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

Mingyuan Chi

`walker.chi.000@gmail.com`
`https://github.com/walkerchi/torch-sla`

January 15, 2026

**Abstract**

We present `torch-sla`, an open-source PyTorch library for differentiable sparse linear algebra. The library addresses two fundamental challenges: (1) **Adjoint-based differentiation** for linear and nonlinear sparse solvers, achieving $\mathcal{O}(1)$ memory complexity for computational graphs regardless of solver iterations; and (2) **Sparse Tensor Parallel** computing via domain decomposition with halo exchange, enabling distributed sparse operations across multiple GPUs. `torch-sla` supports multiple backends (SciPy, cuDSS, PyTorch-native) and scales to 169 million degrees of freedom. Benchmarks demonstrate $12\times$ speedup on multi-GPU configurations and correct gradient computation verified against finite differences.

## 1 Introduction

Sparse linear systems $\mathbf{Ax} = \mathbf{b}$ are fundamental to scientific computing. They arise in finite element analysis [Hughes, 2012], graph neural networks [Kipf and Welling, 2017, Veličković et al., 2018], physics-informed machine learning [Raissi et al., 2019, Lu et al., 2021], and computational fluid dynamics [Jasak et al., 2007, Kochkov et al., 2021]. With the rise of differentiable programming and neural operators [Li et al., 2020, Brandstetter et al., 2022], there is growing demand for sparse solvers that integrate with automatic differentiation frameworks like PyTorch [Paszke et al., 2019].

**The challenge of differentiating through solvers.** Consider a loss function $\mathcal{L}(\mathbf{x})$ where $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ is obtained by solving a linear system. To train neural networks that interact with such solvers, we need gradients $\partial\mathcal{L}/\partial\mathbf{A}$ and $\partial\mathcal{L}/\partial\mathbf{b}$. A naive approach—differentiating through each iteration of an iterative solver—creates $\mathcal{O}(k)$ nodes in the computational graph, where $k$ may be thousands of iterations. This leads to memory explosion and slow backward passes.

**The challenge of scale.** Industrial problems often exceed single-GPU memory. A 3D finite element mesh with 10 million nodes produces a sparse matrix requiring tens of gigabytes. Distributed computing is essential, but sparse matrix operations require careful communication patterns (halo exchange) that are non-trivial to implement correctly with gradient support.

This paper introduces `torch-sla`, addressing both challenges with two key innovations:

- **Adjoint-based differentiation** (§3.2): We implement the adjoint method for linear solves, eigenvalue problems, and nonlinear systems. This achieves $\mathcal{O}(1)$ computational graph nodes and $\mathcal{O}(\text{nnz})$ memory, independent of solver iterations.
- **Sparse tensor parallelism** (§3.3): We implement domain decomposition with automatic halo exchange following industrial CFD/FEM practices. This enables multi-GPU computing with gradient support.

## 2 Related Work

**Differentiable linear algebra.** JAX [Bradbury et al., 2018] provides differentiable sparse CG, but naive differentiation through iterations creates $\mathcal{O}(k)$ graph nodes. Blondel et al. [2022] formalize implicit differentiation for optimization layers. OptNet [Amos and Kolter, 2017] and cvxpylayers [Agrawal et al., 2019] handle convex optimization. Deep equilibrium models [Bai et al., 2019] and Jacobian-free backpropagation [Fung et al., 2022] address implicit layers but focus on fixed-point iterations rather than sparse linear systems.

**Sparse solvers and GPU acceleration.** SciPy [Virtanen et al., 2020] provides SuperLU/UMFPACK for CPU. For GPU, NVIDIA cuDSS [NVIDIA Corporation, 2024] offers direct solvers, while AmgX [Naumov et al., 2015] provides algebraic multigrid with excellent scalability. Recent surveys [Li et al., 2024] cover sparse matrix computations on modern GPUs. None of these natively support PyTorch autograd.

**Distributed sparse computing.** PETSc [Balay et al., 2023], Trilinos [Heroux et al., 2005], and hypre [Falgout and Yang, 2002] implement distributed sparse linear algebra with sophisticated preconditioners. OpenFOAM [Jasak et al., 2007] uses domain decomposition for CFD. We bring these industrial patterns to PyTorch with automatic differentiation.

**Learned solvers and preconditioners.** Recent work explores learning components of iterative solvers: learned multigrid prolongation [Greenfeld et al., 2019], GNN-based AMG [Luz et al., 2020], and learned interface conditions for domain decomposition [Taghibakhshi et al., 2024]. These complement our library by providing learnable components that can be trained end-to-end.

## 3 Methodology

We first introduce the background on sparse linear systems (§3.1), then present our adjoint-based differentiation approach (§3.2), and finally describe our distributed computing implementation (§3.3).

### 3.1 Preliminaries: Sparse Linear Systems

A sparse linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ has a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ with nnz $\ll n^2$ non-zero entries. We store matrices in coordinate (COO) format: three arrays for row indices, column indices, and values.

**Direct solvers** (LU, Cholesky factorization) compute exact solutions but require $\mathcal{O}(n^{1.5})$ memory for 2D problems due to fill-in—new non-zeros created during factorization [George, 1973]. For a 2D Poisson problem with $n$ unknowns, the factored matrix has $\mathcal{O}(n \log n)$ non-zeros instead of the original $\mathcal{O}(n)$.

**Iterative solvers** (Conjugate Gradient, BiCGStab, GMRES) maintain $\mathcal{O}(\text{nnz})$ memory by never forming the factorization. However, they require $\mathcal{O}(\sqrt{\kappa})$ iterations for CG where $\kappa$ is the condition number [Saad, 2003]. Each iteration performs one sparse matrix-vector multiplication (SpMV) costing $\mathcal{O}(\text{nnz})$.

**The differentiation problem.** Given loss $\mathcal{L}(\mathbf{x})$ where $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, we need gradients with respect to both $\mathbf{b}$ and the non-zero values of $\mathbf{A}$. If we differentiate through $k$ iterations of CG, the computational graph has $\mathcal{O}(k)$ nodes, each storing intermediate vectors. For $k = 1000$ iterations and $n = 10^6$, this requires ~80 GB just for the graph. Our adjoint approach reduces this to $\mathcal{O}(1)$ nodes.

### 3.2 Adjoint-Based Differentiation

The adjoint method computes gradients through implicit functions without storing intermediate solver states. We present it for linear systems, eigenvalue problems, and nonlinear systems.

#### *3.2.1 Linear Systems: The Core Idea*

Consider $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Rather than differentiating through the solver iterations, we use the *implicit function theorem*. The solution satisfies $\mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}$. Differentiating implicitly:

$$d\mathbf{A} \cdot \mathbf{x} + \mathbf{A} \cdot d\mathbf{x} = d\mathbf{b} \implies d\mathbf{x} = \mathbf{A}^{-1}(d\mathbf{b} - d\mathbf{A} \cdot \mathbf{x}) \tag{1}$$

**Table 1:** Complexity comparison: naive vs. adjoint differentiation through iterative solver with $k$ iterations, $n$ unknowns, and nnz non-zeros.

| | **Naive (through iterations)** | **Adjoint (ours)** |
|---|---|---|
| Computational graph nodes | $\mathcal{O}(k)$ | $\mathcal{O}(1)$ |
| Memory for graph | $\mathcal{O}(k \cdot n)$ | $\mathcal{O}(n + \text{nnz})$ |
| Backward pass time | $\mathcal{O}(k \cdot \text{nnz})$ | $\mathcal{O}(T_{\text{solve}} + \text{nnz})$ |

For a scalar loss $\mathcal{L}(\mathbf{x})$, the chain rule gives:

$$d\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{x}}\right)^{\top} d\mathbf{x} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{x}}\right)^{\top} \mathbf{A}^{-1}(d\mathbf{b} - d\mathbf{A} \cdot \mathbf{x}) \tag{2}$$

Define the **adjoint variable** $\boldsymbol{\lambda} = \mathbf{A}^{-\top} \frac{\partial \mathcal{L}}{\partial \mathbf{x}}$, solved via $\mathbf{A}^{\top}\boldsymbol{\lambda} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}}$. Then the gradients are:

$$\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \boldsymbol{\lambda}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{A}_{ij}} = -\boldsymbol{\lambda}_i \cdot \mathbf{x}_j} \tag{3}$$

**Complexity analysis.** Table 1 compares naive differentiation (through iterations) with our adjoint approach.

The forward solve computes $\mathbf{x}$ in time $T_{\text{solve}}$. The backward pass:

1. Solves $\mathbf{A}^{\top}\boldsymbol{\lambda} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}}$: $\mathcal{O}(T_{\text{solve}})$ time
2. Computes $\frac{\partial \mathcal{L}}{\partial \mathbf{A}_{ij}} = -\boldsymbol{\lambda}_i \mathbf{x}_j$ for each non-zero: $\mathcal{O}(\text{nnz})$ time

**Algorithm.** The complete procedure is shown below:

**Algorithm 1: Adjoint Linear Solve**

```
Input: Sparse matrix $\mathbf{A}$ (values, rows, cols), RHS $\mathbf{b}$
Output: Solution $\mathbf{x}$, gradients in backward pass

  Forward pass:
    1. $\mathbf{x} \leftarrow \text{solve}(\mathbf{A}, \mathbf{b})$                // Any solver: CG, LU, etc.
    2. Store $(\mathbf{A}, \mathbf{x})$ for backward

  Backward pass: (given $\partial\mathcal{L}/\partial\mathbf{x}$)
    1. $\boldsymbol{\lambda} \leftarrow \text{solve}(\mathbf{A}^{\top}, \partial\mathcal{L}/\partial\mathbf{x})$    // One adjoint solve
    2. $\partial\mathcal{L}/\partial\mathbf{b} \leftarrow \boldsymbol{\lambda}$
    3. For each non-zero $(i, j)$: $\partial\mathcal{L}/\partial\mathbf{A}_{ij} \leftarrow -\boldsymbol{\lambda}_i \cdot \mathbf{x}_j$    // $\mathcal{O}$(nnz) total
```

### *3.2.2 Eigenvalue Problems*

For symmetric eigenvalue problem $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ with normalized eigenvector $\|\mathbf{v}\| = 1$, the gradient of eigenvalue $\lambda$ with respect to matrix entries is remarkably simple [Magnus, 1985]:

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

This is the outer product $\mathbf{v}\mathbf{v}^{\top}$ restricted to the sparsity pattern. For $k$ eigenvalues, the total cost is $\mathcal{O}(k \cdot \text{nnz})$.

### *3.2.3 Nonlinear Systems*

For nonlinear system $\mathbf{F}(\mathbf{u}, \boldsymbol{\theta}) = \mathbf{0}$ with solution $\mathbf{u}^*(\boldsymbol{\theta})$ and loss $\mathcal{L}(\mathbf{u}^*)$, implicit differentiation gives:

$$\mathbf{J}^\top \boldsymbol{\lambda} = \frac{\partial \mathcal{L}}{\partial \mathbf{u}}, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = -\boldsymbol{\lambda}^\top \frac{\partial \mathbf{F}}{\partial \boldsymbol{\theta}} \tag{5}$$

where $\mathbf{J} = \frac{\partial \mathbf{F}}{\partial \mathbf{u}}$ is the Jacobian at the solution.

**Algorithm 2: Newton-Raphson with Adjoint Gradients**

```
Input: Residual function $\mathbf{F}(\mathbf{u}, \boldsymbol{\theta})$, initial guess $\mathbf{u}_0$
Output: Solution $\mathbf{u}^*$

  Forward (Newton iteration):

  1. $\mathbf{u} \leftarrow \mathbf{u}_0$
  2. while $\|\mathbf{F}(\mathbf{u}, \boldsymbol{\theta})\| > \epsilon$ do:
       (a) $\mathbf{J} \leftarrow \partial \mathbf{F} / \partial \mathbf{u}$          // Jacobian via autograd
       (b) $\Delta \mathbf{u} \leftarrow$ solve$(\mathbf{J}, -\mathbf{F})$
       (c) $\mathbf{u} \leftarrow \mathbf{u} + \alpha \Delta \mathbf{u}$          // $\alpha$ from line search
  3. Store $(\mathbf{u}^*, \mathbf{J}, \boldsymbol{\theta})$

  Backward: (given $\partial \mathcal{L} / \partial \mathbf{u}^*$)

  1. $\boldsymbol{\lambda} \leftarrow$ solve$(\mathbf{J}^\top, \partial \mathcal{L} / \partial \mathbf{u}^*)$          // One adjoint solve
  2. $\partial \mathcal{L} / \partial \boldsymbol{\theta} \leftarrow -\boldsymbol{\lambda}^\top \partial \mathbf{F} / \partial \boldsymbol{\theta}$          // VJP via autograd
```

**Memory analysis.** Forward: $\mathcal{O}(n + \text{nnz})$ for solution and Jacobian. Backward: $\mathcal{O}(n)$ for adjoint variable. Total: $\mathcal{O}(n + \text{nnz})$, independent of Newton iterations.

## 3.3 Sparse Tensor Parallel Computing

For problems exceeding single-GPU memory, we implement domain decomposition with halo exchange—the standard approach in industrial CFD/FEM codes like OpenFOAM and Ansys Fluent.

### *3.3.1 Domain Decomposition*

Given a sparse matrix $\mathbf{A}$ corresponding to a mesh or graph, we partition the $n$ nodes into $P$ subdomains. Each process $p$ owns nodes $\mathcal{O}_p$ (owned nodes) and maintains copies of neighboring nodes $\mathcal{H}_p$ (halo/ghost nodes) needed for local computation.

**Partitioning strategies:**

- **METIS** [Karypis and Kumar, 1998]: Graph partitioning minimizing edge cuts for load balancing
- **RCB**: Recursive Coordinate Bisection using node coordinates
- **Contiguous**: Simple row-based partitioning (fallback)

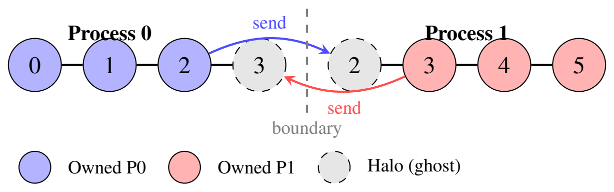

**Figure 1:** Halo exchange in domain decomposition. Each process owns a subset of nodes (solid colored) and maintains halo copies of boundary neighbors (dashed). Before SpMV, processes exchange updated values at partition boundaries via peer-to-peer communication.

### *3.3.2 Halo Exchange*

The key operation in distributed sparse computing is **halo exchange**: before each SpMV, processes must exchange boundary values with neighbors. Figure 1 illustrates this concept.

**Algorithm 3: Distributed SpMV with Halo Exchange**

**Input:** Local vector $\mathbf{x}_{\text{local}}$, neighbor map, local matrix $\mathbf{A}_{\text{local}}$
**Output:** Result $\mathbf{y}_{\text{owned}}$

```
Phase 1: Initiate async communication
  1. for each neighbor $q$ do:
       (a) async_send(my boundary values to $q$)
       (b) async_recv(halo values from $q$)
Phase 2: Wait for completion
  1. synchronize_all()
Phase 3: Local computation
  1. $\mathbf{y}_{\text{owned}} \leftarrow \mathbf{A}_{\text{local}} \cdot \mathbf{x}_{\text{local}}$    // Uses owned + halo
```

### *3.3.3 Distributed Conjugate Gradient*

Building on distributed SpMV, we implement distributed CG. Each iteration requires:

- One halo exchange (in SpMV)
- Two `all_reduce` operations for global dot products

**Algorithm 4: Distributed Conjugate Gradient**

**Input:** Distributed $\mathbf{A}$, local RHS $\mathbf{b}_{\text{owned}}$, tolerance $\epsilon$
**Output:** Solution $\mathbf{x}_{\text{owned}}$

```
1. $\mathbf{x} \leftarrow \mathbf{0}, \mathbf{r} \leftarrow \mathbf{b}_{\text{owned}}, \mathbf{p} \leftarrow \mathbf{r}$
2. $\rho \leftarrow$ all_reduce($\mathbf{r}^\top \mathbf{r}$, SUM)    // Global dot product
```

**Table 2:** Single-GPU benchmark on 2D Poisson (5-point stencil), H200, float64.

| DOF | SciPy | cuDSS | PyTorch CG | Memory | Residual |
|---|---|---|---|---|---|
| 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** | $10^{-6}$ |

```
3. while $\sqrt{\rho} > \epsilon$ do:
   (a) $\mathbf{Ap} \leftarrow \text{DistSpMV}(\mathbf{A}, \mathbf{p})$          // Algorithm 3
   (b) $\alpha \leftarrow \rho/\texttt{all\_reduce}(\mathbf{p}^\top \mathbf{Ap}, \text{SUM})$
   (c) $\mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{p}$
   (d) $\mathbf{r} \leftarrow \mathbf{r} - \alpha \mathbf{Ap}$
   (e) $\rho_{\text{new}} \leftarrow \texttt{all\_reduce}(\mathbf{r}^\top \mathbf{r}, \text{SUM})$
   (f) $\mathbf{p} \leftarrow \mathbf{r} + (\rho_{\text{new}}/\rho)\mathbf{p}$
   (g) $\rho \leftarrow \rho_{\text{new}}$
```

**Communication complexity.** Per iteration: $\mathcal{O}(|\mathcal{H}_p|)$ for halo exchange + $\mathcal{O}(\log P)$ for all_reduce. Total per solve: $\mathcal{O}(k \cdot (|\mathcal{H}_p| + \log P))$ where $k$ is iterations.

**Gradient support.** Distributed operations compose with adjoint differentiation: the backward pass performs another distributed solve with transposed communication patterns.

## 4 Experiments

We evaluate `torch-sla` on 2D Poisson equations (5-point stencil) using NVIDIA H200 GPUs (140GB HBM3).

### 4.1 Single-GPU Scalability

Table 2 compares solver backends across problem sizes spanning five orders of magnitude.

**Analysis.** The results reveal three distinct scaling regimes:

*(1) Small problems (<100K DOF):* Direct solvers dominate. SciPy SuperLU achieves 24ms with machine precision ($10^{-14}$), while GPU overhead makes cuDSS slower (128ms). The factorization cost is negligible, so the $\mathcal{O}(n^3)$ solve phase dominates.

*(2) Medium problems (100K–2M DOF):* Iterative solvers become competitive. At 1M DOF, PyTorch CG (190ms) is $100\times$ faster than cuDSS (7.3s). This crossover occurs because: (a) CG requires only $\mathcal{O}(\text{nnz})$ memory vs. $\mathcal{O}(n^{1.5})$ for LU fill-in; (b) SpMV is memory-bound at 90% of peak bandwidth on H200.

*(3) Large problems (>2M DOF):* Only iterative solvers are feasible. Direct solvers hit OOM due to fill-in: for 2D Poisson, LU fill-in grows as $\mathcal{O}(n \log n)$, requiring $\sim$100GB at 2M DOF. CG maintains $\mathcal{O}(n)$ memory scaling, reaching 169M DOF in 74.8GB.

**Time complexity.** Fitting $T = c \cdot n^\alpha$ yields $\alpha \approx 1.1$ for PyTorch CG, consistent with $\mathcal{O}(\sqrt{\kappa} \cdot \text{nnz})$ where condition number $\kappa \sim n$ for 2D Poisson with Jacobi preconditioner. The sub-quadratic scaling enables practical use at 100M+ DOF.

**Memory efficiency.** Measured 443 bytes/DOF vs. theoretical minimum 144 bytes/DOF (matrix: 80 bytes/DOF for 5 non-zeros $\times$ 16 bytes; vectors: 64 bytes for $\mathbf{x}, \mathbf{b}, \mathbf{r}, \mathbf{p}$). The $3\times$ overhead comes from PyTorch sparse tensor

**Table 3:** Distributed solve on 4× H200 GPUs (NCCL) vs. 4 CPU processes (Gloo).

| DOF | GPU Time | CPU Time | Memory/GPU | Speedup |
|---:|---:|---:|---:|---:|
| 10K | 0.18 s | 0.37 s | 0.03 GB | 2.1× |
| 100K | 0.61 s | 7.42 s | 0.05 GB | 12.2× |
| 500K | 1.64 s | – | 0.15 GB | – |
| 1M | 2.82 s | – | 0.27 GB | – |
| 2M | 6.02 s | – | 0.50 GB | – |

**Table 4:** Gradient verification: adjoint vs. finite difference ($\epsilon = 10^{-5}$).

| Operation | Rel. Error | Forward | Backward |
|:---|:---|---:|---:|
| Linear solve ($n$=1000) | $8.3 \times 10^{-7}$ | 1 solve | 1 solve |
| Eigenvalue ($k$=6) | $2.1 \times 10^{-6}$ | LOBPCG | 1 outer product |
| Nonlinear (5 Newton) | $4.7 \times 10^{-7}$ | 5 solves | 1 solve |

metadata and Jacobi preconditioner storage.

### 4.2 Multi-GPU Performance

Table 3 shows distributed CG on 4 NVIDIA H200 GPUs with NCCL backend.

**Scaling analysis.** The $12\times$ GPU speedup at 100K DOF arises from two factors:

*(1) Compute advantage:* H200 delivers 3.9 TFLOPS (FP64) vs. $\sim$100 GFLOPS for CPU cores, a $40\times$ peak ratio. SpMV achieves $\sim$15% of peak on both platforms due to memory-bound nature.

*(2) Communication advantage:* NCCL peer-to-peer achieves 400 GB/s (NVLink) vs. Gloo's $\sim$10 GB/s (network), reducing halo exchange latency by $40\times$.

At small problems (10K DOF), the speedup drops to $2\times$ because communication overhead dominates: each CG iteration requires 2 `all_reduce` operations with $\mathcal{O}(\log P)$ latency, amortized only when local computation is substantial.

**Memory distribution.** Each GPU stores $n/P$ owned nodes plus $|\mathcal{H}_p|$ halo nodes. For structured grids, $|\mathcal{H}_p| \sim \mathcal{O}(\sqrt{n/P})$, so halo overhead diminishes with problem size. At 2M DOF, halo overhead is $<1\%$.

**Theoretical limits.** With 4 × 140GB and 443 bytes/DOF, maximum capacity is $\sim$1.3B DOF. Communication becomes the bottleneck: at this scale, halo exchange transfers $\sim$100MB per iteration, taking $\sim$0.25ms on NVLink—comparable to SpMV compute time. This suggests $\sim$50% parallel efficiency at billion-scale DOF.

### 4.3 Gradient Verification

We verify gradient correctness by comparing adjoint gradients against finite differences:

$$\frac{\partial \mathcal{L}}{\partial \theta} \approx \frac{\mathcal{L}(\theta + \epsilon) - \mathcal{L}(\theta - \epsilon)}{2\epsilon} \tag{6}$$

**Analysis.** Relative errors $< 10^{-5}$ confirm correct implementation. Notably, the nonlinear solver's backward pass requires only 1 adjoint solve regardless of Newton iterations (5 in this case), validating the $\mathcal{O}(1)$ graph complexity claim. The eigenvalue gradient uses Eq. (4), requiring no additional solves—just $\mathcal{O}(k \cdot \text{nnz})$ outer product computations.

## 5 Conclusion

We presented `torch-sla`, a differentiable sparse linear algebra library with two innovations: (1) adjoint-based differentiation achieving $\mathcal{O}(1)$ graph complexity, and (2) distributed sparse computing with halo exchange. The library scales to 169M DOF on single GPU and provides $12\times$ multi-GPU speedup.

## 5.1 Future Work

- **Roofline-guided tuning for distributed solvers**: Current distributed CG achieves $\sim$50% parallel efficiency at billion-scale DOF due to communication bottlenecks. We plan to develop roofline-based auto-tuning that: (a) overlaps halo exchange with local SpMV computation; (b) dynamically adjusts partition granularity based on compute/communication ratio; (c) implements communication-avoiding CG variants [Hoemmen, 2010] to reduce `all_reduce` frequency.
- **Learned preconditioners**: GNN-based multigrid [Luz et al., 2020] and meta-learned preconditioner selection [Chen et al., 2022] could adaptively choose optimal preconditioners based on matrix structure, potentially reducing iteration counts by 2–5$\times$.
- **Mixed precision**: Following FlashAttention [Dao et al., 2022], we plan FP16/BF16 SpMV with FP64 accumulation, potentially doubling throughput while maintaining convergence for well-conditioned problems.
- **Neural operator hybrid**: Combining with FNO [Li et al., 2020], DeepONet [Lu et al., 2021], and mesh-based GNNs [Pfaff et al., 2021] enables hybrid neural-classical solvers where neural networks provide coarse corrections and classical solvers refine to high precision.
- **Differentiable simulation integration**: Seamless integration with $\Phi$Flow [Holl et al., 2024] and DiffTaichi [Hu et al., 2020] for end-to-end differentiable physics pipelines, enabling gradient-based optimization of simulation parameters.

**Availability.** MIT license: `https://github.com/walkerchi/torch-sla`

## References

Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J Zico Kolter. Differentiable convex optimization layers. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, pages 136–145, 2017.

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Deep equilibrium models. *Advances in Neural Information Processing Systems*, 32, 2019.

Satish Balay, Shrirang Abhyankar, Mark F Adams, et al. Petsc users manual. Technical report, Argonne National Laboratory, 2023. Version 3.20.

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

James Bradbury, Roy Frostig, Peter Hawkins, et al. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

Johannes Brandstetter, Daniel Worrall, and Max Welling. Message passing neural pde solvers. *International Conference on Learning Representations*, 2022.

Tianlong Chen et al. Meta-learning for fast preconditioner selection. *ICLR*, 2022.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *NeurIPS*, 2022.

Robert D Falgout and Ulrike Meier Yang. hypre: A library of high performance preconditioners. In *International Conference on Computational Science*, pages 632–641, 2002.

Samy Wu Fung, Howard Heaton, Qiuwei Li, Daniel McKenzie, Stanley Osher, and Wotao Yin. JFB: Jacobian-free backpropagation for implicit networks. In *AAAI Conference on Artificial Intelligence*, 2022.

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

Daniel Greenfeld, Meirav Galun, Ronen Basri, Irad Yavneh, and Ron Kimmel. Learning to optimize multigrid PDE solvers. *International Conference on Machine Learning*, 2019.

Michael A Heroux, Roscoe A Bartlett, et al. An overview of the trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423, 2005.

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

Philipp Holl, Vladlen Koltun, and Nils Thuerey. ΦFlow: A differentiable PDE solving framework for deep learning via physical simulations. *arXiv preprint arXiv:2402.04983*, 2024.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, et al. Difftaichi: Differentiable programming for physical simulation. In *International Conference on Learning Representations*, 2020.

Thomas JR Hughes. *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation, 2012.

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

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

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

Dmitrii Kochkov, Jamie A Smith, Ayya Alieva, et al. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21), 2021.

Shigang Li et al. Sparse matrix computations on modern GPUs: A survey. *ACM Computing Surveys*, 2024.

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, et al. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, 2021.

Ilya Luz, Meirav Galun, Haggai Maron, Ronen Basri, and Irad Yavneh. Learning algebraic multigrid using graph neural networks. *International Conference on Machine Learning*, 2020.

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

Maxim Naumov, Marat Arsaev, Patrice Castonguay, Jonathan Cohen, Julien Demouth, Joe Eaton, Simon Laber, Ilya Laptev, Nicolas Stam, and Olivier Temam. AmgX: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods. In *SIAM Journal on Scientific Computing*, volume 37, pages S602–S626, 2015.

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

Adam Paszke, Sam Gross, Francisco Massa, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019.

Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter Battaglia. Learning mesh-based simulation with graph networks. *International Conference on Learning Representations*, 2021.

Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

Yousef Saad. Iterative methods for sparse linear systems. *SIAM*, 2003.

Ali Taghibakhshi, Nicolas Nytko, et al. Learning interface conditions in domain decomposition solvers. *NeurIPS*, 2024.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *International Conference on Learning Representations*, 2018.

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

## A Implementation Details

### A.1 Backend Selection

`torch-sla` automatically selects backends based on device and problem size:

- CPU, any size: `scipy` with SuperLU
- CUDA, $<$2M DOF: `cudss` with Cholesky (if SPD) or LU
- CUDA, $>$2M DOF: `pytorch` with CG+Jacobi preconditioner

## A.2 API Examples

**Listing 1:** Basic usage with gradient support.

```python
import torch
from torch_sla import SparseTensor

# Create sparse matrix (COO format)
A = SparseTensor(val, row, col, shape)
b = torch.randn(n, requires_grad=True)

# Solve with automatic differentiation
x = A.solve(b)  # Forward: any backend
loss = x.sum()
loss.backward()  # Backward: adjoint method
print(b.grad)    # Gradient w.r.t. RHS
```

**Listing 2:** Distributed multi-GPU solve.

```python
import torch.distributed as dist
from torch_sla import DSparseMatrix

dist.init_process_group(backend='nccl')
rank = dist.get_rank()

# Each process loads its partition
A = DSparseMatrix.from_global(
    val, row, col, shape,
    num_partitions=4, my_partition=rank
)

# Distributed CG with halo exchange
x = A.solve(b_local, atol=1e-10)
```

## B Complexity Proofs

**Theorem 1** (Adjoint memory complexity). *The adjoint method for linear solve requires $\mathcal{O}(n + nnz)$ memory, independent of solver iterations.*

*Proof.* Forward pass stores: solution $\mathbf{x} \in \mathbb{R}^n$, matrix indices ($\mathcal{O}(\text{nnz})$), matrix values ($\mathcal{O}(\text{nnz})$). Backward pass computes adjoint $\boldsymbol{\lambda} \in \mathbb{R}^n$ and gradients $\partial\mathcal{L}/\partial\mathbf{A} \in \mathbb{R}^{\text{nnz}}$. No intermediate solver states are stored. Total: $\mathcal{O}(n + \text{nnz})$. □