torch-sla: Differentiable Sparse Linear Algebra with Adjoint Solvers and Sparse Tensor Parallelism for PyTorch

Mingyuan Chi [*1] **Shizheng Wen** [*1]

The library is available at https://www.torchsla.com/

Abstract

Differentiable sparse linear algebra is foundational for scientific machine learning, yet PyTorch lacks a unified library for it: `torch.sparse` provides only low-level kernels and a non-differentiable, CPU-only `spsolve`, and `torch.linalg` is dense-only. We present `torch-sla`, an open-source library that fills this gap. It exposes a single autograd-aware API for direct, iterative, nonlinear, and eigenvalue solvers across five interchangeable backends—SciPy and Eigen on CPU, cuDSS, CuPy, and a PyTorch-native iterative solver on GPU—with automatic dispatch by device and problem size. The library further supports batched solves over shared or distinct sparsity patterns and distributed multi-GPU execution via domain decomposition with halo exchange. These capabilities are made scalable by an O(1)-graph adjoint differentiation framework and an autograd-compatible distributed halo-exchange layer.

1. Introduction

Sparse linear systems arise naturally in many scenarios of machine learning and scientific computing. For example, graph neural networks operate on sparse adjacency matrices (Kipf & Welling, 2017; Veličković et al., 2018); neural operators on unstructured meshes often discretize local operators sparsely (Li et al., 2020; Brandstetter et al., 2022; Wen et al., 2025; Shi et al., 2025); and differentiable simulation pipelines require gradients through sparse linear, nonlinear, and eigenvalue solves (Holl & Thuerey, 2024; Hu et al., 2020; Blondel et al., 2022). These applications also tend to be batched, and a training step may contain one sparse system per sample, time step, or mesh.

In the community of JAX (Bradbury et al., 2018), `jax.lax.custom_linear_solve` provides an implicit differentiation primitive, `jax.scipy.sparse.linalg` builds differentiable iterative solvers on top of it, and downstream libraries extend the same idea to nonlinear and fixed-point problems. By contrast, PyTorch has a larger research community, but does not provide an analogous stack. `torch.sparse` exposes sparse matrix–vector and matrix–matrix multiplication (SpMV and SpMM), but its only sparse solve, `spsolve`, is non-differentiable and implemented as a CPU-only SuperLU (Demmel et al., 1999) wrapper for square systems. `torch.linalg` is dense-only. Calling external solvers such as cuDSS or PETSc from PyTorch breaks the autograd graph, and simply differentiating through a hand-written iterative solver loop builds an $\mathcal{O}(k)$-node computational graph over the iterations. In practice, users are left to drop gradients, move the relevant part of the code to JAX, or write a custom solver layer for each problem.

Supporting these workloads in PyTorch takes more than a thin `torch.autograd.Function` wrapper around existing solvers. The first issue is backend choice. Direct solvers such as cuDSS and SuperLU are often fastest below $\sim 10^5$ degrees of freedom (DOF), but their $\mathcal{O}(n^{1.5})$ fill-in (George, 1973) can exhaust GPU memory above $\sim 2\times10^6$ DOF. Iterative methods such as conjugate gradient (CG) (Hestenes & Stiefel, 1952) and biconjugate gradient stabilized (BiCGStab) (Van der Vorst, 1992) have $\mathcal{O}(\text{nnz})$ memory cost (with nnz the number of non-zeros) and can reach more than 10^8 DOF, but small problems are dominated by kernel-launch overhead. Therefore, a useful PyTorch interface should dispatch across heterogeneous backends. The second issue is differentiation. Backpropagating through k iterations of an iterative solver stores the intermediate vectors and creates an $\mathcal{O}(k)$-node graph. For a 1M-DOF problem with 1000 CG iterations, the saved vectors alone occupy roughly 80 GB. The third issue is scalability. Problems with 10^7–10^9 unknowns require distributed domain decom-

[*]Equal contribution [1]ETH Zurich, Zurich, Switzerland. Correspondence to: Mingyuan Chi <walker.chi.000@gmail.com>, Shizheng Wen <shizheng.wen@sam.math.ethz.ch>.

Proceedings of the AI4Physics Workshop at the 43rd International Conference on Machine Learning (AI4Physics@ICML 2026), Seoul, South Korea. 2026. Copyright 2026 by the author(s).

position, and the halo exchange used in the forward solve must be transposed in the backward pass. Existing PyTorch distributed primitives do not provide this sparse-operator adjoint directly.

Contributions. To address these requirements, `torch-sla` is an open-source PyTorch library for differentiable sparse linear algebra. It provides:

- **Unified backend abstraction** (§3.1). A single autograd-aware API dispatches across five backends—SciPy, Eigen, cuDSS, CuPy, and a PyTorch-native iterative solver—selected by device and problem size. The backend interface is also extensible: adding libraries such as PETSc (Balay et al., 2023), Trilinos (Mayr et al., 2026), or hypre (Falgout & Yang, 2002) requires only implementing a common backend interface.
- **Adjoint differentiation framework** (§3.2). We faithfully port the well-established implicit-function-theorem (IFT) adjoint (§3.2 cites the JAX and FEniCS equivalents) and apply it uniformly across linear, nonlinear, and eigenvalue solves and all five backends, keeping the autograd graph at $\mathcal{O}(1)$ nodes and $\mathcal{O}(\text{nnz})$ memory, independent of solver iterations or backend.
- **Distributed solvers with autograd-compatible halo exchange** (§3.3). Domain decomposition with halo exchange uses transposed communication in the backward pass, so distributed solves remain compatible with end-to-end differentiation.

On 2D Poisson benchmarks, `torch-sla` scales to 169M DOF on one H200 and 400M DOF on three H200s, with gradients verified analytically for linear solves and against finite differences for nonlinear and eigenvalue solves. Moreover, an inverse-coefficient learning task is conducted to demonstrate the end-to-end usability of the library.

2. Related Work

The PyTorch sparse ecosystem. PyTorch's native sparse support is fragmented and incomplete. `torch.sparse` provides COO/CSR storage and basic kernels (sparse matrix–vector and matrix–matrix multiplication, SpMV/SpMM, plus elementwise operations); its only sparse solve, `torch.sparse.spsolve`, is non-differentiable, CPU-only, and restricted to square systems. On the other hand, `torch.linalg` is dense-only. Domain libraries such as `torch-scatter`, `torch-sparse`, and PyTorch Geometric (Fey & Lenssen, 2019) target message-passing primitives for graph neural networks (GNNs) rather than general-purpose linear algebra.

Differentiable solvers in PyTorch. Several PyTorch libraries already provide differentiable solves, each on a different slice. Theseus (Pineda et al., 2022) is a nonlinear least-squares *optimizer* whose batched sparse solver is an internal component; CoLA (Potapczynski et al., 2023), the closest analogue, differentiates an *implicit* linear-operator abstraction that exploits *algebraic* structure (Kronecker, low-rank) rather than a sparsity pattern, so its matrix-free operators preclude the sparse *direct* factorizations (cuDSS) and pattern-based preconditioners (ILU, AMG) we rely on; PhiFlow (Holl & Thuerey, 2024) offers differentiable preconditioned *iterative* solves but is GPU-iterative-only. Narrower utilities (`torchsparsegradutils`, `torch_sparse_solve`, the Firedrake–PyTorch bridge) and direct external-solver calls (which break the autograd graph) cover the rest. No prior PyTorch library spans direct, iterative, nonlinear, and eigenvalue solves with batched *and* distributed dispatch at once; `torch-sla` is distinguished by a GPU sparse-*direct* backend (cuDSS) under autograd, tensor-parallel distribution with a transposed-halo backward pass, and this unified coverage at 10^8+ DOF on one GPU (4×10^8 distributed). Appendix E details the axes that separate these libraries.

Differentiable solvers in JAX. JAX provides a layered stack for differentiable sparse linear algebra. At the primitive layer, `jax.lax.custom_linear_solve` wraps arbitrary linear solvers and obtains gradients via the implicit function theorem. Built on top, `jax.scipy.sparse.linalg.{cg, bicgstab, gmres}` provide differentiable iterative solvers; notably, the experimental `sparse.linalg.spsolve` is *not* autodiff-compatible, illustrating that even in JAX, differentiable direct solvers require deliberate adjoint wrapping. At the third-party layer, Lineax (Rader et al., 2023) unifies linear solvers behind a common interface, while JAXopt (Blondel et al., 2022) and Optimistix (Rader et al., 2024) extend implicit differentiation to nonlinear and fixed-point problems. The JAX counterpart to our single-library scope is thus the Lineax + JAXopt + Optimistix trio together; `torch-sla` unifies these axes in one PyTorch library (Table 1).

Established sparse libraries and GPU backends. Mature sparse-solver libraries span CPU and GPU, direct and iterative, single-node and distributed regimes. SciPy (Virtanen et al., 2020) provides SuperLU on CPU; NVIDIA cuDSS (NVIDIA Corporation, 2024) offers GPU direct solvers (LU, Cholesky, LDLT); CuPy (Okuta et al., 2017) exposes iterative and direct solvers on GPU; and AmgX (Naumov et al., 2015) provides algebraic multigrid with strong industrial scalability. At larger scale, PETSc (Balay et al., 2023), Trilinos (Mayr et al., 2026), and hypre (Falgout & Yang, 2002) provide distributed sparse linear algebra with

Table 1. Differentiable sparse linear-algebra capability matrix. JAX provides a layered ecosystem; PyTorch's native stack stops at non-differentiable CPU `spsolve`; `torch-sla` closes the gap in a single library. The matrix compares *first-class* support; some "—" cells are assemblable by hand (e.g. distributed autograd-compatible solves via `shard_map`+`custom_linear_solve` in JAX) but are not provided as a feature.

Capability	JAX (native + 3rd party)	PyTorch (native)	**`torch-sla`**
Differentiable linear-solve primitive	`custom_linear_solve`	—	✓
Differentiable iterative solvers	`jax.scipy.sparse.linalg`	—	✓
Differentiable direct solvers	via Lineax / manual	—	✓
GPU sparse direct backend	exp. `spsolve` (no autodiff)	—	cuDSS, CuPy
Nonlinear solve with adjoint	JAXopt	—	✓
Eigenvalue solve with adjoint	dense / problem-specific	—	✓
Batched solves (shared pattern)	via `vmap`	—	✓
Batched solves (distinct patterns)	manual	—	`SparseTensorList`
Distributed sparse solvers + autograd	—	—	✓

mature preconditioners; however, these systems generally do not compose with PyTorch automatic differentiation, and exposing them through PyTorch typically requires copying tensors across language, device, or process boundaries. `torch-sla` instead treats such solvers as interchangeable, autograd-aware backends behind a single PyTorch API. New backends only need to conform to a common solver interface, while the distributed domain-decomposition patterns used in PETSc, Trilinos, and OpenFOAM (Jasak et al., 2007) are adapted to PyTorch with autograd-compatible communication (§3.3).

3. Methodology

`torch-sla` is built on three components: a unified backend abstraction (§3.1) that dispatches across five solvers behind one autograd-aware API, an adjoint differentiation framework (§3.2) that keeps the autograd graph at $\mathcal{O}(1)$ nodes regardless of solver iterations, and an autograd-compatible distributed layer (§3.3) implementing domain decomposition with halo exchange. Figure 1 gives a system overview.

3.1. Unified Backend Abstraction

Typed sparse-tensor hierarchy. `torch-sla` exposes four sparse-tensor types organized along two orthogonal axes—single matrix versus list, local versus distributed:

Layout	Single matrix	Matrix list
Local	`SparseTensor`	`SparseTensorList`
Distributed	`DSparseTensor`	`DSparseTensorList`

`SparseTensor` holds a single matrix or a batch sharing one sparsity pattern, so one symbolic factorization (direct) or set of communication buffers (iterative) is reused across the batch. `SparseTensorList` holds a batch with *distinct* patterns (GNN minibatches, neural operators on irregular meshes), dispatching each element with an isolated autograd graph. `DSparseTensor` and `DSparseTensorList` are distributed variants where each process owns a row partition with halo metadata. All types expose the same methods (`.solve`, `.matvec`, `.eigsh`, `.det`) plus conversion utilities (`partition`, `gather_global`, etc.).

Five backends and auto-dispatch. `torch-sla` ships with five backends, each specialized to a device and a problem-size regime: `scipy` (Virtanen et al., 2020) (CPU; SuperLU, UMFPACK (Davis, 2004), CG, BiCGStab; CPU default with machine precision), `eigen` (Guennebaud et al., 2010) (CPU; CG, BiCGStab; alternative CPU iterative), `cudss` (CUDA; LU, Cholesky, LDLT; fastest direct solver below $\sim$ 2M DOF), `cupy` (Okuta et al., 2017) (CUDA; LU, CG, and generalized minimal residual (GMRES) via `cupyx`), and a `pytorch`-native backend (CUDA; CG, BiCGStab; beyond $\sim$2M DOF on a single GPU). We list the backend library versions used in our experiments in Appendix A; full method-level coverage per backend is also given there.

The auto-dispatch policy follows three rules in priority order: (i) match the device of the input tensors; (ii) for CUDA devices, prefer `cudss` when the matrix fits within the fill-in budget ($n \lesssim 2 \times 10^6$ for typical 2D/3D PDE matrices) and fall back to the `pytorch`-native iterative backend for larger problems; (iii) for CPU devices, prefer `scipy` for direct solves and offer `eigen` as an iterative alternative. Symmetry and symmetric positive-definiteness (SPD) are detected on the matrix values and used to upgrade general LU to Cholesky or LDLT where applicable. Adding a new backend—PETSc, Trilinos, hypre, or a learned preconditioner—requires only implementing the solver methods exposed by the backend and registering its applicability conditions through `select_backend`.

Bridging backends into autograd. All five backends are wrapped uniformly in `torch.autograd.Function`: `forward` invokes the backend and stashes $(\mathbf{A}, \mathbf{x})$,

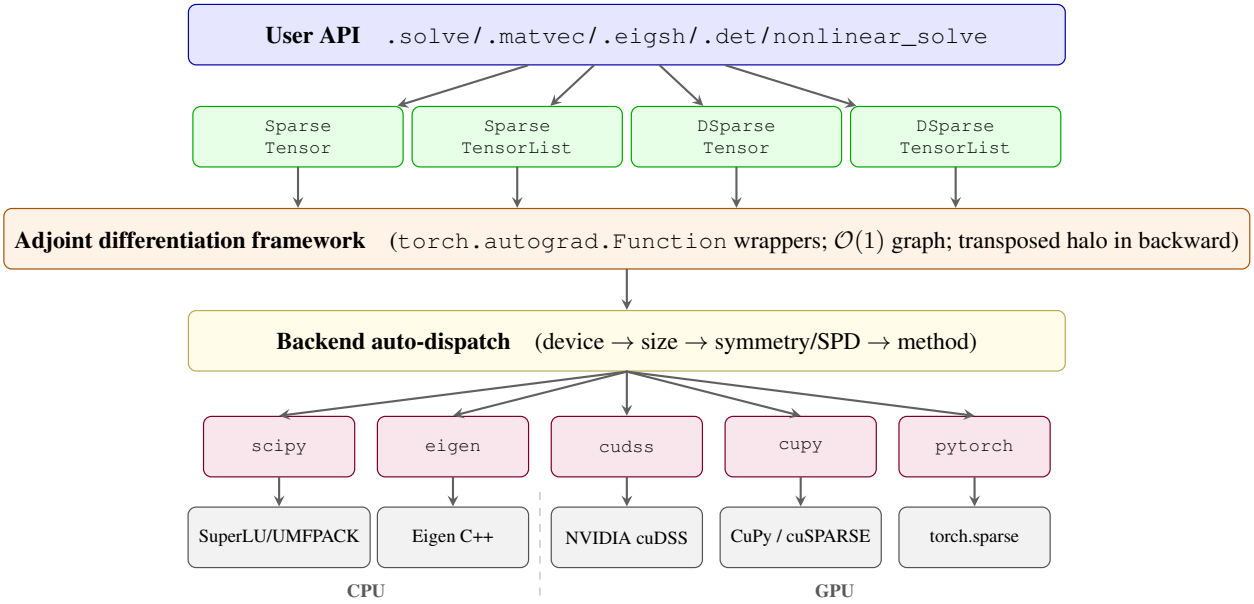

Figure 1. `torch-sla` system architecture. User-facing solve calls on any of the four typed sparse tensors flow through a single adjoint differentiation layer (implemented as `torch.autograd.Function` wrappers) and a unified auto-dispatch policy that selects among five interchangeable backends. Backends in turn delegate to established CPU and GPU libraries; the same path is reused—in transposed form—during the backward pass.

`backward` runs the adjoint solve of §3.2. For the external backends this is the only path preserving the autograd graph across the C++/CUDA boundary; the PyTorch-native backend (otherwise an $\mathcal{O}(k)$-node graph) is wrapped the same way under `torch.no_grad()`, so the adjoint framework is the single gradient path for all backends.

API. A single `.solve(b)` call dispatches uniformly across all five backends and four tensor types—single, batched (shared or distinct patterns), nonlinear, and distributed solves—with gradients flowing through unchanged; `backend` and `method` keywords override auto-dispatch. Appendix B gives end-to-end code.

3.2. Adjoint Differentiation Framework

The adjoint differentiation in `torch-sla` faithfully ports a classical, well-understood technique—the implicit-function-theorem (IFT) adjoint underlying `jax.lax.custom_linear_solve` (Bradbury et al., 2018), JAXopt (Blondel et al., 2022), Optimistix (Rader et al., 2024), CoLA (Potapczynski et al., 2023), and the dolfin-adjoint/pyadjoint tape (Mitusch et al., 2019). Our contribution is its uniform, backend-agnostic integration into PyTorch autograd at scale, not the derivation; we restate it here to fix notation.

Naively backpropagating through k iterations of an iterative solver builds a graph with $\mathcal{O}(k)$ nodes and stores intermediate vectors at each step; for a 1M-DOF problem with 1000 CG iterations, the saved vectors alone occupy roughly 80 GB. The adjoint framework collapses this to $\mathcal{O}(1)$ nodes and $\mathcal{O}(n + \text{nnz})$ memory, independent of k and of the backend used for the forward solve.

3.2.1. The General Setting

Let $\mathbf{x}^*(\boldsymbol{\theta}) \in \mathbb{R}^n$ be the solution of an implicit residual equation

$$\mathbf{F}(\mathbf{x}^*, \boldsymbol{\theta}) \;=\; \mathbf{0}, \qquad \mathbf{F} : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}^n, \tag{1}$$

where $\boldsymbol{\theta} \in \mathbb{R}^d$ collects all differentiable inputs—matrix nonzero values, right-hand side entries, or physical parameters. For a downstream scalar loss $\mathcal{L}(\mathbf{x}^*)$, the gradient follows from the method of Lagrange multipliers: stationarity of the augmented objective $\mathcal{A} = \mathcal{L}(\mathbf{x}) + \boldsymbol{\lambda}^\top \mathbf{F}(\mathbf{x}, \boldsymbol{\theta})$ in $\mathbf{x}$ and $\boldsymbol{\lambda}$ identifies $\boldsymbol{\lambda}$ as the multiplier enforcing the constraint and gives, equivalently to the implicit function theorem (Krantz & Parks, 2002),

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = -\boldsymbol{\lambda}^\top \frac{\partial \mathbf{F}}{\partial \boldsymbol{\theta}}, \quad \mathbf{J}^\top \boldsymbol{\lambda} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^*}, \qquad \mathbf{J} \equiv \left.\frac{\partial \mathbf{F}}{\partial \mathbf{x}^*}\right|_{\mathbf{x}^*}. \tag{2}$$

The backward pass thus reduces to one adjoint solve $\mathbf{J}^\top \boldsymbol{\lambda} = \partial \mathcal{L} / \partial \mathbf{x}^*$ at the converged solution, followed by a vector–Jacobian product $-\boldsymbol{\lambda}^\top \partial \mathbf{F} / \partial \boldsymbol{\theta}$. Only $\mathbf{x}^*$ and the data needed to apply $\mathbf{J}$ are stashed during the forward pass; intermediate solver iterates are not referenced.

3.2.2. Three Instances

We instantiate Eq. (2) for the three solver types in `torch-sla`.

Linear systems. With residual $\mathbf{F}(\mathbf{x}, \mathbf{A}, \mathbf{b}) = \mathbf{A}\mathbf{x} - \mathbf{b}$, the Jacobian is $\mathbf{J} = \mathbf{A}$ and Eq. (2) specializes to the single adjoint linear system $\mathbf{A}^\top \boldsymbol{\lambda} = \partial \mathcal{L}/\partial \mathbf{x}$, yielding

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \boldsymbol{\lambda}, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{A}_{ij}} = -\boldsymbol{\lambda}_i \, x_j. \tag{3}$$

The matrix gradient is materialized only on the sparsity pattern at total cost $\mathcal{O}(\text{nnz})$.

Nonlinear systems. For a general residual $\mathbf{F}(\mathbf{u}, \boldsymbol{\theta}) = \mathbf{0}$ converged by Newton, Picard, or Anderson acceleration (Anderson, 1965; Kelley, 1995) to a fixed point $\mathbf{u}^*$, Eq. (2) applies directly with $\mathbf{J} = \partial \mathbf{F}/\partial \mathbf{u}^*$ evaluated at the solution. The forward pass may take many nonlinear iterations, each itself involving a linear solve; the backward pass is one adjoint linear solve plus one vector–Jacobian product. The Jacobian-vector and vector-Jacobian products required to apply $\mathbf{J}$ and $\mathbf{J}^\top$ are obtained from PyTorch's autograd via `torch.autograd.functional.{jvp,vjp}`, so users supplying a nonlinear residual as a Python function obtain a matrix-free adjoint without writing additional code. Because the adjoint is taken at the converged state, it is exact only once $\mathbf{F}(\mathbf{u}^*, \boldsymbol{\theta}) \approx \mathbf{0}$; early termination biases the gradient. The $\mathcal{O}(\text{nnz})$ matrix-gradient assembly of Eq. (3) carries over to this non-affine case.

Eigenvalue problems. For the symmetric eigenvalue problem $\mathbf{A}\mathbf{v} = \lambda \mathbf{v}$ with $\|\mathbf{v}\|_2 = 1$, the Hellmann–Feynman theorem (Magnus, 1985) gives the closed form

$$\frac{\partial \lambda}{\partial \mathbf{A}_{ij}} = v_i \, v_j, \tag{4}$$

which is Eq. (2) specialized to the constrained residual $\mathbf{F}(\mathbf{v}, \lambda; \mathbf{A}) = \big(\mathbf{A}\mathbf{v} - \lambda \mathbf{v},\ \frac{1}{2}(\mathbf{v}^\top \mathbf{v} - 1)\big)$ after eliminating the Lagrange multiplier on the normalization. Eigenvector gradients require one additional deflated linear solve per eigenpair; eigenvalue gradients reduce to an outer product on the sparsity pattern at cost $\mathcal{O}(\text{nnz})$. Eq. (4) assumes a *simple* eigenvalue with a smoothly varying eigenvector; at crossings or clusters the eigenvector gradient is ill-defined and needs degenerate-perturbation handling, so `torch-sla` targets the simple case (§5).

3.2.3. Complexity

Table 2 summarizes the savings for the linear case; the nonlinear and eigenvalue cases inherit the same $\mathcal{O}(1)$ graph-node count.

The forward-pass cost is unchanged at T_{solve}. The backward pass executes (i) one adjoint solve of the same size and sparsity pattern as the forward—reusing the same backend and, where applicable, the same factorization—and (ii) an $\mathcal{O}(\text{nnz})$ outer-product evaluation to assemble the matrix gradient. Since Eq. (2) treats the forward solve as a black box, any of the five backends of §3.1 may be used for the forward solve, and the adjoint solve may even use a different backend. Composition with the distributed layer of §3.3 is via the transposed halo exchange formalized in Appendix C.

Table 2. Complexity of naive autograd-through-iterations versus `torch-sla`'s adjoint backward, for a sparse linear solve with n unknowns, nnz non-zero entries, k solver iterations, and forward solve time T_{solve}.

	Naive	**Adjoint**
Graph nodes	$\mathcal{O}(k)$	$\mathcal{O}(1)$
Graph memory	$\mathcal{O}(k\,n)$	$\mathcal{O}(n + \text{nnz})$
Backward time	$\mathcal{O}(k \cdot \text{nnz})$	$\mathcal{O}(T_{\text{solve}}) + \mathcal{O}(\text{nnz})$

3.3. Distributed Layer with Halo Exchange

For problems beyond single-device memory, `torch-sla` partitions $\mathbf{A}$ across processes following the standard PETSc/Trilinos/OpenFOAM pattern: row-block ownership with halo metadata, halo exchange before each SpMV, and `all_reduce` for global inner products. Each forward halo exchange induces a *transposed* halo exchange in the backward pass, and the distributed adjoint solve is itself an instance of Eq. (2) on the distributed residual.

Domain decomposition. A sparse matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is partitioned across P processes. Process p owns a contiguous block of rows $\mathcal{O}_p \subset \{1, \ldots, n\}$ and stores $\mathbf{A}[\mathcal{O}_p, :]$ together with a halo index set $\mathcal{H}_p$ enumerating the column indices outside $\mathcal{O}_p$ that appear in any locally owned row. Before each local SpMV, processes exchange boundary-owned entries with neighbors so that halo values are current; the local SpMV $\mathbf{y}_{\mathcal{O}_p} \leftarrow \mathbf{A}[\mathcal{O}_p, :]\, \mathbf{x}_{\text{local}}$ is then purely local. `torch-sla` supports contiguous row-based partitioning, recursive coordinate bisection (Berger & Bokhari, 1987) when node coordinates are available, and edge-cut minimization via METIS (Karypis & Kumar, 1998) for unstructured meshes through `partition_simple` and related utilities.

Distributed Krylov solvers. Distributed CG performs one halo exchange per iteration (inside the SpMV) plus two `all_reduce` operations for the inner products $\langle \mathbf{r}, \mathbf{r} \rangle$ and $\langle \mathbf{p}, \mathbf{A}\mathbf{p} \rangle$, giving per-iteration cost $\mathcal{O}(|\mathcal{H}_p| + \log P)$ with $|\mathcal{H}_p| \sim \mathcal{O}((n/P)^{(d-1)/d})$ on d-dimensional meshes with balanced partitioning. All tensor operations remain on-device, so the NVIDIA Collective Communications Library (NCCL) (NVIDIA Corporation, 2020) backend moves data directly between GPU memories without host staging.

BiCGStab and the locally optimal block preconditioned conjugate gradient (LOBPCG) eigensolver (Knyazev, 2001) follow the same template, substituting the Krylov recurrence and adding global reductions for additional inner products as needed. Pseudocode is in Appendix C.

Autograd composition: the transposed halo exchange. Treating the halo exchange as a linear operator $\mathbf{H} : \mathbb{R}^{|\mathcal{O}_p|} \to \mathbb{R}^{|\mathcal{O}_p|+|\mathcal{H}_p|}$ that scatters owned values into the halo positions of neighbor processes, the forward distributed SpMV factors as

$$\mathbf{y}_{\mathcal{O}_p} \;=\; \mathbf{A}_{\text{local}}\,\mathbf{H}(\mathbf{x}_{\mathcal{O}_p}), \tag{5}$$

with adjoint

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\mathcal{O}_p}} \;=\; \mathbf{H}^\top\left(\mathbf{A}_{\text{local}}^\top \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{\mathcal{O}_p}}\right). \tag{6}$$

The transposed halo exchange $\mathbf{H}^\top$ uses the same neighbor graph and message sizes as $\mathbf{H}$, with reversed sender/receiver roles and summation rather than overwrite at the receive site: where $\mathbf{H}$ sends owned boundary values to neighbor halos, $\mathbf{H}^\top$ sums incoming halo gradients back into the owning process's boundary entries. `torch-sla` implements both $\mathbf{H}$ and $\mathbf{H}^\top$ as PyTorch `autograd.Functions` on top of `torch.distributed` primitives, so each call to distributed SpMV inside CG, BiCGStab, or LOBPCG generates the correct backward communication automatically. Composing this with §3.2, a distributed linear solve runs distributed CG forward and one adjoint distributed solve $\mathbf{A}^\top\boldsymbol{\lambda} = \partial\mathcal{L}/\partial\mathbf{x}$ backward, reusing the same decomposition, backend, and neighbor graph; matrix gradients $-\lambda_i x_j$ are assembled locally on each process's owned non-zeros, with no additional communication.

Scope of distributed gradients. `matvec`, `solve`, and `eigsh` support distributed gradient flow with NCCL on GPU and Gloo on CPU. `det` is a global scalar that cannot be computed without full matrix information; the distributed implementation gathers all partitions onto one rank and emits a runtime warning, and is documented as not scaling to distributed sizes (a distributed factorization or stochastic log-determinant is left to future work).

4. Experiments

We evaluate `torch-sla` along four axes corresponding to the three components of §3 plus an end-to-end demonstration: scalability across single- and multi-GPU regimes (§4.1), the $\mathcal{O}(1)$ autograd-graph claim of the adjoint framework (§4.2), gradient verification (§4.3), and end-to-end usability through an inverse coefficient-learning task (§4.4).

Table 3. Single-GPU benchmark on 2D Poisson, H200 GPU, `float64`. SciPy and cuDSS are direct solvers; the **CG** column is `torch-sla`'s `pytorch`-native iterative backend (Jacobi-preconditioned), with its peak memory (**Mem.**) and final residual (**Resid.**).

DOF	SciPy	cuDSS	CG	Mem.	Resid.
10K	24 ms	128 ms	20 ms	36 MB	10^{-9}
100K	29 ms	630 ms	43 ms	76 MB	10^{-7}
1M	19.4 s	7.3 s	190 ms	474 MB	10^{-7}
2M	52.9 s	15.6 s	418 ms	916 MB	10^{-7}
16M	OOM	OOM	7.3 s	7.1 GB	10^{-6}
169M	OOM	OOM	224 s	74.8 GB	10^{-6}

Table 4. Distributed CG on H200 GPUs with NCCL, fixed 1000-iteration budget (Jacobi-preconditioned). The residual (**Resid.**) is the state after the budget, not convergence (see text and §5).

DOF	GPUs	Time	Mem./GPU	Resid.
100M	4	36.1 s	23.3 GB	1.0×10^{-2}
200M	3	120 s	53.7 GB	1.5×10^{-2}
300M	3	217 s	80.5 GB	1.9×10^{-2}
400M	3	331 s	110 GB	2.3×10^{-2}

4.1. Scalability

Single-GPU scalability. Table 3 compares solver backends across problem sizes spanning five orders of magnitude. Three regimes emerge along the scale axis. Below 100K DOF, direct solvers dominate: SciPy SuperLU reaches machine precision (10^{-14}) in 24 ms, while GPU launch overhead leaves cuDSS at 128 ms. Between 100K and 2M DOF, iterative solvers become competitive in runtime, while direct solvers remain valuable when near-machine-precision accuracy is required. At 1M DOF, the `torch-sla` CG backend (190 ms) outperforms cuDSS (7.3 s) by 38×. Above 2M DOF, direct solvers run out of memory while CG's near-linear scaling extends to 169M DOF within 74.8 GB. Fitting $T = c \cdot n^\alpha$ to the CG measurements gives $\alpha \approx 1.1$, consistent with the theoretical $\mathcal{O}(\sqrt{\kappa} \cdot \text{nnz})$ iteration cost and condition number $\kappa \sim n$ for 2D Poisson under Jacobi preconditioning (Trefethen & Bau, 1997); the measured 443 bytes/DOF is roughly 3× a minimal-storage estimate, broken down in Appendix A.

Multi-GPU scaling. Table 4 summarizes distributed CG performance on multiple H200 GPUs with the NCCL backend. This experiment demonstrates *memory capacity and per-iteration throughput* of the distributed forward/backward path, not solver convergence: under a fixed 1000-iteration budget with only Jacobi preconditioning, the residual at 10^8–4×10^8 DOF stays in the 10^{-2} range, far from a converged tolerance. Reaching a meaningful tolerance at this scale needs a stronger preconditioner (e.g. algebraic multigrid via AmgX/hypre), which we leave to future work (§5). In memory and throughput, `torch-sla` scales to

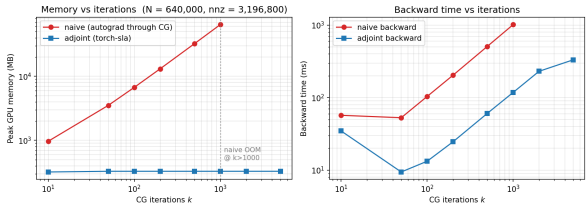

Figure 2. Adjoint vs. naive backprop through k CG iterations (2D Poisson, $N = 640{,}000$; RTX PRO 6000, `float64`). **Left**: adjoint memory is flat ($\sim$ 330 MB); naive grows linearly and OOMs at $k \geq 2000$. **Right**: naive backward time grows linearly in k, while the adjoint path is dominated by the constant backward-solve cost T_{solve} (its mild 35–117 ms growth over $k = 10$–1000 is that solve run to the same k, not a k-dependent $\mathcal{O}(1)$ graph; cf. Table 2).

400M DOF on 3 H200 GPUs. From 1M to 100M DOF time scales near-linearly ($T \propto n^{1.05}$), reaching 2.8M DOF/s—near the aggregate bandwidth of four H200s. Above 100M DOF we use 3 GPUs (a node-allocation constraint, not memory: per-GPU peak is 110 GB of the 140 GB H200), sustaining 1.2M DOF/s. Per-GPU memory reaches 275 B/DOF and scales as $\mathcal{O}(n/P + |\mathcal{H}_p|)$, $|\mathcal{H}_p| \sim \mathcal{O}(\sqrt{n/P})$ on 2D grids, as predicted in §3.3.

4.2. Adjoint vs. Naive Backpropagation

This experiment directly tests the central claim of §3.2: that the adjoint backward pass produces an $\mathcal{O}(1)$ autograd graph regardless of forward iteration count, whereas naive backpropagation through CG iterations produces an $\mathcal{O}(k)$-node graph that grows linearly in memory and time. We solve the same 2D Poisson problem with two paths through the same `pytorch`-native CG forward kernel: **naive**, a manual CG implemented entirely in autograd-tracked PyTorch ops so that every iteration adds nodes to the graph, and **adjoint**, the default `torch-sla` path which wraps the solve in a `torch.autograd.Function` and computes gradients via Eq. (2). Both paths use vanilla unpreconditioned CG forced to run exactly k iterations, and we sweep $k \in \{10, 50, 100, 200, 500, 1000, 2000, 5000\}$. To isolate the autograd-graph cost from the SpMV implementation, the naive baseline uses a hand-coded scatter-based SpMV (`val · x[col]` followed by `index_add`); the otherwise tempting `torch.sparse.mm` produces a *dense* backward gradient with respect to the matrix values and OOMs at $k = 1$ on this problem size. In this experiment, we use a single NVIDIA RTX PRO 6000 Blackwell (96 GB).

Results. The naive path materializes $\sim$64 MB of autograd-tracked intermediates per CG iteration—two nnz-sized tensors from the scatter-based SpMV ($\sim$ 51 MB at nnz $= 3.2 \times 10^6$) plus a handful of N-vectors from the Krylov recurrence ($\sim$ 13 MB)—growing to 64.1 GB at $k = 1000$ and OOM-ing at $k = 2000$ on the 96 GB device, where it would need $\approx$128 GB of intermediates. The adjoint path holds steady at $\sim$328 MB across the entire sweep—a near-perfect slope-one line for naive versus a flat line for adjoint on the log-log left panel of Figure 2—in agreement with the $\mathcal{O}(n + \text{nnz})$ floor of Table 2. At $k = 1000$ this is a $195\times$ memory reduction. On the time axis (right panel), naive backward grows linearly with k at roughly 1 ms per iteration and is $4-9\times$ slower than adjoint across the regime where naive runs at all ($k = 100 \ldots 1000$). On a smaller version of the same 2D Poisson problem as in §4.2 (identical five-point stencil and homogeneous Dirichlet boundary convention, $n_{\text{grid}} = 64$, $N = n_{\text{grid}}^2 = 4096$ interior unknowns), where both paths can be run to full convergence (atol $= 10^{-12}$, $k = 3000$), the loss values agree to machine precision (1.96×10^{-16} relative error)and to $10^{-14} - 10^{-4}$ on gradients with respect to $\mathbf{b}$ and $\mathbf{A}$. Together, these results validate both halves of the central claim of §3.2: the autograd graph is $\mathcal{O}(1)$ in k, and the resulting gradients are correct. Appendix D reports the per-k computational cost and correctness analysis.

Table 5. Gradient verification for the nonlinear and eigenvalue paths vs. central finite differences (FD); forward/backward cost in units of forward operations.

Operation	Rel. err.	Fwd	Bwd
Eigenvalue (k=6)†	2.1×10^{-6}	1 LOBPCG	outer prod.
Nonlinear (5 Newton)	4.7×10^{-7}	5 solves	1 solve

4.3. Gradient Verification

In §4.2, we verify the linear adjoint gradients against naive autograd; in this section, we extend the verification to the nonlinear and eigenvalue cases of §3.2.2 by comparing adjoint gradients against centered finite differences,

$$\frac{\partial \mathcal{L}}{\partial \theta} \approx \frac{\mathcal{L}(\theta + \epsilon) - \mathcal{L}(\theta - \epsilon)}{2\epsilon}, \quad \epsilon = 10^{-5}, \tag{7}$$

on randomly perturbed entries of $\mathbf{A}$ and $\mathbf{b}$.

† LOBPCG and Lanczos eigensolvers are exposed by `torch-sla`'s `.eigsh` entry point on top of the iterative backends; the autograd wrapper itself is independent of the eigensolver choice and applies the Hellmann–Feynman gradient of Eq. (4) once at convergence.

Table 5 shows relative errors below 10^{-5} for both paths. The nonlinear backward pass runs a single adjoint solve regardless of the five Newton iterations performed in the forward; the eigenvalue gradient uses Eq. (4) directly and requires only an $\mathcal{O}(\text{nnz})$ outer-product evaluation, with no additional linear solves.

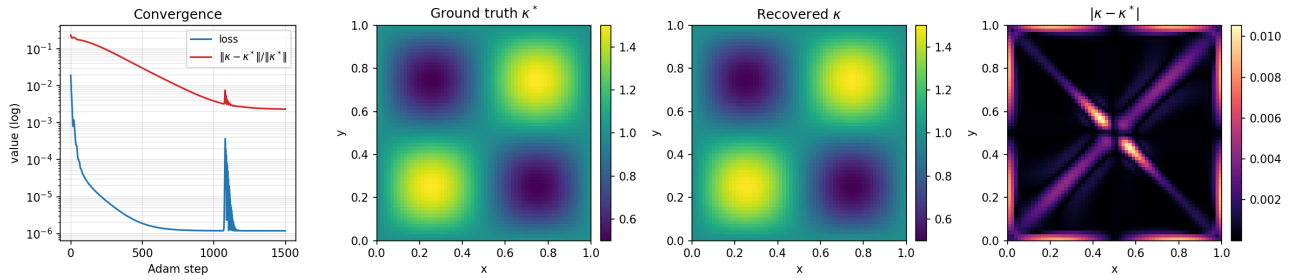

Figure 3. Inverse coefficient learning on the variable-coefficient Poisson equation, 64×64 grid (3,844 unknowns). **Left:** loss and relative L^2 error $\|\kappa - \kappa^*\|_2/\|\kappa^*\|_2$ per Adam step (log scale), both decreasing monotonically over 1500 steps. **Center:** ground-truth κ^* and recovered κ (shared color scale). **Right:** pointwise error $|\kappa - \kappa^*|$, below 1.1×10^{-2} everywhere and largest near the boundary and diagonals where ∇u is small.

4.4. End-to-End: Inverse Coefficient Learning

To demonstrate that gradients flow through `torch-sla` solves inside a realistic training loop, we consider an inverse problem on the variable-coefficient Poisson equation

$$\begin{aligned} -\nabla \cdot \big(\kappa(\mathbf{x})\nabla u(\mathbf{x})\big) &= f(\mathbf{x}) \quad \text{on } (0,1)^2, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{8}$$

with ground-truth conductivity $\kappa^*(x, y) = 1 + 0.5 \sin(2\pi x) \sin(2\pi y)$ and $f \equiv 1$. We discretize with finite differences on a 64×64 grid, generate observed solutions u_{obs} by a forward solve with κ^*, and then learn κ from u_{obs} alone.

We parametrize $\kappa = \text{softplus}(\theta)$ with $\theta \in \mathbb{R}^{64\times 64}$ a single `torch.nn.Parameter` (softplus enforcing $\kappa > 0$). At every step we assemble the cell-centered five-point discretization of $-\nabla \cdot (\kappa \nabla u) = f$ as a `torch-sla SparseTensor`, solve $A(\kappa)\, u_{\text{pred}} = f$ via `A.solve(f)`, and update θ with Adam (Kingma & Ba, 2015) (lr $= 5 \times 10^{-2}$) on $\|u_{\text{pred}} - u_{\text{obs}}\|_2^2$ plus a Tikhonov (Tikhonov & Arsenin, 1977) smoothness regularizer $10^{-3} \cdot \|\nabla_h \kappa\|_2^2/N$. Gradients with respect to κ flow through the solve via the adjoint path of §3.2, with no custom `autograd.Function` written at the user level.

Results. Figure 3 shows the optimization trajectory and recovered coefficient field. After 1500 Adam steps (48.6 s on a single NVIDIA RTX PRO 6000 Blackwell, ∼32 ms/step), the recovered conductivity satisfies $\|\kappa - \kappa^*\|_2/\|\kappa^*\|_2 = 2.3 \times 10^{-3}$ and the corresponding forward solution satisfies $\|u(\kappa) - u_{\text{obs}}\|_2/\|u_{\text{obs}}\|_2 = 3.0 \times 10^{-5}$. The recovered field stays in $[0.503, 1.495]$, a tight match to the ground-truth range $[0.5, 1.5]$. Recovering κ to 0.23% relative error in under a minute confirms that the adjoint path of §3.2 delivers usable, well-scaled gradients inside a standard PyTorch `Adam` loop—the only line of code specific to differentiable sparse linear algebra is `A.solve(f)`.

5. Conclusion

We presented `torch-sla`, an open-source PyTorch library for differentiable sparse linear algebra that closes a long-standing ecosystem gap: a single autograd-aware API spanning direct, iterative, nonlinear, and eigenvalue solvers across five interchangeable backends (SciPy, Eigen, cuDSS, CuPy, PyTorch-native), with batched dispatch through the `SparseTensor`/`SparseTensorList` hierarchy and distributed multi-GPU execution through `DSparseTensor`. A unified IFT adjoint keeps the autograd graph at $\mathcal{O}(1)$ nodes regardless of iterations or backend, and an autograd-compatible distributed layer composes domain decomposition with PyTorch autograd via a transposed halo exchange. On 2D Poisson with H200 GPUs, `torch-sla` reaches 169M DOF on one GPU and 400M across three, the adjoint backward gives a $195\times$ memory reduction at 1000 CG iterations, and an inverse coefficient-learning task recovers a varying conductivity to 0.23% relative L^2 error in under a minute—the only solver-specific line being `A.solve(f)`.

Limitations and future work. The `pytorch`-native iterative backend currently supports only Jacobi preconditioning, insufficient at large DOF—hence the 10^{-2} residuals in our multi-GPU runs under a fixed 1000-iteration budget—and our benchmarks focus on 2D Poisson; broader validation on 3D PDEs, indefinite systems, and GNN graph-Laplacians remains future work. The backend abstraction (§3.1) is designed for two extensions: wrapping mature distributed-sparse libraries (PETSc (Balay et al., 2023), Trilinos (Mayr et al., 2026), hypre (Falgout & Yang, 2002), AmgX (Naumov et al., 2015)) as backends, and registering *learned* preconditioners (learned CG (Li et al., 2023), graph-neural (Chen, 2024), and multigrid-inspired neural solvers (Lyu et al., 2026)) trained end-to-end against full sparse solves—making `torch-sla` a substrate for *learnable* sparse solvers at scale, not merely a faster solver.

References

Anderson, D. G. Iterative procedures for nonlinear integral equations. *Journal of the ACM*, 12(4):547–560, 1965.

Balay, S., Abhyankar, S., Adams, M. F., et al. Petsc users manual. Technical report, Argonne National Laboratory, 2023. Version 3.20.

Berger, M. J. and Bokhari, S. H. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580, 1987.

Blondel, M., Berthet, Q., Cuturi, M., et al. Efficient and modular implicit differentiation. *Advances in Neural Information Processing Systems*, 35:5230–5242, 2022.

Bradbury, J., Frostig, R., Hawkins, P., et al. JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax, 2018.

Brandstetter, J., Worrall, D., and Welling, M. Message passing neural pde solvers. *International Conference on Learning Representations*, 2022.

Chen, J. Graph neural preconditioners for iterative solutions of sparse linear systems. *arXiv preprint arXiv:2406.00809*, 2024.

Davis, T. A. Algorithm 832: UMFPACK v4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.

Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S., and Liu, J. W. H. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

Falgout, R. D. and Yang, U. M. hypre: A library of high performance preconditioners. In *International Conference on Computational Science*, pp. 632–641, 2002.

Fey, M. and Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

George, A. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

Guennebaud, G., Jacob, B., et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

Hestenes, M. R. and Stiefel, E. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

Hoemmen, M. *Communication-avoiding Krylov subspace methods*. PhD thesis, University of California, Berkeley, 2010.

Holl, P. and Thuerey, N. Φ_{flow} (PhiFlow): Differentiable simulations for pytorch, tensorflow and jax. In *International Conference on Machine Learning*. PMLR, 2024.

Hu, Y., Anderson, L., Li, T.-M., et al. Difftaichi: Differentiable programming for physical simulation. In *International Conference on Learning Representations*, 2020.

Jasak, H., Jemcov, A., and Tukovic, Z. Openfoam: A c++ library for complex physics simulations. *International Workshop on Coupled Methods in Numerical Dynamics*, 2007.

Karypis, G. and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

Kelley, C. T. *Iterative Methods for Linear and Nonlinear Equations*. SIAM, 1995.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

Knyazev, A. V. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM Journal on Scientific Computing*, 23(2):517–541, 2001.

Krantz, S. G. and Parks, H. R. *The Implicit Function Theorem: History, Theory, and Applications*. Birkhäuser, 2002.

Li, Y., Chen, P. Y., Du, T., and Matusik, W. Learning preconditioners for conjugate gradient pde solvers. In *International Conference on Machine Learning*, pp. 19425–19439. PMLR, 2023.

Li, Z., Kovachki, N., Azizzadenesheli, K., et al. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

Lyu, K., Cen, R., Wu, Y., and Du, T. A multigrid-inspired neural iterative solver for poisson equations on large voxel grids. In *AI&PDE: ICLR 2026 Workshop on AI and Partial Differential Equations*, 2026. URL https://openreview.net/forum?id=lNcbGSWhJo.

Magnus, J. R. On differentiating eigenvalues and eigenvectors. *Econometric Theory*, 1(2):179–191, 1985.

Mayr, M., Heinlein, A., Glusa, C. A., Rajamanickam, S., Arnst, M., Bartlett, R. A., Berger-Vergiat, L., Boman, E. G., Devine, K. D., Harper, G., Heroux, M. A., Hoemmen, M., Hu, J. J., Kelley, B., Kouri, D. P., Kuberry, P.,

Kim, K., Liegois, K., Ober, C. C., Pawlowski, R. P., Pearson, C., Perego, M., Phipps, E. T., Ridzal, D., Roberts, N. V., Siefert, C. M., Thornquist, H. K., Tomasetti, R., Trott, C. R., Tuminaro, R. S., Willenbring, J. M., Wolf, M., and Yamazaki, I. Trilinos: Enabling Scientific Computing across Diverse Hardware Architectures at Scale. *ACM Transactions on Mathematical Software*, published online ahead of print, 2026. doi: 10.1145/3802822. URL https://doi.org/10.1145/3802822.

Mitusch, S. K., Funke, S. W., and Dokken, J. S. dolfin-adjoint 2018.1: automated adjoints for FEniCS and Firedrake. *Journal of Open Source Software*, 4(38):1292, 2019.

Naumov, M., Arsaev, M., Castonguay, P., Cohen, J., Demouth, J., Eaton, J., Laber, S., Laptev, I., Stam, N., and Temam, O. AmgX: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 37(5):S602–S626, 2015.

NVIDIA Corporation. NVIDIA collective communications library (NCCL). https://developer.nvidia.com/nccl, 2020.

NVIDIA Corporation. cuDSS: NVIDIA CUDA direct sparse solver library. https://developer.nvidia.com/cudss, 2024.

Okuta, R., Unno, Y., Nishino, D., Hido, S., and Loomis, C. CuPy: A NumPy-compatible library for NVIDIA GPU calculations. In *Workshop on Machine Learning Systems (LearningSys) at NeurIPS*, 2017.

Pineda, L., Fan, T., Monge, M., Venkataraman, S., Sodhi, P., Chen, R. T. Q., Ortiz, J., DeTone, D., Wang, A., Anderson, S., Dong, J., Amos, B., and Mukadam, M. Theseus: A library for differentiable nonlinear optimization. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Potapczynski, A., Finzi, M., Pleiss, G., and Wilson, A. G. CoLA: Exploiting compositional structure for automatic and efficient numerical linear algebra. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Rader, J., Lyons, T., and Kidger, P. Lineax: unified linear solves and linear least-squares in jax and equinox. *AI for science workshop at Neural Information Processing Systems 2023, arXiv:2311.17283*, 2023.

Rader, J., Lyons, T., and Kidger, P. Optimistix: modular optimisation in JAX and Equinox. *arXiv preprint arXiv:2402.09983*, 2024.

Shi, Y., Ross, Z. E., Asimaki, D., and Azizzadenesheli, K. Mesh-informed neural operator: A transformer generative approach. *arXiv preprint arXiv:2506.16656*, 2025.

Tikhonov, A. N. and Arsenin, V. Y. *Solutions of Ill-Posed Problems*. Winston & Sons, Washington, D.C., 1977.

Trefethen, L. N. and Bau, D. *Numerical Linear Algebra*. SIAM, 1997.

Van der Vorst, H. A. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. Graph attention networks. *International Conference on Learning Representations*, 2018.

Virtanen, P., Gommers, R., Oliphant, T. E., et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature Methods*, 17(3):261–272, 2020.

Wen, S., Kumbhat, A., Lingsch, L., Mousavi, S., Zhao, Y., Chandrashekar, P., and Mishra, S. Geometry aware operator transformer as an efficient and accurate neural surrogate for PDEs on arbitrary domains. *arXiv preprint arXiv:2505.18781*, 2025.

A. Backend Method Coverage

This appendix expands the backend abstraction described in §3.1. Table 6 lists the concrete solver families exposed through each backend and the problem regimes for which the auto-dispatch policy selects them by default.

Table 6. Backends supported by `torch-sla`. The library auto-selects a backend by device, problem size, and matrix properties (symmetry and positive-definiteness are detected automatically); users can override the choice through a single keyword argument. We list the principal solver methods exposed by each backend; additional Krylov variants (e.g. GMRES, LGMRES, MINRES, QMR, LSQR) are wrapped where the underlying library provides them. `torch-sla` does not reimplement these solvers—it provides a uniform autograd-aware adapter on top of established backends.

Backend	Device	Solver methods	Recommended for
`scipy`	CPU	SuperLU, UMFPACK, CG, BiCGStab	CPU default; machine precision
`eigen`	CPU	CG, BiCGStab	Alternative CPU iterative
`cudss`	CUDA	LU, Cholesky, LDLT	CUDA default; fastest direct ($<$2M DOF)
`cupy`	CUDA	LU, CG, GMRES	GPU direct + iterative via `cupyx`
`pytorch`	CUDA	CG, BiCGStab	Very large problems ($>$2M DOF)

Software versions. For reproducibility—the auto-dispatch heuristics and especially the cuDSS API are version-sensitive—we report the backend versions used for all experiments: PyTorch 2.10.0 (CUDA 12.8), NVIDIA cuDSS 0.7.1 (`nvidia-cudss-cu12`), CuPy 14.0.1, SciPy 1.15.3, and NumPy 2.4.4. Multi-GPU runs use the NCCL backend bundled with the above PyTorch build.

Single-GPU memory breakdown. The 443 bytes/DOF reported for the `pytorch`-native CG backend in Table 3 is $\approx 3\times$ a minimal-storage estimate of ≈ 150 B/DOF: the COO matrix at ~ 5 non-zeros/row costs 24 B/nnz (an `int64` row and column index plus a `float64` value), and the Jacobi diagonal and the CG work vectors ($\mathbf{x}, \mathbf{r}, \mathbf{p}, \mathbf{Ap}$) add a few 8-B/DOF vectors. The remaining $\sim 2\times$ is Krylov temporaries, autograd-saved forward tensors, and CUDA allocator fragmentation.

B. API Examples

Listing 1 shows the user-facing API surfaces used throughout the paper. The examples emphasize that single solves, backend overrides, batched solves, nonlinear solves, and distributed solves all preserve the same PyTorch autograd interface: users call a solver method in the forward pass and receive adjoint gradients through ordinary `loss.backward()`.

Listing 1. `torch-sla` API across single, batched, and distributed solves. Gradients flow through every variant via the same adjoint path.

```python
import torch
from torch_sla import SparseTensor, nonlinear_solve

# 1. Single solve with auto-dispatched backend
val = torch.randn(nnz, requires_grad=True)
A = SparseTensor(val, row, col, shape=(n, n))
b = torch.randn(n, requires_grad=True)
x = A.solve(b)                    # cudss / pytorch / scipy
loss = x.pow(2).sum()
loss.backward()                   # adjoint gradients, O(1) graph

# 2. Explicit backend / method override
x = A.cuda().solve(b.cuda(), backend='cudss', method='cholesky')
x = A.cuda().solve(b.cuda(), backend='pytorch', method='cg')

# 3. Batched solve with shared sparsity pattern
A_batch = SparseTensor(val_batch, row, col, shape=(B, n, n))
x_batch = A_batch.solve(b_batch)  # one symbolic factorization

# 4. Nonlinear solve with adjoint gradients (module-level function)
def residual(u, A_val, f):
    A_local = SparseTensor(A_val, row, col, shape=(n, n))
    return A_local @ u + u**2 - f
u = nonlinear_solve(residual, torch.zeros(n), val, f, method='newton')
```

```python

# 5. Distributed solve via domain decomposition (multi-GPU)
from torch_sla.distributed import DSparseTensor
A_dist = DSparseTensor.from_global(
    val, row, col, shape, num_partitions=world_size,
    my_partition=rank, device=f'cuda:{rank}')
x_dist = A_dist.solve(b_local, atol=1e-10)
```

C. Distributed Conjugate Gradient and Halo Exchange

This section gives the operational details behind the distributed layer of §3.3. Figure 4 illustrates the halo exchange pattern underlying distributed SpMV, while Algorithm 1 shows where halo exchange and global reductions enter the distributed CG loop. Together they make explicit which parts of the computation are local to a partition and which parts require inter-process communication.

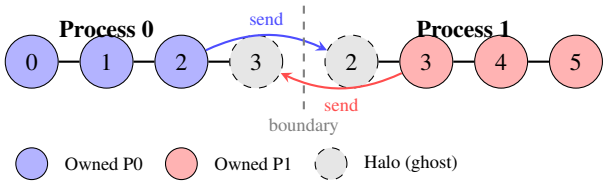

Figure 4. Halo exchange in domain decomposition. Each process owns a subset of nodes (solid colored) and maintains halo copies of boundary neighbors (dashed). Before each distributed SpMV, processes exchange updated values at partition boundaries; the local SpMV then proceeds independently using owned and halo values.

The same halo-exchange primitive appears once per SpMV inside CG. The remaining operations are local vector updates or scalar reductions, so the algorithm follows the standard Krylov recurrence while replacing each matrix-vector product by a distributed SpMV.

Algorithm 1: Distributed Conjugate Gradient in `torch-sla`.
Input: distributed $\mathbf{A}$ (each process holds $\mathbf{A}[\mathcal{O}_p, :]$ and halo metadata), local RHS $\mathbf{b}_{\mathcal{O}_p}$, tolerance ϵ
Output: solution $\mathbf{x}_{\mathcal{O}_p}$

1. $\mathbf{x} \leftarrow \mathbf{0},\ \mathbf{r} \leftarrow \mathbf{b}_{\mathcal{O}_p},\ \mathbf{p} \leftarrow \mathbf{r}$
2. $\rho \leftarrow \texttt{all_reduce}(\mathbf{r}^\top \mathbf{r})$
3. **while** $\sqrt{\rho} > \epsilon$ **do**:
   (a) $\mathbf{Ap} \leftarrow \texttt{DistSpMV}(\mathbf{A}, \mathbf{p})$ *// halo exchange + local SpMV*
   (b) $\alpha \leftarrow \rho \,/\, \texttt{all_reduce}(\mathbf{p}^\top \mathbf{Ap})$
   (c) $\mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{p}$
   (d) $\mathbf{r} \leftarrow \mathbf{r} - \alpha \mathbf{Ap}$
   (e) $\rho_{\text{new}} \leftarrow \texttt{all_reduce}(\mathbf{r}^\top \mathbf{r})$
   (f) $\mathbf{p} \leftarrow \mathbf{r} + (\rho_{\text{new}}/\rho)\,\mathbf{p},\ \rho \leftarrow \rho_{\text{new}}$

The loop issues two `all_reduce` operations per iteration, the standard form. For very large process counts P the latency of these two reductions per iteration becomes the bottleneck; pipelined and communication-avoiding (s-step) CG variants (Hoemmen, 2010) that merge or defer the inner-product reductions are a natural roadmap item, and compose with the same transposed-halo backward pass since they only reorganize the reductions rather than the SpMV.

D. Adjoint vs. Naive: Additional Correctness Analysis

This appendix provides the full sweep behind the memory and timing summary in §4.2. Table 7 reports the measured backward cost at each forced CG iteration count, and the following paragraph explains the small-problem correctness check used to compare adjoint gradients against naive autograd after full convergence.

Table 7. Adjoint vs. naive CG backpropagation, full sweep across $k \in \{10, 50, 100, 200, 500, 1000, 2000, 5000\}$.

k	**Adj. mem**	**Naive mem**	**Adj. bwd**	**Naive bwd**	**Ratio**
10	320 MB	0.97 GB	35 ms	57 ms	3.0×
50	328 MB	3.52 GB	9 ms	53 ms	11×
100	328 MB	6.71 GB	13 ms	104 ms	20×
200	328 MB	13.1 GB	25 ms	204 ms	41×
500	328 MB	32.2 GB	60 ms	508 ms	100×
1000	328 MB	64.1 GB	117 ms	1015 ms	195×
2000	328 MB	OOM	230 ms	—	—
5000	328 MB	OOM	332 ms	—	—

On a smaller problem ($n_{\text{grid}} = 64$, $N = 4096$), where both paths can be run to full convergence (atol $= 10^{-12}$, $k = 3000$), the loss values agree to machine precision (1.96×10^{-16} relative error), the gradient $\partial\mathcal{L}/\partial\mathbf{b}$ matches to 2.6×10^{-14}, and the matrix gradient $\partial\mathcal{L}/\partial\mathbf{A}$ matches to 6.8×10^{-4} relative error. The looser agreement on the matrix gradient reflects floating-point round-off accumulated by naive backpropagation over 3000 iterations of CG recurrences, whereas the adjoint path evaluates the closed-form sparse outer product $-\lambda_i x_j$ once at the converged solution. This explains why the adjoint method remains numerically robust at high iteration counts, in addition to reducing memory and time.

E. Extended Comparison with Related Libraries

Table 1 and §2 place `torch-sla` among differentiable linear-algebra libraries; here we expand the two largely orthogonal axes that most cleanly separate them.

Operator representation: explicit vs. implicit. `torch-sla` stores an *explicit* sparse matrix (`val`/`row`/`col`), which is exactly what heavy sparse *direct* factorizations (cuDSS LU/Cholesky/LDLT) and domain-decomposition partitioners consume. CoLA instead composes *implicit*, matrix-free `LinearOperators`: it can factor $\mathbf{A}\otimes$ and apply matrix functions $f(\mathbf{A})$ efficiently, but it does not expose the non-zero pattern a sparse direct solver or an ILU/AMG preconditioner needs. Consequently CoLA reaches `logdet`, `trace`, and $f(\mathbf{A})$ on structured operators that `torch-sla` does not target, while `torch-sla` reaches GPU sparse-direct factorization and tensor-parallel distribution that CoLA does not.

Structure exploited: sparsity vs. algebraic composition. `torch-sla` and PhiFlow exploit the *sparsity pattern*; CoLA exploits *algebraic* composition (Kronecker, block, low-rank, FFT). The two are orthogonal, and the choice dictates the admissible preconditioners: a pure-matvec black box admits only Jacobi/polynomial preconditioners, whereas ILU/IC/AMG need the explicit non-zeros. This is why PhiFlow materializes its operators (via `jit_compile_linear`) precisely to enable ILU, and why `torch-sla`'s explicit representation makes the AmgX/PETSc backends on its roadmap natural.

Backend reach. Theseus's sparse solvers (CHOLMOD, cudaLU, BaSpaCho) are internal to its nonlinear-least-squares optimizer rather than a general-purpose API. PhiFlow's Krylov solvers are backend-agnostic (JAX/PyTorch/TF) but offer no GPU sparse-direct path (its direct solve routes to CPU SciPy/SuperLU). In JAX, the capability `torch-sla` exposes in one library corresponds to the Lineax + JAXopt + Optimistix trio together; `torch-sla`'s contribution is to unify these axes in PyTorch with a GPU sparse-direct backend and a tensor-parallel distributed layer.