

Machine Learning Operator Lexicon for Partial Differential Equation Solvers on Irregular and Sparse Domains

JAEHYUN LEE, University of Wisconsin-Madison, USA

EFTYCHIOS SIFAKIS, University of Wisconsin-Madison, USA and NVIDIA, USA

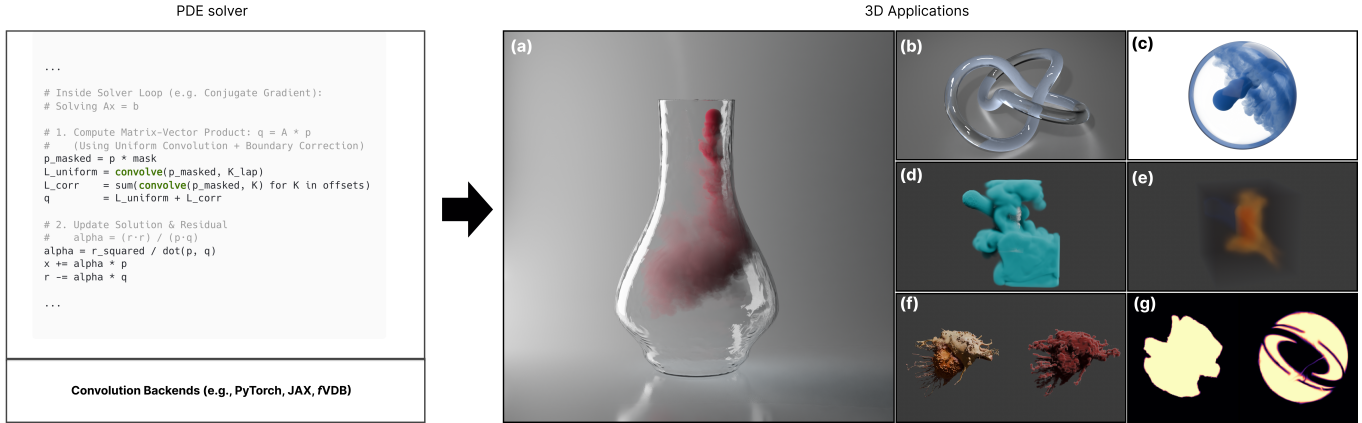


Fig. 1. Overview of the proposed operator lexicon and its 3D PDE applications. Our formulation expresses PDE solvers through machine-learning operators, applying differential operators via convolution and handling boundary conditions through localized masking and shifting operations.

This unified abstraction supports a broad range of PDEs and 3D applications, including sparse fluid simulation in (a–c), dense fluid simulation in (d), heat-transfer PDE coupled with the fluid simulation in (e), inverse coefficient reconstruction in (f), and inverse fluidic control in (g).

Solvers for grid-based discretizations of Partial Differential Equations (PDEs) are ubiquitous in computer graphics. However, efficient implementation of such solvers is contingent on specific design choices for grid data structures (especially for sparse domains), hand-tuned kernels for specific platforms (CPU vs GPU), and typically relies on the lexicon of sparse linear algebra to express its algorithmic building blocks. Meanwhile, modern hardware and software stacks are increasingly optimized around machine learning (ML) workloads, where tensor operators such as convolution, masking, and pooling form the dominant computational vocabulary. We introduce an ML-operator abstraction for discrete PDE solvers on sparse and irregular domains. Our formulation separates solver computation into uniform stencil actions, pointwise coefficient operations, masking, reductions, and localized boundary corrections, yielding a compact operator pipeline that supports practical boundary conditions and matrix-free execution. Unlike neural PDE methods, our approach does not learn a solver. It uses ML operators to express and execute established numerical discretizations. We demonstrate that this abstraction accommodates multigrid, multigrid-preconditioned Krylov solvers, and adjoint differentiation on practical 3D problems, including sparse, irregular, and inverse settings. The same operator-level view also exposes concrete backend optimization targets. Through a sparse domain

optimization case study, we show that targeted improvements to a key operator can substantially improve performance over an existing baseline.

CCS Concepts: • **Mathematics of computing** → **Solvers**; • **Computing methodologies** → *Machine learning*; **Physical simulation**.

Additional Key Words and Phrases: Partial Differential Equations, Convolution, Sparse Solvers

ACM Reference Format:

JaeHyun Lee and Eftychios Sifakis. 2026. Machine Learning Operator Lexicon for Partial Differential Equation Solvers on Irregular and Sparse Domains. In *SIGGRAPH Asia 2026 Conference Papers (SA Conference Papers '26)*, December 01–04, 2026, Kuala Lumpur, Malaysia. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3829340.3842319>

1 Introduction

Partial differential equations (PDEs) are ubiquitous in computer graphics, but the computational vocabulary used to implement their solvers has changed far less than the hardware on which they run. Solver implementations are still largely written in the language of sparse linear algebra, BLAS-level primitives, and hand-tuned kernels. By contrast, modern hardware acceleration has increasingly shifted toward machine learning (ML) workloads, making tensor operators such as convolution, reduction, and pooling the primary targets of optimization. This paper aims to express PDE solvers in terms of ML operators and to *expose the new operator lexicon* needed to represent practical numerical solvers in that language. We emphasize that our solver is not learned; rather, ML operators serve as a new computational abstraction for implementing established numerical methods.

Authors' Contact Information: JaeHyun Lee, University of Wisconsin-Madison, Madison, Wisconsin, USA; Eftychios Sifakis, University of Wisconsin-Madison, Madison, Wisconsin, USA and NVIDIA, Santa Clara, California, USA.



This work is licensed under a Creative Commons Attribution 4.0 International License. *SA Conference Papers '26, Kuala Lumpur, Malaysia*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2842-6/2026/12
<https://doi.org/10.1145/3829340.3842319>

Expressing PDE solvers in this new ML-oriented computational abstraction brings advantages on both sides of the API: the application level and the backend level. At the application level, it provides a much more convenient programming model by allowing solvers to be composed from higher-level ML operators instead of extensive loops, indexing, and data-structure-specific logic. Since modern ML libraries already support these operators with built-in differentiability, gradients become substantially easier to obtain. The same abstraction improves portability across hardware backends, preserves the matrix-free nature of practical solvers, and decouples solver logic from data representation, allowing the same numerical method to target both dense tensors and sparse domains, as illustrated by our roughly 200-line supplemental *fVDB* sparse fluid solver.

At the backend level, ML operators are already the focus of modern hardware optimization. Beyond that, expressing PDE solvers in this operator language reveals which ML-style primitives actually become bottlenecks in numerical workloads, thereby creating clear opportunities for backend optimization. This gives systems designers a much more precise target and shifts optimization from the entire solver as a monolithic code path to the specific operators that matter most. In this paper, we demonstrate this through a case study, showing that even relatively simple operator level optimization can already produce a sparse domain implementation that outperforms the existing baseline.

This mapping, however, is nontrivial. Practical PDE solvers must handle sparse domains and irregular boundary interfaces, as illustrated in Figure 1, along with mixed boundary conditions and spatially varying stencils. In contrast, modern tensor frameworks are built around regular, uniform computation. A useful abstraction must therefore preserve the expressive convenience of ML operators while remaining compatible with the numerical realities of sparse and irregular simulation workloads.

To address this challenge, we introduce a computational abstraction that expresses discrete PDE solvers through a compact set of ML-level operators while retaining compatibility with irregular and sparse domains, practical boundary conditions, and matrix-free numerical structure. This abstraction is not only expressive enough to represent practical solvers, but also structured enough to expose backend level optimization opportunities.

We summarize our contributions as follows:

- **Versatile API abstraction.** We introduce an ML-level computational lexicon for expressing discrete PDE operators on irregular or sparse domains with practical boundary conditions as pipelines of standard tensor operators.
- **Practical realization.** We show that this abstraction is practical and expressive enough to accommodate multigrid, multigrid-preconditioned Krylov methods, and adjoint differentiation, and demonstrate it on several 3D examples including sparse, irregular, and inverse problems.
- **Backend optimization.** We present a case study of a targeted, high-impact operator optimization, showing how the proposed formulation exposes concrete bottlenecks and substantially improves performance in sparse numerical workloads.

2 Related Work

Computational Substrate Mapping. Efforts to map PDE solvers onto the dominant computational substrate of the time have appeared repeatedly over the years, including shader-based implementations on early GPUs [Bolz et al. 2003; Goodnight et al. 2005; Krüger and Westermann 2005], FPGA realizations [Motuk et al. 2005], CUDA-oriented stencil kernels [Micikevicius 2009], GPU support in sparse linear solver libraries such as Hypre [Falgout et al. 2006] and NVIDIA AmgX [Naumov et al. 2015], ASIC acceleration [Chen et al. 2019], and more recently TPU-based approaches [Pierce et al. 2022]. Today, a similar shift is taking place around modern ML stacks, where tensor operators have become primary targets of software and hardware optimization. Our work is motivated by the same broader question, but focuses on the practical graphics setting of irregular and sparse domains.

Mapping PDEs to ML Stacks. Several works have explored implementing PDE solvers on top of modern ML and tensor-programming stacks. In computer graphics, the closest example is PhiFlow [Holl et al. 2020], a high-level differentiable framework built on ML infrastructure with a primary focus on fluid simulation. PhiFlow provides convenient high-level simulation operators, whereas our goal is to expose the lower-level operator lexicon required to construct elliptic solvers themselves. This explicit decomposition provides reusable boundary-correction primitives for irregular Dirichlet, Neumann, and Robin conditions, exposes the diagonal, transpose, and inter-grid transfer operators needed for multigrid and adjoint differentiation, and separates solver logic from data representation so that the same formulation can target both dense tensors and sparse active-set backends. PhiFlow supports matrix-free fluid solves and irregular geometries, but these components are not exposed through the same general elliptic-operator interface, and sparse execution is not a primary design target. Related directions have also appeared in computational physics, such as NeuralPhysics [Chen et al. 2026] and JAX-Fluids [Bezgin et al. 2023], as well as in the HPC community [Boulasikis et al. 2025]. However, these works typically focus on different PDE classes [Chen et al. 2024; Li et al. 2025; Phillips et al. 2023] or simpler domain and boundary settings, rather than elliptic problems on irregular and sparse graphics domains. We provide a more detailed discussion of these distinctions in the supplemental material.

Sparse Backends. Sparse domains are central to modern computer graphics workloads, especially large-scale simulations where dense representations quickly become impractical. Prior work has addressed this challenge through specialized sparse simulation systems, including SPGrid [Setaluri et al. 2014] and MSBG [Braun et al. 2025] (both CPU-oriented), as well as other large-scale sparse fluid solvers. While effective, these systems are often tightly coupled to specific algorithms, discretizations, or hardware assumptions. Sparse support can instead be exposed at the domain-specific language (DSL) level, as in Taichi [Hu et al. 2019], which provides hierarchical sparse execution models within a programmable framework. However, users must still program against the underlying sparse structure and rely on an additional compilation layer. Another approach is to build simulation methods on general-purpose

sparse data structures such as OpenVDB [Museth et al. 2013] and its descendants [Kim et al. 2024; Museth 2021]. The main challenge then becomes mapping PDE components to the backend’s operator API, as illustrated by Wu et al. [2018] and Shao et al. [2022]. As the OpenVDB ecosystem evolves toward GPU support, differentiability, and ML-oriented APIs, exemplified by fVDB [Williams et al. 2024], expressing PDE solvers in terms of ML operators provides a natural bridge to modern sparse backends.

Convolution as a Numerical Primitive. Convolution has also been used as a computational primitive for numerical problems independently of learning. Farbman et al. [2011] introduced convolution pyramids for efficiently approximating large-support convolutions, with applications including Poisson image editing and scattered-data interpolation. Pereira et al. [2020] presented a matrix-free formulation for large-scale geostatistical filtering, illustrating a related effort to avoid explicit matrix assembly when applying spatial differential operators. Multigrid-inspired neural architectures such as MgNet [He and Xu 2019] and subsequent methods [Han et al. 2024; He et al. 2023; Hsieh et al. 2019; Huang et al. 2022] further connect hierarchical convolutional computation with multilevel numerical methods, although primarily in learning-centric settings. Our work instead uses standard tensor operators as a deterministic implementation vocabulary for complete numerical solvers, including boundary treatment, Krylov methods, multigrid, and adjoint differentiation.

Boundary-Aware Convolution. Prior work has also modified convolution near missing data or domain boundaries. Normalized and differential convolution [Knutsson and Westin 1993] incorporates local confidence or applicability information when filtering incomplete and uncertain data, while partial convolution [Liu et al. 2018] renormalizes convolution over valid pixels for irregular-hole inpainting. Innamorati et al. [2018] instead learn explicit filters for handling image boundaries in CNNs. These methods address truncated or uncertain convolution support in image-processing and learning settings. In contrast, our corrections are derived from prescribed PDE boundary conditions: ghost-point elimination converts Dirichlet, Neumann, and Robin conditions into local operator and right-hand-side corrections while retaining a uniform bulk stencil.

3 Problem Statement

We focus on steady-state second-order linear elliptic PDEs, a compact but expressive class that covers many recurring subproblems in computer graphics, including geometry processing, diffusion-type systems, and pressure projection in fluid simulation. Specifically, we consider a scalar unknown $u : \Omega \subset \mathbb{R}^d \rightarrow \mathbb{R}$ satisfying

$$L(u) = \nabla \cdot (\alpha \nabla u) + \omega \cdot \nabla u + \sigma u = f, \quad \text{in } \Omega, \quad (1)$$

with Robin boundary conditions

$$a u + b \partial_n u = g, \quad \text{on } \partial\Omega. \quad (2)$$

This form subsumes Dirichlet conditions when $b = 0$ and Neumann conditions when $a = 0$. In incompressible fluid simulation, for example, this reduces to the constant coefficient Poisson pressure equation with $\alpha = 1$, $\omega = 0$, and $\sigma = 0$, where u is the pressure and $f = \nabla \cdot \mathbf{v}$ up to sign convention.

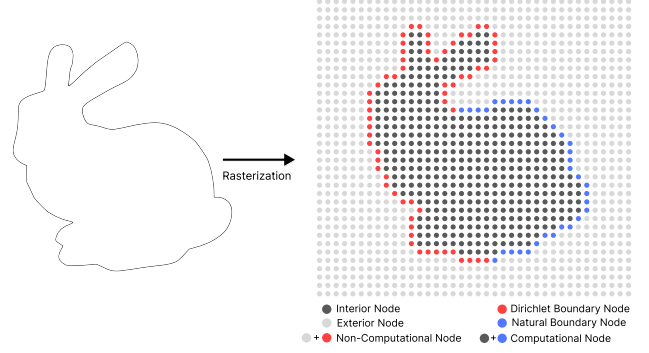


Fig. 2. Discretization of Ω . Nodes are partitioned into interior ($\mathcal{I}$, black), exterior ($\mathcal{E}$, gray), Dirichlet boundary ($\mathcal{D}$, red), and natural boundary ($\mathcal{N}$, blue). We define $\mathcal{I} \cup \mathcal{N}$ as **computational nodes** and $\mathcal{E} \cup \mathcal{D}$ as **non-computational nodes**.

We rewrite this PDE class in terms of local stencil actions and pointwise operations. Expanding the diffusion term using the product rule separates variable-coefficient diffusion into a Laplacian stencil, a gradient stencil, and pointwise coefficient terms:

$$\nabla \cdot (\alpha \nabla u) = \alpha \nabla^2 u + \nabla \alpha \cdot \nabla u.$$

At the continuous level, this yields the equivalent non-divergence form

$$L(u) = \alpha(\nabla^2 u) + (\nabla \alpha + \omega) \cdot (\nabla u) + \sigma u = f. \quad (3)$$

We define

$$\omega_{\text{eff}} = \nabla \alpha + \omega$$

as the effective first-order coefficient.

We discretize Ω on a regular Cartesian grid with spacing h using a node-centered layout. Grid nodes are partitioned into exterior nodes $\mathcal{E}$, interior nodes $\mathcal{I}$, and boundary nodes $\mathcal{B}$, where $\mathcal{B}$ consists of nodes inside the domain whose stencil neighborhood touches the exterior. Boundary nodes are further divided into Dirichlet nodes $\mathcal{D}$ and Neumann or Robin boundary nodes $\mathcal{N}$. Nodes in $\mathcal{D}$ are fixed by the boundary condition, while nodes in $\mathcal{N}$ remain active unknowns and are solved together with the interior. For brevity, we refer to the latter as *natural-boundary nodes* throughout the paper; here, this term denotes Neumann or Robin boundary nodes that remain active unknowns. Figure 2 illustrates this partition. Throughout the paper, we use italic symbols for continuous fields such as u , α , and ω , and bold symbols for their discretized tensor counterparts such as $\mathbf{u}$, $\boldsymbol{\alpha}$, and $\boldsymbol{\omega}$.

This Cartesian discretization preserves a regular local neighborhood structure for stencil operators. Boundary-adjacent nodes therefore have axis-aligned exterior directions, so the normal derivative $\partial_n u$ can be discretized as a one-dimensional finite difference along the grid axis pointing outside the domain.

Unless otherwise stated, we discretize all derivatives with centered finite differences. Under this discretization, applying the operator $\mathbf{u} \mapsto \mathbf{L}\mathbf{u}$ reduces to combining local Laplacian and gradient stencil actions with pointwise coefficient operations, where juxtaposition of discrete fields denotes elementwise multiplication. Boundary effects are handled separately through correction terms in the next section.

4 Operator Formulation

We now express the discrete PDE operator in an ML-oriented form. The key idea is to separate the computation into uniform stencil actions, pointwise coefficient operations, masking, and localized boundary corrections.

Our construction proceeds in three steps. We first define a uniform operator that is valid away from boundary irregularities. We then introduce localized correction terms near boundary nodes. Finally, we assemble these components into a compact operator pipeline.

4.1 Uniform Operator

For any node set $\mathcal{A}$, let $\mathbf{m}_{\mathcal{A}}$ denote the pointwise mask that keeps entries on $\mathcal{A}$ and zeros out all other entries. Let

$$\mathbf{m}_{\text{comp}} = \mathbf{m}_{\mathcal{I} \cup \mathcal{N}}$$

denote the computational mask, which selects active unknowns: interior nodes and natural-boundary nodes. Dirichlet and exterior nodes are treated as non-computational and are zeroed out before applying spatial stencils:

$$\mathbf{u}_{\text{comp}} = \mathbf{m}_{\text{comp}} \mathbf{u}.$$

The uniform part of the operator is then

$$\mathbf{L}_0 \mathbf{u} = \alpha (\mathbf{K}_{\nabla^2} * \mathbf{u}_{\text{comp}}) + \omega_{\text{eff}} \cdot (\mathbf{K}_{\nabla} * \mathbf{u}_{\text{comp}}) + \sigma \mathbf{u}_{\text{comp}}. \quad (4)$$

Here $\mathbf{K}_{\nabla^2}$ is the standard finite-difference Laplacian stencil, $\mathbf{K}_{\nabla}$ denotes the collection of centered finite-difference gradient stencils, and $*$ denotes applying a local stencil, equivalently a convolution in the convention used by ML libraries.

Equation (4) is correct at interior nodes whose stencil support lies entirely inside the computational domain. Near boundaries, however, some stencil accesses reach Dirichlet or exterior nodes. We handle these cases through localized correction terms rather than changing the bulk stencil.

4.2 Boundary Corrections

Near boundaries, some stencil entries point to Dirichlet or exterior nodes that have been masked out to zero. Boundary corrections replace these missing entries using the prescribed boundary conditions. Dirichlet nodes contribute only an affine right-hand-side correction, while natural boundary conditions modify both the operator and the right-hand side.

Dirichlet correction. For Dirichlet nodes, stencil reads should use the prescribed values $\mathbf{g}$ rather than zero. Equation (4), however, applies the stencil to a field in which Dirichlet nodes have been zeroed out. By linearity, the missing contribution is obtained by applying the same uniform stencil expression to the prescribed Dirichlet values and restricting the result to computational nodes:

$$C_{\mathcal{D}}^{\text{rhs}} = \mathbf{m}_{\text{comp}} \left[\alpha (\mathbf{K}_{\nabla^2} * \mathbf{m}_{\mathcal{D}} \mathbf{g}) + \omega_{\text{eff}} \cdot (\mathbf{K}_{\nabla} * \mathbf{m}_{\mathcal{D}} \mathbf{g}) + \sigma \mathbf{m}_{\mathcal{D}} \mathbf{g} \right]. \quad (5)$$

This term is independent of $\mathbf{u}$ and can be precomputed and absorbed into the right-hand side, corresponding to the standard homogenization of Dirichlet boundary conditions.

Natural-boundary correction. For Neumann and Robin boundaries, the stencil value outside the domain is replaced using the boundary condition. Consider a natural-boundary node $i \in \mathcal{N}$ and a stencil offset p such that $i + p \in \mathcal{E}$. Using a centered finite difference along the axis defined by p , the Robin condition gives

$$a_i u_i + b_i \frac{u_{i+p} - u_{i-p}}{2h} = g_i,$$

and therefore

$$u_{i+p} = u_{i-p} - \frac{2ha_i}{b_i} u_i + \frac{2h}{b_i} g_i.$$

Thus, an exterior stencil read is replaced by a reflected interior read, a local diagonal correction, and an affine boundary term.

To express this substitution in operator form, let $\mathbf{S}_p$ denote the shift/fetch operator

$$(\mathbf{S}_p \mathbf{u})_i = u_{i+p}.$$

We use a fetch convention: the output at node i reads the input value from $i + p$, rather than moving the value at i to $i + p$. Equivalently, $\mathbf{S}_p$ can be viewed as convolution with a one-hot stencil at offset p .

For each offset p , define the directional boundary mask

$$(\mathbf{m}_p)_i = \begin{cases} 1, & i \in \mathcal{N} \text{ and } i + p \in \mathcal{E}, \\ 0, & \text{otherwise.} \end{cases} \quad (6)$$

With this fetch convention, the same mask can be constructed by evaluating the exterior mask at the neighbor in direction p :

$$\mathbf{m}_p = \mathbf{m}_{\mathcal{N}} (\mathbf{S}_p \mathbf{m}_{\mathcal{E}}). \quad (7)$$

Indeed, $(\mathbf{S}_p \mathbf{m}_{\mathcal{E}})_i = (\mathbf{m}_{\mathcal{E}})_{i+p}$, so $\mathbf{m}_p$ selects natural boundary nodes i whose neighbor $i + p$ lies in the exterior.

For a stencil entry with weight $w_p^{(\mathbf{K})}$, the masked uniform operator reads zero when $i + p \in \mathcal{E}$, since exterior nodes have been zeroed out. The boundary condition instead requires the contribution

$$w_p^{(\mathbf{K})} \left(u_{i-p} - \frac{2ha_i}{b_i} u_i + \frac{2h}{b_i} g_i \right).$$

We therefore add this contribution only at nodes selected by $\mathbf{m}_p$, separating the $\mathbf{u}$ -dependent part from the boundary-data term.

For a stencil $\mathbf{K}$ with coefficients $w_p^{(\mathbf{K})}$, the natural-boundary operator correction is

$$C_{\mathcal{N}}(\mathbf{u}, \mathbf{K}) = \sum_p w_p^{(\mathbf{K})} \mathbf{m}_p \left(\mathbf{S}_{-p} \mathbf{u} - \frac{2ha}{b} \mathbf{u} \right), \quad (8)$$

and the corresponding right-hand-side correction is

$$C_{\mathcal{N}}^{\text{rhs}}(\mathbf{K}) = \sum_p w_p^{(\mathbf{K})} \mathbf{m}_p \left(\frac{2h}{b} g \right). \quad (9)$$

The first term modifies the linear operator, while the second is absorbed into the right-hand side as part of the boundary homogenization.

4.3 Full Operator Action

Combining the uniform operator with the natural-boundary corrections, the final linear operator is

$$\mathbf{L} \mathbf{u} = \mathbf{L}_0 \mathbf{u} + \alpha C_{\mathcal{N}}(\mathbf{u}, \mathbf{K}_{\nabla^2}) + \omega_{\text{eff}} \cdot C_{\mathcal{N}}(\mathbf{u}, \mathbf{K}_{\nabla}). \quad (10)$$

Algorithm 1 ApplyOperator $\mathbf{u} \mapsto \mathbf{L}\mathbf{u}$ **Input:** $\mathbf{u}, \mathbf{f}, \theta$, Kernels $\mathbf{K}$, Masks $\mathbf{m}$ **Output:** $\mathbf{y} = \mathbf{L}\mathbf{u}$

```

1:  $\mathbf{u} \leftarrow \mathbf{m}_{\text{comp}} \mathbf{u}$  ▷ Mask non-computational nodes
2:  $\mathbf{y}_{\nabla^2} \leftarrow \text{Conv}(\mathbf{u}, \mathbf{K}_{\nabla^2})$ 
3:  $\mathbf{y}_{\nabla} \leftarrow \text{Conv}(\mathbf{u}, \mathbf{K}_{\nabla})$  ▷ Uniform stencil actions
4:  $\mathbf{c}_{\nabla^2} \leftarrow \sum_p \mathbf{w}_p^{(\nabla^2)} \mathbf{m}_p \left( \mathbf{S}_{-p} \mathbf{u} - \frac{2ha}{b} \mathbf{u} \right)$ 
5:  $\mathbf{c}_{\nabla} \leftarrow \sum_p \mathbf{w}_p^{(\nabla)} \mathbf{m}_p \left( \mathbf{S}_{-p} \mathbf{u} - \frac{2ha}{b} \mathbf{u} \right)$  ▷ Natural-boundary corrections
6:  $\mathbf{y} \leftarrow \alpha(\mathbf{y}_{\nabla^2} + \mathbf{c}_{\nabla^2}) + \omega_{\text{eff}} \cdot (\mathbf{y}_{\nabla} + \mathbf{c}_{\nabla}) + \sigma \mathbf{u}$  ▷ Full operator
7: return  $\mathbf{m}_{\text{comp}} \mathbf{y}$ 

```

All boundary terms that do not depend on $\mathbf{u}$ are moved to the right-hand side:

$$\tilde{\mathbf{f}} = \mathbf{f} - \mathbf{C}_D^{\text{rhs}} - \alpha \mathbf{C}_N^{\text{rhs}}(\mathbf{K}_{\nabla^2}) - \omega_{\text{eff}} \cdot \mathbf{C}_N^{\text{rhs}}(\mathbf{K}_{\nabla}).$$

This yields a homogenized linear system in which the bulk computation is expressed through uniform stencil operators, while all boundary complexity is isolated into localized masks, shifts, pointwise products, and reductions over stencil offsets. Algorithm 1 summarizes the resulting operator evaluation.

5 Extending the Operator Lexicon

The action $\mathbf{u} \mapsto \mathbf{L}\mathbf{u}$ is sufficient for basic matrix-free Krylov methods, but practical solvers require additional operator access. Stationary smoothers require the diagonal of $\mathbf{L}$, adjoint methods require the transpose action $\mathbf{L}^\top \mathbf{u}$, and multigrid requires inter-grid transfer operators. Fluid simulation further introduces staggered-grid differential operators and advection. These components use the same operator lexicon as Section 4, including stencil actions, shifts, masks, and pointwise operations. The only additional primitive needed for fluid advection is a sampling operator.

5.1 Diagonal Operator

Weighted Jacobi and related stationary smoothers require the diagonal of the discretized operator. Given the residual

$$\mathbf{r}^{(k)} = \tilde{\mathbf{f}} - \mathbf{L}\mathbf{u}^{(k)},$$

a weighted Jacobi update is

$$\mathbf{u}^{(k+1)} = \mathbf{u}^{(k)} + \omega_J \mathbf{D}^{-1} \mathbf{r}^{(k)}, \quad (11)$$

where ω_J is the relaxation weight and $\mathbf{D} = \text{diag}(\mathbf{L})$ is applied pointwise on computational nodes.

In our formulation, the diagonal is obtained directly from the center weights of the uniform stencil operator and the local diagonal shifts introduced by natural-boundary corrections. The uniform operator contributes

$$\text{diag}(\mathbf{L}_0) = \alpha \mathbf{w}_0^{(\nabla^2)} + \sigma,$$

since centered gradient stencils have zero center weight. The natural-boundary correction contributes through its local diagonal term only. The reflected shift term has no diagonal contribution for $p \neq 0$.

Combining these terms gives

$$\mathbf{D} = \alpha \mathbf{w}_0^{(\nabla^2)} + \sigma - \frac{2ha}{b} \sum_p \mathbf{m}_p \left(\alpha \mathbf{w}_p^{(\nabla^2)} + \sum_{j=1}^d \omega_{\text{eff},j} \mathbf{w}_p^{(\nabla_j)} \right). \quad (12)$$

The diagonal is independent of $\mathbf{u}$ and can therefore be cached. Thus Jacobi smoothing reduces to one operator evaluation, a pointwise scaling by $\mathbf{D}^{-1}$, and a masked pointwise update. Red-black Gauss-Seidel can be expressed similarly using alternating color masks, avoiding in-place updates while retaining compatibility with tensor backends.

5.2 Transpose Operator

The transpose action $\mathbf{u} \mapsto \mathbf{L}^\top \mathbf{u}$ is needed for adjoint-based differentiation [Giles and Pierce 2000] and BiCG-type Krylov solvers. The same operator decomposition used for the forward action makes the transpose straightforward: stencil actions transpose to flipped stencils [Goodfellow et al. 2016], shifts transpose to opposite shifts, and masks and pointwise coefficient multiplications remain pointwise operations.

For the uniform operator, this gives

$$\mathbf{L}_0^\top \mathbf{u} = \text{flip}(\mathbf{K}_{\nabla^2}) * (\alpha \mathbf{u}) + \sum_{j=1}^d \text{flip}(\mathbf{K}_{\nabla_j}) * (\omega_{\text{eff},j} \mathbf{u}) + \sigma \mathbf{u}. \quad (13)$$

With centered differences, $\text{flip}(\mathbf{K}_{\nabla^2}) = \mathbf{K}_{\nabla^2}$ and $\text{flip}(\mathbf{K}_{\nabla_j}) = -\mathbf{K}_{\nabla_j}$.

For the natural-boundary correction, the forward operator first applies a reflected shift and then masks the affected boundary nodes. Its transpose applies the mask first and then shifts in the opposite direction:

$$\mathbf{C}_N^\top(\mathbf{u}; \mathbf{K}) = \sum_p \mathbf{w}_p^{(\mathbf{K})} \left[\mathbf{S}_p(\mathbf{m}_p \mathbf{u}) - \frac{2ha}{b} \mathbf{m}_p \mathbf{u} \right]. \quad (14)$$

Combining this with Eq. (10), the full transpose action is

$$\mathbf{L}^\top \mathbf{u} = \mathbf{L}_0^\top \mathbf{u} + \mathbf{C}_N^\top(\alpha \mathbf{u}, \mathbf{K}_{\nabla^2}) + \mathbf{C}_N^\top(\omega_{\text{eff}} \mathbf{u}, \mathbf{K}_{\nabla}). \quad (15)$$

This transpose action enables implicit differentiation via the adjoint method [Giles and Pierce 2000] without unrolling the forward iterative solver. For a parameterized system

$$\mathbf{L}(\theta) \mathbf{u} = \tilde{\mathbf{f}},$$

and loss gradient $\partial \mathcal{L} / \partial \mathbf{u}$, we solve the adjoint system

$$\mathbf{L}^\top \boldsymbol{\lambda} = \frac{\partial \mathcal{L}}{\partial \mathbf{u}}.$$

The gradient with respect to the right-hand side is $\partial \mathcal{L} / \partial \tilde{\mathbf{f}} = \boldsymbol{\lambda}$, and parameter gradients are computed as

$$\frac{\partial \mathcal{L}}{\partial \theta} = -\boldsymbol{\lambda}^\top \left(\frac{\partial \mathbf{L}}{\partial \theta} \mathbf{u} \right). \quad (16)$$

In practice, the final vector-Jacobian product is obtained by back-propagating through one evaluation of the forward operator with seed vector $\boldsymbol{\lambda}$, allowing coefficient dependencies to be handled by reverse-mode automatic differentiation (autodiff) [Griewank and Walther 2008] while keeping the solver itself matrix-free and memory efficient.

This adjoint solve also avoids the instability of unrolling Krylov iterations. Krylov solvers such as BiCGSTAB contain state-dependent

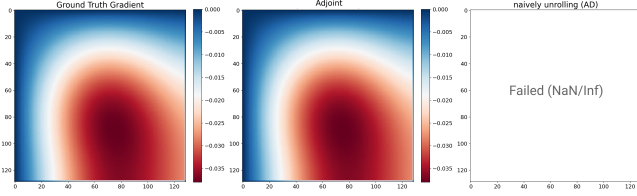


Fig. 3. Adjoint differentiation avoids unstable Krylov gradients. Unrolling BiCGSTAB through reverse-mode autodiff leads to gradient explosion, while an adjoint solve with L^T remains stable and matches the analytic gradient.

coefficients and divisions, which can produce exploding reverse-mode gradients when naively unrolled. In Figure 3, unrolled autodiff through a multigrid solver with a BiCGSTAB coarse solve deviates from the analytic gradient, while implicit differentiation through the adjoint system remains stable and accurate.

5.3 Inter-Grid Transfer Operators

Geometric multigrid methods require operators that transfer residuals and corrections between grid levels. These transfers also fit naturally into the ML-operator view: restriction is a fixed strided stencil operation, analogous to pooling, and prolongation is a fixed transposed stencil operation, analogous to upsampling or transposed convolution. We use full-weighting restriction R and matched linear prolongation P . In tensor form, R is implemented as a fixed strided convolution with stride 2, and P as a stride-2 transposed convolution with interpolation weights. Together with the smoother from Section 5.1, residual evaluation using L , and a coarse-level solver, these transfers form a standard geometric multigrid V-cycle. The resulting multigrid solver remains matrix-free: all operations are expressed as stencil actions, pointwise updates, masks, and inter-grid transfers. The same V-cycle can be used either as a standalone solver or as a preconditioner for Krylov methods. As a concrete illustration of the compactness of this formulation, our supplemental material includes a 77-line PyTorch implementation of the complete multigrid solver for a Poisson problem.

5.4 Fluid Operators

Fluid simulation adds two operators to the formulation. In a MAC-grid discretization, pressure is stored at cell centers, velocity components at face centers, and obstacle boundaries are represented by grid faces separating fluid and solid cells. Thus, pressure projection requires adapting the boundary correction from Section 4.2 to this staggered setting, while smoke and velocity transport require advection operators.

For pressure projection, the unknown u is the cell-centered pressure. When a pressure stencil from a fluid cell i crosses a solid face toward $i + p$, the homogeneous Neumann wall condition, i.e., the zero-normal-derivative condition $\partial_n u = 0$, replaces the exterior pressure by the interior pressure:

$$u_{i+p} = u_i.$$

Thus the correction does not require a reflected fetch.

Because the wall condition is homogeneous, the affine right-hand-side term vanishes. The MAC-grid obstacle correction therefore only changes the center coefficient:

$$C_{N,MAC}(u, K) = \sum_p w_p^{(K)} m_p u.$$

Here m_p selects fluid cells whose p -neighbor lies across a solid face. For the standard 7-point pressure Laplacian, $w_p^{(K)} = 1$ for all axis-aligned neighbors, up to the global $1/h^2$ scaling. The correction then reduces to

$$C_{N,MAC}(u) = \left(\sum_p m_p \right) u = \sigma_{obs} u, \quad \sigma_{obs} = \sum_p m_p.$$

Thus σ_{obs} is simply the per-cell obstacle count. Since this count depends only on the domain geometry, it can be precomputed and cached before the pressure solve. The staggered correction is therefore a cached pointwise diagonal correction, with no reflected fetch and no affine right-hand-side term.

Advection requires a different primitive from fixed stencil operators. Semi-Lagrangian advection [Stam 1999] backtraces each degree of freedom and interpolates the transported field from neighboring grid samples. On sparse grids, this becomes a sparse interpolation query around each backtraced position rather than a fixed convolution. We include this operator in our lexicon because it is essential for fluid simulation and exposes another concrete target for backend optimization.

6 Sparse Domain Adaptation

Our formulation maps naturally to sparse convolution backends such as fVDB [Williams et al. 2024], but efficient execution requires translating the dense masking logic into sparse active-set operations. Sparse convolution distinguishes between a *source grid*, which defines active input degrees of freedom, and a *target grid*, which defines where outputs are produced. This provides a sparse-domain replacement for the masks in Algorithm 1: instead of evaluating an operator everywhere and multiplying by a mask, we choose the target grid to contain only the nodes where the output is required. The bulk stencil is therefore evaluated on the active simulation domain, whereas boundary correction terms use boundary-adjacent nodes as their target grid. Since these corrections are only needed near the boundary, restricting their target grid avoids evaluating them over the full sparse domain.

Limitations of naive sparse convolution. A direct sparse convolution call, however, is not optimal for PDE stencil workloads. Generic sparse convolution assumes a full stencil support, such as the 27 offsets of a $3 \times 3 \times 3$ kernel. Common PDE stencils are much sparser. The 3D Laplacian, for example, uses only 7 nonzero entries. The resulting overhead is not limited to redundant arithmetic on zero-weight offsets. Sparse convolution backends also build and traverse *kernel maps* [Choy et al. 2019; Williams et al. 2024] that store source-target relationships for stencil offsets, so unused offsets can still incur bookkeeping, memory traffic, and lookup cost.

This overhead is pronounced for PDE solvers because they typically operate on scalar fields with batch size one and channel count

one. Unlike high-channel neural network layers, there is little arithmetic with which to amortize sparse-topology traversal. The operation is therefore memory-bandwidth bound, dominated by neighbor fetches and kernel map access rather than floating point work. On the backend side, this motivates stencil-specialized sparse convolution implementations that evaluate only the nonzero stencil entries and avoid unnecessary kernel-map construction or traversal.

Backend Optimization Strategy. The differential operators involved in pressure projection lie in a few-channel stencil regime, unlike typical ML convolutions over high-dimensional feature tensors. Their performance is therefore dominated by sparse topology lookup and data movement rather than arithmetic throughput. We specialize the *fVDB* backend using a shared sparse-stencil kernel template that assigns one CUDA thread block to each 8^3 target leaf, with one thread per output voxel. The block first probes the surrounding $3 \times 3 \times 3$ source-leaf neighborhood and caches the resulting 27 leaf pointers in shared memory. It then stages only the feature values required by the one-voxel halo, forming a local 10^3 dense tile in shared memory; missing leaves and inactive voxels contribute zero. This replaces repeated voxel-wise sparse-tree traversal with 27 leaf probes followed by leaf-local accesses.

The stencil support is specified at compile time using only its nonzero entries. Consequently, a 7-point Laplacian evaluates seven entries rather than traversing all 27 offsets of a dense $3 \times 3 \times 3$ kernel, while other differential operators use the same kernel structure by supplying a different set of stencil offsets and weights. This computation is unrolled at compile time, and the same template supports Laplacian, gradient, divergence, and MAC-grid variants. This makes the optimization a reusable sparse differential-operator primitive rather than a routine-specific implementation. Section 7 evaluates our implementation following this design.

7 Results

In this section, we design multiple tasks to evaluate our method across different PDEs, domain geometries, backends, and boundary-condition settings. All experiments were conducted on a machine equipped with an Intel(R) Xeon(R) Gold 6312U CPU @ 2.40 GHz and an NVIDIA RTX A6000 GPU.

7.1 Evaluation

Solver Verification. We first verify correctness and robustness of our formulation on a 3D Poisson problem with a smooth analytical reference. As shown in Fig. 4, baseline stationary and unpreconditioned Krylov solvers stagnate on this large, ill-conditioned system, while our geometric multigrid and multigrid-preconditioned BiCGSTAB converge rapidly to machine precision and match the ground truth.

Sparse benchmark. Table 1a reports timings for the sparse MAC-grid smoke benchmark in Figure 5. With the optimized differential-operator kernel from Section 6, the Laplacian, divergence, and gradient reach effective bandwidths of 17%, 14%, and 11%, respectively. For the Laplacian operator, our formulation on *fVDB* with optimized convolution is 3.2× faster than Taichi. As a result, we achieved 3.9× and 2.9× speedups over the Taichi baseline for pressure projection

Table 1. Fluid benchmark results. The sparse benchmark uses a 384^3 domain with 29.65M active cells, while the dense benchmark uses a 256^3 domain. Both benchmarks use 200 CG iterations and MacCormack advection. All timings are in milliseconds. Percentages indicate effective bandwidth relative to the 768 GB/s peak bandwidth of the NVIDIA RTX A6000 GPU.

(a) Sparse benchmark using *fVDB* as the backend. Ours (opt) denotes our formulation with optimized differential convolution operators. Red entries indicate operators accelerated by our few-channel convolution backend optimization.

Component	Ours (naive)	Ours (opt)	Taichi
Density advection	100.25 (1%)	100.18 (1%)	20.53 (4%)
Velocity advection	376.15 (0%)	376.01 (0%)	28.22 (3%)
Divergence	229.86 (0%)	4.51 (14%)	3.96 (16%)
Laplacian	173.59 (0%)	1.79 (17%)	5.79 (5%)
Gradient	286.58 (0%)	5.49 (11%)	7.75 (8%)
Projection	36373.58	1393.13	5489.16
Full step	37012.03	1887.51	5541.04

(b) Dense benchmark. Both PhiFlow and our method use PyTorch as the backend.

Component	PhiFlow	Ours	Taichi
Density advection	72.87 (1%)	7.56 (6%)	2.38 (18%)
Velocity advection	238.47 (0%)	22.97 (2%)	6.03 (9%)
Divergence	8.62 (4%)	0.39 (90%)	0.56 (62%)
Laplacian	21.28 (1%)	0.77 (23%)	0.59 (29%)
Gradient	14.62 (2%)	0.60 (58%)	1.25 (28%)
Projection	43202.63	1069.69	490.04
Full step	43558.50	1101.98	500.86

and full simulation step respectively. Figure 6 breaks down the full simulation step at the ML-operator level. Naive sparse convolution dominates the step, whereas the optimized convolution accounts for about 11% of the total runtime. Figure 7 reports peak CUDA memory under a sphere-radius sweep. As the active set grows from 0.6M to 70.3M cells with increasing sphere radius, peak reserved memory increases from 0.24 GiB to 28.92 GiB.

Dense benchmark. Table 1b reports timings for the dense 256^3 benchmark. All methods use JIT compilation for a fair comparison. Our PyTorch implementation is 39.5× faster than PhiFlow and 2.2× slower than the hand-tuned Taichi baseline. We note that the performance gap between PhiFlow and our implementation in Table 1b should not be interpreted as evidence that our formulation in Section 4 is intrinsically superior to PhiFlow’s numerical formulation. In our profiling, much of PhiFlow’s overhead comes from intermediate tensor creation and framework-level abstraction costs. The distinction we emphasize in this benchmark is the exposure of the API abstraction. By making the ML-operator API explicit, our formulation allows sparse linear-algebra operations to be assembled directly from the underlying ML primitives, reducing abstraction overhead and enabling a more general simulation framework. We further discuss the source of PhiFlow’s overhead in our supplementary material.

7.2 Applications

Sparse Fluid Simulation. We simulate several sparse smoke scenarios with the fVDB backend and a staggered MAC discretization on a prescribed fixed domain topology. Our examples include a spherical-topology smoke jet used for benchmarking (Fig. 5), a decanter scene on a 1024^3 grid with 7.5% active cells and MacCormack [Selle et al. 2008] advection (Fig. 8), and a highly sparse 1024^3 smoke simulation on a trefoil topology with approximately 0.1% active voxels, solved with CG (Fig. 9). The first two use multigrid-preconditioned conjugate gradients (MGPCG) for pressure projection. Because uniform stencil evaluation and localized boundary correction are separated, computations are restricted to active regions and boundary-adjacent targets rather than the dense embedding grid. These examples use the same operator formulation on sparse data structures for both large-scale smoke simulation and sparse operator benchmarking.

Cooling Simulation. We simulate cooling of the *David* statue to demonstrate the generality of our formulation. Using Backward Euler for transient convection–diffusion, each time step requires solving Eq. (1) with adjusted coefficients, effective reaction $\sigma = -1/\Delta t$ and source $f = -(1/\Delta t)u^n$. Here α varies spatially (statue vs. air) and ω is imported from the fluid simulation. We use mixed boundary conditions, with Dirichlet conditions at the cold-air inlet, homogeneous Neumann conditions on the insulated walls, and Robin conditions on the partially insulated outlet. Figure 10 shows the resulting cooling dynamics.

Reconstructing a Coefficient. We test differentiability by reconstructing a spatially varying coefficient α in

$$\nabla \cdot (\alpha \nabla u) = f \quad \text{in } \Omega, \quad (17)$$

given a target field u over Ω . This can be viewed as a simplified Electrical Impedance Tomography (EIT) setting, where we assume dense observations of u rather than boundary-only data. More realistic EIT setups with partial measurements are discussed by Yilmazer et al. [2024]. We optimize α by minimizing the residual loss

$$\mathcal{L}(\alpha) = \|\nabla \cdot (\alpha \nabla u) - f\|_2^2, \quad (18)$$

where the residual is evaluated with our forward operator and gradients w.r.t. α follow directly from autodiff. Figure 11 shows the reconstructed coefficient field.

Inverse fluid control. We optimize the initial velocity field of a 2D 256^2 smoke simulation so that the final density matches a SIGGRAPH logo target. Following the adjoint-based fluid control approach of McNamara et al. [2004], we compute gradients through the simulation using our adjoint differentiation pipeline. Figure 12 shows the resulting simulation trajectory and target frame.

8 Limitations and Future Work

Our current formulation has several limitations. Our method is not intended to replace general-purpose GPU programming systems such as Taichi or Warp, which provide greater flexibility for arbitrary computation. Instead, we focus on a narrower class of grid-based PDE solvers that can be expressed through ML operators, trading some generality for a compact operator-level abstraction. Our optimization case study focuses on sparse stencil evaluation, but

other operators, especially semi-Lagrangian advection, may benefit from dedicated sparse implementations. Our Cartesian discretization preserves regular stencil structure but yields staircase boundaries. Ghost-fluid or cut-cell extensions could provide sharper boundary geometry. The current formulation targets dense and sparse Cartesian grids, and extending the operator view to unstructured meshes would require mesh-specific neighborhood and gather/scatter primitives. The core operator formulation is limited in scope, primarily targeting second-order linear elliptic operators and the elliptic subproblems that arise in simulation. Non-elliptic components such as fluid advection are handled through separate sampling-based operators. The method could also be combined with learned components, such as learned preconditioners or learned upsampling operators, while retaining established numerical solvers as the backbone.

References

- Deniz A Bezgin, Aaron B Buhendwa, and Nikolaus A Adams. 2023. JAX-Fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows. *Computer Physics Communications* 282 (2023), 108527.
- Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM transactions on graphics (TOG)* 22, 3 (2003), 917–924.
- Michail Boulasikis, Flavius Gruian, and Robert-Zoltan Szasz. 2025. Using machine learning hardware to solve linear partial differential equations with finite difference methods. *International Journal of Parallel Programming* 53, 2 (2025), 15.
- Bernhard Braun, Jan Bender, and Nils Thuerey. 2025. Adaptive Phase-Field-FLIP for Very Large Scale Two-Phase Fluid Simulation. *ACM Transactions on Graphics (TOG)* 44, 4 (2025), 1–23.
- Boyang Chen, Claire E Heaney, Jefferson LMA Gomes, Omar K Matar, and Christopher C Pain. 2024. Solving the discretised multiphase flow equations with interface capturing on structured grids using machine learning libraries. *Computer Methods in Applied Mechanics and Engineering* 426 (2024), 116974.
- Boyang Chen, Claire E Heaney, and Christopher C Pain. 2026. Neural Physics: Using AI Libraries to Develop Physics-Based Solvers for Incompressible Computational Fluid Dynamics. *Computers & Fluids* (2026), 106981.
- Thomas Chen, Jacob Botimer, Teyuh Chou, and Zhengya Zhang. 2019. An SRAM-based accelerator for solving partial differential equations. In *2019 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE, 1–4.
- Christopher Choy, JunYoung Gwak, and Silvio Savarese. 2019. 4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3075–3084.
- Robert D Falgout, Jim E Jones, and Ulrike Meier Yang. 2006. The design and implementation of hypre, a library of parallel high performance preconditioners. In *Numerical solution of partial differential equations on parallel computers*. Springer, 267–294.
- Zeev Farbman, Raanan Fattal, and Dani Lischinski. 2011. Convolution pyramids. *ACM Trans. Graph.* 30, 6 (2011), 175.
- Michael B Giles and Niles A Pierce. 2000. An introduction to the adjoint approach to design. *Flow, turbulence and combustion* 65, 3 (2000), 393–415.
- Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. Vol. 1. MIT press Cambridge.
- Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. 2005. A multigrid solver for boundary value problems using programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses*. 193–es.
- Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM.
- Xi Han, Fei Hou, and Hong Qin. 2024. UGrid: An efficient-and-rigorous neural multigrid solver for linear PDEs. *arXiv preprint arXiv:2408.04846* (2024).
- Juncai He, Xinliang Liu, and Jinchao Xu. 2023. MgNO: Efficient parameterization of linear operators via multigrid. *arXiv preprint arXiv:2310.19809* (2023).
- Juncai He and Jinchao Xu. 2019. MgNet: A unified framework of multigrid and convolutional neural network. *Science china mathematics* 62, 7 (2019), 1331–1354.
- Philipp Holl, Vladlen Koltun, Kiwon Um, and Nils Thuerey. 2020. phiflow: A differentiable pde solving framework for deep learning via physical simulations. In *NeurIPS workshop*, Vol. 2.
- Jun-Ting Hsieh, Shengjia Zhao, Stephan Eismann, Lucia Mirabella, and Stefano Ermon. 2019. Learning neural PDE solvers with convergence guarantees. *arXiv preprint arXiv:1906.01200* (2019).
- Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–16.

- Ru Huang, Ruipeng Li, and Yuanzhe Xi. 2022. Learning optimal multigrid smoothers via neural networks. *SIAM Journal on Scientific Computing* 45, 3 (2022), S199–S225.
- Carlo Innamorati, Tobias Ritschel, Tim Weyrich, and Niloy J Mitra. 2018. Learning on the edge: Explicit boundary handling in CNNs. *arXiv preprint arXiv:1805.03106* (2018).
- Doyub Kim, Minjae Lee, and Ken Museth. 2024. Neuralvdb: High-resolution sparse volume representation using hierarchical neural networks. *ACM Transactions on Graphics* 43, 2 (2024), 1–21.
- Hans Knutsson and C-F Westin. 1993. Normalized and differential convolution. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 515–523.
- Jens Krüger and Rüdiger Westermann. 2005. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM SIGGRAPH 2005 Courses*. 234–es.
- Lin Feng Li, Jiansheng Xiang, Boyang Chen, Claire E Heaney, Steven Dargaville, and Christopher C Pain. 2025. Implementing the discontinuous-Galerkin finite element method using graph neural networks with application to diffusion equations. *Neural Networks* 185 (2025), 107061.
- Guilin Liu, Fitsum A Reda, Kevin J Shih, Ting-Chun Wang, Andrew Tao, and Bryan Catanzaro. 2018. Image inpainting for irregular holes using partial convolutions. In *European conference on computer vision*. Springer, 89–105.
- Antoine McNamara, Adrien Treuille, Zoran Popović, and Jos Stam. 2004. Fluid control using the adjoint method. *ACM Transactions On Graphics (TOG)* 23, 3 (2004), 449–456.
- Paulius Micikevicius. 2009. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd workshop on general purpose processing on graphics processing units*. 79–84.
- Erdem Motuk, Roger Woods, and Stefan Bilbao. 2005. Implementation of finite difference schemes for the wave equation on FPGA. In *Proceedings (ICASSP'05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, Vol. 3. IEEE, iii–237.
- Ken Museth. 2021. NanoVDB: A GPU-friendly and portable VDB data structure for real-time rendering and simulation. In *ACM SIGGRAPH 2021 Talks*. 1–2.
- Ken Museth, Jeff Lait, John Johanson, Jeff Budsberg, Ron Henderson, Mihai Alden, Peter Cucka, David Hill, and Andrew Pearce. 2013. OpenVDB: an open-source data structure and toolkit for high-resolution volumes. In *Acm siggraph 2013 courses*. 1–1.
- Maxim Naumov, Marat Arsaev, Patrice Castonguay, Jonathan Cohen, Julien Demouth, Joe Eaton, Simon Layton, Nikolay Markovskiy, István Reguly, Nikolai Sakharlykh, et al. 2015. AmgX: A library for GPU accelerated algebraic multigrid and pre-conditioned iterative methods. *SIAM Journal on Scientific Computing* 37, 5 (2015), S602–S626.
- Mike Pereira, Nicolas Desassis, Cédric Magneron, and Nathan Palmer. 2020. A matrix-free approach to geostatistical filtering. *arXiv preprint arXiv:2004.02799* (2020).
- Toby RF Phillips, Claire E Heaney, Boyang Chen, Andrew G Buchan, and Christopher C Pain. 2023. Solving the discretised neutron diffusion equations using neural networks. *Internat. J. Numer. Methods Engrg.* 124, 21 (2023), 4659–4686.
- Damien Pierce, R Lily Hu, Yusef Shafi, Anudhyan Boral, Vladimir Anisimov, Sella Nevo, and Yi-fan Chen. 2022. Accelerating Physics Simulations with TPUs: An Inundation Modeling Example. *arXiv preprint arXiv:2204.10323* (2022).
- Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. 2008. An unconditionally stable MacCormack method. *Journal of Scientific Computing* 35, 2 (2008), 350–371.
- Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. 2014. SPGrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Transactions on Graphics (TOG)* 33, 6 (2014), 1–12.
- Han Shao, Libo Huang, and Dominik L Michels. 2022. A fast unsmoothed aggregation algebraic multigrid framework for the large-scale simulation of incompressible flow. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–18.
- Jos Stam. 1999. Stable Fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*. 121–128. doi:10.1145/311535.311548
- Francis Williams, Jiahui Huang, Jonathan Swartz, Gergely Klar, Vijay Thakkar, Matthew Cong, Xuanchi Ren, Ruilong Li, Clement Fuji-Tsang, Sanja Fidler, et al. 2024. fvdB: A deep-learning framework for sparse, large scale, and high performance spatial intelligence. *ACM Transactions on Graphics (TOG)* 43, 4 (2024), 1–15.
- Kui Wu, Nghia Truong, Cem Yuksel, and Rama Hoetzlein. 2018. Fast fluid simulations with sparse volumes on the GPU. In *Computer Graphics Forum*, Vol. 37. Wiley Online Library, 157–167.
- Ekrem Fatih Yilmazer, Delio Vicini, and Wenzel Jakob. 2024. Solving Inverse PDE Problems using Grid-Free Monte Carlo Estimators. *ACM Trans. Graph.* 43, 6 (2024), 175–1.

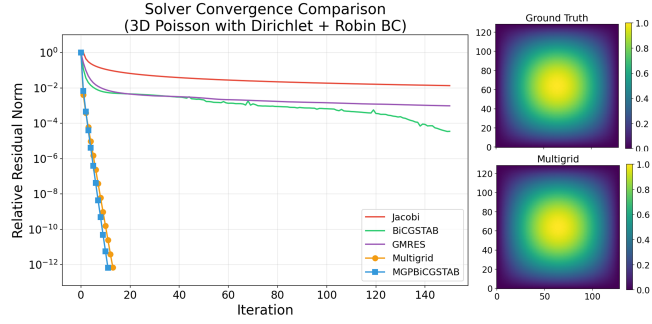


Fig. 4. **Convergence verification on a 3D benchmark (129^3) with mixed Dirichlet/Robin BCs.** Our solution matches the analytical ground truth visually, confirming numerical correctness. Multigrid and multigrid-preconditioned BiCGSTAB show robust convergence, reducing the relative residual by roughly one order of magnitude per iteration.

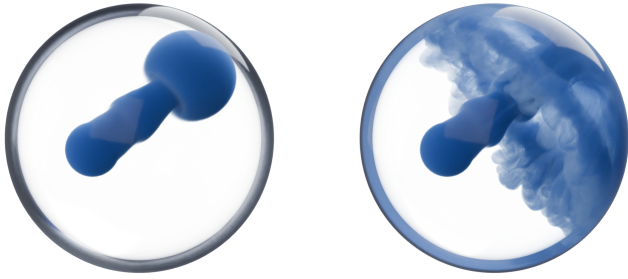


Fig. 5. Benchmark scenario: a smoke jet evolving inside a spherical domain. We use this setup for both dense and sparse performance measurements.

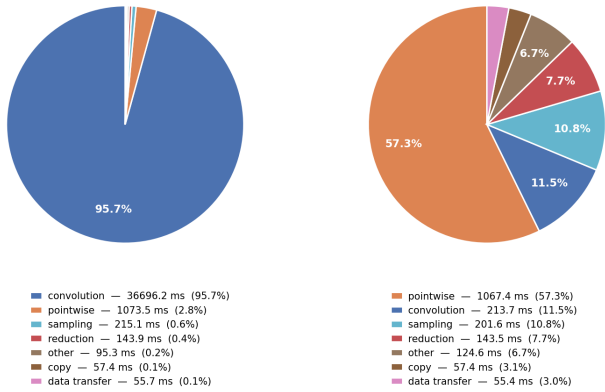


Fig. 6. **ML-operator runtime breakdown for a sparse fluid simulation step.** With naive fVDB sparse convolution (left), convolution dominates the runtime. After our operator-level optimization (right), convolution accounts for only about 11% of the total runtime.

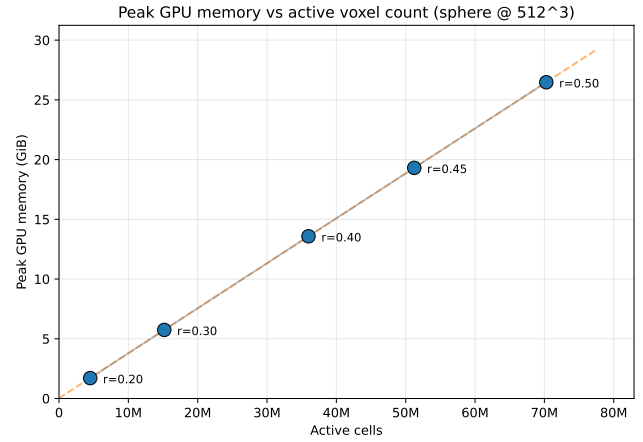


Fig. 7. Peak CUDA memory for a fixed-resolution sphere-radius sweep using MGPCG. As the sphere radius increases, the active set grows from 0.6M to 70.3M cells, and peak reserved memory increases accordingly from 0.24 GiB to 28.92 GiB.

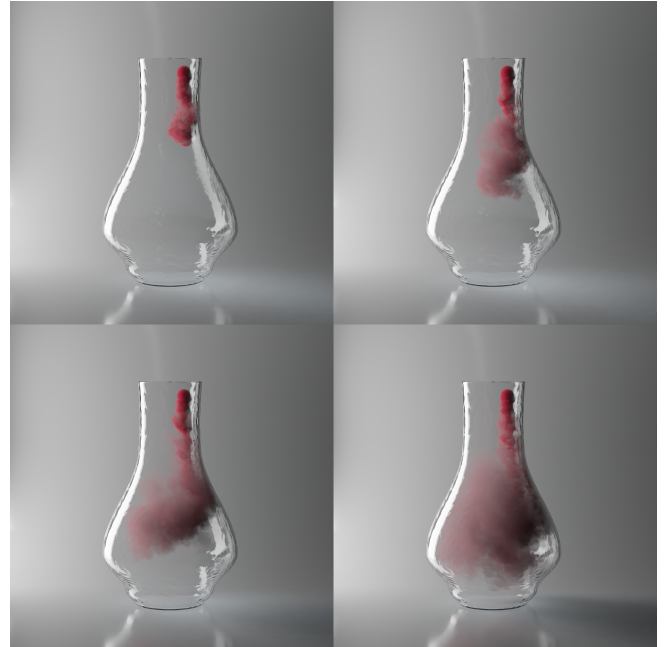


Fig. 8. Sparse smoke simulation of a wine-colored fluid pouring into a decanter. The simulation runs on a 1024^3 grid with 7.5% active cells and uses MacCormack advection.

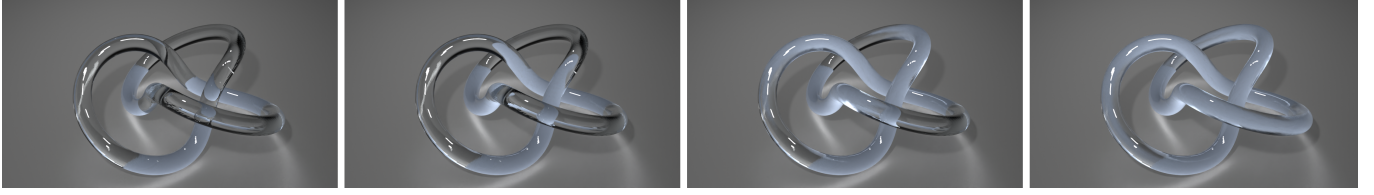


Fig. 9. Giga-scale sparse fluid simulation on a 1024^3 grid using $fVDB$, where only the trefoil-shaped domain is active (1.26M voxels, $\approx 0.1\%$ of the full grid). We implement the solver entirely with $fVDB$ primitives. A gentle tangential force drives the flow along the trefoil, and the differential operators are expressed as high-level convolution calls executed only on the sparse active region.

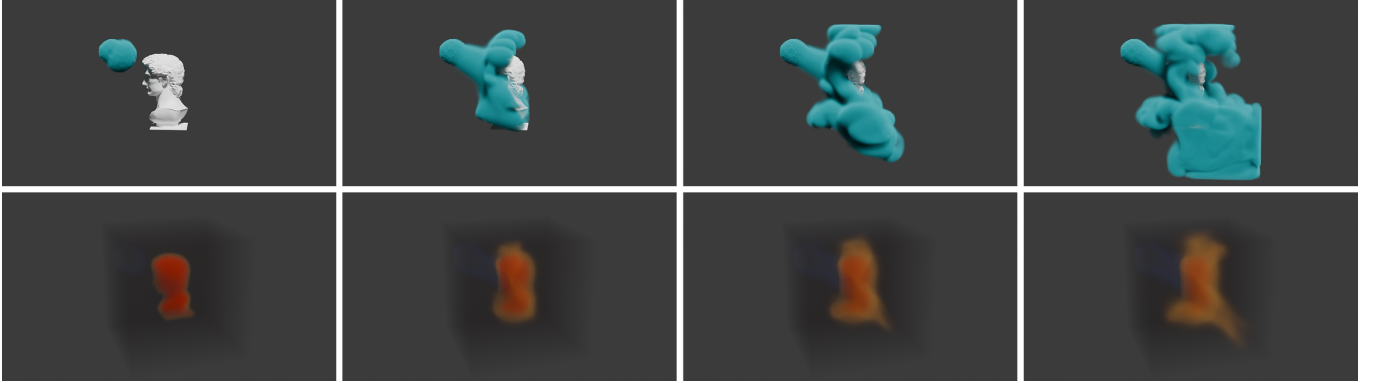


Fig. 10. Cooling simulation on the *David* statue. The upper row shows the driving fluid simulation, and the lower row visualizes the resulting temperature field. We solve a transient convection–diffusion equation with Backward Euler, resulting in an elliptic solve at each time step. The spatially varying conductivity α distinguishes the statue from the surrounding air, and the advection field ω is taken from the fluid simulation. Boundary conditions are mixed: Dirichlet at the cold-air inlet, homogeneous Neumann on insulated walls, and Robin (partially insulated) at the outlet.

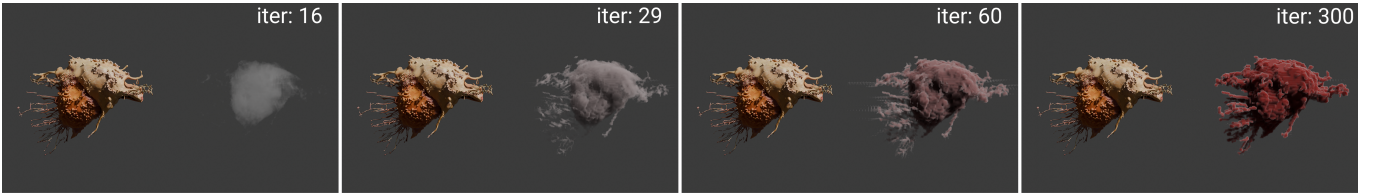


Fig. 11. A simplified EIT-like inverse problem. The tumor has higher electrical conductivity α than the surrounding background. Given voltage measurements $\mathbf{u}$ and current density $\mathbf{f}$, we reconstruct α . Left: tumor mesh (ground truth). Right: reconstructed α , where more reddish values are closer to the tumor conductivity and more whitish values are closer to the background. This example highlights the differentiability of our convolutional solver.



Fig. 12. Inverse fluid control by optimizing the initial velocity field to match a SIGGRAPH logo target. Frames progress left to right; the red rectangle indicates the target frame.