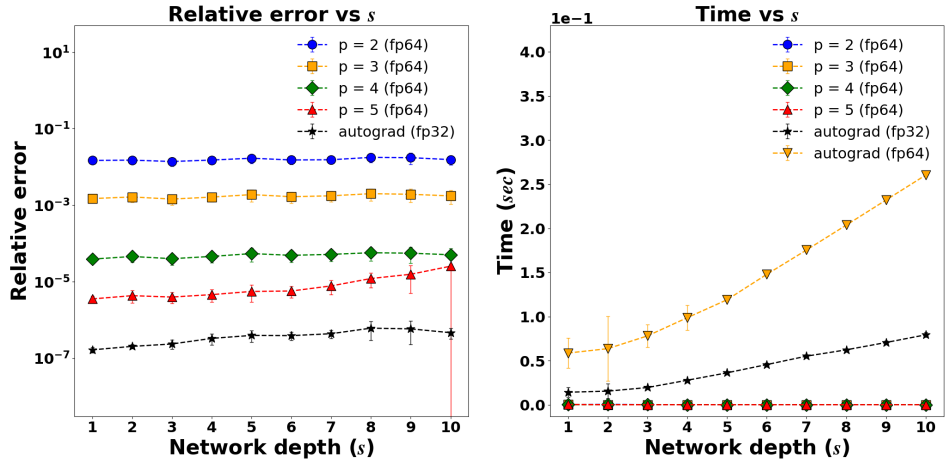
A Appendix

A.1 Additional results

In this section, we present additional results that help clarify details of our method.

A.1.1 Higher order derivatives



(a) (b)

Figure 1: Autograd properties as a function of network depth s. The figure shows (a) effect of neural network depth s on the relative error (with respect to fp64 autograd) and (b) time taken for one application of autograd on fp32 and fp64, compared to the time taken for SpMV using RBF-FD for computing $\Delta^2 u(x)$ in $d = 2$. Error bars over 15 random runs are shown.

We also study the effect of PINN depth on computing the biharmonic operator Δ^2 of the output with respect to the spatial variable x using either autograd or RBF-FD. We compute errors against fp64 autograd for fp32 autograd and for RBF-FD with $p = 2, 3, 4$, and 5. All errors were computed on $N = 4977$ quasi-uniform collocation points. The results are in shown in Figure 1a. We see that fp32 autograd is the most accurate, and similar to the Laplacian Δ, increasing p increases the accuracy of RBF-FD by about two orders of magnitude. In Figure 1b, we report the time taken for the same test. It is clear that RBF-FD offers even greater speedups for approximating high-order differential operators.

A.1.2 fp32 DT-PINN

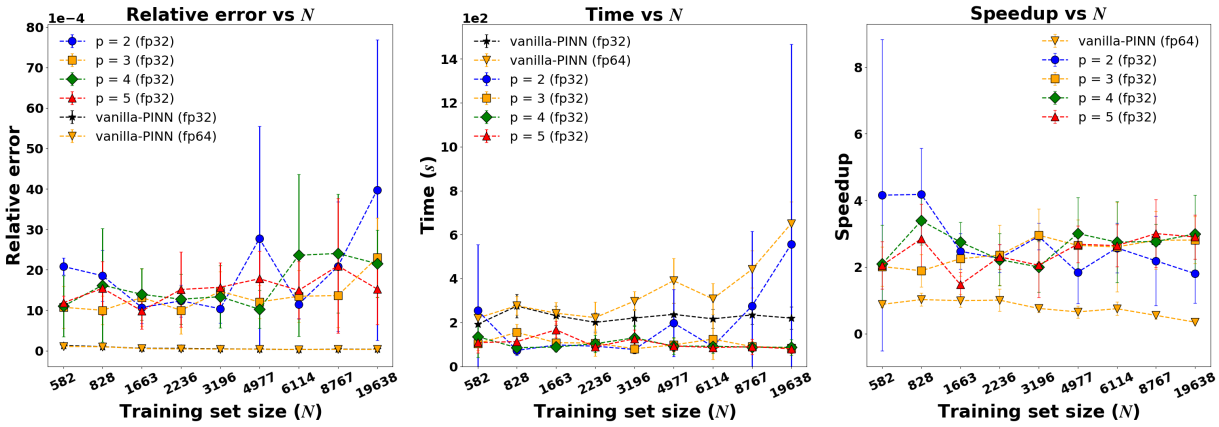
Figure 2 shows results for fp32 DT-PINNs on the linear Poisson equation. While speedups over vanilla-PINNs are similar to fp64 DT-PINNs, they suffer from a degradation in accuracy for all values of p compared to both fp64 DT-PINNs and vanilla-PINNs. These results demonstrate that the combination of fp64 and discrete training is key to achieving training speedups without a loss in accuracy, but that fp32 DT-PINNs are also viable alternatives to vanilla-PINNs.

A.1.3 Star shaped domain

In Figure 3a, we show $N = 2180$ interior and boundary collocation points on a star-shaped domain from (3). Figure 3b shows the manufactured solution as specified in (19). In Figure 4 we present results for fp64 DT-PINNs and fp32 vanilla-PINN on the linear Poisson equation with a star shaped domain. DT-PINNs with $p > 2$ achieve the same relative errors as vanilla-PINN. DT-PINNs also obtain a maximum of 2x speedup over vanilla-PINN ignoring $p = 2$. Similar to the results in unit disk domain, DT-PINNs are faster than vanilla-PINNs while getting competitive or even better accuracies.

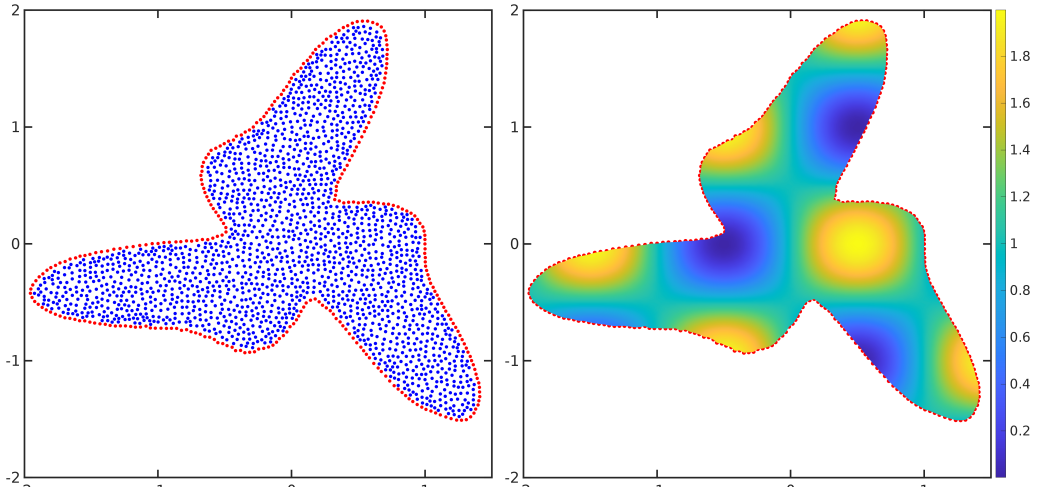
A.2 Implementation

From a practical implementation standpoint, DT-PINNs differ from vanilla-PINNs in several ways beyond the replacement of autograd with differentiation matrices. We discuss some of the details of our implementation within the PyTorch framework (2).



(a) (b) (c)

Figure 2: fp32 DT-PINNs on the linear Poisson equation (3) for different numbers of collocation points (N) and orders of accuracy (p). We show (a) the relative error in the PINN solution; (b) the time taken to converge to lowest relative error; and (c) the speedup attained by fp32 DT-PINNs relative to fp32 vanilla-PINN for those times.

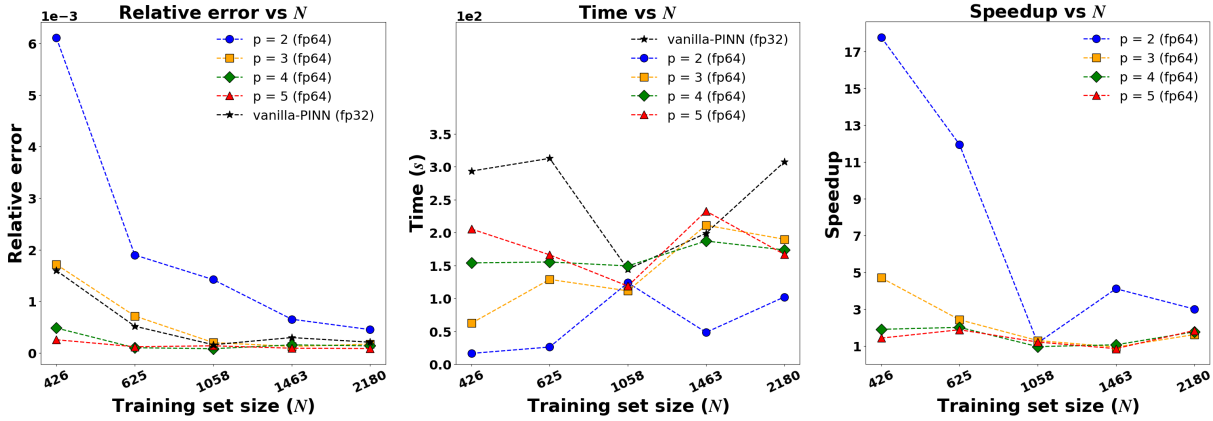


(a) (b)

Figure 3: Quasi-uniform collocation points and the true solution on a star-shaped domain. The figure shows (a) $N = 2180$ interior and boundary collocation points and (b) the manufactured solution given by (19) and used in Section 4.2.

Efficient sparse matrix storage and operations on the GPU Since DT-PINNs replace all autograd computations in the loss function with an SpMV, we use the compressed sparse row (CSR) format for storing L and B; this ensures both ease of parallelization and compact storage. As of this writing, PyTorch support for SpMV operations in the CSR format on the GPU is limited and somewhat inefficient. To overcome this limitation, we used the CuPy library (1) for all SpMV operations. We also use the DLPack library to enable memory and context sharing between CuPy and PyTorch.

Custom autograd implementation As of this writing, PyTorch is in general unable to apply autograd with respect to the weight vector $\mathbf{w}$ to loss terms involving CuPy CSR matrices. To overcome this issue, we wrote custom autograd classes in PyTorch. Consider the second term in (12), $\|L\tilde{\boldsymbol{u}} - \mathbf{f}\|_2^2$. According to the PyTorch API, a custom autograd class for handling this term must supply two methods: a **forward** method that tells PyTorch how to evaluate it, and a **backward** method that tells



(a) (b) (c)

Figure 4: fp64 DT-PINNs on the linear Poisson equation (3) for different numbers of collocation points (N) and orders of accuracy (p) with a star shaped domain. We show (a) the relative error in the PINN solution; (b) the time taken to converge to lowest relative error; and (c) the speedup attained by fp32 DT-PINNs relative to fp32 vanilla-PINN for those times. Results shown over a single random seed.

PyTorch how to compute the product $(\nabla_{\tilde{\boldsymbol{u}}}(L\tilde{\boldsymbol{u}}-\mathbf{f}))(\nabla_{\mathbf{w}}\tilde{\boldsymbol{u}})$. As $\nabla_{\tilde{\boldsymbol{u}}}(L\tilde{\boldsymbol{u}}-\mathbf{f})=L^T$, our custom PyTorch **backward** method can be expressed as a CuPy-based SpMV of L^T and $\nabla_{\mathbf{w}}\tilde{\boldsymbol{u}}$, the latter of which PyTorch automatically supplies; loss terms involving B are handled similarly. The **forward** method to evaluate the loss term is also an SpMV between L and $\tilde{\boldsymbol{u}}$. The overall procedure is similar for the heat equation as well.

Code listings We first show the CuPy custom autograd methods for the SpMV operations in the autograd term. The required imports are:

```
import torch
from torch.utils.dlpack import to_dlpack, from_dlpack
import cupy
from cupy.sparse import csr_matrix
```

The following Python class defines the **forward** and **backward** methods to connect the CuPy SpMV with the native PyTorch tensors.

```
class Cupy_L(torch.autograd.Function):
    @staticmethod
    def forward(ctx, pinn_pred, sparse_mat):
        return from_dlpack(sparse_mat.dot(
               cupy.from_dlpack(to_dlpack(pinn_pred))
                               ).toDlpack())

    @staticmethod
    def backward(ctx, grad_output):
        return from_dlpack(L_t.dot(
               cupy.from_dlpack(to_dlpack(grad_output))
                               ).toDlpack()), None

# class Cupy_B can be similarly defined
```

where `L_t` is the transpose of the L matrix:

```
L_t = csr_matrix(self.L.transpose(), dtype=np.float64)
```

Each DT-PINN training step in linear Poisson can then be written as follows:

```
# we use the L-BFGS optimizer provided by PyTorch
def closure():
    self.optimizer.zero_grad()
    u_tilde = self.pinn_weights.forward(self.X_tilde)

    pde_residual = L_mul(u_tilde, self.L) - self.f
    bdry_residual = B_mul(u_tilde, self.B) - self.g

    pde_loss = torch.mean(torch.square(torch.flatten(pde_residual)))
    bdry_loss = torch.mean(torch.square(torch.flatten(bdry_residual)))

    train_loss = interior_loss + boundary_loss
    train_loss.backward(retain_graph=True)
    return train_loss.item()

loss_value = self.optimizer.step(closure)
```

where `self.X_tilde` is $\tilde{X}$ and `L_mul` and `B_mul` are:

```
L_mul = Cupy_L.apply
B_mul = Cupy_B.apply
```

References

[1] Okuta, R., Unno, Y., Nishino, D., Hido, S., and Loomis, C. (2017). Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*.

[2] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

[3] Shankar, V., Kirby, R., and Fogelson, A. (2018). Robust node generation for mesh-free discretizations on irregular domains and surfaces. *SIAM Journal on Scientific Computing*, 40(4):A2584–A2608.