Tensorised Simulation of Place/Transition-Petri Nets in PyTorch

Heiko Rölke[1]

[1]University of Applied Sciences of the Grisons (FHGR), 7000 Chur, Switzerland

Abstract

Place/Transition Petri nets have a direct algebraic representation: markings are vectors, pre- and postconditions are matrices, and firing can be described by linear updates. We investigate whether this representation can be implemented as a tensor program in *PyTorch*. The aim is not to replace established Petri net tools, but to evaluate whether batched simulation of many markings can benefit from the same execution model that is used in modern numerical and machine-learning frameworks. A research prototype represents P/T nets as tensors, computes enabled transitions through tensor comparisons and reductions, and fires one enabled transition per marking according to a deterministic or random policy. Benchmarks on CPU and GPU indicate that PyTorch is a viable execution platform for tensorised Petri net simulation, especially for sufficiently large batched workloads. At the same time, the results show that GPU acceleration is workload-dependent and that enablement checking is the dominant performance bottleneck in the current prototype.

Keywords

Petri net simulator, PyTorch, CUDA, tensor computation, batched simulation

1. Motivation and Research Question

Petri nets are a classical formalism for modelling concurrent, distributed, and discrete-event systems. In the case of Place/Transition Petri nets (P/T nets), the formal model has a particularly clear algebraic structure. A marking can be represented as a vector, the consumption and production of tokens can be represented by matrices, and a firing step can be expressed as an update based on an incidence matrix. This makes P/T nets an interesting candidate for implementation in tensor-oriented numerical frameworks.

Modern frameworks such as *PyTorch* execute tensor operations on CPUs and GPUs without requiring the programmer to write hardware-specific CUDA code. This raises the question whether Petri net simulation can be reformulated as a tensor program that benefits from batched execution and GPU acceleration. Related work has already shown that matrix-based Petri net simulation and GPU-oriented Petri net analysis can be effective in specialised settings [1, 4, 5]. However, a compact and general PyTorch implementation of the basic P/T net firing semantics is still not a standard approach.

The guiding research question of this work is therefore:

> How suitable is PyTorch as an execution platform for the tensorised simulation of P/T Petri nets, particularly with regard to batched execution and GPU acceleration?

This question is investigated through a research prototype and an empirical evaluation. The focus is deliberately limited to simulation of the firing semantics. Full verification, reachability analysis, high-level net classes, and advanced state-space methods are outside the scope of this abstract.

2. Background and Related Work

The theoretical basis of the approach is the standard matrix representation of P/T nets. If P is the set of places and T the set of transitions, a net can be represented by a pre-incidence matrix $Pre \in \mathbb{N}^{|P|\times|T|}$ and

PNSE'26, International Workshop on Petri Nets and Software Engineering, 2026

0000-0002-9141-0886 (H. Rölke)

a post-incidence matrix $Post \in \mathbb{N}^{|P|\times|T|}$. The incidence matrix is $C = Post - Pre$. A marking $m \in \mathbb{N}^{|P|}$ enables a transition t if $m(p) \geq Pre(p,t)$ for every place p. Firing t then produces the successor marking $m' = m + C(:,t)$.

Ai et al. propose the simulation of elementary one-bounded nets using Boolean matrices and report substantial speed-ups compared with several symbolic logic-based systems [1]. The CUDA-PetriNet-Toolkit demonstrates that GPU parallelisation can be useful for Petri-net-related analysis tasks, in particular reachability graph computation [4]. Zaitsev et al. present a notation for mass-parallel algorithms and use Petri net state-space computation on GPUs as a case study, reporting strong speed-ups compared with Tina [5]. Other recent approaches connect Petri nets with learning frameworks, for example PetriRL for job-shop scheduling [2] or neural surrogates for stochastic Petri nets [3]. These works show that Petri nets and modern numerical computation can be combined fruitfully, but they do not directly provide a general PyTorch-based formulation of the P/T net firing semantics.

3. Tensorised Firing Semantics

The prototype represents the static structure of a P/T net by the tensors *Pre*, *Post*, and *C*. A single marking is represented by a vector m, while a batch of B markings is represented as

$$M \in \mathbb{N}^{B\times|P|}.$$

This batch representation is central to the approach: instead of simulating one marking after another, many markings are processed simultaneously by the same tensor operations.

The simulation step consists of three phases. First, the simulator computes the enablement relation between all markings in the batch and all transitions of the net. Conceptually, this checks the condition

$$M[b,p] \geq Pre[p,t] \quad \text{for all } p,$$

for every batch index b and transition t. In PyTorch, this can be implemented by broadcasting, comparison, and reduction operations. The result is a Boolean tensor indicating which transitions are enabled in which markings.

Second, one enabled transition is selected for each marking. The current prototype supports two policies. The deterministic *first-enabled* policy selects the first enabled transition according to the transition order. The *random-enabled* policy selects one enabled transition at random. This separation between enablement computation and transition selection makes it possible to compare policy overhead without changing the underlying representation of the net.

Third, the selected transitions are fired. For each marking in the batch, the simulator adds the corresponding column of the incidence matrix C. In the current prototype exactly one transition is fired per marking and per simulation step. Parallel firing of conflict-free transition sets is intentionally left for future work, because it raises additional semantic and implementation questions.

The implementation follows a tensor-centric rather than an object-centric design. Places and transitions are not represented as individual objects during simulation. Instead, the operational semantics is expressed as tensor operations over the matrix representation. The purpose of this design is transparency, correctness, and performance measurability rather than feature completeness.

4. Benchmark Design

The evaluation compares CPU and GPU execution of the same PyTorch implementation. The experiments were carried out in a Google Colab environment with a T4 GPU enabled. All benchmark instances are randomly generated weighted P/T nets. For each configuration, the program constructs pre- and post-incidence matrices with a controlled density of non-zero entries and samples random initial markings. Fixed random seeds are used to make CPU and GPU runs comparable.

The benchmark measures the runtime of batched simulation over a fixed number of steps. Each step performs enablement checking, transition selection, and marking update. Before timed runs, warm-up

executions are performed in order to reduce initialization effects. On CUDA, explicit synchronization is used before and after timing so that the measured values reflect actual GPU execution rather than only kernel launch overhead.

The benchmark varies the number of places, the number of transitions, the density of the incidence matrices, the batch size, and the number of simulation steps. The reported metrics are average runtime, runtime standard deviation, simulation steps per second, and state updates per second. The latter metric is particularly useful because it combines batch size and trajectory length and therefore measures the effective throughput of individual marking updates.

Three main experiments were conducted. The first experiment varies the batch size for a fixed net size in order to determine when GPU execution becomes advantageous. The second experiment varies the net size while keeping the batch size large, thereby testing how structural complexity affects throughput. The third experiment compares deterministic and random transition selection policies. In addition, a profiled version of the simulator measures the relative cost of enablement checking, policy selection, and marking update.

5. Results and Interpretation

The correctness evaluation was successful for a collection of manually designed and automatically generated test nets. The resulting traces were checked against the expected classical firing semantics. This confirms that the tensor formulation can reproduce the intended P/T net behaviour for the supported one-transition-per-step execution model.

The performance results show that GPU acceleration is not uniformly beneficial. For very small batch sizes, CPU execution can outperform GPU execution because the overhead of CUDA kernel launches, synchronization, and data management outweighs the available parallelism. This is especially visible for batch size one, where the computation is too small to exploit the GPU effectively.

As the batch size increases, GPU throughput improves substantially relative to CPU throughput. This supports the main hypothesis of the work: tensorised Petri net simulation is most useful when many markings are processed simultaneously. In such scenarios, batched tensor operations provide enough parallel work to amortize the GPU overhead. The approach is therefore particularly promising for parameter sweeps, repeated simulation, Monte-Carlo-style exploration, or other settings in which large numbers of markings or trajectories must be evaluated.

The net-scaling experiment shows that increasing the number of places and transitions reduces throughput on both CPU and GPU. Larger nets increase the cost of enablement checking because each transition has to be compared against more places. However, the relative GPU advantage tends to become more visible when large nets are combined with large batch sizes. This indicates that the suitability of PyTorch cannot be attributed to model size alone; it depends on the interaction between model dimensions, batch size, and tensor-operation overhead.

The policy comparison shows that randomized transition selection is more expensive than deterministic first-enabled selection, as expected. Nevertheless, the overhead remains secondary compared with the main semantic operations. This suggests that the prototype can support different firing policies without fundamentally changing its performance profile.

The phase-level runtime breakdown identifies enablement checking as the dominant cost component in most configurations. This is plausible because enablement checking requires comparisons across the product of batch size, number of places, and number of transitions. Marking update and transition selection are usually less expensive, although update costs can become visible on GPU for small and medium batch sizes. The main optimization target for future work is therefore the enablement computation, for example by using sparse representations, specialised layouts, or fused operations.

6. Limitations and Outlook

The present prototype is deliberately limited. It supports batched simulation in which one enabled transition is fired per marking and per step. It does not yet implement parallel firing of conflict-free transition sets, sparse tensor encodings, reachability analysis, or high-level extensions such as Coloured Petri Nets. Furthermore, the benchmark uses randomly generated dense tensors, which is useful for controlled measurement but does not cover the full variety of real Petri net models.

These limitations do not invalidate the main result. The work demonstrates that the firing semantics of P/T nets can be formulated naturally and correctly as a PyTorch tensor program. It also shows that the performance benefits are conditional: PyTorch and GPU execution are especially useful for workloads with sufficient parallel structure, but they should not be expected to improve isolated single-state simulation.

Future work will address two directions. First, the simulator should be extended towards richer execution semantics, especially conflict handling and parallel firing of independent transitions. Second, the prototype should be connected to existing graphical Petri net tools so that tensorised execution can be combined with established modelling workflows. Further optimization should focus on the enablement phase, since the current measurements identify it as the primary bottleneck.

7. Conclusion

PyTorch is a viable platform for tensorised simulation of P/T Petri nets. The algebraic structure of P/T nets maps cleanly to tensor operations, and batched execution makes it possible to exploit GPU parallelism when the workload is sufficiently large. The main contribution of this work is a proof-of-concept formulation and empirical characterization of this execution model. The results show both the potential and the boundary conditions of the approach: GPU acceleration is promising for large batched simulations, while small workloads remain better suited to conventional CPU execution.

8. References

References

[1] Ai, L., Muggleton, S. H., Liang, S.-S., and Baldwin, G. S. (2024). Simulating Petri nets with Boolean Matrix Logic Programming. *arXiv preprint*. Available at: https://arxiv.org/abs/2405.11412

[2] Lassoued, S. et al. (2024). Introducing PetriRL: An innovative framework for JSSP resolution integrating Petri nets and deep reinforcement learning. *Journal of Manufacturing Systems*. Available at: https://www.sciencedirect.com/science/article/pii/S0278612524000943

[3] Manu, B. K., Reckell, T., Sterner, B., and Jevtic, P. (2025). A Simple Approximate Bayesian Inference Neural Surrogate for Stochastic Petri Net Models. *arXiv preprint*. Available at: https://arxiv.org/abs/2507.10714

[4] Zhang, Z. (n.d.). CUDA-PetriNet-Toolkit. GitHub repository. Available at: https://github.com/zhaolongzhang995/CUDA-PetriNet-Toolkit

[5] Zaitsev, D. A., Zhang, Z., Liu, D., & Shmeleva, T. R. (2025). Notation for mass parallel algorithms: computing Petri net state space on GPU case study. *International Journal of Parallel, Emergent and Distributed Systems*, 40(2), 101–115. https://doi.org/10.1080/17445760.2024.2431545

Declaration on Generative AI

During the preparation of this work, the author used ChatGPT-5 for grammar and spelling checks, literature search support, and condensation of the manuscript into an extended abstract. The author reviewed and edited the content as needed and takes full responsibility for the publication's content.