 $\pm$ 0.010	0.355 $\pm$ 0.004	0.365 $\pm$ 0.004	0.319 $\pm$ 0.000	0.394 $\pm$ 0.001	0.360 $\pm$ 0.003
Word2Vec	ρ	0.529 $\pm$ 0.006	0.591 $\pm$ 0.002	0.558 $\pm$ 0.004	0.471 $\pm$ 0.002	0.561 $\pm$ 0.002	0.542 $\pm$ 0.001
	τ	0.357 $\pm$ 0.007	0.415 $\pm$ 0.002	0.389 $\pm$ 0.003	0.324 $\pm$ 0.002	0.391 $\pm$ 0.002	0.375 $\pm$ 0.001
FastText	ρ	0.508 $\pm$ 0.007	0.601 $\pm$ 0.003	0.574 $\pm$ 0.004	0.486 $\pm$ 0.003	0.580 $\pm$ 0.002	0.550 $\pm$ 0.001
	τ	0.348 $\pm$ 0.007	0.420 $\pm$ 0.002	0.399 $\pm$ 0.004	0.334 $\pm$ 0.002	0.406 $\pm$ 0.001	0.382 $\pm$ 0.002
Omni TM-AE	ρ	0.633 $\pm$ 0.020	0.478 $\pm$ 0.013	0.519 $\pm$ 0.006	0.487 $\pm$ 0.004	0.596 $\pm$ 0.002	0.543 $\pm$ 0.005
	τ	0.452 $\pm$ 0.016	0.344 $\pm$ 0.012	0.367 $\pm$ 0.005	0.350 $\pm$ 0.003	0.439 $\pm$ 0.002	0.390 $\pm$ 0.004
FastOmniTMAE (CPU)	ρ	0.656 $\pm$ 0.016	0.470 $\pm$ 0.007	0.498 $\pm$ 0.009	0.477 $\pm$ 0.003	0.586 $\pm$ 0.002	0.537 $\pm$ 0.003
	τ	0.468 $\pm$ 0.014	0.343 $\pm$ 0.006	0.356 $\pm$ 0.009	0.350 $\pm$ 0.001	0.441 $\pm$ 0.001	0.392 $\pm$ 0.003
FastOmniTMAE (CUDA)	ρ	0.649 $\pm$ 0.020	0.469 $\pm$ 0.014	0.492 $\pm$ 0.006	0.472 $\pm$ 0.006	0.576 $\pm$ 0.002	0.532 $\pm$ 0.008
	τ	0.465 $\pm$ 0.015	0.347 $\pm$ 0.009	0.352 $\pm$ 0.005	0.347 $\pm$ 0.004	0.434 $\pm$ 0.002	0.389 $\pm$ 0.006

to ensure a fair comparison. The benchmark included three tests: classification, similarity, and clustering. The additional baseline models used in the evaluation were executed in separate environments. The benchmark code will be released publicly after double-blind review.

3.1.1 Classification Performance

For the classification task, 100 tokens were selected. Training was performed on a One Billion Words dataset using 1,000 examples per token, followed by 500 examples for inference. The dataset vocabulary contained 40k tokens, and each token's documents were labeled as True or False. True indicates that the document supports the target token, while False indicates that the document does not contain the target token. Figure 4 shows the precision, recall, and F1-score for FastOmniTMAE and Omni TM-AE. Each experiment was repeated 5 times, with 100 epochs per run. The 100 target tokens are a configurable benchmark subset and can be expanded to larger or full-vocabulary runs.

In Figure 4, the dark curve represents the mean over paired runs, while the lighter curves in the background represent individual runs. Both models were trained using identical settings: $clauses = 160$, $T = 8000$, $s = 2$, and $seed = 42$. FastOmniTMAE demonstrated stable performance throughout training, while Omni TM-AE showed a gradual decline as training progressed. FastOmniTMAE consistently achieved F1-scores above 0.6 across epochs, whereas Omni TM-AE started below 0.6 and decreased to 0.56 in the final epochs. This decline is attributed to the use of $class_sum$ to reduce the clause update rate, which limits the model's ability to adapt to previously unseen patterns. As discussed earlier, this behavior does not provide a reliable convergence measure and may prevent continued learning when further updates are needed. In addition to its higher classification performance, FastOmniTMAE achieved more than a **5×** speedup. Omni TM-AE required 1709 seconds to train for 100 epochs, while FastOmniTMAE completed the same training in 310 seconds. This demonstrates the effectiveness of the proposed model structure in improving both training speed and accuracy.

3.2 Semantic Similarity Evaluation

In the second test, we used the similarity datasets RG65, MTurk287, MTurk771, WordSim353, and MEN. These datasets contain human-annotated word pairs, where each pair is assigned a similarity score. Table 1 compares FastOmniTMAE with other models included in the benchmark, namely Word2Vec, FastText, GloVe, and Omni TM-AE. For Omni TM-AE and FastOmniTMAE, the same hyperparameters were used: $number_of_examples = 2000$, $accumulation = 24$, $clauses = 32$, $T = 20000$, $s = 1.0$, $epochs = 4$, and $number_of_state_bits = 8$. The remaining models were trained using $vector_size = 100$, $window = 5$, and $epochs = 25$. All models were trained on the CPU using the same DGX H100 server, except that FastOmniTMAE was also trained on GPU to evaluate the suitability of GPU acceleration for TM-based training. A supplementary comparison with lightweight pretrained Transformers, TinyBERT and DistilBERT, including their word-similarity performance and architectural trade-offs, is provided in Appendix A.7.

From Table 1, FastText achieved the highest average Spearman correlation ($\rho = 0.550$), followed closely by FastOmniTMAE on CPU ($\rho = 0.537$). In terms of Kendall correlation, FastOmniTMAE achieved the highest average score ($\tau = 0.392$), slightly outperforming Omni TM-AE ($\tau = 0.390$) and FastText ($\tau = 0.382$). FastOmniTMAE also achieved the highest individual score on the RG65 dataset for both Spearman and Kendall correlations, representing the best single-dataset similarity result in the experiment. Overall, FastOmniTMAE achieved comparable embedding quality while improving training efficiency over Omni TM-AE on CPU under identical resource allocation. Both models were trained using approximately 80% of the DGX H100 CPU resources, corresponding to around 179 cores. FastOmniTMAE required an average of 242.9 seconds, while Omni TM-AE required 293.1 seconds, giving FastOmniTMAE a **1.2**× speedup on the same hardware. The training times of the remaining models are not directly compared, as they were executed under different resource conditions. For GPU execution, FastOmniTMAE showed longer training time due to the mismatch between GPU architecture and TM-based logical operations. This issue is discussed further in the Multi-Hardware Benchmark section.

3.3 Clustering Analysis

The previous tests provide structured numerical evaluation. The final test in this benchmark provides a visual representation of embedding quality. In this test, t-SNE was used to cluster a set of words based on their embedding vectors. The selected words were pre-divided into distinguishable semantic groups, such as food, geography, and vehicles. The embeddings used for this visualization were generated with 32 clauses, $T = 20000$, $s = 1.0$, accumulation $= 24$, 4 epochs, and 2000 training examples per token. Figure 5 shows the resulting clusters, demonstrating the ability of FastOmniTMAE to generate meaningful embedding vectors. Words belonging to the same semantic group are generally clustered together, indicating that the model captures useful semantic relationships.

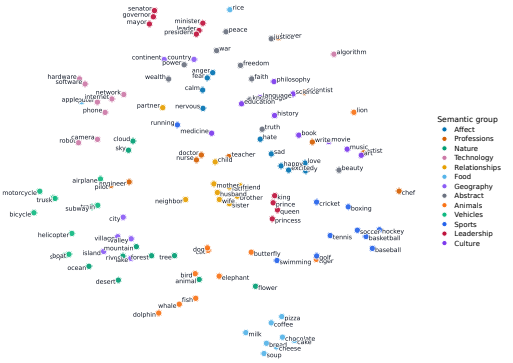

Figure 5: t-SNE visualization of FastOmniTMAE embeddings for pre-defined semantic word groups.

3.4 Multi-Hardware Benchmark

This benchmark compares the performance of the same FastOmniTMAE model across different hardware architectures. FastOmniTMAE was implemented using multiple technologies to target CPUs, GPUs, and SoC-FPGA platforms. Accordingly, four configurations were developed: a C implementation for CPU execution, an HDL implementation for FPGA acceleration, a C/OpenCL implementation for Intel GPU execution, and a CUDA implementation for NVIDIA GPU execution. These configurations were tested on five devices: a mid-range Intel Core i5 PC, an Intel Core i9 Ultra workstation, an NVIDIA DGX H100 server, a Xilinx ZCU104 board, and a Digilent Zybo board.

Table 2: FastOmniTMAE hardware benchmark on the RG65 44-token workload (352k examples total). Each row reports the mean with standard deviation over 5 runs.

Platform	Configuration	Backend	Time (s) ↓	Speedup vs. local CPU ↑	RG65 ρ ↑
ZCU104 Board	ARM CPU (1 of 4 cores)	CPU	848.7 $\pm$ 1.5	1.00×	0.680
ZCU104 Board	FPGA design (12 engines)	SoC	119.9 $\pm$ 0.9	7.08×	0.696 $\pm$ 0.005
Zybo Board	ARM CPU (1 of 2 cores)	CPU	3667.5 $\pm$ 21.8	1.00×	0.680
Zybo Board	FPGA design (2 engines)	SoC	657.4 $\pm$ 1.1	5.58×	0.669 $\pm$ 0.008
Core i5 PC	Host CPU (1 of 8 threads)	CPU	137.3 $\pm$ 37.1	1.00×	0.680
Core i5 PC	Intel Iris Xe Graphics	OpenCL	475.6 $\pm$ 7.2	0.29×	0.668
Core i9 PC	Host CPU (1 of 24 threads)	CPU	57.7 $\pm$ 1.9	1.00×	0.680
Core i9 PC	Intel Graphics	OpenCL	262.2 $\pm$ 0.4	0.22×	0.668
H100 Server	Host CPU (1 of 224 threads)	CPU	87.3 $\pm$ 0.8	1.00×	0.680
H100 Server	NVIDIA H100 80GB HBM3	CUDA	493.1 $\pm$ 0.5	0.18×	0.681

To ensure a fair comparison, the benchmark was limited to training only, excluding example preparation. Preparing examples requires a corpus larger than 4 GB, which is unsuitable for constrained hardware such as the Zybo board with 1 GB of RAM. Therefore, examples for each token were prepared as small files containing 8,000 examples per token. Training consisted of two stages: clause evaluation and clause update. The examples were extracted from a One Billion Word dataset with a vocabulary of 40,000 entries. The RG65 word similarity dataset, which contains 44 word pairs, was selected as the evaluation task. This dataset was chosen based on previous benchmark observations, as it provides sufficient sensitivity to evaluate embedding quality. The hyperparameters were set as follows: 32 clauses, $T = 20000$, and $s = 1.0$.

Table 2 presents the training results across the different hardware configurations. The fastest training time was achieved using a single-threaded implementation on the Intel Core i9 Ultra processor operating at 3.7 GHz. However, on the same machine, execution on the integrated GPU reduced performance by a factor of 0.22. The DGX H100 server, with CPU frequencies ranging from 2.0 to 2.9 GHz during training[1], also achieved strong CPU performance. In contrast, CUDA execution on the DGX H100 did not provide the expected acceleration, despite full GPU utilization. As discussed earlier, this is attributed to the mismatch between GPU architectures and the logical operation structure of the TM. The Core i5 PC achieved moderate CPU performance and acceptable GPU performance, with better GPU training time than the DGX H100 CUDA implementation.

Apart from CPU execution, the best training performance was achieved on the ZCU104 SoC-FPGA board. This platform provided sufficient resources to instantiate 12 FastOmniTMAE cores, allowing clauses to be trained in parallel. In addition to efficient execution time, this configuration achieved a high similarity score of **0.696**, making it the best-performing hardware configuration in terms of embedding quality. The design was also executable on the more resource-constrained Zybo board, showing the scalability of the proposed architecture. Despite its limited resources, the Zybo implementation achieved a competitive similarity score with a smaller hardware footprint. Further details on hardware resource utilization for ZCU104 and Zybo are provided in Appendix A.6.

4 Conclusion

This work presented FastOmniTMAE, a parallel reformulation of Omni TM-AE that removes global training dependencies and improves training efficiency while preserving comparable embedding quality. Across classification, similarity, and clustering evaluations, FastOmniTMAE demonstrated stable performance and faster training than Omni TM-AE. The SoC-FPGA implementation further showed that TM-based embedding can benefit from dedicated logic-based hardware, achieving efficient training with a small hardware footprint.

[1]Although the processor can reach up to 3.9 GHz under specific conditions, this mode was not used in our experiments.

AI USE STATEMENT

Generative AI tools were used to assist with grammar, clarity, language polishing, and manuscript formatting. They were not used to generate research ideas, methodology, experimental results, or scientific conclusions. All AI-assisted content was reviewed and verified by the authors, who take full responsibility for the manuscript.

REPRODUCIBILITY STATEMENT

The experimental configurations, implementation details, and additional evaluation results are provided in the appendix. An anonymized source-code package is included as supplementary material to facilitate reproducibility. The source code will be made publicly available after the double-blind review period.

REFERENCES

Kuruge Darshana Abeyrathna, Bimal Bhattarai, Morten Goodwin, Saeed Rahimi Gorji, Ole-Christoffer Granmo, Lei Jiao, Rupsa Saha, and Rohan K. Yadav. Massively parallel and asynchronous Tsetlin machine architecture supporting almost constant-time scaling. In Marina Meila and Tong Zhang (eds.), *ICML*, volume 139 of *Proceedings of Machine Learning Research*, pp. 10–20. PMLR, 2021.

Kuruge Darshana Abeyrathna, Ahmed A. O. Abouzeid, Bimal Bhattarai, Charul Giri, Sondre Glimsdal, Ole-Christoffer Granmo, Lei Jiao, Rupsa Saha, Jivitesh Sharma, Svein A. Tunheim, and Xuan Zhang. Building concise logical patterns by constraining Tsetlin machine clause size. In Edith Elkind (ed.), *IJCAI*, pp. 3395–3403, 8 2023.

Abu Bakar, Tousif Rahman, Alessandro Montanari, Jie Lei, Rishad Shafik, and Fahim Kawsar. Logic-based intelligence for batteryless sensors. In *Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications*, pp. 22–28. Association for Computing Machinery, 2022.

Abu Bakar, Tousif Rahman, Rishad Shafik, Fahim Kawsar, and Alessandro Montanari. Adaptive intelligence for batteryless sensors using software-accelerated Tsetlin machines. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, pp. 236–249. Association for Computing Machinery, 2023.

Bimal Bhattarai, Ole-Christoffer Granmo, Lei Jiao, Rohan Yadav, and Jivitesh Sharma. Tsetlin machine embedding: Representing words using logical expressions. *Findings of EACL*, pp. 1512–1522, 2024.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling, 2014. URL https://arxiv.org/abs/1312.3005.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Sondre Glimsdal and Ole-Christoffer Granmo. Coalesced multi-output Tsetlin machines with clause sharing. *CoRR*, 2021. URL https://arxiv.org/abs/2108.07594.

Ole-Christoffer Granmo. The Tsetlin machine - a game theoretic bandit driven approach to optimal pattern recognition with propositional logic, 2018. URL https://arxiv.org/abs/1804.01508.

Sindhusha Jeeru, Lei Jiao, Per-Arne Andersen, and Ole-Christoffer Granmo. Interpretable rule-based architecture for GNSS jamming signal classification. *IEEE Sensors Journal*, 2025.

Ahmed K. Kadhim, Ole-Christoffer Granmo, Lei Jiao, and Rishad Shafik. Exploring state space and reasoning by elimination in Tsetlin machines. In *2024 International Symposium on the Tsetlin Machine (ISTM)*, pp. 1–8. IEEE Computer Society, 2024.

Ahmed K. Kadhim, Lei Jiao, Rishad Shafik, and Ole-Christoffer Granmo. Adversarial attacks on AI-generated text detection models: A token probability-based approach using embeddings, 2025a. URL https://arxiv.org/abs/2501.18998.

Ahmed K. Kadhim, Lei Jiao, Rishad Shafik, and Ole-Christoffer Granmo. Omni TM-AE: A scalable and interpretable embedding model using the full Tsetlin machine state space, 2025b. URL https://arxiv.org/abs/2505.16386.

Ahmed K. Kadhim, Lei Jiao, Rishad Shafik, Ole-Christoffer Granmo, and Bimal Bhattarai. Scalable multi-phase word embedding using conjunctive propositional clauses. In *2025 International Symposium on the Tsetlin Machine (ISTM)*, pp. 107–115, 2025c.

C Kishore, Santhosh Sivasubramani, Rishad Shafik, and Amit Acharyya. Nano-magnetic logic based architecture for edge inference using Tsetlin machine. In *2023 21st IEEE Interregional NEWCAS Conference (NEWCAS)*, pp. 1–5, 2023.

Jie Lei, Tousif Rahman, Rishad Shafik, Adrian Wheeldon, Alex Yakovlev, Ole-Christoffer Granmo, Fahim Kawsar, and Akhil Mathur. Low-power audio keyword spotting using Tsetlin machines. *Journal of Low Power Electronics and Applications*, 11(2), 2021.

Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pp. 142–150. Association for Computational Linguistics, 2011.

Sidharth Maheshwari, Tousif Rahman, Rishad Shafik, Alex Yakovlev, Ashur Rafiev, Lei Jiao, and Ole-Christoffer Granmo. REDRESS: Generating compressed models for edge inference using Tsetlin machines. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(9):11152–11168, 2023.

Gang Mao, Alex Yakovlev, Fei Xia, Tian Lan, Shengqi Yu, and Rishad Shafik. Automated synthesis of asynchronous Tsetlin machines on FPGA. In *2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 1–4, 2022.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. URL https://arxiv.org/abs/1301.3781.

Jordan Morris, Ashur Rafiev, Fei Xia, Rishad Shafik, Alex Yakovlev, and Andrew Brown. An alternate feedback mechanism for Tsetlin machines on parallel architectures. In *2022 International Symposium on the Tsetlin Machine (ISTM)*, pp. 53–56, 2022.

NVIDIA. NVIDIA H100 Tensor Core GPU Architecture. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, 2022. Accessed: 2026-05-04.

Rebekka Olsson Omslandseter, Lei Jiao, Xuan Zhang, Anis Yazidi, and B. John Oommen. The hierarchical discrete pursuit learning automaton: A novel scheme with fast convergence and epsilon-optimality. *IEEE Transactions on Neural Networks and Learning Systems*, pp. 8278–8292, 2024.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, pp. 1532–1543, 2014.

Samuel Prescott, Adrian Wheeldon, Rishad Shafik, Tousif Rahman, Alex Yakovlev, and Ole-Christoffer Granmo. An FPGA architecture for online learning using the Tsetlin machine, 2023. URL https://arxiv.org/abs/2306.01027.

Tousif Rahman, Adrian Wheeldon, Rishad Shafik, Alex Yakovlev, Jie Lei, Ole-Christoffer Granmo, and Shidhartha Das. Data booleanization for energy efficient on-chip learning using logic driven AI. In *2022 International Symposium on the Tsetlin Machine (ISTM)*, pp. 29–36, 2022.

Rupsa Saha, Ole-Christoffer Granmo, and Morten Goodwin. Using Tsetlin machine to discover interpretable rules in natural language processing applications. *Expert Systems*, 40(4):e12873, 2023.

Rishad Shafik, Tousif Rahman, Adrian Wheeldon, Ole-Christoffer Granmo, and Alex Yakovlev. Energy-frugal and interpretable AI hardware design using learning automata, 2023. URL https://arxiv.org/abs/2305.11928.

Svein Anders Tunheim, Lei Jiao, Rishad Shafik, Alex Yakovlev, and Ole-Christoffer Granmo. A convolutional Tsetlin machine-based field programmable gate array accelerator for image classification. In *2022 International Symposium on the Tsetlin Machine (ISTM)*, pp. 21–28, 2022.

Svein Anders Tunheim, Lei Jiao, Rishad Shafik, Alex Yakovlev, and Ole-Christoffer Granmo. Tsetlin machine-based image classification FPGA accelerator with on-device training. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 72(2):830–843, 2025.

Svein Anders Tunheim, Yujin Zheng, Lei Jiao, Rishad Shafik, Alex Yakovlev, and Ole Christoffer Granmo. An all-digital 8.6-nJ/Frame 65-nm Tsetlin machine image classification accelerator. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 73(2):1107–1120, 2026.

Adrian Wheeldon, Rishad Shafik, Tousif Rahman, Jie Lei, Alex Yakovlev, and Ole-Christoffer Granmo. Learning automata based energy-efficient AI hardware design for IoT applications. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378(2182):20190593, 09 2020.

Xuan Zhang, Lei Jiao, and Ole-Christoffer Granmo. Convergence analysis of Tsetlin machines under noise-free and noisy training conditions: From 2 bits to k bits. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=feOrSQdD9Y.



Figure 6: Example interfaces of the Tsetlin State-Space Explorer (TSSE). The left panel shows clause-level literal states for a selected target word, highlighting included and excluded literals in the learned clause structure. The right panel shows training-history tracing, linking literal-state transitions to feedback events and the supporting documents used to construct the synthesized input.

A Technical Appendices and Supplementary Material

A.1 Related Work

Most hardware-oriented TM research has focused on classification tasks, particularly in low-power and edge-computing settings. Early implementations demonstrated the suitability of TMs for constrained platforms such as microcontrollers and batteryless sensing systems (Lei et al., 2021; Bakar et al., 2023; 2022). Other studies explored TM execution on FPGAs, asynchronous hardware, and mixed software–hardware systems, showing that the logical structure of TMs can be mapped efficiently to diverse hardware targets (Wheeldon et al., 2020; Mao et al., 2022; Morris et al., 2022; Shafik et al., 2023; Prescott et al., 2023).

More recent work has investigated specialized TM accelerators for image classification. In particular, convolutional and coalesced variants have been implemented on FPGAs and custom digital hardware, including designs with on-device training and highly energy-efficient inference (Tunheim et al., 2022; 2025; 2026). These works demonstrate the hardware efficiency of TMs, but they primarily target image classification and do not address embedding generation or NLP representation learning.

In NLP, previous TM work has mostly focused on classification and interpretability, while embedding-oriented models remain comparatively limited. Omni TM-AE introduced a static embedding model based on the CoTM structure, where clauses are trained per token and the full automaton state space is used to construct embeddings (Glimsdal & Granmo, 2021; Kadhim et al., 2025b). Unlike conventional static embedding models such as Word2Vec and GloVe (Mikolov et al., 2013; Pennington et al., 2014), Omni TM-AE derives representations directly from clause-state distributions. However, prior TM-AE work has mainly emphasized embedding quality rather than training efficiency or hardware execution.

This work differs from previous hardware and embedding studies by targeting full-scale TM-based embedding training for NLP. Instead of accelerating inference for classification, we redesign the Omni TM-AE training procedure and evaluate FastOmniTMAE across CPUs, GPUs, and SoC-FPGA platforms.

A.2 Interpretability Trace Example

FastOmniTMAE retains the vocabulary-aligned representation of Omni TM-AE: for a vocabulary of 40,000 words, each target word is represented by a 40,000-dimensional integer vector, where each coordinate corresponds directly to a named vocabulary word and is derived from the states of its original and negated literals across the clauses. Thus, individual embedding dimensions can be traced back to the logical structures and learning events from which they were constructed.

To illustrate this property, we use the Tsetlin State-Space Explorer (TSSE) introduced with Omni TM-AE. Figure 6 shows the two main TSSE views used in this analysis: clause-level state inspection and training-history tracing. The tool records clause states and training events, allowing individual literals to be followed through the learning process. For the target word *driving*, for example, high-

state literals in a clause can correspond to semantically related vocabulary words such as *vehicle* and *license*. Because the clause is represented explicitly by its literal states, its learned structure can be inspected directly.

The trace can then be followed at the level of an individual literal. For example, one recorded update for the literal *vehicle* is:

Table 3: Example of a recorded state transition during training of the target word *driving*.

Example	Clause	Literal	State	Feedback	Target
2015	0	vehicle	$112 \rightarrow 113$	Type I	True

For each such transition, the diagnostic trace also records the documents used to construct the corresponding synthesized input X. In this example, the supporting documents include #8782, #16490, #17156, #5014, #17022, #11525, #11382, #5998, #15508, and #24240. These documents can be reconstructed as vocabulary tokens, providing a path from the final embedding coordinate to the training evidence that influenced it.

A.3 FastOmniTMAE Training Procedure

Algorithm 1 summarizes the training procedure used in FastOmniTMAE. The key difference from the original Omni TM-AE training procedure is that clause evaluation and clause update are performed locally at the clause level. Instead of computing a global *class_sum* over all clauses before updating, each clause uses its own weighted contribution to determine its update probability. This removes the global synchronization step and allows clauses to be evaluated and updated independently.

For each target token, a binary autoencoder example is generated by sampling whether the target token is present or absent. The input vector X is constructed by accumulating documents according to this target label. Each clause C_j is then evaluated against X, producing a binary output o_j. The local clause contribution $r_j = w_j o_j$ is used to compute the update probability. If selected for update, the clause receives one of the standard feedback types: Type Ia reinforcement, Type Ib penalization, or Type II false-positive suppression. This procedure preserves the original TM feedback mechanism while enabling parallel clause-level training Granmo (2018).

A.4 State-Space Distribution of Literals

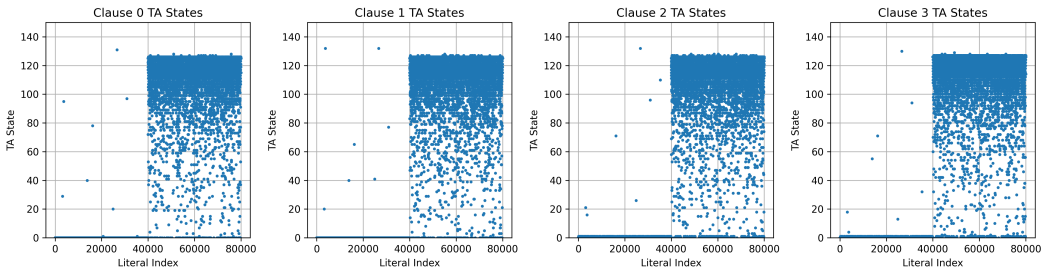

Figure 7: Distribution of TA states across clauses after training. The first half of the literal index range corresponds to original vocabulary features, while the second half corresponds to negated literals. The distribution shows that original literals are mostly pushed toward low states, whereas negated literals form a richer high-state distribution, indicating that absence-based information is important for Omni-style embedding construction.

Figure 7 visualizes the distribution of automaton states after training FastOmniTMAE. Each subplot corresponds to one clause, where the horizontal axis represents the literal index and the vertical axis represents the learned TA state. Since the vocabulary contains 40,000 original features, the full literal space contains 80,000 literals after including negations. Thus, the first half of the horizontal axis corresponds to original literals, while the second half corresponds to their negated counterparts.

Algorithm 1 FastOmniTMAE training procedure

Require: Training data $\mathcal{D}$, target tokens $\mathcal{Y}$, clauses $\mathcal{C}$, weights W, threshold T, specificity s, accumulation a

for each training iteration **do**
  for each target token $y \in \mathcal{Y}$ **do**
    Sample binary target label $Y \in \{0, 1\}$
    Construct input X by accumulating a documents according to Y
    for each clause $C_j \in \mathcal{C}$ **do**
      Evaluate clause output:

$$o_j \leftarrow \textsc{EvaluateClause}(X, C_j)$$

      Compute local clause contribution:

$$r_j \leftarrow \mathrm{clip}(w_j o_j, -T, T)$$

      Compute update probability:

$$p_j \leftarrow \begin{cases} \dfrac{T - r_j}{2T}, & Y = 1, \\ \dfrac{T + r_j}{2T}, & Y = 0. \end{cases}$$

      if $\textsc{SampleUpdate}(p_j)$ and $w_j \geq 0$ **then**
        if $Y = 1$ **then**
          if $o_j = 1$ **then**
            Apply Type Ia feedback:
              reinforce literals active in X
              penalize inactive literals
          else
            Apply Type Ib feedback:
              penalize clause states
          end if
        else
          if $o_j = 1$ **then**
            Apply Type II feedback:
              reinforce literals absent from X to suppress false positives
          end if
        end if
      end if
    end for
  end for
end for

The visualization shows that the model rapidly separates original and negated literals in the state space. Most original literals are pushed toward low states, indicating that they are not selected for clause evaluation. In contrast, the negated literals form a dense distribution across higher states, with many automata approaching or exceeding the inclusion threshold. This behavior suggests that the model does not rely only on the direct presence of words, but also learns from their absence. In other words, negated literals provide a form of reasoning by elimination, allowing clauses to characterize a target token not only by what supports it, but also by what should be excluded.

This observation is important for embedding construction. In conventional TM usage, only literals whose automata exceed the inclusion threshold contribute directly to the clause expression. However, in Omni TM-AE and FastOmniTMAE, the full automaton state space is informative. Even excluded literals retain trained state values, and these values reflect how each feature interacts with the target token during training. Therefore, both included and excluded literals contribute to the final embedding representation through their signed state values.

The strong distribution in the negated literal region also explains why feature negation is important in the Omni embedding formulation. If only original literals were used, much of the learned information would be discarded, especially in large-vocabulary settings where many original features are suppressed. By combining the state values of original literals with the corresponding negated literals, the embedding vector captures a more complete representation of the target token.

However, the distribution should not be interpreted as a direct measure of embedding quality. Although a clear separation between original and negated literals indicates that the model has learned structured clause states, there is currently no established mapping between a specific state-distribution pattern and downstream embedding performance. This supports the observation made in the main text: state-space structure can provide useful insight into training dynamics, but it cannot yet serve as a reliable convergence criterion or early-stopping rule.

A.5 CUDA Profiling and Tensor-Mask Analysis

We conducted a standalone profiling experiment on a single NVIDIA H100 80 GB GPU using five randomly sampled target words and the same configuration as the semantic similarity evaluation: 32 clauses, 40,000 features, eight state bits, four epochs, and 2,000 examples per epoch.

Table 4 presents the execution-time breakdown of the CUDA implementation. The fused training kernel accounts for 99.09% of the total execution time, indicating that the primary bottleneck lies in the training computation rather than data transfers or initialization.

Table 4: CUDA workflow profiling on a single NVIDIA H100 80 GB GPU.

Workflow phase	**Time (ms)**	**Share (%)**
Setup and H2D transfer	459.464	0.9025
State initialization	0.063	0.0001
Fused training kernel	50,447.880	99.0897
Embedding initialization and collection	0.866	0.0017
D2H transfer and finalization	1.342	0.0026
Teardown	1.729	0.0034

Table 5: Cycle distribution and mask characterization of the fused CUDA training kernel.

Kernel stage	**Cycles (%)**	**Mask characterization**
Example construction	1.43	Partially maskable
Clause evaluation	29.82	Fully mask-expressible
Feedback decision	37.25	Irregular stochastic control
Include-action scan	2.72	Fully mask-expressible
Automata-state update	28.79	Fully mask-expressible and already bit-sliced

To further investigate the training bottleneck, we instrumented the fused kernel to estimate the execution-cycle distribution across its logical stages (see Table 5). We also classified each stage according to whether its operations can be expressed using bitwise masks.

The results indicate that approximately 61.32% of the kernel's execution cycles are spent in fully mask-expressible operations, while 37.25% involve irregular stochastic control and 1.43% are partially maskable. Despite the high proportion of mask-expressible operations, none of the identified stages directly maps to Tensor Core matrix-multiply-accumulate (MMA) operations. Instead, the kernel relies on integer AND/XOR operations, population counts, random feedback, conditional branches, carry propagation, and atomic reductions.

The automata-state update is already implemented using bit-slicing, processing 32 automata per `uint32`. However, full-state scans and memory traffic remain significant limitations.

Furthermore, each update launches a single CUDA block containing 64 threads (two warps). Consequently, at most one of the H100's 132 streaming multiprocessors (SMs) can execute the kernel at a time, corresponding to a maximum active-SM coverage of 0.758% and only 0.0237% of the GPU-wide resident-warp capacity.

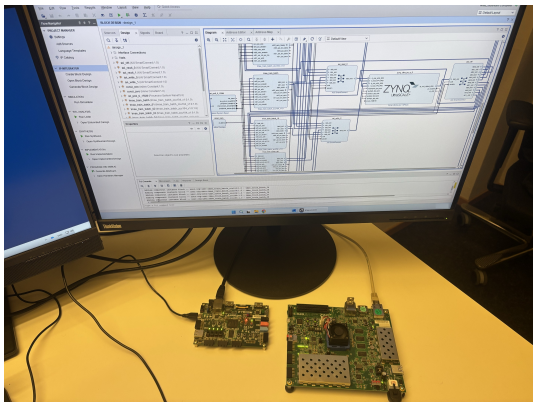

Figure 8: Hardware development setup for the FastOmniTMAE FPGA accelerator, showing the Zybo platform (left), the ZCU104 platform (right), and the corresponding Vivado block design used for hardware integration and validation.

These observations identify three primary limitations of the profiled implementation: insufficient launch parallelism, irregular stochastic feedback, and automaton-state memory traffic. Although several stages are suitable for bitwise-mask processing, their execution does not directly benefit from the dense matrix acceleration provided by Tensor Cores.

A.6 Resource Utilization for ZCU104 and Zybo

Table 6 summarizes the post-synthesis resource utilization of the proposed FastOmniTMAE accelerator on the two SoC-FPGA targets. The ZCU104 design targets the Zynq UltraScale+ MPSoC device `xczu7ev-ffvc1156-2-e` and instantiates 12 accelerator engines. The design uses a 150 MHz PL clock and distributes memory traffic across multiple high-performance ports, with HP0/HP1 used for reads and HP2/HP3 used for writes. In contrast, the Zybo design targets the Zynq-7000 device `xc7z010clg400-1` and instantiates two accelerator engines using a 75 MHz PL clock. The Zybo version uses one AXI-GP control path and two HP data paths, where HP0 is used for reads and HP1 for writes.

Table 6: Post-synthesis resource utilization for the ZCU104 and Zybo SoC-FPGA implementations.

Resource	**ZCU104, 12 engines**	**Zybo, 2 engines**
LUTs	59,228 / 230,400 (25.71%)	12,453 / 17,600 (70.76%)
Registers	50,349 / 460,800 (10.93%)	11,493 / 35,200 (32.65%)
BRAM tiles	270 / 312 (86.54%)	45 / 60 (75.00%)
DSPs	12 / 1,728 (0.69%)	2 / 80 (2.50%)
Clock buffers	1 / 96 (1.04%)	1 / 32 (3.13%)

The results show that the accelerator is mainly constrained by on-chip memory rather than arithmetic resources. This is expected because TM training relies primarily on logical operations and automaton-state storage rather than dense floating-point computation. On the ZCU104, the larger programmable logic allows 12 engines to be instantiated while using only 25.71% of LUTs and 10.93% of registers, although BRAM utilization reaches 86.54%. On the Zybo, the design remains feasible with two engines, but the smaller device is more tightly constrained, using 70.76% of LUTs and 75.00% of BRAM tiles. The low DSP utilization on both devices confirms that the architecture does not depend on conventional multiply-accumulate resources, supporting the logic-oriented design motivation of FastOmniTMAE. Figure 8 shows the hardware development setup used to implement and validate the accelerator on the Zybo and ZCU104 platforms.

Table 7: Average word-similarity correlations across the five evaluation datasets.

Model	Spearman (ρ)	Kendall (τ)
FastOmniTMAE	0.537	0.392
TinyBERT	0.613	0.444
DistilBERT	0.579	0.414

A.7 Comparison with Lightweight Pretrained Transformers

To further contextualize the embedding quality of FastOmniTMAE, we conducted a preliminary comparison with two lightweight pretrained Transformer models, TinyBERT and DistilBERT, using the same five word-similarity datasets and evaluation procedure described in Section 3.2. Table 7 reports the average Spearman and Kendall correlations across these datasets.

Table 8: Architectural and computational characteristics of FastOmniTMAE and lightweight pretrained Transformers.

Aspect	FastOmniTMAE	TinyBERT	DistilBERT
Model footprint	2.44 MiB of automaton states per target word during training	14.35M parameters (54.75 MiB)	66.36M parameters (253.16 MiB)
Embedding size	40,000 vocabulary-aligned integer dimensions	312 pooled dimensions	768 pooled dimensions
Vector storage	39.06 KiB with 8-bit storage	1.22 KiB with FP32 storage	3.00 KiB with FP32 storage
Training	Corpus-based training per target word; four epochs, 2,000 examples per epoch	Pretrained; forward-pass extraction	Pretrained; forward-pass extraction
Hardware	Logical and bitwise operations suited to CPU and FPGA implementations	Dense tensor operations suited to GPU acceleration	Dense tensor operations suited to GPU acceleration
Interpretability	Vocabulary-aligned dimensions, logical clauses, automaton states, and traceable feedback	Dense latent representations without direct vocabulary alignment	Dense latent representations without direct vocabulary alignment

TinyBERT uses four Transformer layers with 312 hidden dimensions, while DistilBERT uses six layers with 768 hidden dimensions. Both pretrained models achieve higher average correlations than FastOmniTMAE in this experiment. However, the models differ in their training requirements, memory footprints, hardware characteristics, and interpretability, as summarized in Table 8.

The pretrained Transformer models achieve higher average word-similarity correlations and produce more compact cached vectors. However, their network parameters must remain available to generate embeddings for new inputs. In contrast, FastOmniTMAE produces static, vocabulary-aligned embeddings that can be stored independently of the training model. Its learning process also supports inspection of logical clauses, automaton-state transitions, feedback operations, and supporting training documents.

These results illustrate the trade-offs between the representation quality of pretrained Transformers and the interpretable, logic-based learning and hardware characteristics of FastOmniTMAE.