Towards Large-Scale Learned Solvers for Parametric PDEs with Model-Parallel Fourier Neural Operators

1st Thomas J. Grady II
Georgia Institute of Technology
Atlanta, GA, USA
tgrady@gatech.edu

2nd Rishi Khan
Extreme Scale Solutions
Wilmington, DE, USA
rishi@extreme-scale.com

3rd Mathias Louboutin
Georgia Institute of Technology
Atlanta, GA, USA
mlouboutin3@gatech.edu

4th Ziyi Yin
Georgia Institute of Technology
Atlanta, GA, USA
ziyi.yin@gatech.edu

5th Philipp A. Witte
Microsoft
Redmond, WA, USA
pwitte@microsoft.com

6th Ranveer Chandra
Microsoft
Redmond, WA, USA
ranveer@microsoft.com

7th Russell J. Hewett
Virginia Tech
Blacksburg, VA, USA
rhewett@vt.edu

8th Felix J. Herrmann
Georgia Institute of Technology
Atlanta, GA, USA
felix.herrmann@gatech.edu

***Abstract*—Fourier neural operators (FNOs) are a recently introduced neural network architecture for learning solution operators of partial differential equations (PDEs), which have been shown to perform significantly better than comparable approaches based on convolutional networks. Once trained, FNOs can achieve speed-ups of multiple orders of magnitude over conventional numerical PDE solvers. However, due to the high dimensionality of their input data and network weights, FNOs have so far only been applied to two-dimensional or small three-dimensional problems. To remove this limited problem-size barrier, we propose a model-parallel version of FNOs based on domain-decomposition of both the input data and network weights. We demonstrate that our model-parallel FNO is able to predict time-varying PDE solutions of over 3.2 billion variables on Summit using up to 768 GPUs and show an example of training a distributed FNO on the Azure cloud for simulating multiphase CO$_2$ dynamics in the Earth's subsurface.**

***Index Terms*—Operator Learning, Deep Learning, Model Parallelism, Multiphase Flow**

1. Introduction

1.1. Background - Fourier Neural Operators

Neural operators (NOs) [1] are a recently introduced neural network architecture which learn mappings between infinite-dimensional function spaces, in contrast to traditional neural networks wherein mappings are learned between large (but ultimately finite) dimensional vector spaces. To do this, neural operators often employ architectural tricks to ensure that their output does not significantly vary with changes to the discretization of the corresponding input. In the context of learning solution operators to families of elliptic PDEs, NOs attempt to learn the mapping

$$\mathcal{G} : \mathcal{A} \to \mathcal{U} \tag{1}$$

where $\mathcal{A}$ and $\mathcal{U}$ are function spaces which represent the initial/boundary conditions and parameters of the PDE and its (time-varying) solution respectively [1] [2]. To learn this mapping, NOs employ an iterative architecture, constructing a sequence of functions $\nu_1, \ldots, \nu_K$ in a lifted space and then projecting the last value of this sequence down to an output u, with the training objective that u matches the solution of the PDE in a given norm (e.g. L^2, Sobolev). The iterative update between elements of this sequence is given by

$$\nu_{k+1}(x) = \sigma\left(W\nu_k(x) + \left(\mathcal{K}(a;\phi)\nu_k\right)(x)\right) \tag{2}$$

where σ is a nonlinear pointwise function, W a learned linear transformation, and $\mathcal{K}(a;\phi)$ a kernel integral operator with learned parameterization ϕ. Fourier neural operators (FNOs) choose this kernel operator to be

$$\left(\mathcal{K}(\phi)\nu_k\right)(x) = \mathcal{F}^{-1}\left(R_\phi \cdot (\mathcal{F}\nu_k)\right)(x), \tag{3}$$

where $\mathcal{F}$ is the Fourier transform, and R_ϕ is a restriction operator, which contains a low-pass filter and learned pointwise weight multiplication parameterized by ϕ [2]. This operator is referred to as a *spectral convolution*. The cutoff of the low pass filter in R_ϕ will depend on a user-defined parameter describing how many Fourier-modes to keep in each dimension.

When learning solutions to time-dependent PDEs, FNOs are most often trained on discretized pairs of input data (x, y), where x is a multidimensional tensor containing a discretization of the initial state $a(x) \in \mathcal{A}$ and y is a discretization of the time-evolving solution to the PDE, $u(x,t) \in \mathcal{U}$.

1.2. Motivation

Traditional approaches to numerical simulations based on finite differences, volumes, or elements are designed to be highly accurate, meaning that errors of numerical approximations are quantifiable and numerical solutions are consistent with the original (continuous) problem formulation [3], [4]. In addition, traditional numerical methods are also generic, which means that a discretized PDE can be solved for any set of initial/boundary conditions and input parameters, as long as the stability criteria of the respective discretization are met. Due to these stability, consistency and convergence requirements, numerical solvers for PDEs have strict sets of conditions of how problems are discretized (both in space and time), which results in very large-scale sets of linear and non-linear equations for many problem types [5]. As such, numerical simulators used in many real-world applications such as weather and climate forecasting, exploration seismology, or aerodynamical shape optimization are among some of the largest current high-performance computing workloads [6]–[9].

AI-driven approaches to numerical simulations based on deep learning offer an opportunity to frontload the high computational cost of numerical simulations to the network training time, while offering speedups of multiple orders of magnitude over conventional numerical solvers during inference (i.e. after a network has been trained) [10]–[12] . While deep learning based PDE solvers have, for the time being, no quantifiable error estimates and typically generalize to a much narrower set of simulation scenarios (i.e., in terms of models or initial conditions), they provide fast surrogate models that can be evaluated in fractions of a second. As such, they pave the way for applications that require a large number of individual simulations such as uncertainty quantification and inverse problems, which are often prohibitively expensive using conventional simulators. FNOs in particular have been shown to perform exceedingly well in comparison to alternative approaches to learning PDE solvers based on convolutional neural networks, as shown in various recent adoptions of FNOs for problems ranging from CO_2 flow simulations to weather forecasting [13]–[16].

In addition to very fast simulation times during inference, FNOs and other deep learning-based approaches offer the possibility to compute gradients/sensitivities of PDEs using automatic differentiation (AD), thus making it possible to solve inverse problems without requiring users to manually differentiate and implement adjoints and/or gradients. Here, we highlight this opportunity with a recent example from [17] on subsurface CO_2 flow and seismic imaging. The goal of the example is to estimate subsurface medium parameters such as permeability from seismic data, which is an example of a coupled multi-physics problem. Contrary to solving PDEs in the forward problem, the authors employ a learned FNO trained to predict the CO_2 concentration history in the subsurface from a given permeability field, which is then converted to a model of the acoustic wave speed. The changes wave speed induced by the expanding CO_2 plume can then be indirectly observed through seismic data. In the corresponding inverse problem, we are given seismic data at different points in time (i.e. during the expansion of the CO_2 plume), and are attempting to estimate the unknown permeability from this data (Figure 1). This task is not (easily) possible if conventional simulators such as Open Porous Media (OPM) [18] or GEOSX [19] are used for the CO_2 flow simulation, as neither framework offers sensitivities of the simulated CO_2 concentration with respect to the permeability. However, for an FNO implemented in deep learning frameworks like PyTorch [20] or Tensorflow [21], these sensitivities are readily available through AD, thus making it possible to implement a coupled inversion framework that enables us to directly invert for permeability from seismic data (Figure 1).

One of the main remaining challenges of adopting AI-driven solvers for real-world use cases is to scale FNOs and related networks to relevant problem sizes beyond 2D and small-scale 3D scenarios. Current applications of FNOs in the literature are based on data parallelism and as such, are limited to problem sizes that are supported by the amount of available memory on a single GPU [15], [22]. Leveraging distributed GPU memory for networks that do not fit onto a single GPU requires the use of model parallelism, in which networks are partitioned along the feature and/or channel dimension, rather than only along the data/batch dimension. Current research on model parallelism for neural networks has focused around exploiting parallelizable features of specific architectures such as multi-head attention in transformers [23], or parallelism between the generator and discriminator in generative adversarial networks [24]. However, solving PDEs with FNOs involves large amounts of input data, hidden state variables, as well as networks weights and gradients, and as such requires that the entire network (including the in- and output) is parallelized across a distributed-memory architecture. For this reason, we employ a parallelization approach to FNOs based on domain decomposition, in which we not only partition the network and state variables across multiple GPUs, but are also able to process each domain concurrently in a true model-parallel fashion. We base our FNO implementation on DistDL [25], a Python package that provides domain decomposition support for PyTorch by integrating communication primitives as linear operators into PyTorch's AD tool [26].

2. Parallel Implementation

2.1. Background - Abstraction of Parallelism

In order to successfully implement complex dimensionality-independent parallel algorithms within an automatic differentiation framework (such as PyTorch's `autograd` [27]), it is important to have a clear and expressive abstraction for parallel primitives and their adjoints [28]. To achieve this, we utilize the linear-algebraic formulation of communication primitives for domain-decomposed tensors in neural networks partitioned over a Cartesian worker topology as described in [26]. One can think of these abstractions as ways of describing the global action of a parallel operator across all workers distributed over an MPI Cartesian communicator. Here we will forgo the rederivation of basic memory operations such as copy, clear, send, and receive as seen in [26], and instead focus on

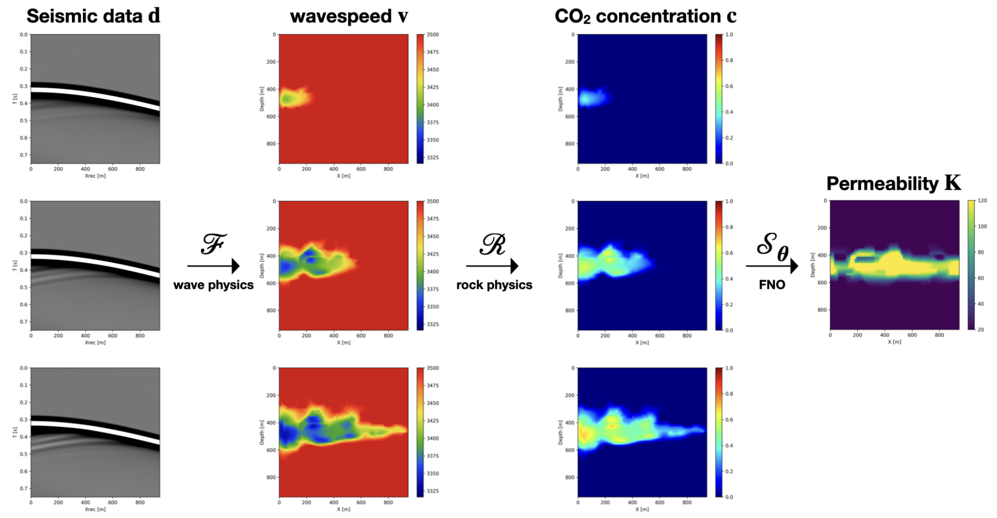

Figure 1: Coupled multi-physics inversion to estimate the subsurface permeability of a porous medium from seismic data measurements. To invert for the permeability, the authors in [17] first train a FNO that maps a permeability field to the CO_2 concentration history, which in turn is converted to the acoustic wave speed and used for simulating the seismic response. In the inverse problem, changes in the seismic data are first mapped to changes in the wave speed and the corresponding perturbations of the CO_2 concentration. Using the trained FNO, perturbations in the permeability can then be directly computed from changes in the CO_2 concentration via algorithmic differentiation and used to solve an inverse problem for estimating the permeability from seismic measurements.

the two communication primitives necessary to implement a distributed FNO.

The first of these primitives is *broadcast*, which copies subtensors of a tensor partitioned on one set of workers to another. As described in [26], the broadcast operation on tensors extends beyond the classical parallel broadcast primitive. While it can trivially represent the classical operation, i.e., copying data from one worker to many, it can also represent an extension of this operation to tensor partitions. As long as the input and output partitions satisfy the DistDL broadcast rules (a subset of the NumPy broadcast rules [29]) the action of this operator between two partitions P_x and P_y, $B_{\{P_x\}\to\{P_y\}}$, will copy an input tensor x along the appropriate dimensions [26]. Following the definition of the adjoint, a broadcast in the forward evaluation will induce a sum-reduction in the gradient calculation.

The second of these primitives is *repartition*, a high-dimensional generalization of *all-to-all*. The action of this operator, $T_{\{P\}\to\{Q\}}$, changes the distribution of the data from one Cartesian partition P to another Cartesian partition Q. While P and Q are not required to have the same number of workers, they are required to have the same number of dimensions as the input tensor. In higher dimensions, an implementation of this primitive is not trivial as any worker may need to send or receive subtensors to or from any or all other workers (e.g., a many-to-many operation). Repartition is in some sense the most “general” possible communication primitive for tensors and its adjoint is also a repartitioning, from Q to P.

Ultimately, this linear algebraic formulation of parallel primitives on domain-decomposed tensors allows for non-trivial parallel operations to be cleanly expressed within the mathematical formulation of FNOs, instead of as disjoint algorithm listings.

2.2. Pointwise Affine Transformations

One of the two core components of FNOs that must be adapted for a distributed setting are the affine transformations along particular dimensions

$$y = W : x + b, \tag{4}$$

where “:” represents Einstein summation [30] along said dimension. These transformations are used to lift an input function a to the initial function ν_0 in the iterative update sequence $\nu_0 \to \nu_1 \to \ldots, \to \nu_K$, as part of the spectral convolution blocks within this sequence, and to project the last function in the sequence ν_K back to the output of the network u. Because these transformations are pointwise with respect to the dimensions representing the infinite-dimensional function, we can use the aforementioned broadcast operator to copy the weight tensor W and bias vector b from a root worker on a size one partition P_r to all workers on the partition of interest P_x, where P_x has size 1 in the dimension along which the Einstein summation occurs. The distributed pointwise affine transformation is then written

$$y = (B_{\{P_r\}\to\{P_x\}}W) : x + B_{\{P_r\}\to\{P_x\}}b. \tag{5}$$

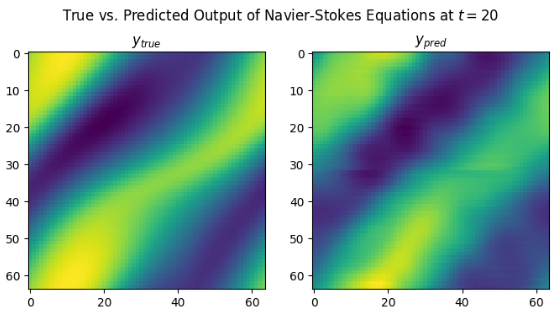

Figure 2: Artifacting at parallel worker boundaries and solution inaccuracies on the validation dataset caused by training a distributed FNO over a 2×2 partition without broadcasting the weight matrices and bias vectors before applying pointwise transformations.

The inclusion of a broadcast of the weight and bias terms ensures no artifacting occurs at worker boundaries as in figure 2.

2.3. Distributed Fourier Transform

Performing fast discrete Fourier Transforms (FFTs) on domain-decomposed tensors is a well-studied problem. Current state-of-the-art approaches [31] [32] use an iterative repartition pattern, wherein a chain of repartition operators and sequential FFTs are used to compute the entire FFT over distributed data without ever having to compute a parallel FFT along any dimension. See figure 3 for an illustration. Note that this works only because the Fourier transform is separable. From a linear algebra viewpoint, the distributed Fourier transform (DFFT) of an n-dimensional tensor x distributed over a partition P can then be written[1]

$$\mathcal{F}_{dist} x = \mathcal{F}_{\mathcal{I}_k} T_{\{P_{\mathcal{I}_{k-1}}\} \rightarrow \{P_{\mathcal{I}_k}\}} \dots \mathcal{F}_{\mathcal{I}_1} T_{\{P\} \rightarrow \{P_{\mathcal{I}_1}\}} x \quad (6)$$

where

$$\bigcup_{j=1}^{k} \mathcal{I}_j = \{1, 2, \dots, n\}, \quad \mathcal{I}_{j_1} \cap \mathcal{I}_{j_2} = \emptyset \; \forall j_1 \neq j_2. \quad (7)$$

Each $\mathcal{I}_j$ describes an index set of dimensions over which to apply the sequential multidimensional Fourier transform $\mathcal{F}_{\mathcal{I}_j}$, and denotes which dimensions of x are present in their entirety on each worker. Combining this formulation with PyTorch's native multidimensional FFT yields a clean and powerful implementation of a distributed, differentiable n-dimensional FFT, which can also be further generalized to other combinations of separable transforms. Choosing this selection of index sets is generally problem specific, but a good rule of thumb is to select them such that the number of repartition operators (and thus generalized all-to-all communications) is minimized while still being able to fit all subtensors within the memory constraints of their corresponding parallel workers at all steps of the DFFT. We can then take this DFFT and simply replace the standard FFT in the spectral convolution to get the distributed spectral convolution operator

$$\left(\mathcal{S}_{dist} \nu_k\right)(x) = \left(\mathcal{F}_{dist}^{\top} (R_\phi \cdot (\mathcal{F}_{dist} \nu_k))\right)(x). \quad (8)$$

Practical considerations require that only workers containing nonzero values in the Fourier domain after the application of the low pass filter perform any work. To achieve this, each worker uses information about the underlying partition of the data and corresponding index in that partition to compute whether it will contain a nonzero value after application of R_ϕ. If so, it applies the restriction and pointwise weight multiplication, otherwise the value is multiplied by zero. Note that all of this is happening on the output partition $P_{\mathcal{I}_k}$ of the forward DFFT $\mathcal{F}_{dist}$, as to remove the need for an unnecessary all-to-all communication to get the data back to the original partition before applying R_ϕ. See figure 4 for an illustration and appendix B for how this index calculation is performed.

2.4. Full Network

Combining the broadcast operator, repartition operator, and DFFT, we are now able to describe the distributed FNO (DFNO) in its entirety. FNOs traditionally consist of the following sequence of transformations. First, the input function is projected to the correct number of output timesteps and lifted to a higher-dimensional space via two affine transformations along the time and channel dimensions

$$a_1(x, t) = (W_t : a + b_t)(x, t) \quad (9)$$
$$\nu_0(x, t) = (W_c : a_1 + b_c)(x, t). \quad (10)$$

To distribute these two transformations, we apply repatition operators before each to ensure the dimensions on which they act are present entirely on each worker. These partitions are denoted P_t and P_x respectively. We then apply the broadcasted affine operator as in equation 5. Assuming that each weight tensor and bias vector is stored on the same size 1 root partition P_r, this is written as the sequence of transformations

$$W_t^{bc} = B_{\{P_r\} \rightarrow \{P_t\}} W_t \quad (11)$$
$$W_c^{bc} = B_{\{P_r\} \rightarrow \{P_c\}} W_c \quad (12)$$
$$b_t^{bc} = B_{\{P_r\} \rightarrow \{P_t\}} b_t \quad (13)$$
$$b_c^{bc} = B_{\{P_r\} \rightarrow \{P_c\}} b_c \quad (14)$$
$$a_1 = (W_t^{bc} : T_{\{P_x\} \rightarrow \{P_t\}} a + b_t^{bc})(x, t) \quad (15)$$
$$\nu_0 = (W_c^{bc} : T_{\{P_t\} \rightarrow \{P_c\}} a_1 + b_c^{bc})(x, t), \quad (16)$$

where W_t, b_t and W_c, b_c are the weights and biases for each affine transformation respectively. The FNO architecture then calls for a sequence of K blocks consisting of a linear transformation along the channel dimension added with a spectral convolution and passed through a pointwise

1. Optionally, there is an additional repartition operator $T_{\{P_{\mathcal{I}_k}\} \rightarrow \{Q\}}$ to change the partition of the data to some output partition Q, but due to the structure of FNOs, this step is not performed.

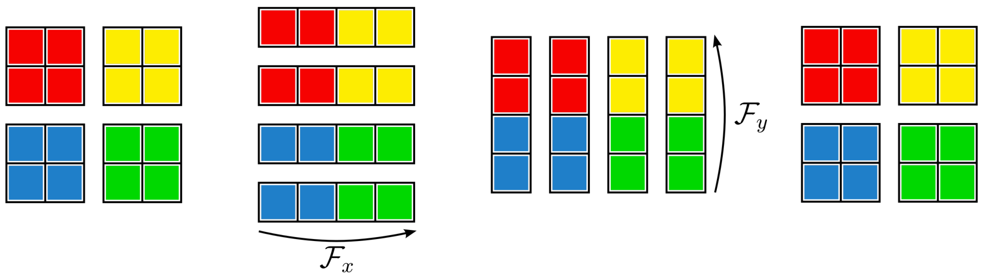

Figure 3: Distributed FFT using pencil decompositions acting on an input initially distributed over a 2×2 partition. Repartition operators are used to ensure each worker has the full data it needs to calculate the sequential FFT in each dimension.

nonlinearity. A single iterative update in this sequence can be written

$$\nu_{k+1}(x,t) = \sigma\left(W : \nu_k + \mathcal{S}\nu_k\right)(x,t), \tag{17}$$

where $k = 1, \ldots, K-1$ and σ is a pointwise nonlinear activation function. Applying the same procedure as in equation 5 for distributing pointwise linear transformations then gives the sequence of transformations for the DFNO block

$$W^{bc} = B_{\{P_r\}\to\{P_x\}}W \tag{18}$$

$$\nu_{k+1}(x,t) = \sigma\left(W^{bc} : \nu_k + \mathcal{S}_{dist}\nu_k\right)(x,t). \tag{19}$$

Finally, a linear operator is used to project the output to the correct number of channels. Using a similar derivation to equation 16, the distributed variant is

$$u(x,t) = \left(W^{bc} : T_{\{P_x\}\to\{P_c\}}\nu_K\right)(x,t) \tag{20}$$

where $u(x,t)$ is the approximate solution of the PDE given by the learned operator represented by the distributed FNO, and is distributed over the partition P_c.

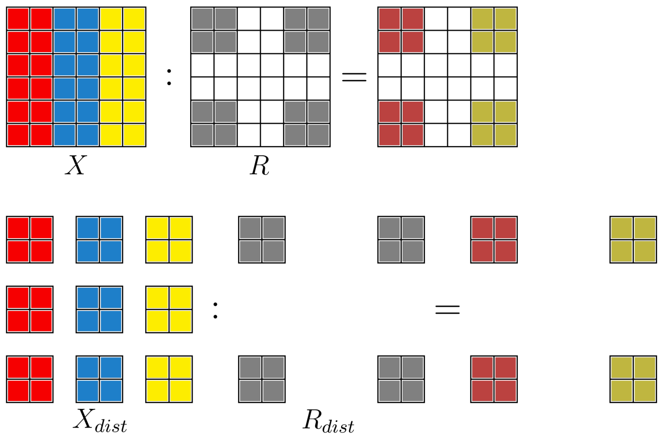

Figure 4: Sequential versus distributed application of R_ϕ on 2D data distributed over a 3×3 partition.

3. Experimental Results

3.1. Scaling

All of our scaling experiments were performed on the Summit system at Oak Ridge National Laboratory. Summit is a world-class supercomputer consisting of 4,608 nodes each with 2 IBM Power9 CPUs, 6 NVIDIA Volta V100 GPUs each with 16 GB HBM2 memory, and 512GB of DDR4 main memory. Each node is connected to a non-blocking fat-tree interconnect topology using a dual-rail Mellanox EDR InfiniBand interconnect [33]. Each node is running PyTorch v1.10, and DistDL v0.5.0.

We conducted a weak scaling study measuring the time taken to apply the network forward without saving gradients, forward with saving gradients, and perform backpropagation. For each of these scenarios, we also compared scaling the size of the spatial dimensions versus scaling the number of output timesteps. In the spatial scaling study, we scaled from a minimum $x \times y \times z$ problem size of $64 \times 64 \times 64$ to $768 \times 512 \times 512$. In the temporal scaling study, we scaled from a minimum number of output timesteps $nt = 16$ to a maximum number of output timesteps $nt = 6144$. The network was set up to use 4 spectral convolution blocks and have a lifted dimension of size 20, as originally presented by Li et al. [2]. The run configurations and timing results can be seen in appendix A, and a plot of the timing results in figure 5. We observe imperfect weak scaling, and an overall dominance by communication costs in the network due to the relative simplicity of computation in the network (i.e. matrix multiplications, pointwise multiplications, and FFTs) compared with the expensive communications (predominantly generalized all-to-all).

3.2. Distributed Training Example

The ability to scale FNOs with domain decomposition to large problem sizes opens up the possibility to apply them to real-world simulation use cases. To showcase the value of being able to train model-parallel networks for solving large-scale PDEs, we train a distributed FNO to simulate subsurface CO_2 flow by solving the 3D time-varying two-phase flow equations [18], [19]. Simulating CO_2 flow in

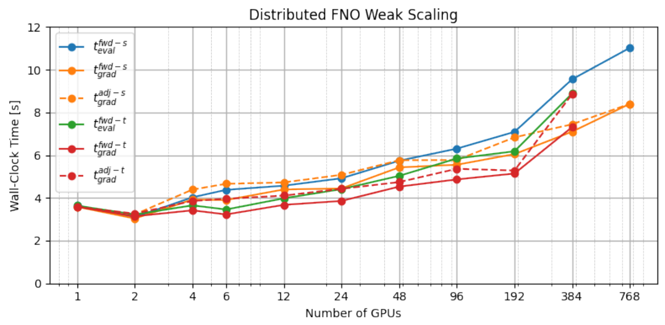

Figure 5: Weak scaling of distributed FNO up to 768 GPUs (128 nodes) on Summit. Any timing result with $eval$ indicates that the network was run with `torch.no_grad()` and `network.eval()`, while any timing result with $grad$ indicates that the network was run without `torch.no_grad()` and with `network.train()`. $-s$ indicates a spatial scaling study, while $-t$ indicates a temporal scaling study. fwd indicates a forward pass of the network `y = network(x)`, while adj indicates backpropagation, `y.backward()`.

porous media plays an important role in carbon capture and storage (CCS), where simulations are required to optimize the CO_2 injection location and verify that CO_2 does not leak from the storage site [34], [35]. A variety of deep-learning based approaches, including FNOs, have been proposed for simulating subsurface CO_2 flow, but so far current examples in the literature are limited to either two-dimensional or small to medium-scale three-dimensional problems [13], [22], [36]. On the other hand, our model-parallel FNO can scale to realistic 3D problem sizes using widely available Nvidia V100 GPUs with 16 GB memory.

We train an FNO to predict the evolution of a 3D subsurface CO_2 plume over a given number of time steps n_t. The input into the network is a permeability tensor $\mathcal{K}(x)$, where x is the spatial position in three dimensions as well as a tensor describing the grid topography (i.e. column-wise vertical displacement). In contrast to the original FNO from Li et al. [2], our input does not include the first 10 time steps of the predicted saturation history, which is zero almost everywhere. We train our model on the CO_2 simulation dataset from [37], which was derived from the Sleipner benchmark [38], a reservoir simulation model from the world's first industrial-scale CO_2 injection site off the coast of Norway [39], [40]. The training dataset consists of $1,000$ randomly generated permeability models that were generated in analogy to the original benchmark, as well as the corresponding simulated saturation histories for each sample. Each individual input sample to the network has a shape of $batch \times 2 \times 60 \times 60 \times 64 \times 1$ (NCXYZT), where permeability and (geographical) topography are provided as the two input channels. The output has a shape of $batch \times 1 \times 60 \times 60 \times 64 \times n_t$.

Our network is again identical in structure to the original proposed by Li et al. [2], consisting of a total of 4 spectral convolution blocks, and a lifted space dimension of 20. The network was trained for 30 epochs using 800 training data points and 100 validation data points with a batch size of 1, and used the Adam optimizer [41] with a learning rate of 10^{-3}. As in the original FNO paper, we measure the data misfit using the relative L^p loss with $p = 2$ [2]:

$$L(y, \hat{y}) = \frac{\|y - \hat{y}\|_p}{\|\hat{y}\|_p}.$$

We train our model on a Standard_NC24 virtual machine on the Azure cloud, which has 4 Nvidia Volta V100 GPUs with 16 GB memory each. We therefore create a worker partition of shape $1 \times 1 \times 1 \times 4 \times 1 \times 1$, which results in an input shape per worker of $batch \times 2 \times 60 \times n_y \times 64 \times n_t$ where $n_y = 15$. It should be noted that this training setup does not fit within the memory of a single V100 GPU and thus constitutes true model-parallel training. While our FNO implementation is in principle able to handle much larger data sizes, as shown in our previous scaling experiments, there are currently no larger public datasets for training multi-phase flow simulators available. The training dataset in our experiment is stored in Azure's object store (Blob storage) in the Zarr format, whose corresponding Python packages provides an API for storing chunked n-dimensional tensors on both file systems and object stores [42]. During training, each MPI rank reads its corresponding domain of the input data directly from the object store, so no single GPU has to fit the full input (or output) data into its memory at any given time.

Figure 6 shows the history of the training and validation loss as a function of the training epoch. It is noticeable that the validation loss stalls around a value of 0.4, while the training loss decreases until the final epoch. Nevertheless, the trained network performs reasonably well on unseen test samples, as shown in figure 8 by a comparison of the predicted CO_2 saturation with the corresponding simulated data samples. A three-dimensional plot of a test samples that correctly visualizes the varying grid topography is shown in Figure 7. The inference time of predicting the CO_2 saturation history for a single input sample was

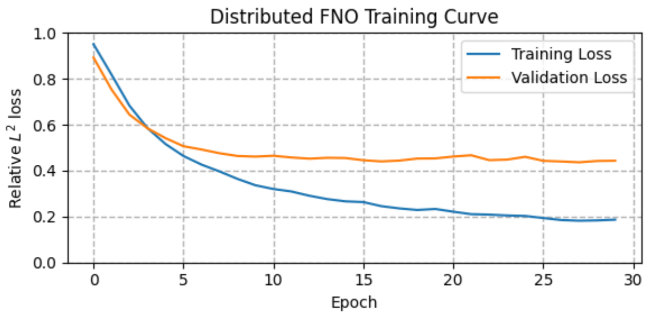

Figure 6: Training and validation loss curves for running distributed FNO training experiment. Validation loss plateaus around a value of 0.4, so the network training was halted after 30 epochs.

measured to be in the range of 0.45 seconds, including the data transfer to and from the main CPU memory to the GPU. In contrast, the simulation time of a single sample with the Open Porous Media simulator, which was used to compute the original training examples in [37], lies in the order of 3 minutes per sample on an Azure HBv2 HPC node with 120 CPU cores. The achieved 400x speedup with FNOs thus brings the ability to solve 3D inverse problems as motivated earlier in Section 1.2 within reach. Our implementation of distributed FNOs is made available at https://github.com/slimgroup/dfno.

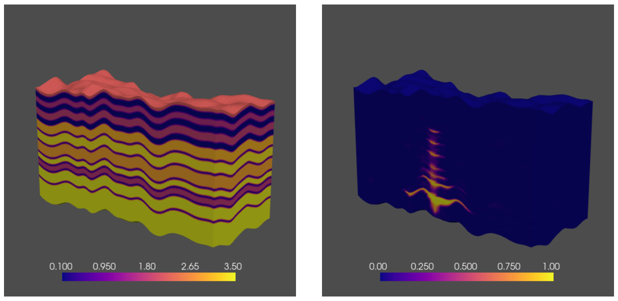

Figure 7: Input permeability/topography map and output CO_2 plume at the final timestep for a validation sample run through our trained 4D two-phase flow FNO.

4. Conclusion

In this work, we have presented a domain-decomposition based implementation of model-parallel Fourier neural operators for data of arbitrary size and dimensionality. Using a linear-algebraic formulation of parallelism, we derive mathematically all requisite components of distributed FNOs and provide an implementation of our distributed network in PyTorch using DistDL. We demonstrate our network's weak scaling capabilities on Summit and show an example of training a model-parallel 4D FNO to learn solutions to the two-phase flow equations for predicting the time-evolution of subsurface CO_2 plumes. To our knowledge, this is the first implementation of an FNO to scale beyond 64^3 in the spatial dimensions. Our work provides a critical first step in the ability to solve coupled inverse problems on realistically-sized data by rapidly accelerating inference and gradient calculations on large volumetric problems via distributed operator learning.

Acknowledgments

This research was carried out with the support of Georgia Research Alliance and partners of the ML4Seismic Center. The authors acknowledge Erik Skjetne (Equinor) and Tugrul Konuk (formerly Colorado School of Mines and Microsoft Research intern) for their contributions to the generation of the Sleipner CO_2 training dataset. Russell J. Hewett was funded by Department of Energy, Office of Science, Early Career Research Program award DE-SC0022041. This work is supported by the Department of Energy under Grant No. DE-SC0021515. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

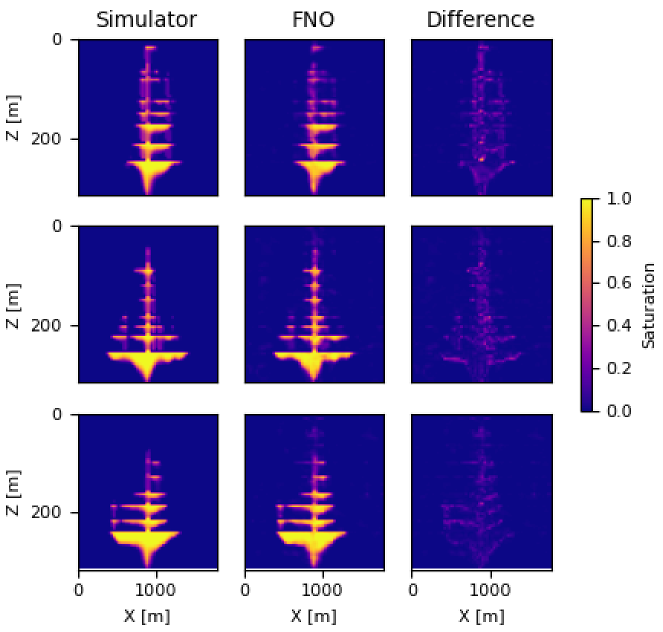

Figure 8: Vertical slices out of the last timestep of our 4D FNO trained for two-phase flow simulations.

References

[1] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, "Neural operator: Graph kernel network for partial differential equations," *arXiv preprint arXiv:2003.03485*, 2020.

[2] ——, "Fourier neural operator for parametric partial differential equations," *arXiv preprint arXiv:2010.08895*, 2020.

[3] R. J. LeVeque, *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. SIAM, 2007.

[4] T. J. Hughes, *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation, 2012.

[5] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical analysis*. Cengage learning, 2015.

[6] A. Gokhberg and A. Fichtner, "Full-waveform inversion on heterogeneous hpc systems," *Computers & Geosciences*, vol. 89, pp. 260–268, 2016.

[7] T. C. Schulthess, P. Bauer, N. Wedi, O. Fuhrer, T. Hoefler, and C. Schär, "Reflecting on the goal and baseline for exascale computing: a roadmap based on weather and climate simulations," *Computing in Science & Engineering*, vol. 21, no. 1, pp. 30–41, 2018.

[8] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman, "Devito (v3. 1.0): an embedded domain-specific language for finite differences and geophysical exploration," *Geoscientific Model Development*, vol. 12, no. 3, pp. 1165–1187, 2019.

[9] D. Su, K. U. Mayer, and K. T. MacQuarrie, "Min3p-hpc: a high-performance unstructured grid code for subsurface flow and reactive transport simulation," *Mathematical Geosciences*, vol. 53, no. 4, pp. 517–550, 2021.

[10] J. Sirignano and K. Spiliopoulos, “Dgm: A deep learning algorithm for solving partial differential equations,” *Journal of computational physics*, vol. 375, pp. 1339–1364, 2018.

[11] L. Lu, P. Jin, and G. E. Karniadakis, “Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators,” *arXiv preprint arXiv:1910.03193*, 2019.

[12] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, “Physics-informed machine learning,” *Nature Reviews Physics*, vol. 3, no. 6, pp. 422–440, 2021.

[13] G. Wen, Z. Li, K. Azizzadenesheli, A. Anandkumar, and S. M. Benson, “U-fno–an enhanced fourier neural operator based-deep learning model for multiphase flow,” *arXiv preprint arXiv:2109.03697*, 2021.

[14] J. Guibas, M. Mardani, Z. Li, A. Tao, A. Anandkumar, and B. Catanzaro, “Adaptive fourier neural operators: Efficient token mixers for transformers,” *arXiv preprint arXiv:2111.13587*, 2021.

[15] J. Pathak, S. Subramanian, P. Harrington, S. Raja, A. Chattopadhyay, M. Mardani, T. Kurth, D. Hall, Z. Li, K. Azizzadenesheli *et al.*, “Fourcastnet: A global data-driven high-resolution weather model using adaptive fourier neural operators,” *arXiv Preprints*, 2022.

[16] M. V. de Hoop, D. Z. Huang, E. Qian, and A. M. Stuart, “The cost-accuracy trade-off in operator learning with neural networks,” 2022. [Online]. Available: https://arxiv.org/abs/2203.13181

[17] Z. Yin, A. Siahkoohi, M. Louboutin, and F. J. Herrmann, “Learned coupled inversion for carbon sequestration monitoring and forecasting with fourier neural operators,” *arXiv preprint arXiv:2203.14396*, 2022.

[18] A. F. Rasmussen, T. H. Sandve, K. Bao, A. Lauser, J. Hove, B. Skaflestad, R. Klöfkorn, M. Blatt, A. B. Rustad, O. Sævareid *et al.*, “The open porous media flow reservoir simulator,” *Computers & Mathematics with Applications*, vol. 81, pp. 159–185, 2021.

[19] H. Gross and A. Mazuyer, “Geosx: A multiphysics, multilevel simulator designed for exascale computing,” in *SPE Reservoir Simulation Conference*. OnePetro, 2021.

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

[21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “{TensorFlow}: A system for {Large-Scale} machine learning,” in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[22] B. Yan, B. Chen, D. R. Harp, and R. J. Pawar, “A robust deep learning workflow to predict multiphase flow behavior during geological co2 sequestration injection and post-injection periods,” *arXiv preprint arXiv:2107.07274*, 2021.

[23] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, “Megatron-lm: Training multi-billion parameter language models using model parallelism,” *arXiv preprint arXiv:1909.08053*, 2019.

[24] L. Yang, S. Treichler, T. Kurth, K. Fischer, D. Barajas-Solano, J. Romero, V. Churavy, A. Tartakovsky, M. Houston, M. Prabhat *et al.*, “Highly-scalable, physics-informed gans for learning solutions of stochastic pdes,” in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 2019, pp. 1–11.

[25] R. J. Hewett, T. Grady, and J. Merizian, “distdl/distdl: Version 0.4.0 release,” Sep. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5360401

[26] R. J. Hewett and T. J. Grady II, “A linear algebraic approach to model parallelism in deep learning,” *arXiv preprint arXiv:2006.03108*, 2020.

[27] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.

[28] J. Utke, L. Hascoet, P. Heimbach, C. Hill, P. Hovland, and U. Naumann, “Toward adjoinable mpi,” in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–8.

[29] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith *et al.*, “Array programming with numpy,” *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[30] A. H. Barr, “The einstein summation notation,” *An Introduction to Physically Based Modeling (Course Notes 19), pages E*, vol. 1, p. 57, 1991.

[31] L. Dalcin, M. Mortensen, and D. E. Keyes, “Fast parallel multidimensional fft using advanced mpi,” *Journal of Parallel and Distributed Computing*, vol. 128, pp. 137–150, 2019.

[32] M. Pippig, “Pfft: An extension of fftw to massively parallel architectures,” *SIAM Journal on Scientific Computing*, vol. 35, no. 3, pp. C213–C236, 2013.

[33] “Summit - ibm power system ac922, ibm power9 22c 3.07ghz, nvidia volta gv100, dual-rail mellanox edr infiniband.” [Online]. Available: https://www.top500.org/system/179397/

[34] J. Gibbins and H. Chalmers, “Carbon capture and storage,” *Energy policy*, vol. 36, no. 12, pp. 4317–4322, 2008.

[35] P. Ringrose, *How to Store CO2 Underground: insights from early-mover CCS Projects*. Springer, 2020.

[36] M. Tang, X. Ju, and L. J. Durlofsky, “Deep-learning-based coupled flow-geomechanics surrogate model for co _2 sequestration,” *arXiv preprint arXiv:2105.01334*, 2021.

[37] P. A. Witte, T. Konuk, E. Skjetne, and R. Chandra, “Carbon capture in geological formations optimized by machine learning,” in *Climate Change AI Webinar series*, Nov 2021.

[38] “Sleipner 2019 benchmark model,” https://co2datashare.org/dataset/sleipner-2019-benchmark-model, accessed: 2021-12-20.

[39] J. C. Andrew, R. S. Haszeldine, and B. Nazarian, “The sleipner co2 storage site: using a basin model to understand reservoir simulations of plume dynamics,” *First Break*, vol. 33, no. 6, 2015.

[40] A.-K. Furre, O. Eiken, H. Alnes, J. N. Vevatne, and A. F. Kiær, “20 years of monitoring co2-injection at sleipner,” *Energy procedia*, vol. 114, pp. 3916–3926, 2017.

[41] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[42] “Zarr,” https://github.com/zarr-developers/zarr-python, accessed: 2021-12-28.

Appendix A. Scaling Experiment Run Configurations

In figures 9 and 10 we show all run configurations for weak scaling experiments. Each timing result reported is the maximum value given by any single worker. p indicates the number of workers. "Partition Shape" indicates the Cartesian partition topology applied over the global input/output shape, given by "Input Shape" and "Output Shape" respectively. Any timing result with $eval$ indicates that the network was run with `torch.no_grad()` and `network.eval()`, while any timing result with $grad$ indicates that the network was run without `torch.no_grad()` and with `network.train()`. fwd indicates a forward pass of the network `y = network(x)`, while adj indicates backpropagation, `y.backward()`.

Appendix B. Index Calculation in Application of R_ϕ

Given an n-dimensional partition P of shape $d_1 \times d_2 \times \dots d_n$ of an n-dimensional tensor x, we can calculate whether the nonzero image of R_ϕ is within the subtensor of an individual worker i. To do this, we first compute (and subsequently cache for later applications of the network) the following information via a collective communication:

- The global shape of the input tensor.
- The length-n list of tuples (a_i^k, b_i^k) describing the start/stop of the subtensor relative to to the global tensor in each dimension k.

Then, because the nonzero image of R_ϕ is defined as a low pass filter, we are looking only for the overlap between the i-th worker's subtensor and the corners of an n-dimensional volume representing the input to the layer. These corners correspond to the low frequency modes $(m_1, \dots, m_n)$ in each dimension, a hyperparameter set by the user. The algorithm for computing the size of this overlap on a particular worker i is defined in algorithm 1.

p	Partition Shape	Input Shape	Output shape	t_{eval}^{fwd}	t_{grad}^{fwd}	t_{grad}^{adj}
1	(1,1,1,1,1,1)	(1,1,64,64,64,1)	(1,1,64,64,64,16)	3.64	3.58	3.18
2	(1,1,2,1,1,1)	(1,1,128,64,64,1)	(1,1,128,64,64,16)	3.09	3.03	3.24
4	(1,1,2,2,1,1)	(1,1,128,128,64,1)	(1,1,128,128,64,16)	4.02	3.92	4.39
6	(1,1,3,2,1,1)	(1,1,196,128,64,1)	(1,1,196,128,64,16)	4.37	3.90	4.77
12	(1,1,3,2,2,1)	(1,1,196,128,128,1)	(1,1,196,128,128,16)	4.57	4.39	4.83
24	(1,1,4,3,2,1)	(1,1,256,196,128,1)	(1,1,256,196,128,16)	4.91	4.44	5.13
48	(1,1,4,4,3,1)	(1,1,256,256,196,1)	(1,1,256,256,196,16)	5.80	5.42	5.76
96	(1,1,6,4,4,1)	(1,1,384,256,256,1)	(1,1,384,256,256,16)	6.31	5.55	5.91
192	(1,1,8,6,4,1)	(1,1,512,384,256,1)	(1,1,512,384,256,16)	7.09	6.04	7.00
384	(1,1,8,8,6,1)	(1,1,512,512,384,1)	(1,1,512,512,384,16)	9.55	7.09	7.71
768	(1,1,12,8,8,1)	(1,1,768,512,512,1)	(1,1,768,512,512,16)	11.02	8.40	8.5

Figure 9: Table of run configurations and timings results for all spatial weak scaling experiments.

p	Partition Shape	Input Shape	Output shape	t_{eval}^{fwd}	t_{grad}^{fwd}	t_{grad}^{adj}
1	(1,1,1,1,1,1)	(1,1,64,64,64,1)	(1,1,64,64,64,16)	3.64	3.58	3.58
2	(1,1,2,1,1,1)	(1,1,64,64,64,1)	(1,1,64,64,64,32)	3.22	3.14	3.26
4	(1,1,2,2,1,1)	(1,1,64,64,64,1)	(1,1,64,64,64,64)	3.65	3.41	3.86
6	(1,1,3,2,1,1)	(1,1,64,64,64,1)	(1,1,64,64,64,96)	3.46	3.22	3.96
12	(1,1,3,2,2,1)	(1,1,64,64,64,1)	(1,1,64,64,64,192)	3.98	3.67	4.11
24	(1,1,4,3,2,1)	(1,1,64,64,64,1)	(1,1,64,64,64,384)	4.41	3.85	4.42
48	(1,1,4,4,3,1)	(1,1,64,64,64,1)	(1,1,64,64,64,768)	5.03	4.52	4.74
96	(1,1,6,4,4,1)	(1,1,64,64,64,1)	(1,1,64,64,64,1536)	5.84	4.87	5.36
192	(1,1,8,6,4,1)	(1,1,64,64,64,1)	(1,1,64,64,64,3072)	6.18	5.14	5.28
384	(1,1,8,8,6,1)	(1,1,64,64,64,1)	(1,1,64,64,64,6144)	8.88	7.31	8.86

Figure 10: Table of run configurations and timings results for all temporal weak scaling experiments.

Algorithm 1: Calculation of weight shapes in the spectral convolution on worker i

```
Input: Global shape of the input tensor $(s_1, \dots, s_n)$
Input: Global start/stop for each dimension on worker $i$ $[(a_i^1, b_i^1), \dots, (a_i^n, b_i^n)]$
Input: Number of modes to keep in each dimension $(m_1, \dots, m_n)$
Output: List of weight sizes on worker $i$
weight_shapes ← []
for $j \leftarrow 0$ to $2^n$ do
    $b \leftarrow$ binary($j$)
    shape → []
    for $k \leftarrow 0$ to $n$ do
        dim ← $n - k - 1$
        digit ← $b[dim]$
        start ← $a_i^{dim}$
        stop ← $b_j^{dim}$
        gs ← $s_{dim}$
        mode ← $m_{dim}$
        if digit == 0 then
            shape.append($(\max\{0, start\} - start, \min\{mode, stop\} - start)$)
        end
        else
            shape.append($(\max\{gs - mode, start\} - start, \min\{gs, stop\} - start)$)
        end
    end
    weight_shapes.append(shape)
end
return weight_shapes
```