Optimized Sparse Matrix Operations for Reverse Mode Automatic Differentiation

Nicolas Nytko [1] **Ali Taghibakhshi** [2] **Tareq Uz Zaman** [3] **Scott MacLachlan** [4] **Luke N. Olson** [1] **Matt West** [2]

Abstract

Sparse matrix representations are ubiquitous in computational science and machine learning, leading to significant reductions in compute time, in comparison to dense representation, for problems that have local connectivity. The adoption of sparse representation in leading ML frameworks such as PyTorch is incomplete, however, with support for both automatic differentiation and GPU acceleration missing. In this work, we present an implementation of a CSR-based sparse matrix wrapper for PyTorch with CUDA acceleration for basic matrix operations, as well as automatic differentiability. We also present several applications of the resulting sparse kernels to optimization problems, demonstrating ease of implementation and performance measurements versus their dense counterparts.

1. Introduction

The popularity of automatic differentiation (AD) support in libraries such as PyTorch (Paszke et al., 2019), Tensorflow (Abadi et al., 2016), or Jax (Bradbury et al., 2018) has exploded in recent years, allowing rapid development of models and optimization problems without requiring analytical gradients before training. These libraries allow taking the gradient of complex chains of operations by decomposing them into small, atomic operations then linking them together by copious usage of the chain rule. As an example, differentiation through traditional matrix-matrix multiplication or matrix addition is straightforward and efficient.

While dense matrix linear algebra routines are readily implemented and available in these frameworks, their sparse counterparts have received less attention. In the case of PyTorch, Tensorflow, and Jax, all three have preliminary support for sparse matrices stored in the coordinate (COO) format. Yet, the use of sparse linear algebra is extensive in computational science, for example in the solution to partial differential equations (PDEs) (Briggs et al., 2000; Saad, 2003), circuit analysis (Bonfatti et al., 1973), graph neighborhood analysis (Saad, 2003; George et al., 1993), among many others, motivating the need for a more complete treatment of sparse operations. Moreover, there is growing demand for sparse kernels in the broader area of scientific machine learning (SciML), where machine learning techniques augment traditional scientific computing methods. The authors in (Greenfeld et al., 2019; Luz et al., 2020) learn methods to accelerate matrix linear solve routines, though develop various simplifications in the problems themselves due to the lack of sparse linear algebra routines. Similarly, the method presented in (Taghibakhshi et al., 2022) is constrained to training on small, dense matrices, then evaluated on arbitrarily sized inputs without any AD considerations. We thus seek an implementation of such sparse routines to allow machine learning researchers and computational scientists to develop scalable, learned methods.

The main contributions of this paper are structured as follows. We present preliminary background on AD internals and the sparse CSR matrix format in Section 2, then state the backwards-mode gradient derivations for several sparse operations and their implementation on both CPU and GPU (CUDA) processors in Section 3. To help motivate further possible uses of these sparse routines, we introduce several applications to optimization problems in Section 4, including graph neural networks (Wu et al., 2021; Kipf & Welling, 2017). Finally, we present performance results for our implementation in Section 5, underscoring the efficiency of the methods in the forward pass by comparing to existing sparse-matrix packages.

2. Background

Unless noted, $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ and $\boldsymbol{b}, \boldsymbol{x} \in \mathbb{R}^n$ represent the variables in the (sparse) system of linear equations

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}. \tag{1}$$

[1]Department of Computer Science, University of Illinois at Urbana-Champaign [2]Department of Mechanical Science and Engineering, University of Illinois at Urbana-Champaign [3]Scientific Computing Program, Memorial University of Newfoundland [4]Department of Mathematics and Statistics, Memorial University of Newfoundland. Correspondence to: Nicolas Nytko <nnytko2@illinois.edu>.

Proceedings of the 39^{th} International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022.

For iterative solvers, $\boldsymbol{x}_k$ denotes the approximate solution to Equation (1) at iteration k, along with residual $\boldsymbol{r}_k = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k$, and error $\boldsymbol{e}_k = \boldsymbol{x} - \boldsymbol{x}_k$.

Given a sparse matrix $\boldsymbol{M}$, let mask $(\boldsymbol{M})$ denote the *sparsity mask* of $\boldsymbol{M}$ as in

$$[\text{mask}\,(\boldsymbol{M})]_{ij} = \begin{cases} 1 & m_{ij} \neq 0 \\ 0 & \text{otherwise} \end{cases}. \tag{2}$$

Letting $\odot$ denote the standard Hadamard product, $A \odot \text{mask}\,(\boldsymbol{M})$ then represents (sparse) matrix $\boldsymbol{A}$ masked to the sparsity of $\boldsymbol{M}$. The sparsity mask also leads to the identity $\boldsymbol{M} \odot \text{mask}\,(\boldsymbol{M}) = \boldsymbol{M}$, and, by convention, if $\boldsymbol{M}$ is dense then mask $(\boldsymbol{M})$ is also dense.

2.1. Chain Rule

For background, we recall the multivariate chain rule (Johnson, 2017) in Theorem 2.1.

Theorem 2.1. *Given function $f : \mathbb{R}^{n_x} \to \mathbb{R}$ and vectors $\boldsymbol{t} \in \mathbb{R}^{n_t}$ and $\boldsymbol{x}(\boldsymbol{t}) \in \mathbb{R}^{n_x}$, the partial derivative $\frac{\partial z}{\partial t_i}$ for $z = f(\boldsymbol{x}(\boldsymbol{t}))$ is given by*

$$\frac{\partial z}{\partial t_i} = \sum_{j=1}^{n_x} \frac{\partial f}{\partial x_j} \frac{\partial x_j}{\partial t_i}. \tag{3}$$

For the general case, we can extend Theorem 2.1 for arbitrary tensor-valued functions, with tensors representing n-dimensional arrays (Kolda & Bader, 2009), as in Corollary 2.2.

Corollary 2.2. *Let $I_1, I_2, \ldots I_N$ denote the indices of an N-mode tensor. Given function $f : \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_{N_X}} \to \mathbb{R}$, and tensors $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_{N_T}}$ and $\boldsymbol{\mathcal{X}}(\boldsymbol{\mathcal{T}}) \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_{N_X}}$, the partial derivative $\frac{\partial z}{\partial t_{j_1, j_2, \ldots, j_{N_T}}}$ for $z = f(\boldsymbol{\mathcal{X}}(\boldsymbol{\mathcal{T}}))$ is given by*

$$\frac{\partial z}{\partial t_{j_1,\ldots,j_{N_T}}} = \sum_{i_1=1}^{I_1} \cdots \sum_{i_{N_X}=1}^{I_{N_X}} \frac{\partial z}{\partial x_{i_1,\ldots,i_{N_X}}} \frac{\partial x_{i_1,\ldots,i_{N_X}}}{\partial t_{j_1,\ldots,j_{N_T}}}. \tag{4}$$

From Corollary 2.2, we denote the term on the left side in the summation as the *generalized gradient*:

$$[\nabla_{\boldsymbol{\mathcal{X}}}(z)]_{i_1,i_2,\ldots,i_{N_X}} = \frac{\partial z}{\partial x_{i_1,i_2,\ldots,i_{N_X}}}. \tag{5}$$

This represents "the gradient of z with respect to $\boldsymbol{\mathcal{X}}$." Moreover, the value attains the same shape (dimensionality and size in each dimension) as $\boldsymbol{\mathcal{X}}$ itself.

Likewise, from Corollary 2.2, we refer to the term on the right in the summation as the *generalized Jacobian*:

$$[J_{\boldsymbol{\mathcal{T}}}(\boldsymbol{\mathcal{X}})]_{(i_1,i_2,\ldots,i_{N_X}),(j_1,j_2,\ldots,j_{N_T})} = \frac{\partial x_{i_1,i_2,\ldots,i_{N_X}}}{\partial t_{j_1,j_2,\ldots,j_{N_T}}}, \tag{6}$$

where the parentheses in the indexing are used only to emphasize that the first N_X indices are used for the input, while the last N_T indices are used for the output. From this, we have that Equation (4) can be alternatively denoted as the tensor contraction $\nabla_{\boldsymbol{\mathcal{X}}}(z)^T J_{\boldsymbol{\mathcal{X}}(\boldsymbol{\mathcal{T}})}$ over the indices $i_1, i_2, \ldots, i_{N_X}$.

2.2. Reverse Mode Auto-Differentiation

Contemporary machine learning frameworks use *reverse-mode automatic differentiation* to compute gradient information, which is an efficient means of computing gradients of scalar-valued functions with respect to tensor-valued inputs (Baydin et al., 2015). The computation, in essence, is first done normally in a *forward pass*, with compositions of elementary functions being recorded into a *computation graph*. A second *backward pass* is then executed, tracing the computation graph backwards from the scalar output back to each input node and computing intermediate gradients at each step.

To help illustrate, we consider the function

$$f(\boldsymbol{x}, \boldsymbol{y}) = 2\sin(\langle \boldsymbol{x}, \boldsymbol{y} \rangle), \tag{7}$$

for which we seek to evaluate the derivative with respect to the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. At each node of the computation

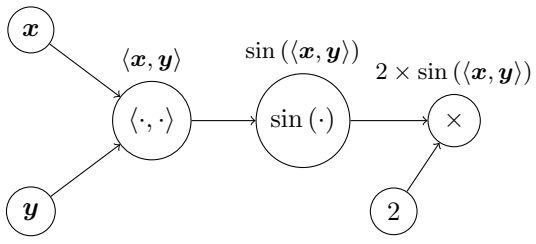

Figure 1. Example computation graph corresponding to evaluating Equation (7).

graph (see Figure 1), the gradient of the output is calculated with respect to the input of that particular node; this is then passed further back in the graph. Formally, letting z be the scalar output of our computation and $\boldsymbol{\mathcal{Y}} = f_i(\boldsymbol{\mathcal{X}})$ be the computation done at node i, we find the intermediate gradient with the tensor contraction

$$\nabla_{\boldsymbol{\mathcal{X}}}(z) = \nabla_{\boldsymbol{\mathcal{Y}}}(z)^T J_{f_i(\boldsymbol{\mathcal{X}})}(\boldsymbol{\mathcal{X}}). \tag{8}$$

This operation is referred to as the *vector-Jacobian product* (VJP) in automatic differentiation. In the case that functions are vector valued ($\mathbb{R}^n \to \mathbb{R}^m$) this simplifies to vector-matrix products.

2.3. Compressed Sparse Row format

The *compressed sparse row* format is a row-oriented representation for storage of sparse matrices (Saad, 2003). In

this representation, three arrays are needed to determine a single sparse matrix, $\boldsymbol{A} \in \mathbb{R}^{n \times m}$. Denoting $\text{nnz}(\boldsymbol{A})$ as the number of nonzero entries in $\boldsymbol{A}$, the three arrays become:

- `data` of size $\text{nnz}(\boldsymbol{A})$ storing each nonzero data entry sorted in row then column order,
- `indices` of size $\text{nnz}(\boldsymbol{A})$ storing the column number of each respective nonzero entry, and
- `indptr` of size $m+1$ that stores the *offset* into `data` of the starting point of each row.

By convention, the last entry of `indptr` stores the number of nonzeros in $\boldsymbol{A}$.

$$\boldsymbol{C} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 5 \end{bmatrix}$$

$$\begin{aligned} \texttt{data}(\boldsymbol{C}) &= \begin{bmatrix} 1 & 3 & 2 & 7 & 5 \end{bmatrix} \\ \texttt{indices}(\boldsymbol{C}) &= \begin{bmatrix} 0 & 1 & 2 & 0 & 3 \end{bmatrix} \\ \texttt{indptr}(\boldsymbol{C}) &= \begin{bmatrix} 0 & 1 & 3 & 3 & 5 \end{bmatrix} \end{aligned}$$

Figure 2. An example sparse matrix and its CSR representation.

We use the CSR representation in this work because it provides good memory access to the matrix rows, especially useful in products where the matrix is left-multiplying, for example in the sparse matrix-vector product $\boldsymbol{Ax}$ or sparse-times-dense matrix product $\boldsymbol{AD}$, where $\boldsymbol{A}$ is sparse and $\boldsymbol{x}$, $\boldsymbol{D}$ are a dense vector and matrix.

3. Sparse Kernels

For this work, we implement optimized forward and backward passes (VJP implementations) for various sparse operations with an underlying CSR data structure. We propagate *sparse* gradients whereby functions that take sparse inputs will have gradients whose sparsity mask will match those of the inputs. This trade-off leads to much smaller memory and computational costs (as observed indirectly in Table 2) as, during optimization, only nonzero entries in the inputs will receive gradients. For example, optimizing a problem with a sparse-matrix input and scalar output will only lead to optimization of the nonzero entries of the input.

We next detail the forward implementations and backwards (VJP) products of several sparse operations. For derivations of the matrix reverse-mode updates, we point the reader to (Giles, 2008) and remark that with minor consideration of the sparsity patterns, the results extend to sparse matrices. In the following, not all routines have highly tuned implementations for both CPU and CUDA processors.

Table 1. Definitions of vector-Jacobian products for the different sparse operations. The vector in the VJP (denoted by $\boldsymbol{v}$ or $\boldsymbol{V}$) is the intermediate gradient with respect to the output of the operation when running backpropagation.

METHOD	OPERATION	WRT	VJP
SPMV	$\boldsymbol{Ax}$	$\boldsymbol{A}$	$\boldsymbol{vx}^T \odot \text{mask}(\boldsymbol{A})$
SPMV	$\boldsymbol{Ax}$	$\boldsymbol{x}$	$\boldsymbol{A}^T\boldsymbol{v}$
SPSPMM	$\boldsymbol{AB}$	$\boldsymbol{A}$	$\left(\boldsymbol{VB}^T\right) \odot \text{mask}(\boldsymbol{A})$
SPSPMM	$\boldsymbol{AB}$	$\boldsymbol{B}$	$\left(\boldsymbol{A}^T\boldsymbol{V}\right) \odot \text{mask}(\boldsymbol{B})$
SPDMM	$\boldsymbol{AB}$	$\boldsymbol{A}$	$\left(\boldsymbol{VB}^T\right) \odot \text{mask}(\boldsymbol{A})$
SPDMM	$\boldsymbol{AB}$	$\boldsymbol{B}$	$\boldsymbol{A}^T\boldsymbol{V}$
SP + SP	$\alpha\boldsymbol{A} + \beta\boldsymbol{B}$	$\boldsymbol{A}$	$\alpha\boldsymbol{V} \odot \text{mask}(\boldsymbol{A})$
SP + SP	$\alpha\boldsymbol{A} + \beta\boldsymbol{B}$	$\boldsymbol{B}$	$\beta\boldsymbol{V} \odot \text{mask}(\boldsymbol{B})$
SPSOLVE	$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$	$\boldsymbol{A}$	$-\boldsymbol{A}^{-T}\boldsymbol{vx}^T \odot \text{mask}(\boldsymbol{A})$
SPSOLVE	$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$	$\boldsymbol{b}$	$\boldsymbol{A}^{-T}\boldsymbol{v}$

3.1. Sparse Matrix-Vector (SpMV)

The sparse matrix-vector product computes

$$\boldsymbol{y} \leftarrow \boldsymbol{Ax} \tag{9}$$

for sparse $\boldsymbol{A} \in \mathbb{R}^{n \times m}$, dense $\boldsymbol{x} \in \mathbb{R}^m$, and outputs $\boldsymbol{y} \in \mathbb{R}^n$. The CSR representation allows for direct evaluation of inner products of each row of $\boldsymbol{A}$ and $\boldsymbol{x}$, and the inner products can be executed in parallel on a GPU.

For the backwards pass, we compute $\boldsymbol{A}^T\boldsymbol{v}$ (resp. $\boldsymbol{vx}^T$) for the gradient with respect to (WRT) $\boldsymbol{x}$ (resp. $\boldsymbol{A}$), for intermediate gradient value $\boldsymbol{v}$. Computing the transposed matrix-vector product does not directly align with the sparse structure, therefore we opt to use an atomic reduction-based algorithm to compute $\boldsymbol{A}^T\boldsymbol{v}$ as outlined in (Tao et al., 2014). In this form, we compute inner products of the rows of $\boldsymbol{A}$ and $\boldsymbol{v}$ and atomically reduce the output into their correct entries. Computing $\boldsymbol{vx}^T$ is easily parallelized, since it is the outer product between two dense vectors that is masked to a sparse matrix, requiring computation of only nonzero entries of $\boldsymbol{A}$.

3.2. Sparse-Sparse Matrix Multiply (SpSpMM)

For the sparse-sparse matrix multiply primitive, we compute

$$\boldsymbol{C} \leftarrow \boldsymbol{AB}, \tag{10}$$

where $\boldsymbol{A} \in \mathbb{R}^{i \times j}$, $\boldsymbol{B} \in \mathbb{R}^{j \times k}$, and $\boldsymbol{C} \in \mathbb{R}^{i \times k}$, with all three matrices are stored in CSR format. In the forward pass, we follow the parallel SpGEMM algorithm from (Dalton et al., 2015), where we compute intermediate products of each row of $\boldsymbol{A}$ and the entirety of $\boldsymbol{B}$, then reduce and collect redundant entries to form $\boldsymbol{C}$.

For the backward pass, we compute $\left(\boldsymbol{VB}^T\right) \odot \text{mask}(\boldsymbol{A})$

and $\left(\boldsymbol{A}^T\boldsymbol{V}\right) \odot \text{mask}\left(\boldsymbol{B}\right)$. Noting that the product $\boldsymbol{V}\boldsymbol{B}^T$ is the inner product between rows of both $\boldsymbol{V}$ and $\boldsymbol{B}$, our data accesses aligns directly with the CSR structure. Computing $\boldsymbol{A}^T\boldsymbol{V}$, however, accesses columns of both $\boldsymbol{A}$ and $\boldsymbol{V}$; we thus follow a modified version of the SpGEMM algorithm from above that operates on columns of $\boldsymbol{A}$ instead.

3.3. Sparse-Dense Matrix Multiply (SpDMM)

For the SpDMM routine, we compute

$$\boldsymbol{C} \leftarrow \boldsymbol{A}\boldsymbol{B} \tag{11}$$

as in the SpSpMM case except for $\boldsymbol{B}$ and $\boldsymbol{C}$ now being dense. In the forward pass, we parallelize with each fine-grained operation focusing on an entry of $\boldsymbol{C}$. That is, we compute the respective inner product between a row of $\boldsymbol{A}$ and a column of $\boldsymbol{B}$, which because of its dense structure does not incur any significant penalties for column accesses.

On the backward pass, we compute $\left(\boldsymbol{V}\boldsymbol{B}^T\right) \odot \text{mask}\left(\boldsymbol{A}\right)$ (as before) and $\boldsymbol{A}^T\boldsymbol{V}$. For the latter, we take the sparse transposition of $\boldsymbol{A}$ and re-execute the forward routine to compute the gradient.

3.4. Sparse + Sparse

In a sparse add, we compute the linear combination of two sparse matrices as in

$$\boldsymbol{C} \leftarrow \alpha\boldsymbol{A} + \beta\boldsymbol{B}, \tag{12}$$

which we write in a general form so that both $\boldsymbol{A} + \boldsymbol{B}$ and $\boldsymbol{A} - \boldsymbol{B}$ can be computed with the same method. A key observation is that (with slight abuse of notation),

$$\text{mask}\left(\boldsymbol{C}\right) = \text{mask}\left(\boldsymbol{A}\right) \cup \text{mask}\left(\boldsymbol{B}\right), \tag{13}$$

meaning the computation of $\boldsymbol{C}$ can be viewed as a union over the rows of $\boldsymbol{A}$ and $\boldsymbol{B}$. Moreover, this form can be implemented in parallel over each row.

To compute the backwards pass, we consider the gradient with respect to $\boldsymbol{A}$, $\boldsymbol{B}$ as $\alpha\boldsymbol{V} \odot \text{mask}\left(\boldsymbol{A}\right)$, $\beta\boldsymbol{V} \odot \text{mask}\left(\boldsymbol{B}\right)$, respectively. Each can be found as the row-wise reduction from $\boldsymbol{V}$ to the sparsity mask of $\boldsymbol{A}$ or $\boldsymbol{B}$. Because $\text{mask}\left(\boldsymbol{A}\right) \subseteq \text{mask}\left(\boldsymbol{V}\right)$, $\text{mask}\left(\boldsymbol{B}\right) \subseteq \text{mask}\left(\boldsymbol{V}\right)$, we need only to compute matching nonzero entries in both matrices. Again, this can be implemented in parallel over the rows.

3.5. Sparse Triangular Solve

A sparse triangular solve is an operation to compute the value of $\boldsymbol{x}$ in the matrix equation

$$\boldsymbol{L}\boldsymbol{x} = \boldsymbol{b}, \tag{14}$$

where $\boldsymbol{L}$ has a lower triangular form, i.e. $l_{ij} \neq 0$ if $i \geq j$. Without loss of generality, we can also consider upper-triangular systems $\boldsymbol{U}$ using matrix flip operations to convert $\boldsymbol{U}\boldsymbol{x} = \boldsymbol{b}$ into an equivalent lower-triangular system. Such a system has a (relatively) simple routine for computing the linear solve: each row depends on the intermediate values of previous rows only, and no intermediate preprocessing is needed.

For the forward pass, we elect to use the synchronization-free GPU triangular solve detailed in (Su et al., 2020) to exploit the limited parallelism that may exist in computing $\boldsymbol{x}$.

On the backward pass, we seek $\boldsymbol{L}^{-T}\boldsymbol{v}$ and $-\boldsymbol{L}^{-T}\boldsymbol{v}\boldsymbol{x}^T \odot \text{mask}\left(\boldsymbol{L}\right)$ for gradients with respect to $\boldsymbol{b}$ and $\boldsymbol{L}$, respectively. We first find $\boldsymbol{L}^{-T}\boldsymbol{v}$ with our existing forward triangular solve routine (except flipped to run on upper triangular systems). Then we observe that the gradient with respect to $\boldsymbol{L}$ contains the $\boldsymbol{b}$ gradient term as a masked outer product, so we can re-use the result. The masked outer-product can executed in parallel over the nonzero entries of $\boldsymbol{L}$.

4. Applications of Sparse Kernels

To demonstrate the use of the sparse kernels and differentiability above, we present several optimization problems that exercise sparse-matrix computations. As an example sparse matrix, we define $\boldsymbol{A}$ as

$$\boldsymbol{A} = \begin{bmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{bmatrix}, \tag{15}$$

which is a standard test problem in computational science and comes from the finite-difference discretization of the 1D Poisson problem (Saad, 2003; Quarteroni et al., 2006),

$$-\nabla^2 u = f \qquad \text{in } \Omega, \tag{16}$$

$$u = 0 \qquad \text{on } \partial\Omega, \tag{17}$$

on the domain $\Omega = \left(0, (N+1)^2\right) \subseteq \mathbb{R}$, with N being the number of interior gridpoints in the discretization — the domain is selected to cancel any constant scaling of the matrix. This results in a constant number of nonzero entries per row (3) away from the boundary.

4.1. Entry-wise Jacobi Relaxation

The Jacobi method is an iterative solver for linear systems whereby the inverse matrix is approximated by inverting the diagonal entries only (Saad, 2003). Consider solving

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{18}$$

where $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is a sparse matrix with diagonal D. The Jacobi method is expressed as

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{D}^{-1}\left(\boldsymbol{b} - (\boldsymbol{A} - \boldsymbol{D})\,\boldsymbol{x}^{(k)}\right), \tag{19}$$

though this can often see slow convergence as the iteration tends to largely dampen *high-frequency error modes* only when $\boldsymbol{A}$ comes from discretization of an elliptic differential equation (Briggs et al., 2000).

Convergence can be accelerated by separately weighting the previous value of $\boldsymbol{x}$ and the approximate inverse at each iteration; here, we consider the case where this weighting is determined independently for each entry of $\boldsymbol{x}$. Denoting $\boldsymbol{\Omega}$ as the diagonal scaling matrix of the entries of $\boldsymbol{x}^{(k)}$, we then rewrite the Jacobi iteration as

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{\Omega}\boldsymbol{D}^{-1}\boldsymbol{b} + \left(\boldsymbol{I} - \boldsymbol{\Omega}\boldsymbol{D}^{-1}\boldsymbol{A}\right)\boldsymbol{x}^{(k)}. \tag{20}$$

To simplify the notation, we write $\boldsymbol{j}\left(\boldsymbol{x}_0, \boldsymbol{b};\ \boldsymbol{\Omega}\right)$ as the function that applies one iteration of the weighted Jacobi scheme to $\boldsymbol{x}_0$ in order to solve $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$.

We generate a finite-difference matrix $\boldsymbol{A}$ from Equation (15) of size $n = 16$, then optimize the entries of $\boldsymbol{\Omega}$ by minimizing

$$\ell = \sum_{k=1}^{K} \left(\boldsymbol{j}\left(\boldsymbol{x}_k, \boldsymbol{0};\ \boldsymbol{\Omega}\right)\right)^T \boldsymbol{A}\boldsymbol{j}\left(\boldsymbol{x}_k, \boldsymbol{0};\ \boldsymbol{\Omega}\right), \tag{21}$$

where $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_K\}$ are a set of test vectors in $\mathbb{R}^n$ with random unit-normally distributed entries. This optimizes the one-step Jacobi error when solving for a $\boldsymbol{0}$ right-hand side.

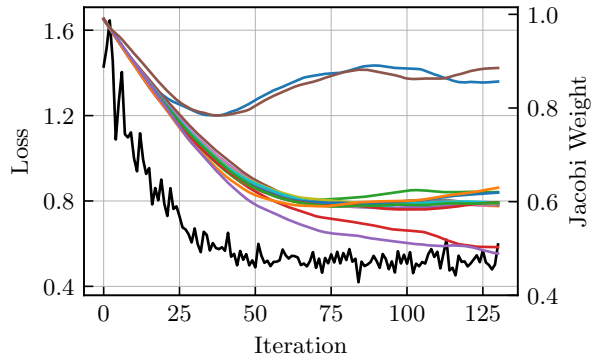

Figure 3. Loss history (in black) obtained from optimizing over the Jacobi relaxation weights. The colored lines are the history of the node-wise weights for each training iteration.

Training loss history and final values of the weights are shown in Figures 3 and 4. In Figure 4, we observe that the nodal weights at the two ends of the domain are maximized; for our problem setup this mimics the behavior of values being propagated from the boundaries inwards.

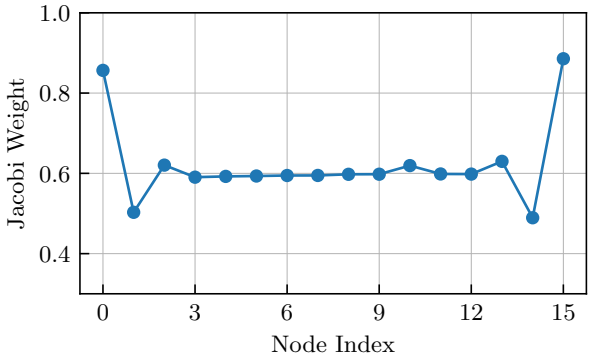

Figure 4. Final node-wise weight values for the Jacobi relaxation.

4.2. Heavyball Iteration

Polyak's heavyball iteration (Polyak, 1964) is a two-step iterative method that uses the information from the last two iterates to generate the next approximation. It can be viewed as a gradient descent with a momentum term.

To minimize a differentiable function f, we consider the update step

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \alpha\nabla_{\boldsymbol{x}} f\left(\boldsymbol{x}^{(k)}\right) + \beta\left(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}\right), \tag{22}$$

for scalars $\alpha, \beta \in \mathbb{R}$. We then apply Equation (22) to solve systems of the form Equation (18) by defining $\boldsymbol{A}$ to be

$$f\left(\boldsymbol{x}; \boldsymbol{A}, \boldsymbol{b}\right) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x}. \tag{23}$$

Next, taking the gradient of f with respect to $\boldsymbol{x}$ yields

$$\nabla_{\boldsymbol{x}} f\left(\boldsymbol{x}; \boldsymbol{A}, \boldsymbol{b}\right) = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}, \tag{24}$$

where we have that $\nabla_{\boldsymbol{x}} f = \boldsymbol{0}$ implies a solution to the matrix system if $\boldsymbol{A}$ is SPD.

Similarly, we consider the example problem in Section 4.1 by generating a 16×16 matrix $\boldsymbol{A}$ and minimizing

$$\ell = \sum_{k=1}^{K} \left(\boldsymbol{h}_{12}\left(\boldsymbol{x}_k, \boldsymbol{0};\ \alpha, \beta\right)\right)^T \boldsymbol{A}\boldsymbol{h}_{12}\left(\boldsymbol{x}_k, \boldsymbol{0};\ \alpha, \beta\right), \tag{25}$$

where $\boldsymbol{h}_L\left(\boldsymbol{x}_k, \boldsymbol{b};\ \alpha, \beta\right)$ is the application of L rounds of heavyball iteration to $\boldsymbol{x}_k$ with right-hand-side $\boldsymbol{b}$ and parameters α, β. Again, the $\{\boldsymbol{x}_k\}$ is a set of K vectors in $\mathbb{R}^N$ with unit-normally distributed entries.

We optimize the error after $\frac{3}{4}N$ iterations of the heavyball method. The training and testing loss history is shown in Figure 5, and the parameter history as a function of training epoch is shown in Figure 6.

4.3. Conjugate Gradient

The conjugate gradient (CG) method is another iterative solver for matrix equations of the form in Equation (18).

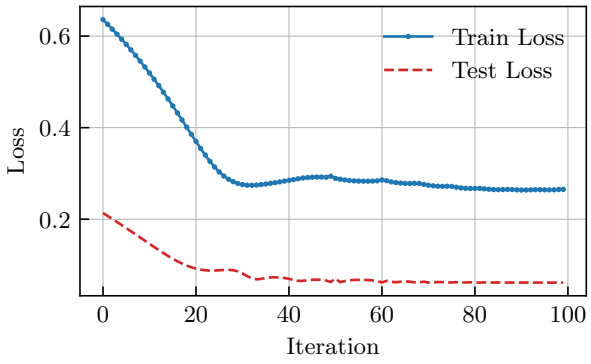

Figure 5. Loss history obtained from optimizing over the heavyball parameters.

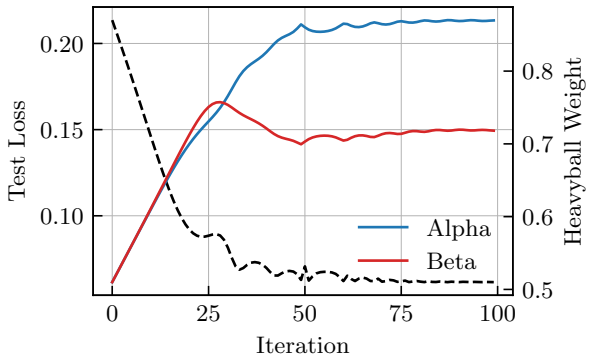

Figure 6. Parameter history for each heavyball training iteration, plotted along with the training loss.

While it typically exhibits faster convergence than both the Jacobi and heavyball methods, convergence can be accelerated further by passing an approximate inverse to $\boldsymbol{A}$, denoted by $\boldsymbol{M} \approx \boldsymbol{A}^{-1}$, with the assumption that $\boldsymbol{M}$ is itself SPD, using the preconditioned conjugate gradient (PCG) method (Saad, 2003).

We extend the one-dimensional problem in Equation (15) to two dimensions via tensor product over the x and y dimensions, as in

$$\boldsymbol{A} = \boldsymbol{A}_{N_x \times N_y} = \left(\boldsymbol{A}_{N_x} \otimes \boldsymbol{I}_{N_y}\right) + \left(\boldsymbol{I}_{N_x} \otimes \boldsymbol{A}_{N_y}\right), \quad (26)$$

where $\otimes$ denotes the standard matrix Kronecker product, and $\boldsymbol{A}_N$ indicates a one-dimensional discretization with N interior grid points. For our test problem we use $N_x = N_y = 8$, resulting in a matrix shape $\boldsymbol{A} \in \mathbb{R}^{64\times 64}$.

The goal is the construction of $\boldsymbol{M}$ that suitably approximates the inverse to $\boldsymbol{A}$. Since, $\boldsymbol{M}$ is an SPD matrix, it has a decomposition,

$$\boldsymbol{M} = \boldsymbol{L}\boldsymbol{L}^T. \quad (27)$$

From this, we optimize over the entries of $\boldsymbol{L}$ to force the preconditioner $\boldsymbol{M}$ to be symmetric, while $\boldsymbol{M}$ is not directly constrained to be positive-definite.

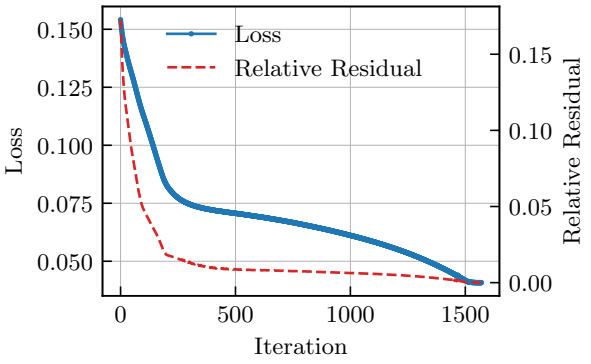

Figure 7. Loss history in constructing M, the CG preconditioner. The blue line (left axis) is the loss and the red dashed line (right axis) is the relative residual.

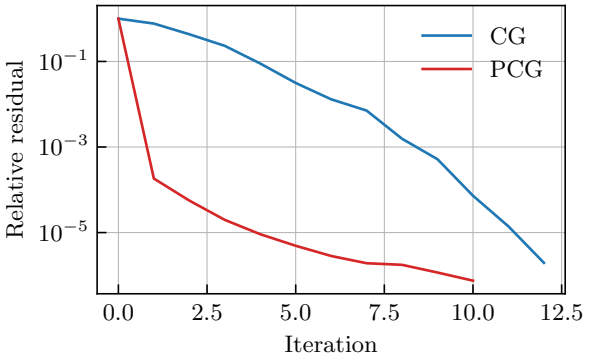

Figure 8. Relative residual history of a regular (non-preconditioned) CG and the preconditioner found by optimization.

To find the entries in $\boldsymbol{M}$, we introduce the weighted loss

$$\ell = \sum_{i=1}^{N_{\text{it}}} \left(\frac{\gamma^{N_{\text{it}}-i}}{\sum_{j=1}^{N_{\text{it}}} \gamma^{N_{\text{it}}-j}} \right) \frac{\left\|\boldsymbol{r}^{(i)}\right\|_2}{\|\boldsymbol{b}\|_2}, \quad (28)$$

where N_{it} is the number of PCG iterations run (we use $N_{\text{it}} = 4$), $\boldsymbol{r}^{(i)}$ is the ith residual of the iteration, $\boldsymbol{b}$ is the right-hand-side, and $\gamma \in (0, 1]$ is a scaling constant (we use $\gamma = 0.6$). With this form, we minimize the overall weighted sum of the residual history with later iterates being weighted more than earlier ones. Experimentally, we observe improved convergence of the optimization problem in comparison to minimizing only over the last iterate. This also avoids the problem of vanishing gradients as the number of iterations is increased, as gradient information is used from each intermediate iterate.

We run the optimization over a lower bidiagonal $\boldsymbol{L}$, which gives a tridiagonal $\boldsymbol{M} = \boldsymbol{L}\boldsymbol{L}^T$. The loss and final relative residual obtained during each training iteration are displayed in Figure 7, while the residual history with and without $\boldsymbol{M}$ is shown in Figure 8.

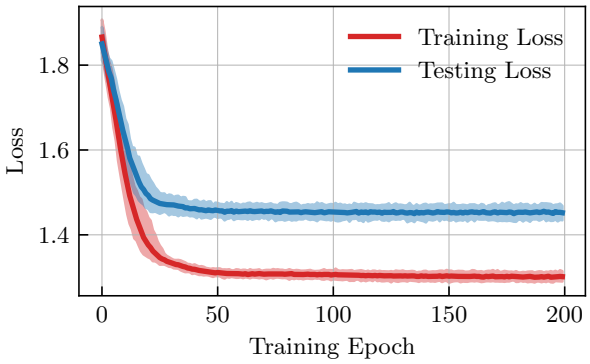

Figure 9. Loss history over training the GNN over the CiteSeer dataset. Averaged over 100 runs, lines denote mean while shaded regions are two standard deviations from mean.

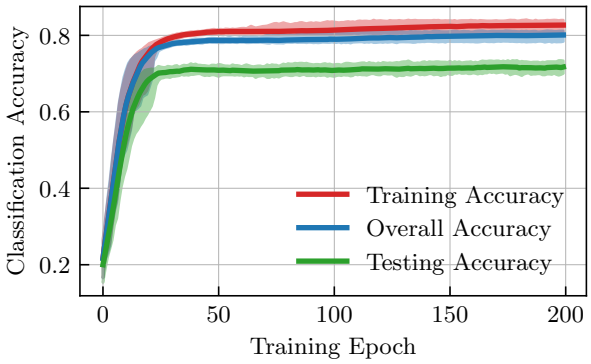

Figure 10. Classification accuracy of the networks per training epoch. Averaged over 100 runs, lines denote mean while shaded regions are two standard deviations from mean.

4.4. Graph Neural Networks

The GCN layer (Kipf & Welling, 2017) is a graph convolutional layer that *convolves* node features on a graph, $G(\boldsymbol{A})$. Denoting the (weighted) adjacency matrix of the graph by $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, and denoting $\boldsymbol{X}^{(i)} \in \mathbb{R}^{n\times C}$ as the C-dimensional node features at layer i, we represent the convolution of the node features as

$$\boldsymbol{X}^{(i+1)} = \tilde{\boldsymbol{D}}^{-\frac{1}{2}}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-\frac{1}{2}}\boldsymbol{X}^{(i)}\boldsymbol{\Theta}^{(i+1)}, \tag{29}$$

where $\boldsymbol{\Theta} \in \mathbb{R}^{C\times F}$ is the weight matrix for the layer, $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$, and $\tilde{\boldsymbol{D}}$ is the diagonal matrix extracted from $\tilde{\boldsymbol{A}}$.

We reimplement the GCN layer using our optimized underlying sparse matrix operations. Depending on the order of operations, computing the output in Equation (29) can be seen as a series of sparse-dense matrix multiplies (multiplying from right-to-left), or as two sparse-sparse multiplies followed by dense multiplies. In the following, we employ the former method.

Using this GCN layer, we reimplement the semi-supervised CiteSeer example from (Kipf & Welling, 2017). This dataset has 3327 nodes, 4732 edges, and 6 classes. We train a two-layer GCN with ReLU and sigmoid activations after the first and second layers and use a cross-entropy loss to optimize the predicted labels on each node, which follows the same setup as in (Kipf & Welling, 2017). Likewise, between each GCN layer is a dropout layer with $p = 0.5$, and we train with an Adam optimizer using a learning rate of 0.01, L2 regularization of 5×10^{-4}, and 16-dimensional representations for the hidden node features. This is trained for 200 epochs over 100 random initializations and training history can be seen in Figure 9.

As seen in Figure 10, we are able to match the roughly 70% classification accuracy on the CiteSeer dataset.

5. Timing Results

We present timing results for our implementation of the sparse kernels outlined in Section 3 along with a comparison of CPU and GPU (CUDA) timings in Table 2. For comparison, we have also included the respective *dense* operation in applicable cases, such as the dense matrix-vector (DMV) or dense-dense matrix-matrix product (DDMM). These dense implementations use the PyTorch built-in matrix routines. We also compare with NumPy/SciPy CPU and CuPy CUDA optimized versions for the forward passes.

Figure 11 highlights scaling results for the sparse matrix-vector product, sparse-sparse matrix product, and sparse-dense matrix product. These show the running time as a factor of the number of nonzeros in the matrix given in Equation (15), and indicate that forward and backward passes for the SpMV and SpSpMM are both linear in the nonzeros, while the forward and backward passes for the SpDMM are quadratic in the number of nonzeros. These timing results are run on a single compute node of the Narval cluster with two AMD Milan 7413 processors and four NVIDIA A100 GPUs, though for our tests we use only one CPU and one GPU. Our CPU-based implementations are single threaded only, so for fair comparison we limited the amount of threads in the dense operations to one.

6. Conclusions

In this work, we have described the implementation of a framework for automatically computing the gradient through computations with sparse matrix operations. We have stated the backwards propagation update rules and described at a high level how one may efficiently implement them on CPUs and GPUs. We demonstrated a range of applications that can be optimized using these sparse matrix primitives. Finally, we show timing and scaling results to show that our implementations are both scalable with re-

Table 2. Timing results for CPU- and CUDA-based implementations. *Iterations* refers to the number of times the specific test was run (times listed are total wall-clock time elapsed); N refers to the problem size used, most usually the size of the matrix in one dimension. *GPU Speedup* is computed as CPU time / GPU time. Methods with *Sp* refer to *Sp*arse operations (ours), while *D* refers to *D*ense (PyTorch).

Test Name	**ours** CPU (s)	SciPy CPU (s)	**ours** GPU (s)	CuPy GPU (s)	GPU Speedup	Iter.	N
SpMV Forward	0.141	0.092	0.045	0.049	3.150	1000	32,768
SpMV Backward	0.595		0.198		3.003	1000	32,768
DMV Forward	143.441	143.614	2.985	2.983	48.058	1000	32,768
DMV Backward	1509.088		21.806		69.206	1000	32,768
SpSpMM Forward	0.184	0.065	0.048	0.032	3.798	100	16,384
SpSpMM Backward	2.150		0.293		7.336	100	16,384
SpDMM Forward	145.156	100.840	1.100	0.569	132.014	100	16,384
SpDMM Backward	303.903		4.923		61.736	100	16,384
DDMM Forward	190.113	202.030	0.985	0.921	193.097	2	16,384
DDMM Backward	597.621		2.846		210.020	2	16,384
Sp + Sp Forward	0.073	0.034	0.011	0.011	6.813	100	32,768
Sp + Sp Backward	0.133		0.025		5.227	100	32,768
D + D Forward	60.978	61.553	0.941	0.947	64.783	100	32,768
D + D Backward	123.519		2.596		47.580	100	32,768
SpTRSV Forward	0.030	32.746	4.710	0.767	0.006	100	32,768
SpTRSV Backward	13.462		10.997		1.224	100	32,768
DTRSV Forward	35.092	65.834	0.695	4.119	50.523	100	32,768
DTRSV Backward	863.779		3.499		246.855	100	32,768

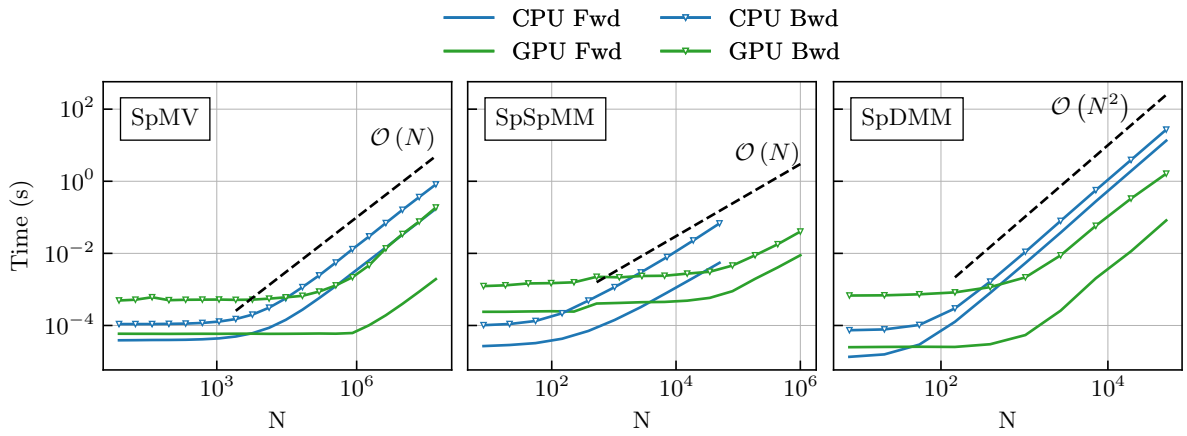

Figure 11. Scaling results for the forward and backward passes on several key sparse operations. The CPU running times are in blue, while the CUDA runtimes are in green. Triangle markers denote the backwards pass routines.

Table 3. Software versions used for testing.

Library	Backend	Version
FlexiBLAS	BLIS	0.8.1
Python	CPython	3.10.2
SciPy		1.8.0
NumPy		1.22.2
CUDA		11.7
CuPy		11.2.0
PyTorch		1.13.0

spect to dense linear algebra routines, and competitive in the forward pass with other sparse linear algebra packages.

There are a numerous directions of future work that we envision from this paper. Perhaps the most straightforward is the implementation of more complex sparse linear algebra operations such as eigensolver routines, linear least squares, etc. In an extension to the solve routine we have presented, one can consider backpropagation through the $\boldsymbol{L}$ and $\boldsymbol{U}$ factors themselves, perhaps in optimization problems where such calculations are necessary. Our CUDA LU factorization itself could also be optimized, there are recent works such as (Gaihre et al., 2022) that claim the symbolic factorization can be implemented to run in a scalable, parallel fashion on the GPU.

Acknowledgments

This research was enabled in part by support provided by ACENET (www.ace-net.ca), and the Digital Research Alliance of Canada (alliancecan.ca). The work of SM was partially supported by an NSERC Discovery Grant.

A. Software Versions

Our sparse kernels are compiled as a PyTorch extension written in C++17 using GCC, with the `-O2` flag enabled as the only non-default option. The list of library versions we use and compare against are detailed in Table 3.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

Baydin, A. G., Pearlmutter, B. A., and Radul, A. A. Automatic differentiation in machine learning: a survey. *CoRR*, abs/1502.05767, 2015. URL http://arxiv.org/abs/1502.05767.

Bonfatti, F., Monaco, V., and Tiberio, P. Microwave circuit analysis by sparse matrix techniques. In *1973 IEEE G-MTT International Microwave Symposium*, pp. 41–43, 1973. doi: 10.1109/GMTT.1973.1123084.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Briggs, W. L., Henson, V. E., and McCormick, S. F. *A Multigrid Tutorial, Second Edition*. Society for Industrial and Applied Mathematics, second edition, 2000. doi: 10.1137/1.9780898719505. URL https://epubs.siam.org/doi/abs/10.1137/1.9780898719505.

Dalton, S., Olson, L., and Bell, N. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Trans. Math. Softw.*, 41(4), oct 2015. ISSN 0098-3500. doi: 10.1145/2699470. URL https://doi.org/10.1145/2699470.

Gaihre, A., Li, X. S., and Liu, H. Gsofa: Scalable sparse symbolic lu factorization on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 33:1015–1026, 4 2022. ISSN 15582183. doi: 10.1109/TPDS.2021.3090316.

George, A., Gilbert, J. R., and Liu, J. W. H. (eds.). *Graph Theory and Sparse Matrix Computation*. Springer New York, 1993. doi: 10.1007/978-1-4613-8369-7. URL https://doi.org/10.1007/978-1-4613-8369-7.

Giles, M. B. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. *Lecture Notes in Computational Science and Engineering*, 64 LNCSE:35–44, 2008. ISSN 14397358. doi: 10.1007/978-3-540-68942-3_4/COVER. URL https://link.springer.com/chapter/10.1007/978-3-540-68942-3_4.

Greenfeld, D., Galun, M., Kimmel, R., Yavneh, I., and Basri, R. Learning to optimize multigrid pde solvers, 2019.

Johnson, J. Derivatives, backpropagation, and vectorization. http://cs231n.stanford.edu/handouts/derivatives.pdf, September 2017.

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

2017. URL https://openreview.net/forum?id=SJU4ayYgl.

Kolda, T. G. and Bader, B. W. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009. doi: 10.1137/07070111X. URL https://doi.org/10.1137/07070111X.

Luz, I., Galun, M., Maron, H., Basri, R., and Yavneh, I. Learning algebraic multigrid using graph neural networks, 2020.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019. URL http://arxiv.org/abs/1912.01703.

Polyak, B. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964. ISSN 0041-5553. doi: https://doi.org/10.1016/0041-5553(64)90137-5. URL https://www.sciencedirect.com/science/article/pii/0041555364901375.

Quarteroni, A., Sacco, R., and Saleri, F. *Numerical Mathematics*. Texts in Applied Mathematics. Springer, Berlin, Germany, 2 edition, October 2006.

Saad, Y. *Iterative Methods for Sparse Linear Systems*. Other Titles in Applied Mathematics. SIAM, second edition, 2003. ISBN 978-0-89871-534-7. doi: 10.1137/1.9780898718003. URL http://www-users.cs.umn.edu/~{}saad/IterMethBook_2ndEd.pdf.

Su, J., Zhang, F., Liu, W., He, B., Wu, R., Du, X., and Wang, R. Capellinisptrsv: A thread-level synchronization-free sparse triangular solve on gpus. pp. 1–11. ACM, 8 2020. ISBN 9781450388160. doi: 10.1145/3404397.3404400. URL https://dl.acm.org/doi/10.1145/3404397.3404400.

Taghibakhshi, A., Nytko, N., Zaman, T., MacLachlan, S., Olson, L., and West, M. Learning interface conditions in domain decomposition solvers. 5 2022. URL http://arxiv.org/abs/2205.09833.

Tao, Y., Deng, Y., Mu, S., Zhu, M., Xiao, L., Ruan, L., and Huang, Z. Atomic reduction based sparse matrix-transpose vector multiplication on gpus. *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, 2015-April:987–992, 2014. ISSN 15219097. doi: 10.1109/PADSW.2014.7097920.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Yu, P. S. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32:4–24, 1 2021. ISSN 2162-237X. doi: 10.1109/TNNLS.2020.2978386. URL https://ieeexplore.ieee.org/document/9046288/.