Learning, Solving & Optimizing PDEs with TensorGalerkin
an Efficient High-Performance Galerkin Assembly Algorithm

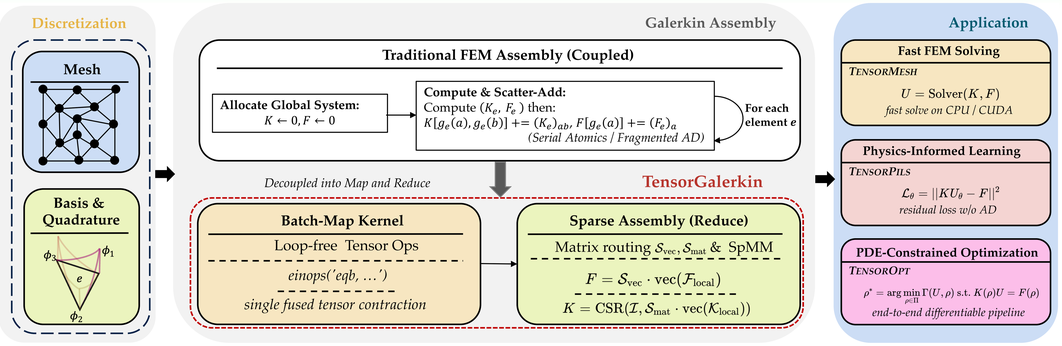
Shizheng Wen[1*] Mingyuan Chi* Tianwei Yu[1] Ben Moseley[2] Mike Y. Michelis[1] Pu Ren[3] Hao Sun[4] Siddhartha Mishra[1]

[1]ETH Zürich [2]Imperial College London [3]Northeastern University [4]Renmin University of China *Equal contribution

TLDR

FEM's real GPU bottleneck isn't the solve — it's the **assembly**. TensorGalerkin recasts Galerkin assembly as a tensorized **Map–Reduce** operation (batched element contractions → sparse-matmul message passing on the mesh graph), yielding:

1. one GPU-native, fully differentiable, JIT-free PyTorch engine that **solves** PDEs at order-of-magnitude speedups over CPU FEM,
2. **learns** PDEs ~5× faster than PINNs while pushing physics-informed operator learning to near data-driven efficiency
3. **optimizes** PDEs 3.7–4.5× over existing differentiable FEM on inverse problems.



Key idea:

Recast Galerkin assembly as a strictly tenderized Map (physics) + Reduce (topology), co-designed inside Pytorch autograd:

Backward graph: $O(Ek^2) \longrightarrow O(1)$ nodes independent of mesh size ‖ Eager execution → Zero JIT recompilation

Motivation

Modern SciML leans on Galerkin/FEM harder than ever:

1. **Many-query** (inverse design, topology opt, UQ) — one PDE, thousands of solves.
2. **Data generation** millions of sims for neural operators.
3. **Physics-informed learning:** weak-form/energy residual.

All three need FEM that's GPU-native, differentiable, PyTorch-native, and all re-run assembly *inside the autodiff graph*.

→ **The real bottleneck is assembly, not the solve.** Naive GPU assembly (element loops + scatter-add) kills throughput, bloats the graph.

Stage I - Batch-Map (tensorized physics)

Lift the element index to a batch dimension; evaluate all local matrices by one dense contraction over quadrature (q) and basis (a, b):

$$(\mathscr{K}_{\text{loc}})_{eab} = \sum_q \hat{w}_q \left|\det \mathscr{J}_{eq}\right| \mathscr{F}\left(\mathscr{G}_{eqa}, \mathscr{G}_{eqb}, \mathscr{C}_{eq}\right)$$

$\mathscr{G} = \mathscr{J}^{-\top} \nabla \hat{\mathscr{B}}$ are physical shape gradients. One kernel = all (q) local matrices - no loops over elements, basis, or quadrature.

Stage II - Sparse-Reduce (topology routing)

Precompute binary routing matrices from mesh topology only, then assemble with on deterministic Sparse Matrix Multiplication (SpMM):

$$F = \mathcal{S}_{\text{vec}} \cdot \text{vec}(\mathcal{F}_{\text{loc}})$$
$$K = \text{CSR}\left(\mathcal{I}, \mathcal{S}_{\text{mat}} \cdot \text{vec}(\mathcal{K}_{\text{loc}})\right)$$

Replaces millions of atomic scatter-adds -> deterministic & reproducible

Fast FEM Solving Engine

$U = \text{Solver}(K, F)$

TensorMesh https://www.tensor-mesh.com/

torch-sla https://www.torchsla.com/

Our two killers for integrating classical numerical methods in modern ML pipelines:

- **TensorMesh**: our fast, differentiable, JIT-Free, debugging-friendly finite element library for PyTorch.
- **torch-sla**: our differentiable cross-platform sparse linear algebra for PyTorch

Physics-Informed Learning System (TensorPILS)

Canonical strong form: $\mathscr{L}_{\text{st}}(\hat{u}) = \left\| \mathscr{D}[\hat{u}] - f \right\|_{L^2}^2$

Powered by TensorGalerkin, we can easily get the matrix of these bilinear forms.

Energy form: $\mathscr{L}_{\text{en}}(\hat{u}) = \frac{1}{2} a(\hat{u}, \hat{u}) - \ell(\hat{u})$

Galerkin-residual (weak form): $\mathscr{L}_{\text{wk}}(\hat{u}) = \sum_j \left| \ell(\phi_j) - a(\hat{u}, \phi_j) \right|^2$

PDE-Constrained Optimization (TensorOPT)

$$\rho^* = \arg\min_{\rho \in \Pi} \Gamma(U, \rho) \text{ s.t. } K(\rho)U = F(\rho),$$

- Due to the TensorMesh (powered by TensorGalerkin) and our differentiable sparse linear solver **torch-sla** are end-to-end differentiable and with $\mathcal{O}(1)$ computational graph. We can easily and efficiently use autograd to do gradient based optimization.

Results: TensorMesh

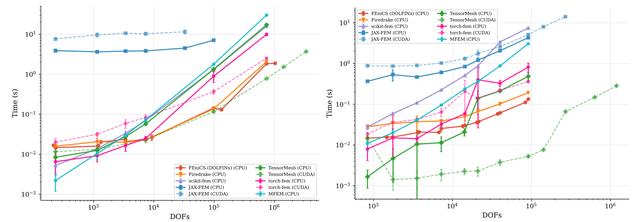
GPU-native FEM that scales past 10^6 DoFs - up to ~2 orders of magnitude faster than best CPU FEM stack, at matched accuracy.

Baselines

- Classical CPU FEM with 8 rank MPI: FEniCS, Firedrake, MFEM
- Classical python scope CPU FEM: scikit-fem
- Complied GPU/differentiable FEM: JAX-FEM, torch-FEM

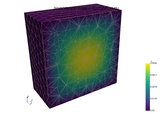
Speed

- **Poisson:** ~1 order of magnitude over the best baseline (FEniCS).
- **Elasticity:** ~2 orders of magnitude over the best baseline — the harder, vector-valued problem widens the lead.

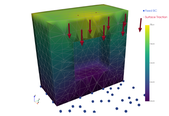


Possion 3D runtime

Elasticity 3D runtime



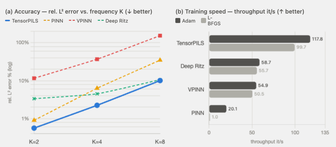
Poisson 3D solution



Elasticity 3D solution

Results: TensorPILS

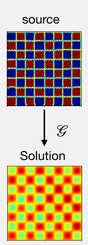
TensorPILS as a stand-alone neural PDE solver



source

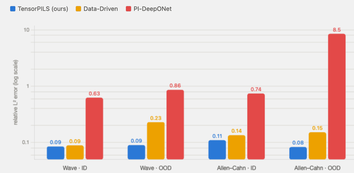
$\mathscr{G}$

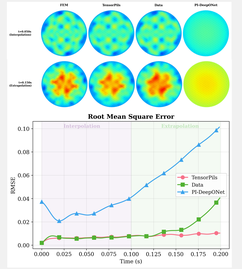
Solution



We learn the discontinuous checkerboard source function for poisson equation.

TensorPILS for physics-informed operator learning





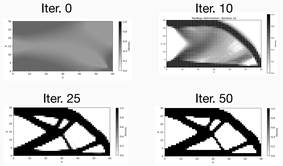
We are learning the time-dependent PDEs generalize different ICs for long-term rollout totally based on physical loss.

Results: TensorOPT

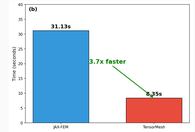
Problem

- Compliance minimization problem for a 2D cantilever beam

Process of Topology evolution

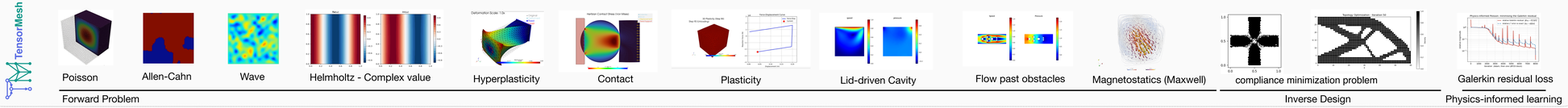


Results



Metric	JAX-FEM	TensorMesh	
ACCURACY			Rel. diff.
Initial compliance	53.71	53.71	0.00%
Final compliance	84.03	83.75	0.33%
WALL-CLOCK TIME			Speedup
Setup	2.62 s	0.58 s	4.5×
Optimization	28.51 s	7.77 s	3.7×
Total	31.13 s	8.35 s	3.7×

- **TensorMesh** (Used for TensorOPT) IS 3.7 times faster than JAX-FEM with final compliance difference < 0.33%.



References

[1] Wen et al. *Learning, Solving and Optimizing PDEs with TensorGalerkin: an efficient high-performance Galerkin assembly algorithm* (2026)
[2] Chi et al. *torch-sla: Differentiable Sparse Linear Algebra with Adjoint Solvers and Sparse Tensor Parallelism for PyTorch* (2026)
[3] Xue et al. *JAX-FEM: A differentiable GPU-accelerated 3D finite element solver for automatic inverse design and mechanistic data science* (2022)
[4] Wen et al. *TensorMesh: A fast, differentiable, JIT-free, debugging-friendly finite element library for PyTorch* (2026)

ICML International Conference On Machine Learning

Seoul, 2026