ShardTensor: Domain Parallelism for Scientific Machine Learning

Corey Adams
NVIDIA
Santa Clara, CA, USA
coreya@nvidia.com

Peter Harrington
NVIDIA
Santa Clara, CA, USA
pharrington@nvidia.com

Akshay Subramaniam
NVIDIA
Santa Clara, CA, USA
asubramaniam@nvidia.com

Mohammad Shoaib Abbas
NVIDIA
Santa Clara, CA, USA
mohammadshoa@nvidia.com

Jaideep Pathak
NVIDIA
Santa Clara, CA, USA
jpathak@nvidia.com

Mike Pritchard
NVIDIA
Santa Clara, CA, USA
mpritchard@nvidia.com

Sanjay Choudhry
NVIDIA
Santa Clara, CA, USA
schoudhry@nvidia.com

***Abstract*—Scientific Machine Learning (SciML) faces unique challenges for extreme-resolution data, with mitigations that often fail to scale or degrade the accuracy of trained models. While some specialized methods have achieved remarkable results in training models or performing inference on massive spatial datasets with bespoke techniques, there is no generalized framework for parallelization over input data below batch size one per device. In this work we introduce `ShardTensor`: a novel paradigm of domain parallelism that enables flexible scaling of input data to arbitrary sizes. By decoupling the spatial dimensionality of input data from hardware constraints, `ShardTensor` enables scientific machine learning workloads to reach new levels of high fidelity training and inference. We demonstrate both strong and weak scaling of workloads during training and inference, showing improved latency with strong scaling and demonstrating the capacity to process higher data sizes with weak scaling. Additionally, we demonstrate multiple dimensions of parallelization, removing barriers to SciML on extreme-scale inputs.**

***Index Terms*—HPC for ML; Parallel and distributed learning algorithms; Model, pipeline, and data parallelism**

I. Introduction

Scientific machine learning applications have become a vehicle for accelerated simulation, scientific discovery, and industrial design. Machine learning has found applications in an incredible breadth of domains: healthcare and medicine [1], [2], industrial design [3]–[5], fluid dynamics [6] and aerodynamics [7], weather and climate forecasting [8], [9], fundamental sciences [10]–[12], and many, many more [13]–[15]. It is not an overstatement to say that machine learning methods are fundamentally changing scientific research, all the way from early development to end user and industrial applications.

Scientific data has several attributes that make it especially challenging to use for both training and inference, leading to reduced adoption or degraded applications of these scientific ML models. First, the data in scientific models is typically of **high spatial resolution**, with scientists working with a "more is better" philosophy - and rightly so. Higher resolution imaging across a breadth of scientific domains often leads to breakthrough results, from the first ever images of a black hole [16] to achieving atomic-resolution protein structures in cryo-electron microscopy [17], and mapping human cerebral cortexes at petavoxel scales [18]. In multi-decadal Earth System projection, climate-critical cloud-forming turbulent processes require tens of meters in space and seconds in time to satisfyingly resolve, which remains far beyond the computational capacity of even the most ambitious global simulation frameworks [19]–[22].

From a scientific perspective, high resolution data is an aspiration. But from a computational perspective, high resolution data is a challenge; and from a machine learning perspective, where GPU memory resources can quickly become a bottleneck, high resolution data is a *major* challenge.

Further, scientific data suffer from a computational curse of dimensionality: doubling the length of text for a language model will increase the number of input tokens by approximately double; doubling the resolution of N dimensional data will increase the size of scientific data by 2^N. Scientific machine learning models rapidly encounter challenges in GPU memory management, especially for model training.

Building machine learning tools and techniques that can train and run inference on models at the native resolution of scientific data is a challenge the High Performance Computing community is well-positioned to address. Our contribution in this paper is a framework for high-resolution SciML that provides the simplicity and accessibility expected of PyTorch and its ecosystem while enabling this native-resolution paradigm.

In this paper, we will describe ShardTensor, an abstraction and extension to a PyTorch `tensor` that allows **domain parallelism**, defined here as parallelism across devices for the input data, below even batch size one. As an example, an input batch of 3D Tensors of shape [B, C, H, W, D] (B=batch, C=channels, H=height, W=width, D=depth) would be partitioned across the B axis in a standard "data parallel" distribution. In domain parallelism, we extend this to partition further: when the limit of B=1 is reached, and each GPU has a single image, further subdivision is possible along the spatial

dimensions. The name "domain parallelism" is taken from the analogous techniques in classical solvers in Computational Fluid Dynamics, numerical methods and other fields, in which this type of domain decomposition has existed for decades [23]–[25].

In this paper, we proceed as follows. First, we will provide a simplified overview of the origins of GPU memory usage for scientific machine learning, especially as it relates to high resolution data. The goal here is to motivate *why* domain parallelism is an avenue worth pursuing. Next, we will provide a short overview of some common techniques for reducing memory usage in scientific machine learning, followed by a description of related works focused on parallelism techniques in machine learning.

Finally, we will introduce `ShardTensor`, starting with the design principles and goals, differentiation from `DTensor` [26], and expected use cases. We highlight some existing applications, performance results, and expected areas where it might be of use to users. `ShardTensor` is already available for use, open source, through the NVIDIA PhysicsNeMo framework [27].

II. What Causes High GPU Memory Usage?

To motivate our discussion of domain parallelism techniques, we begin with an overview of the dominant origins of GPU memory usage in most scientific machine learning workloads. Of course, with the diversity of scientific workloads, it is impossible to provide an exhaustive and prescriptive description of every source of memory usage; unique workloads that require second order optimizers or non-reverse-mode auto-differentiation techniques will not necessarily fit this categorization.

For a standard machine learning workload, we categorize the dominant memory drivers into four broad categories, two of which are specific to training only. For simplicity we only discuss 32-bit numerical precision in this section, but a discussion of reduced precision is found in Section II-B.

1) **Model Parameters** (weights, biases, encodings, etc.) contribute substantial GPU memory usage in Large Language Models (LLMs) but typically only modest usage in scientific models - though that is a trend that is changing. Every parameter in 32-bit precision will require 4 bytes of GPU storage, meaning 1 million parameters requires approximately 1 MB of GPU storage. Model parameters require storage in both training and inference.
2) **Active Data**, or the transient working memory for a single operation - including input/output buffers and any temporary workspace the kernel requires - occupies GPU memory only for the duration of the currently executing computation. Naturally, this is necessary during both training and inference.
3) **Optimizer States** represent the gradients, moments, or other related tensors required to apply updates like Adam [28], RMSProp [29], and other extensions of stochastic gradient descent that store gradients and other additional information. Typically, in 32-bit precision, this is a multiplicative factor of the storage required for the model parameters: a factor of two or three is typical.
4) **Intermediate Activations** hold the cached primals of a layer to enable reverse mode auto differentiation. For each layer, depending on the specifics of the layer, one or more input or output tensors are saved during the forward pass. During the backward pass, these tensors are reused to propagate the gradients backward through the network, according to the chain rule. As we will show below, for high resolution input data and modest parameter counts, **intermediate activations are the dominant GPU memory consumer during training**.

A. An Example of Memory Usage

As a concrete example, let us consider the basic building block of many machine learning models, the `Linear` layer:

$$z = Wx + B \tag{1}$$

Where $x \in \mathbb{R}^{N_{in}}$ is the input vector, and $z \in \mathbb{R}^{N_{out}}$ is the output vector. W is the weight matrix of shape $[N_{out}, N_{in}]$ and B is a learnable bias vector. The total number of parameters in the layer is therefore $N_p = N_{in} \times N_{out} + N_{out}$. The total memory usage by the layer's parameters is simply αN_p, measured in bytes, which depends on the floating point precision used. For float32, α is 4; for half precision, α is 2. Additionally, during training, an optimizer such as AdamW [30] must track the gradients (an additional copy of αN_p) as well as both the momentum and variance vectors for the gradients.

Now let's consider, from the other perspective, the impact of the intermediate activation on GPU memory allocations. For a `Linear` layer, the gradient formulas are

$$\frac{dL}{dx} = \frac{dL}{dz}\frac{dz}{dx} = \frac{dL}{dz}W \tag{2}$$

$$\frac{dL}{dW} = \frac{dL}{dz}\frac{dz}{dW} = \frac{dL}{dz}x \tag{3}$$

$$\frac{dL}{dB} = \frac{dL}{dz}\frac{dz}{dB} = \sum_{batch}\frac{dL}{dz} \tag{4}$$

Where L is the loss, and $\frac{dL}{dx}$, $\frac{dL}{dW}$, and $\frac{dL}{dB}$ represent the gradients with respect to the inputs, weights, and bias respectively. In this case, computing the gradients with respect to the weights requires x, so the forward pass will save the inputs x for the backwards pass - holding these intermediate activations in memory until the backward pass has used them, and they can be released. The size of this allocation is *directly proportional* to the input tensor shape, and it is the total number of elements that matters. For high resolution scientific data, and especially high dimensional data in 3D or higher dimensions where the total number of elements scales to the power of the dimension D, memory allocations can grow exceedingly quickly.

Further, since memory allocation by intermediate activation accumulates per layer, the overall depth of a model in number

Spatial Dimension	Model Layers	Features	N_{params}	Weights (MB)	Activations (MB)
(256,)	20	1024	21.0M	80.1	20
(256,)	20	8192	1.3B	5120.6	160
(256,256)	20	1024	21.0M	80.1	5120
(256,256)	20	8192	1.3B	5120.6	40,960
(256,256, 256)	20	1024	21.0M	80.1	1,310,720
(256,256, 256)	20	8192	1.3B	5120.6	10,485,760

TABLE I
SUMMARY OF THE MEMORY USAGE OF A SEQUENCE OF LINEAR LAYERS ON VARIOUS INPUT DATA SHAPES, AS A FUNCTION OF NUMBER OF LAYERS, SPATIAL SHAPE (INCLUDING DIMENSION), AND NUMBER OF FEATURES. FOR SIMPLICITY, EACH LAYER HAS THE SAME NUMBER OF FEATURES FOR INPUT AND OUTPUT. ALL CALCULATIONS ASSUME BATCH SIZE 1.

of layers (and type of layer) will impact the total amount of data that must be saved for a backward pass. In other words, for high resolution data, deeper models often require more memory in training primarily due to the increased activations saved, and not because of the increased number of parameters - that is a secondary effect.

Typical LLM models use N_{in} and N_{out} in the range of O(10,000) or higher, while frequently scientific operator-learning AI models like FNOs [31], Transolver [32], DoMINO [33], and others work at lower dimensional latent spaces below 1,000. Table I summarizes the impacts of parameters, optimizer states, and intermediate activations on memory usage, as the number of features or number of input points vary.

As seen in Table I, despite being a contrived example, higher resolution data quickly outpaces the memory usage of model weights - especially in higher dimensions. It is exactly this explosion in GPU memory usages that we seek to parallelize over with domain parallelism and `ShardTensor`.

B. Reducing GPU Memory Consumption for High Resolution Data

For scientific data, training even modest parameter-count models at high resolution can become computationally impractical due to memory constraints, leading to a number of workarounds. Naturally, the users of scientific machine learning are interested in, first and foremost, achieving their scientific mission with the least computational difficulties. A number of strategies can be employed to enable high resolution training and inference. Parallelization strategies are discussed instead in Section III, on related works.

- **Reduced Precision** is a common and effective method that is, practically speaking, the first line of attack at reducing both activation and model weight memory usage. Both `bfloat16` and `float16` training [34] are stable and convergent for most models, and LLMs have pioneered many techniques for further lower-precision optimizations [35], [36]. In scientific machine learning, there can sometimes be challenges with sufficient dynamic range in model outputs for surrogate simulations, and computationally reduced precision offers only modest memory savings - typically a factor of 2x when using half precision.
- **Spatial Downsampling** is perhaps the most obvious and simplest path towards reducing the memory cost of intermediate activations in training a scientific ML model. Many problems, especially neural operators [31]–[33], [37], [38] are trained very successfully at reduced spatial sampling, though some evidence [39], [40] indicates higher spatial resolution during training can in fact lead to better convergence of operator models. Other problems, especially imaging problems, inherently suffer lack of information when downsampling and can not be trivially downsampled without more sophisticated algorithmic improvements.
- **Model Reduction** can also lead to significant reduction in memory usage for high resolution scientific models, though not because of the reduction in parameter storage; the reduction in saved activations by reducing the number of layers, or number of channels per layer, can be significant. Unfortunately, this can often come at the cost of reduced application accuracy.
- **Activation Checkpointing** and **CPU Offloading** are the most promising, flexible, and versatile techniques available for resolving memory constraints due to intermediate activations. Recalling our `Linear` layer's `backwards` example, since the input x is not needed until the model reaches this layer in the backward pass, x can be safely moved to CPU memory or further-away storage until needed. Even more extreme, several consecutive layers could drop all but the first x activations and recompute them on-the-fly in the backward pass from the single saved tensor. Both methods incur extra computational bottlenecks: host-to-device transfers, extra GPU computations, or both. However, both methods can reduce GPU memory usage on high resolution data with no detrimental impacts on data resolution or model accuracy. Better still, these optimizations are only needed during **training**, and inference can proceed fully optimized.
- **Sparsity or Lower Dimensional Representations** can enable alternative methods such as SparseConvNets [41], Minkowski Networks [42], FigConvNet [43] and other methods. In many cases, especially as spatial dimensionality rises, taking advantage of inherent structure and sparsity of the data structures of scientific data is crucial to achieving both accurate results and high performance for machine learning.

III. RELATED WORK

Other methods and techniques of parallelization for machine learning have seen success over the past decade, including some recent developments upon which this work is built. Here we summarize the most impactful and relevant works to this research.

A. Within PyTorch

The work described in this paper is built on top of the PyTorch framework, so we first describe the related work in the PyTorch ecosystem. The earliest forms of parallelism

in machine learning were **data parallel** learning, first via `horovod` [44], and now most commonly through PyTorch's `DDP` [45]. Data parallel learning, as the name implies, allows parallelizing over the batch dimension to arbitrary scale (provided the computational resource allows it, and the dataset size is large enough). With data parallel learning came significant research into strong-scaling machine learning algorithms, with focus on optimizers [46], [47] to accelerate convergence and set record training times for challenging problems [48].

As model parameter counts grew in the early 2020s, the era of Large Language Models led to new developments in model parallelization. Some of the earliest billion parameter models via the Megatron [49] framework from NVIDIA led to breakthroughs in convergence of language models. Subsequent work from DeepSpeed [50] made multi-billion parameter model training possible. As of publication of this manuscript, similar technology as DeepSpeed is available through PyTorch's `DTensor` and `FullyShardedDataParallel` abstractions [26], [51].

`DTensor` is a distributed Tensor abstraction that enables parallelization of a generic tensor over a set of GPUs, targeting model parallel training. It uses placement specifications such as `Shard` and `Replicate` to describe how a tensor is distributed across a logical device mesh, and automatically inserts the necessary collective communications (e.g., all-reduce, all-gather) when operating on distributed tensors.

At first glance, `DTensor` itself might possibly be used for domain parallelism on the input data, but it is not possible. Baked into `DTensor` is an assumption on static distribution shapes: because `DTensor` is designed to represent *weights*, not inputs or outputs, it is not expected to change shapes dynamically. As a concrete example, consider a convolution operation: an evenly distributed input tensor, when processed with a convolution that changes the global shape, will produce output chunks that are no longer evenly distributed – violating `DTensor`'s assumption. Further, simplifying assumptions can be made about the distributed memory layout of `DTensor` that can not be made about distributed input and output tensors to a machine learning layer. However, the sophisticated machinery of `DTensor` is sufficient to provide the bulk of the operations needed to build `ShardTensor`, as will be seen below. We extend where necessary and interoperate smoothly where we can.

Additionally, an alternative paradigm of parallelism known as Pipeline Parallelism [52], [53] is useful in certain scenarios. While not necessarily a computationally efficient technique in terms of scaling without careful tuning, pipeline parallelism can offer memory efficiency and has significantly lower overall networking requirements than alternatives such as data parallel training. For inference, and especially with well tuned pipelines, pipeline parallelism can be a compelling option. For the high resolution challenges we seek to address in this paper, it is not necessarily a universally suitable option, and we will not discuss it further here.

Finally, a number of bespoke parallelization efforts have been made that should be considered domain parallelism, including Ring Attention [54], Makani [55], and techniques in Transolver^{++} [39]. While all excellent demonstrations of the power of domain parallelism at reaching higher spatial or input resolution, they are not easily extendable nor as broadly applicable to new models as `ShardTensor`, as described below. Many of the operations and algorithms in those works have been adopted and implemented in `ShardTensor` as optimized dispatch paths for certain layers.

B. Outside of PyTorch

The largest deep learning frameworks outside of PyTorch, such as TensorFlow [56], JAX [57], and PaddlePaddle [58] all support some form of data parallel training. TensorFlow also has native support for a distributed tensor, very similar to PyTorch's DTensor.

JAX uniquely has a very interesting and composable `shard_map` decorator to build single-program, multi-data programs from arbitrary tensor shapes. In many ways, `shard_map` is something of a Swiss-army-knife of parallel programming for scientific computing, and domain parallelism could be implemented for many operations in JAX. However, we are restricting to a single popular framework (PyTorch) and technique here. We encourage interested readers to learn more [57].

IV. ShardTensor

We have, to this point, motivated the challenges facing scientific machine learning when it comes to managing high resolution data and memory management. The goal, then, is to build a usable and generic framework that enables parallelization along dimensions that, to date, have not generically been parallelizable: the high resolution data dimensions. We emphasize a performance limitation of this design from the start: it is almost always more performant to parallelize over the batch dimension, if possible. Domain parallelization should be employed to train models when batch size 1 training is not possible.

We seek to build a framework to enable generic, simple, and performant domain parallelism, and a number of design decisions emerge clearly from the discussion above and successes of other paradigms.

The most flexible framework for domain parallelism must be imperative rather than static. That is, it must dispatch collectives on-demand: the framework must be able to work within the PyTorch paradigm of not necessarily knowing what operation will come next, and therefore every single layer must be computable in a domain parallel way. Further, since domain parallelism will often require communication between devices at any particular layer, an intimate relationship between the collective devices, current GPU operation, and data-under-operation must be maintained. The natural choice is to utilize PyTorch's dispatch methods for `torch.Tensor` extensions, and to extend their distributed tensor class `DTensor`.

At its core, `ShardTensor` is an extension to PyTorch's `DTensor` with a few critical extensions necessary for

TABLE II
COMPARISON OF DTENSOR AND SHARDTENSOR, FEATURES AND EXPECTED USE CASE. SHARDTENSOR IS AN EXTENSION OF DTENSOR, DESIGNED FOR DOMAIN PARALLELISM

	DTensor	**ShardTensor**
Primary use case	Model parallelism (weights)	Domain parallelism (input data)
Tensor metadata	Global shape, mesh, placement	Global shape, mesh, placement, *sharding shapes*
Chunk distribution	Even (`torch.chunk`)	Arbitrary per-rank sizes
Shape assumptions	Static (model weights)	Dynamic (data-dependent)
User extensibility	`__torch_dispatch__`	`__torch_dispatch__`, `__torch_function__`, and `@custom_op`

domain parallelism. As background, a distributed tensor combines three pieces of information: global shape information for the tensor, a description of the devices the tensor resides on (known as a `Mesh`) and a description of *how* the tensor has been sharded across the devices. A `Mesh` can be multi-dimensional, and a tensor can be sharded across more than one dimension as well.

`DTensor`, working with statically-shaped model weights, assumes that tensors are **always** distributed according to `torch.chunk` syntax across a dimension of the `Mesh`. Since the input and output shapes of a function are not static, in general, through any given operation, we can not make a similar assumption in `ShardTensor` – as illustrated by the convolution example above.

Therefore a fourth component of information is essential to describe a `ShardTensor` but not a `DTensor`: "sharding shapes", making each tensor aware of the local chunk shape of each tensor along its sharded mesh axis. This information also enables arbitrary chunking of unstructured or non-uniform data, such as point clouds and meshes.

Both `DTensor` and `ShardTensor` support sharding over an arbitrary number of GPU mesh dimensions, however, it should not be expected (for either tensor extension) that all operations support an arbitrary amount of shardings.

A. User Facing Considerations

First, and most importantly, with `ShardTensor` we seek to provide - as much as reasonably possible - a non-invasive style of domain parallelism in the style of `DDP` and `FSDP`. We expect users to, in general, not apply bespoke patches to layers or models to enable parallelisms; we instead expect users to want to apply a thin wrapper to their model inputs that will enable a set of under-the-hood dispatch paths, in turn enabling layer-by-layer domain parallelism.

Second, we recognize that models, frameworks, and operations evolve, and the pace of evolution has never been more rapid. To this end, `ShardTensor` is inherently extensible. Users can extend both PyTorch operations, as well as custom kernels, layers, or models, through both a high-level functional interface and a low-level dispatch interface.

Finally, since performance with PyTorch is already *excellent* in most cases, we focus performance for domain parallelism where it matters most: when the input data is extremely large.

B. Implementation and Performance

A key consideration of `ShardTensor` is flexibility in user space: distributed operations must be flexibly dispatched by PyTorch depending on user code, and not based on any pre-compiled computational graphs or models. To enable this, we follow closely the philosophy of `DTensor` in upstream PyTorch, though with several user-facing entry points for extensibility deliberately designed for better interoperability with custom user operations.

1) PyTorch Dispatch: PyTorch uses a **dispatch** mechanism [59] to route operations from Python to the correct device and kernel, at run time, dynamically launching kernels onto devices like GPUs or dispatching memory transfers from host to device, or device collectives. The `torch.Tensor` interface allows extensions to PyTorch `Tensor` objects to implement custom dispatch mechanisms: Python objects inheriting from `torch.Tensor` will first pass through `__torch_function__` and `__torch_dispatch__` Python functions for torch "function" and "aten" level operations, respectively. In the standard use case, the dispatch of these operations to a device like the GPU is handled by PyTorch's C++-based dispatcher for optimal performance. `DTensor` implements a custom `__torch_dispatch__` to override this layer, and `ShardTensor` extends this. We specifically allow users to interface with the dispatcher at three locations. At the lowest level, users can implement logic to parallelize `aten` operations, the low-level PyTorch operations. Most DTensor operations are implemented at this level.

At a higher level, `ShardTensor` also allows users to override operations at the `__torch_function__` level, enabling differentiable overrides of PyTorch functions as well as custom named functions through PyTorch's `@custom_op` interface. In fact, by defining such a custom operation, a user is capable of inserting parallelism into their application at whichever depth of complexity they prefer.

Within PhysicsNeMo, where ShardTensor is implemented, many common operations for domain parallelism are implemented. Many operations, such as matrix multiplications and elementwise operations, use a fallback path via `DTensor` in upstream PyTorch. Outputs from the fallback path are promoted to `ShardTensor` before being returned to the user.

The dispatch path and operations can also be seen in Figure 1. We note that, like `DTensor`, this python-based dispatch mechanism carries some additional overhead and for small operations the CPU launch latency can be significant. However, it should be specifically noted that small operations are not the regime that `ShardTensor` has been designed for: we are targeting the highest resolution data and large, compute- and memory-bound operations. Further, ongoing

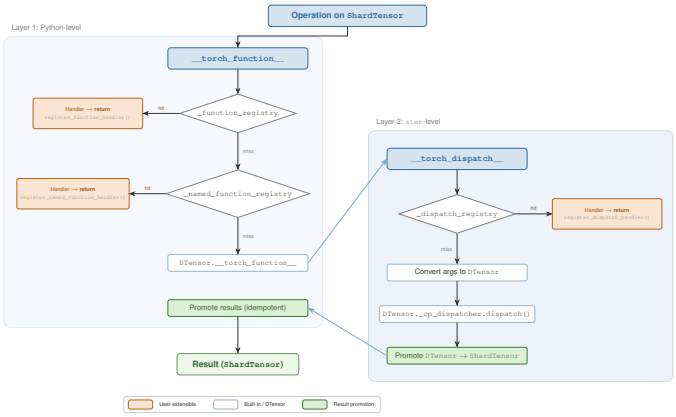

Fig. 1. Dispatch architecture of `ShardTensor`.

work to enable `torch.compile` for static compute graphs will significantly mitigate CPU overheads from the dispatch mechanism.

It should be emphasized that often, in the "Handler" components of Figure 1, collective operations must be dispatched. As an example, a convolution must fetch the adjacent pixels from neighboring devices for numerical consistency, sometimes referred to as a "halo" operation. Alternatively, a normalization layer must aggregate statistics across all ranks to produce global normalizations.

V. Benchmarks and Applications

To validate the framework, we first test performance on single layers and models with synthetic data. All benchmarks and applications are open source and available for reproduction, in the PhysicsNeMo package. All benchmarks and applications were run on Nvidia Blackwell GPUs, installed in an NV72 system, except the StormScope application which used an H100 Cluster instead. Performance benchmarks were run multiple times and the mean latencies are shown.

A. Performance Benchmarks

The `ShardTensor` dispatch model prioritizes flexibility, user friendliness, and performance focused on the usage model it has been designed for: large operations on large data. It is a known limitation that the dispatch and communication overhead of `ShardTensor` on small operations can offset any possible parallelization gains. However, it must be emphasized that small data operations can almost always be parallelized, if necessary, in a more efficient way than via domain decomposition.

In the following sections, we will highlight several benchmarks and applications that we have used to show the performance and benefits of `ShardTensor` and domain parallelism. Performance benchmarks are designed to be reproducible, and applications are also meant to be reproducible but require extra steps of data access and preparation. In all cases, the application programming model follows the same steps:

Algorithm 1 ShardTensor Application Programming Model

1: Initialize PyTorch Distributed Environment with a 1- or 2-D GPU mesh.
2: Load PyTorch model and wrap with `FSDP` along one dimension of the GPU mesh.
3: Load data and promote to `ShardTensor` via collectives along the perpendicular dimension of the mesh, if using a 2-D mesh.
4: Proceed with standard PyTorch syntax as usual.

1) Ring Attention: As a first step in benchmarking `ShardTensor` to understand the scale out performance, we will look at the performance of the standard attention mechanism [60]. The computational complexity of attention has been the subject of much research [61], [62] and here we implement the algorithm "Ring Attention", which naturally enables domain parallelism [54]. Ring attention computes scaled dot product attention locally with K_i, Q_i, V_i, using the optimized flash-attention backend dispatched by PyTorch, and then passes K_i and V_i around the domain in a ring to complete the attention computation. Computation of the current attention block overlaps with message passing of the next K, V tensors, and for numerical stability accumulation of the softmax is performed in log space.

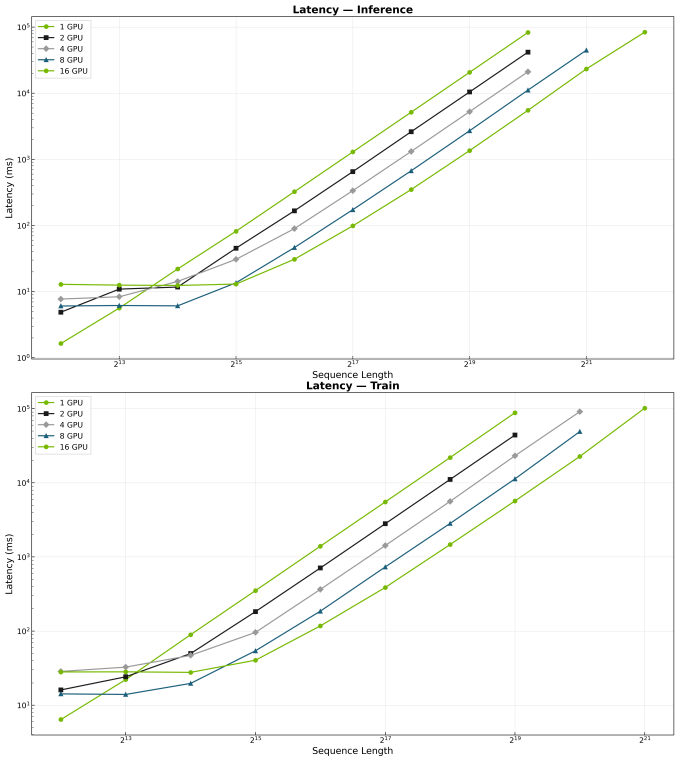

Fig. 2. Ring attention with `ShardTensor`: each device computes full global attention by computing blockwise attention on Q, K, V, while passing K and V around the GPU ring, overlapping computation with communication. Algorithm first published in [54].

As seen in Figure 2, the ring attention layer performance

is poor on many GPUs compared to a single GPU for very small sequence sizes - as expected. After all, there is nearly no benefit to parallelizing such small domains. However, at very large sequence sizes, the scaling becomes nearly linear with GPU count, in both inference and train mode.

2) Vision Transformer: As a more complicated performance benchmark, we next turn our attention to a Vision Transformer model, as popularized in [63]. We use a synthetic data source to perform computational benchmarking and build the model with either 2D or 3D data, using a convolutional tokenizer and 16 layers of standard attention, with approximately 115 million parameters total. The model, though it is using synthetic data, undergoes a synthetic training loop with the AdamW [30] optimizer and FSDP parallelization over the data axis. Since FSDP enables both Data and Model parallelization (over the same axis of GPUs), and Shard Tensor enables domain parallelization (over a perpendicular set of GPUs), this application simulates 2D or 3D parallelism in both training and inference.

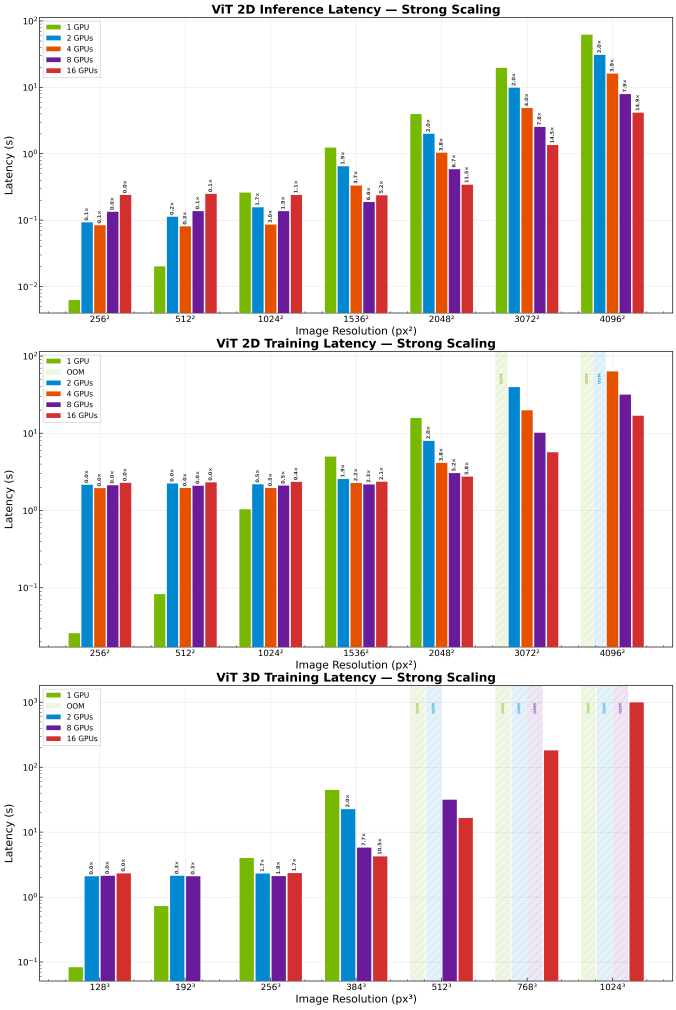

Fig. 3. Latency of the Vision Transformer model for inference on 2D data (top), training on 2D data (middle) and training on 3D data (bottom) as a function of spatial resolution, for varying numbers of GPUs. Each group of bars at fixed resolution represents strong scaling via `ShardTensor`.

Figure 3 shows the latency of the ViT model in 2D and 3D for training and inference, as the data size is increased, for a variety of run sizes. All experiments were performed on NVIDIA GB200 GPUs. Each set of bars at fixed resolution in Figure 3 represents strong scaling of the same problem via `ShardTensor`. At small resolutions, where the image size is not a computational challenge, the strong scaling efficiency is poor: the model is slower in training at 1024^2 resolution. However, at larger sizes, the efficiency improves: 2048^2 is 5x faster at training with 8GPUs than a single GPU, and 15x faster at inference at 4096^2 resolution on 16 GPUs. In 3D, the memory benefits are even more stark: with 16 GPUs, we can train on over 1 billion input points.

The benefits of `ShardTensor` are seen clearly in the memory behavior of the model training. As discussed in Section II, most of the memory usage when training on high resolution data will be from intermediate activations. Indeed, the observed memory usage for 2D data is fit very well by a quadratic function of the spatial resolution, as shown in Figure 4, confirming that intermediate activations dominate. For 3D data, the memory growth follows a cubic relationship as expected. The memory savings achieved by strong scaling the training with `ShardTensor` aligns well with expectations, and even extremely high resolution 3D data is manageable with `ShardTensor` on standard GPU hardware.

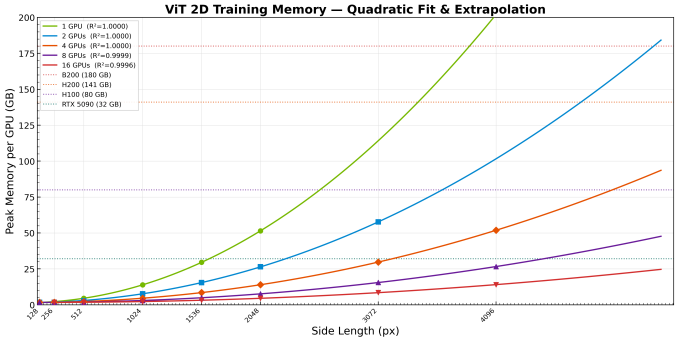

Fig. 4. GPU memory usage during ViT training as a function of spatial resolution for 2D data. Quadratic fits confirm that intermediate activations dominate memory consumption. Strong scaling with `ShardTensor` reduces per-device memory proportionally.

B. Applications

To demonstrate the numerical stability and accuracy of `ShardTensor`, we showcase two applications from industrial use cases that have high-resolution data requirements.

1) Transolver: Transolver [32] is a transformer-like architecture that implements the PhysicsAttention layer to learn physical-state approximations to the attention mechanism, enabling a low rank approximation to standard attention that performs well on physical systems. Transolver++ [39] demonstrated a parallelization strategy that showcased techniques to scale to high resolution input data using methods that are, in effect, domain parallelization. Interestingly, Transolver and `ShardTensor` are both implemented in the PhysicsNeMo

framework. The algorithm described for parallelization in [39] is precisely the path `ShardTensor` takes to parallelize both Transolver and Transolver++, when automatically dispatching collective operations.

For this experiment, we train Transolver on the DrivaerML automotive aerodynamics [64] dataset for 200 epochs, with a minibatch size of 8, and a per-gpu resolution of 200,000 points. We use a Transolver configuration with 8 layers, a hidden dimension of 256, MLP ratio of 2, 512 "slices" in the PhysicsAttention layer, and predict the pressure, velocity, and turbulent velocity properties of the volumetric fields. For the experiments, we increase the domain size by a factor of two per experiment: from 1, to 2, to 4, to 8, for a total of 1.2 million points in the domain.

Figure 5 shows the training and validation performance of Transolver, at a fixed minibatch size of 8, as resolution increases with domain size. We see the training is stable over all resolutions, and the final values for pressure and velocity are competitive with the original Transolver publication [32]. We note that newer models have exceeded the accuracy predictions of these models [37], [38] and some are in progress for domain parallelization; the goal of this study was to demonstrate stability of high resolution training and inference, as compared to standard data-parallel training. Domain-parallel training is both stable and complementary to data parallel training: the 400k, 800k, and 1.2M point resolution runs were all 2D parallelism runs (data parallel + domain parallel).

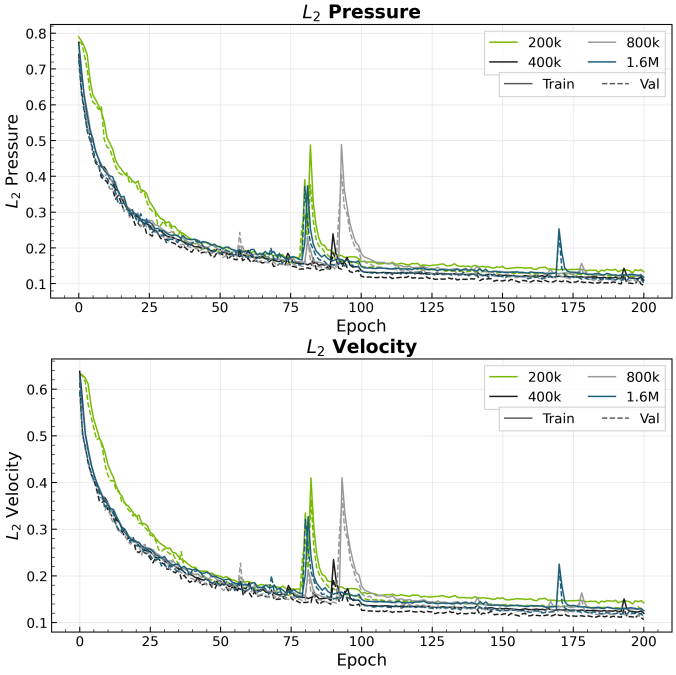

Fig. 5. L2 error for pressure (top) and velocity (bottom) predictions of the Transolver model as domain resolution increases. Domain-parallel training with `ShardTensor` maintains accuracy competitive with the original single-GPU Transolver across all resolutions. All runs are compatible with standard uncertainty at 200 epochs.

One notable component of this application is that the entire preprocessing pipeline, from the data loading all the way to model ingestion, is also parallelized via `ShardTensor`. This enables the entire end-to-end application to scale efficiently, not just the model training.

2) StormScope: Convective storms are among the most impactful weather phenomena, frequently producing heavy precipitation, strong winds, and hail. Individual thunderstorm cells have a spatial extent from a few kilometers to a few tens of kilometers. In order to resolve individual storms, a weather forecasting model needs to have sufficient spatial resolution. Additionally, convective storms involve interactions across many different length scales and are affected by the large-scale environment such as fronts, which can have a spatial extent of several hundred kilometers. Thus, storm-scale models represent processes spanning scales from a few kilometers to hundreds or thousands of kilometers [65]. In practice, this means that the spatial resolution of the model needs to be on the order of a few kilometers and the domain size needs to be on the order of thousands of kilometers. Numerical Weather Prediction (NWP) models address this requirement through varying approaches, including nested approaches that couple coarse-resolution models to fine-resolution regional models (e.g., RAP/HRRR [66]). Beyond weather prediction, for climate projection, achieving km-scale resolution globally for multi-decadal ensemble prediction remains a grand scientific challenge fundamentally limited by the compute demands of spatial resolution [67].

Numerical models with the ability to resolve convection explicitly are known as convection-allowing models (CAMs). These models are operationally used in several countries and often run on rapidly updating forecast cycles. Numerical models have some limitations. They have a long spin-up time, which can be on the order of one to several hours or longer, overlapping with the predictability window of convective events, which can range from minutes to a few hours. They also have limitations related to convective-scale data assimilation which constrains how well the initial condition fed to the forecast model represents the true state of the atmosphere at the initial time.

StormScope [68] is a data-driven AI/ML model that is designed to address some of the limitations of numerical storm-scale models. It operates on a continental-sized domain spanning the contiguous US at 3 km resolution. The model ingests and directly forecasts rapidly updating geostationary satellite imagery and ground-based radar observations, enabling initialization as frequently as every 2–4 minutes with no spin-up time. The high resolution allows the model to resolve the small scale features of convective storms, while the large domain extent preserves the synoptic-scale context that governs storm evolution and structures.

In practice the model processes tensors that have the dimension $(T \times C \times H \times W)$ representing geostationary satellite and radar observations, where T represents the stacked timesteps processed by the model and C represents the channels consisting of different observations at varying sensor wavelengths obtained from the satellite and slices of radar reflectivity

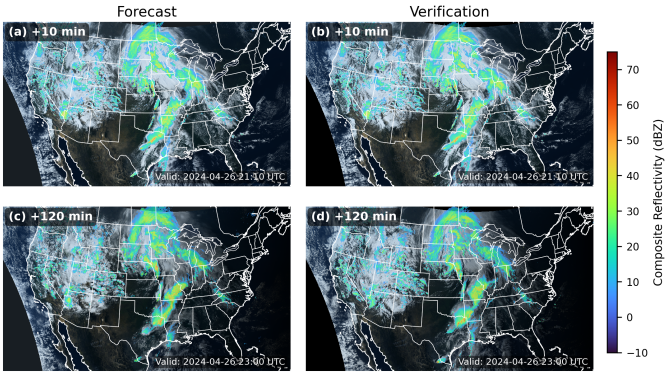

Fig. 6. GOES-16 visible channel composite with MRMS composite reflectivity (dBZ, color shading) overlaid for a forecast initialized at 2024-04-26 21:00 UTC. Left column shows the model forecast; right column shows the corresponding satellite and radar observations (verification). Top row: +10 min lead time; bottom row: +120 min lead time. State boundaries and coastlines are shown in white. The GOES-16 composite is derived from the 0.47, 0.64, and 0.86 μm Advanced Baseline Imager channels.

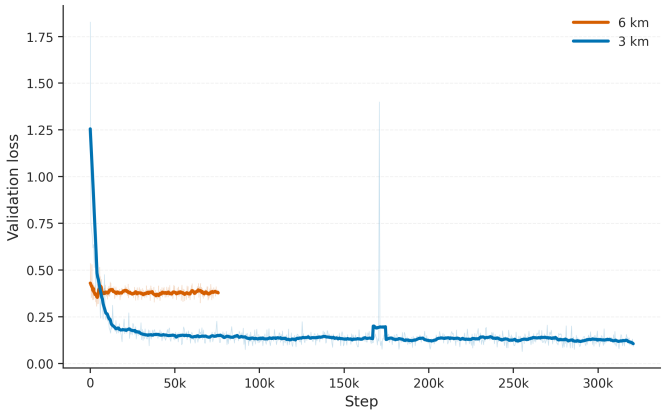

Fig. 7. Validation loss as a function of training step for StormScope, comparing single-GPU 6km resolution runs and `ShardTensor`-distributed training runs at 3km resolution.

composited from a network of ground-based radars across the US. The model processes 8 channels from geostationary satellite observations and two channels from composite radar mosaics representing composite and base radar reflectivity. The model takes in six previous timesteps $[t-50,\ \ t]$ min with a temporal resolution of $\Delta t = 10$ min as input and produces a single timestep at $t+10$ min as output. The model then performs autoregressive inference out to 2 hours. The (H, W) dimensions for the model representing the Continental United States (CONUS) are (1024, 1792) for an effective grid spacing of 3km. The resolution of 3km combined with the large domain size spanning ∼5000km allows the model to learn dynamics of storm evolution across a large range of interacting spatial length scales.

For this experiment, the model is trained on about 300,000 input-output pairs of data from the GOES-16 satellite observations. The model is trained with a denoising diffusion loss following Ref. [69]. The model architecture is based on the Diffusion Transformer [70] with the all-to-all self-attention layers replaced by neighborhood attention (NATTEN [71]) using a neighborhood size of 49. The model has 195 million parameters. We train the model with 32 GPUs by splitting them into 16 data-parallel groups of 2 GPUs each. Within each data-parallel group, we split the activations across 2 GPUs (domain-parallel group) using ShardTensor. The peak memory usage of the model is estimated to be 114GB, beyond the 80GB limit of a single H100 GPU.

Figure 6 shows an example forecast of visible channels from GOES-16 and radar reflectivity using Stormscope with the corresponding verification (ground truth).

As shown in Figure 7, StormScope training at 3 km resolution converges stably and tracks the loss trajectory of the single-GPU 6 km baseline. At 3 km resolution, the CONUS-scale input tensors of shape (1024×1792) per channel exceed the memory capacity of a single GPU during training, making domain parallelism via `ShardTensor` essential. By distributing the spatial dimensions across multiple devices, `ShardTensor` enables StormScope to train at the resolution required to resolve individual convective storms – a capability that was previously inaccessible without sacrificing domain extent or spatial fidelity.

VI. Impact and Conclusions

Domain parallelism, as realized through `ShardTensor`, addresses a bottleneck in scientific machine learning: the inability to train and perform inference on data at the resolution scientists actually need. The results presented in this work demonstrate several concrete impacts and open the door to future developments. The framework to deploy these methods is already in production, used in scientific workloads, and rigorously tested. Across our benchmarks and applications, we observe near-linear strong scaling for ring attention at large sequence lengths, up to 15× inference speedups for a Vision Transformer on 16 GPUs, numerically stable training of Transolver at over one million mesh points, and continental-scale storm forecasting at 3 km resolution that would not fit on a single device.

A. Unblocking Resolution-Limited Workloads

The most immediate impact of `ShardTensor` is the removal of single-GPU memory as a hard ceiling on input resolution. By distributing these activations across a mesh of GPUs, `ShardTensor` converts what was previously an impossibility into a tractable computation. This directly enables scientific domains such as volumetric medical imaging, high-fidelity computational fluid dynamics, and climate modeling to leverage machine learning at resolutions that were previously accessible only to classical numerical solvers.

B. Composability with Existing Parallelism

A key impact of the design philosophy behind `ShardTensor` is its composability with existing parallelism paradigms. As demonstrated in the experiments, domain parallelism operates on an orthogonal mesh axis to data

and model parallelism, enabling 2D and potentially higher-dimensional parallelization strategies. This composability means that scaling scientific ML workloads is no longer a choice between more data, larger models, or higher resolution.

C. Lowering the Barrier to Adoption

By providing a non-invasive programming model, domain parallelism becomes accessible to practitioners who are not distributed systems experts. The extensibility of the dispatch interface further ensures that new layers, custom kernels, and evolving model architectures can be accommodated without redesigning the parallelism strategy from scratch. This stands in contrast to prior bespoke efforts, where parallelization was tightly coupled to a specific model architecture.

D. Limitations

The imperative, layer-by-layer dispatch model that gives `ShardTensor` its flexibility also imposes overhead. Each operation incurs Python-level dispatch latency and, when halo exchanges or other collectives are required, inter-device communication that cannot be amortized across consecutive layers. For small operations or low-resolution data, this overhead can offset parallelization gains; domain parallelism is most beneficial when operations are large and compute- or memory-bound. Scaling efficiency also depends on interconnect bandwidth: hardware with slower interconnects than those benchmarked here will see correspondingly degraded communication performance. Not all PyTorch operations have domain-parallel dispatch paths implemented today; unsupported operations fall back to `DTensor` semantics or require user-written extensions. Finally, `torch.compile` integration is not yet complete, meaning that static-graph optimizations such as kernel fusion and communication/computation overlap across layers are not yet available. More broadly, a framework designed for generality across model architectures and scientific domains cannot simultaneously be optimal for every individual workload.

E. Future Directions

Several avenues remain for further development. First, tighter integration with activation checkpointing and CPU offloading could compound the memory savings of domain parallelism, enabling even deeper models at extreme resolutions. Second, compiler-level optimizations, such as those enabled by `torch.compile`, present an opportunity to reduce the dispatch overhead observed at small problem sizes, broadening the regime in which domain parallelism is beneficial. These optimizations are already underway, though not complete as of this manuscript. It is our hope, as we enter an era of scientific foundation models, that high domain parallelism will become as commonplace in scientific machine learning as data parallelism is today. It is our goal that `ShardTensor` is a step in that direction.

VII. Acknowledgements

On the use of AI Assistants: This paper was written, first and foremost, by humans. AI assistants were used for assisting with latex compilation errors, bibliography errors, spelling and grammar checking, and small miscellaneous tasks. Figure 1 was generated in first draft form via AI. The AI tool used was Claude from Anthropic [72].

References

[1] A. Esteva *et al.*, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, pp. 115–118, 2017. [Online]. Available: https://doi.org/10.1038/nature21056

[2] E. J. Topol, "High-performance medicine: the convergence of human and artificial intelligence," *Nature Medicine*, vol. 25, no. 1, pp. 44–56, 2019. [Online]. Available: https://doi.org/10.1038/s41591-018-0300-7

[3] A. Merchant *et al.*, "Scaling deep learning for materials discovery," *Nature*, vol. 624, no. 7990, pp. 80–85, 2023. [Online]. Available: https://doi.org/10.1038/s41586-023-06735-9

[4] N. J. Szymanski *et al.*, "An autonomous laboratory for the accelerated synthesis of inorganic materials," *Nature*, vol. 624, no. 7990, pp. 86–91, 2023. [Online]. Available: https://doi.org/10.1038/s41586-023-06734-w

[5] M. G. Kapteyn, J. V. R. Pretorius, and K. E. Willcox, "A probabilistic graphical model foundation for enabling predictive digital twins at scale," *Nature Computational Science*, vol. 1, no. 5, pp. 337–347, 2021. [Online]. Available: https://doi.org/10.1038/s43588-021-00069-0

[6] D. Kochkov *et al.*, "Machine learning–accelerated computational fluid dynamics," *Proceedings of the National Academy of Sciences*, vol. 118, no. 21, p. e2101784118, 2021. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.2101784118

[7] S. L. Brunton, B. R. Noack, and P. Koumoutsakos, "Machine learning for fluid mechanics," *Annual Review of Fluid Mechanics*, vol. 52, no. Volume 52, 2020, pp. 477–508, 2020. [Online]. Available: https://www.annualreviews.org/content/journals/10.1146/annurev-fluid-010719-060214

[8] R. Lam *et al.*, "Learning skillful medium-range global weather forecasting," *Science*, vol. 382, no. 6677, pp. 1416–1421, 2023. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.adi2336

[9] K. Bi *et al.*, "Accurate medium-range global weather forecasting with 3d neural networks," *Nature*, vol. 619, no. 7970, pp. 533–538, 2023. [Online]. Available: https://doi.org/10.1038/s41586-023-06185-3

[10] J. Degrave *et al.*, "Magnetic control of tokamak plasmas through deep reinforcement learning," *Nature*, vol. 602, no. 7897, pp. 414–419, 2022. [Online]. Available: https://doi.org/10.1038/s41586-021-04301-9

[11] G. Carleo *et al.*, "Machine learning and the physical sciences," *Rev. Mod. Phys.*, vol. 91, p. 045002, Dec 2019. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.91.045002

[12] A. Karthikeyan and U. D. Priyakumar, "Artificial intelligence: machine learning for chemical sciences," *Journal of Chemical Sciences*, vol. 134, no. 1, p. 2, 2021. [Online]. Available: https://doi.org/10.1007/s12039-021-01995-2

[13] J. Jumper *et al.*, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, no. 7873, pp. 583–589, 2021. [Online]. Available: https://doi.org/10.1038/s41586-021-03819-2

[14] H. Wang *et al.*, "Deep learning enables cross-modality super-resolution in fluorescence microscopy," *Nature Methods*, vol. 16, no. 1, pp. 103–110, 2019. [Online]. Available: https://doi.org/10.1038/s41592-018-0239-0

[15] G. E. Karniadakis *et al.*, "Physics-informed machine learning," *Nature Reviews Physics*, vol. 3, no. 6, pp. 422–440, 2021. [Online]. Available: https://doi.org/10.1038/s42254-021-00314-5

[16] T. E. H. T. Collaboration *et al.*, "First m87 event horizon telescope results. i. the shadow of the supermassive black hole," *The Astrophysical Journal Letters*, vol. 875, no. 1, p. L1, apr 2019. [Online]. Available: https://doi.org/10.3847/2041-8213/ab0ec7

[17] K. M. Yip *et al.*, "Atomic-resolution protein structure determination by cryo-em," *Nature*, vol. 587, no. 7832, pp. 157–161, 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2833-4

[18] A. Shapson-Coe *et al.*, "A petavoxel fragment of human cerebral cortex reconstructed at nanoscale resolution," *Science*, vol. 384, no. 6696, p. eadk4858, 2024. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.adk4858

[19] M. Satoh *et al.*, "Global cloud-resolving models," *Current Climate Change Reports*, vol. 5, no. 3, pp. 172–184, 2019. [Online]. Available: https://doi.org/10.1007/s40641-019-00131-0

[20] C. R. Terai *et al.*, "The impact of resolving subkilometer processes on aerosol-cloud interactions of low-level clouds in global model simulations," *Journal of Advances in Modeling Earth Systems*, vol. 12, no. 11, p. e2020MS002274, 2020, e2020MS002274 10.1029/2020MS002274. [Online]. Available: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2020MS002274

[21] L. Peng *et al.*, "Improving stratocumulus cloud amounts in a 200-m resolution multi-scale modeling framework through tuning of its interior physics," *Journal of Advances in Modeling Earth Systems*, vol. 16, no. 3, p. e2023MS003632, 2024, e2023MS003632 2023MS003632. [Online]. Available: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2023MS003632

[22] H. Segura *et al.*, "nextgems: entering the era of kilometer-scale earth system modeling," *EGUsphere*, vol. 2025, pp. 1–39, 2025. [Online]. Available: https://egusphere.copernicus.org/preprints/2025/egusphere-2025-509/

[23] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998. [Online]. Available: https://doi.org/10.1137/S1064827595287997

[24] C. Farhat and F.-X. Roux, "A method of finite element tearing and interconnecting and its parallel solution algorithm," *International Journal for Numerical Methods in Engineering*, vol. 32, no. 6, pp. 1205–1227, 1991. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.1620320604

[25] A. Toselli and O. Widlund, *Domain Decomposition Methods – Algorithms and Theory*, ser. Springer Series in Computational Mathematics. Springer Science & Business Media, 2005, vol. 34.

[26] Y. Zhao *et al.*, "Pytorch fsdp: Experiences on scaling fully sharded data parallel," 2023. [Online]. Available: https://arxiv.org/abs/2304.11277

[27] PhysicsNeMo Contributors, "NVIDIA PhysicsNeMo: An open-source framework for physics-based deep learning in science and engineering," https://github.com/NVIDIA/physicsnemo, 2023, accessed: 2026. [Online]. Available: https://github.com/NVIDIA/physicsnemo

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017. [Online]. Available: https://arxiv.org/abs/1412.6980

[29] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[30] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019. [Online]. Available: https://arxiv.org/abs/1711.05101

[31] Z. Li *et al.*, "Fourier Neural Operator for Parametric Partial Differential Equations," *arXiv e-prints*, p. arXiv:2010.08895, Oct. 2020.

[32] H. Wu *et al.*, "Transolver: A Fast Transformer Solver for PDEs on General Geometries," *arXiv e-prints*, p. arXiv:2402.02366, Feb. 2024.

[33] R. Ranade *et al.*, "DoMINO: A Decomposable Multi-scale Iterative Neural Operator for Modeling Large Scale Engineering Simulations," *arXiv e-prints*, p. arXiv:2501.13350, Jan. 2025.

[34] D. Kalamkar *et al.*, "A study of bfloat16 for deep learning training," 2019. [Online]. Available: https://arxiv.org/abs/1905.12322

[35] P. Micikevicius *et al.*, "Fp8 formats for deep learning," 2022. [Online]. Available: https://arxiv.org/abs/2209.05433

[36] X. Sun *et al.*, "Ultra-low precision 4-bit training of deep neural networks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[37] C. Adams *et al.*, "GeoTransolver: Learning Physics on Irregular Domains Using Multi-scale Geometry Aware Physics Attention Transformer," *arXiv e-prints*, p. arXiv:2512.20399, Dec. 2025.

[38] B. Alkin *et al.*, "AB-UPT: Scaling Neural CFD Surrogates for High-Fidelity Automotive Aerodynamics Simulations via Anchored-Branched Universal Physics Transformers," *arXiv e-prints*, p. arXiv:2502.09692, Feb. 2025.

[39] H. Luo *et al.*, "Transolver++: An Accurate Neural Solver for PDEs on Million-Scale Geometries," *arXiv e-prints*, p. arXiv:2502.02414, Feb. 2025.

[40] H. Zhou *et al.*, "Transolver-3: Scaling Up Transformer Solvers to Industrial-Scale Geometries," *arXiv e-prints*, p. arXiv:2602.04940, Feb. 2026.

[41] B. Graham and L. van der Maaten, "Submanifold sparse convolutional networks," *arXiv preprint arXiv:1706.01307*, 2017.

[42] C. Choy, J. Gwak, and S. Savarese, "4d spatio-temporal convnets: Minkowski convolutional neural networks," 2019. [Online]. Available: https://arxiv.org/abs/1904.08755

[43] C. Choy *et al.*, "Factorized implicit global convolution for automotive computational fluid dynamics prediction," 2025. [Online]. Available: https://arxiv.org/abs/2502.04317

[44] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[45] S. Li *et al.*, "PyTorch Distributed: Experiences on Accelerating Data Parallel Training," *arXiv e-prints*, p. arXiv:2006.15704, Jun. 2020.

[46] Y. You, I. Gitman, and B. Ginsburg, "Large Batch Training of Convolutional Networks," *arXiv e-prints*, p. arXiv:1708.03888, Aug. 2017.

[47] Y. You *et al.*, "Large Batch Optimization for Deep Learning: Training BERT in 76 minutes," *arXiv e-prints*, p. arXiv:1904.00962, Apr. 2019.

[48] Y. You *et al.*, "ImageNet Training in Minutes," *arXiv e-prints*, p. arXiv:1709.05011, Sep. 2017.

[49] M. Shoeybi *et al.*, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020. [Online]. Available: https://arxiv.org/abs/1909.08053

[50] R. Y. Aminabadi *et al.*, "Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale," 2022. [Online]. Available: https://arxiv.org/abs/2207.00032

[51] PyTorch Contributors, "PyTorch DTensor: Distributed tensor primitives for SPMD distributed training," https://github.com/pytorch/pytorch/issues/88838, 2022, RFC for PyTorch DistributedTensor. Accessed: 2026. [Online]. Available: https://github.com/pytorch/pytorch/issues/88838

[52] Y. Huang *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," 2019. [Online]. Available: https://arxiv.org/abs/1811.06965

[53] D. Narayanan *et al.*, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–15. [Online]. Available: https://doi.org/10.1145/3341301.3359646

[54] H. Liu, M. Zaharia, and P. Abbeel, "Ring attention with blockwise transformers for near-infinite context," *arXiv preprint arXiv:2310.01889*, 2023.

[55] B. Bonev *et al.*, "Fourcastnet 3: A geometric approach to probabilistic machine-learning weather forecasting at scale," 2025. [Online]. Available: https://arxiv.org/abs/2507.12144

[56] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[57] J. Bradbury *et al.*, "JAX: composable transformations of Python+NumPy programs," http://github.com/jax-ml/jax, 2018, version 0.3.13.

[58] Y. Ma *et al.*, "Paddlepaddle: An open-source deep learning platform from industrial practice," *Frontiers of Data and Domputing*, vol. 1, no. 1, p. 105, 2019. [Online]. Available: http://www.jfdc.cnic.cn/EN/abstract/article_2.shtml

[59] J. Ansel *et al.*, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 929–947. [Online]. Available: https://doi.org/10.1145/3620665.3640366

[60] A. Vaswani *et al.*, "Attention Is All You Need," *arXiv e-prints*, p. arXiv:1706.03762, Jun. 2017.

[61] T. Dao *et al.*, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[62] T. Dao, "FlashAttention-2: Faster attention with better parallelism and work partitioning," in *International Conference on Learning Representations (ICLR)*, 2024.

[63] A. Dosovitskiy *et al.*, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *arXiv e-prints*, p. arXiv:2010.11929, Oct. 2020.

[64] N. Ashton *et al.*, "DrivAerML: High-Fidelity Computational Fluid Dynamics Dataset for Road-Car External Aerodynamics," *arXiv e-prints*, p. arXiv:2408.11969, Aug. 2024.

[65] P. Markowski and Y. Richardson, *Mesoscale meteorology in midlatitudes*. John Wiley & Sons, 2011.

[66] D. C. Dowell *et al.*, "The high-resolution rapid refresh (hrrr): An hourly updating convection-allowing forecast model. part i: Motivation and system description," *Weather and Forecasting*, vol. 37, no. 8, pp. 1371–1395, 2022.

[67] H. Segura *et al.*, "nextgems: entering the era of kilometer-scale earth system modeling," *EGUsphere*, vol. 2025, pp. 1–39, 2025. [Online]. Available: https://egusphere.copernicus.org/preprints/2025/egusphere-2025-509/

[68] J. Pathak *et al.*, "Learning accurate storm-scale evolution from observations," *arXiv preprint arXiv:2601.17268*, 2026.

[69] T. Karras *et al.*, "Elucidating the design space of diffusion-based generative models," *Advances in neural information processing systems*, vol. 35, pp. 26 565–26 577, 2022.

[70] W. Peebles and S. Xie, "Scalable diffusion models with transformers," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2023, pp. 4195–4205.

[71] A. Hassani *et al.*, "Neighborhood attention transformer," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

[72] Anthropic, "Claude," https://www.anthropic.com/claude, 2025, aI assistant. Accessed: 2026.