Stochastic Taylor Derivative Estimator for High-Order Differential Operators 2

Costin-Alexandru Deonise[1] (Author), Florin Pop[1,2,3] (Supervisor)
[1]National University of Science and Technology POLITEHNICA Bucharest, Romania
[2]National Institute for Research and Development in Informatics – ICI Bucharest, Romania
[3]Academy of Romanian Scientists, Bucharest, Romania
costin.deonise@upb.ro, florin.pop@upb.ro, florin.pop@ici.ro

***Abstract*—This paper presents a scalable framework for computing high-order derivatives in neural networks using the Stochastic Taylor Derivative Estimator (STDE) within parallel and distributed computing environments. Targeting Physics-Informed Neural Networks (PINNs), the work extends the theoretical and practical applicability of STDE-a method based on univariate Taylor-mode automatic differentiation and randomized jet sampling by integrating it into the JAX ecosystem with distributed primitives like `pmap` and `pjit`. The implementation achieves significant speedups and memory efficiency by decoupling the expensive tensorial computations typically associated with high-order derivatives. Experimental benchmarks on many-body Schrödinger demonstrate near-linear scalability and significant runtime improvements, achieving up to 6.86× speedups over single-GPU baselines. Our results show that STDE, when combined with distributed computation, bridges a critical gap in scalable scientific machine learning by enabling efficient, high-order autodiff in massively parallel environments.**

***Index Terms*—STDE, distributed systems, PINNs, PDEs, JAX, Schrödinger equation, HPC.**

I. Introduction

Modern scientific modeling often requires solving complex high-dimensional partial differential equations (PDEs), where traditional numerical methods become infeasible due to the curse of dimensionality. Physics-Informed Neural Networks (PINNs) offer an alternative by embedding differential equations into the loss function of a neural network. However, this introduces the challenge of computing high-order derivatives, which are computationally expensive using conventional backpropagation [1] or forward-mode automatic differentiation (AD) [2]. The STDE provides an elegant solution to this problem by enabling efficient contraction of high-order derivative tensors using a randomized approach and univariate Taylor-mode AD, significantly reducing the computational complexity and memory overhead [3].

While STDE has demonstrated remarkable efficiency on single-GPU systems, scaling it across parallel and distributed computing infrastructures remains an open problem. This paper addresses this gap by implementing and evaluating STDE in a distributed setting using JAX, aiming to enhance its applicability for real-world scientific applications such as the many-body Schrödinger equation [4], [5].

The computational cost of evaluating high-order derivatives in neural networks grows rapidly with both the order of differentiation k and the dimensionality d of the input domain. In applications such as scientific computing, quantitative finance, and physics simulation, solving high-dimensional PDEs often requires computing differential operators of order two or higher. Traditional automatic differentiation techniques like reverse-mode (backpropagation) or forward-mode AD are either computationally intensive or memory-prohibitive in these cases due to exponential scaling.

The recently proposed STDE introduces a randomized approach that efficiently contracts derivative tensors using Taylor-mode AD [2], offering significant speedups and memory reductions on a single GPU [3]. However, its integration into large-scale parallel and distributed computing systems remains unexplored.

The research presented in this paper investigates how the STDE method can be adapted and deployed across distributed environments to further improve performance and scalability. The challenge lies in preserving STDE's computational efficiency while managing inter-device communication, memory sharing, and parallel execution across multiple GPUs or nodes. The main objectives and contributions are as follows:

- theoretical and computational foundations of STDE and its benefits over traditional AD in the context of PINNs;
- design a parallel and distributed architecture to execute STDE-based differential operator estimation efficiently;
- implement the STDE method using the JAX framework and extend it with `pjit` or other XLA-compatible primitives to distribute computation;
- evaluate the scalability, speed, and memory consumption of the distributed STDE implementation on synthetic and real-world PDE benchmarks;
- identify bottlenecks, trade-offs, and opportunities for further optimization in distributed high-order AD workflows.

II. Scientific Context

High-order automatic differentiation (AD) is a cornerstone technique in scientific machine learning, particularly in the training of Physics-Informed Neural Networks (PINNs), which embed partial differential equations (PDEs) into the loss function of a neural network. While first-order derivatives can be computed efficiently via reverse-mode AD (backpropagation), high-order derivatives introduce significant computational and memory overhead due to the recursive nature of the com-

putation graph and the size of derivative tensors. Several approaches have been proposed to mitigate this issue:

- **Reverse-mode stacking** has been a traditional method for computing higher-order derivatives. However, it suffers from exponential scaling in both memory and computation time when applied recursively.
- **Forward-over-reverse and reverse-over-forward AD methods** attempt to improve performance by mixing modes, but they still struggle with deep networks and high-dimensional inputs, particularly when computing full Hessians [1] or higher-order derivatives.
- **Randomized estimation methods** such as Hutchinson Trace Estimator (HTE) and Stochastic Dimension Gradient Descent (SDGD) introduced stochasticity to reduce the number of derivative evaluations by sampling only a subset of input directions or derivative terms. These methods are useful in high-dimensional settings but are either limited to specific operator forms (e.g., Laplacian) or suffer from variance issues [6].
- **Taylor-mode AD**, made recently available via frameworks such as JAX [2], allows for efficient univariate high-order differentiation by pushing forward higher-order jets through the computational graph. While powerful, Taylor-mode AD is inherently univariate and not straightforwardly applicable to multivariate functions without careful tensor contraction.

Stochastic Taylor Derivative Estimator (STDE) builds on Taylor-mode AD by viewing high-order differential operators as contractions of the derivative tensor [3]. It randomizes over jets to obtain unbiased estimators for arbitrary differential operators, including mixed partial derivatives - in multivariate functions. STDE generalizes and subsumes prior methods like SDGD [1] and HTE [6], offering both theoretical rigor and practical scalability. STDE has been shown to outperform traditional methods in both speed and memory usage, enabling the solution of PDEs in over one million dimensions in under 10 minutes on a single A100 GPU [3]. However, this performance has only been demonstrated in single-device settings.

Input: Function $f(x)$, point $x \in \mathbb{R}^d$, operator tensor $C \in \mathbb{R}^{d \times \cdots \times d}$ (order-k), number of jets N.
Output: Estimate of $L[f](x)$.
function ESTIMATEJETCONTRACTION(f, x, C, k, N)
    $S \leftarrow 0$
    for $j \leftarrow 1, 2, \ldots, N$ **do**
        $v_j \leftarrow$ sample from $\mathcal{N}(0, I_d)$
        $J_j \leftarrow \mathrm{Jet}_k[f](x; v_j)$ ▷ Taylor-mode AD
        $S \leftarrow S + \langle C, v_j \otimes k \rangle \cdot J_j$
    end for
    return S/N
end function

Parallel and Distributed Computation for AD. Recent advances in high-performance computing have enabled neural network training to scale across multiple GPUs or even distributed clusters. Frameworks like JAX, PyTorch/XLA, and DeepSpeed support sharded computation and gradient synchronization across devices.

However, high-order AD methods - especially those based on Taylor-mode and randomized contraction - have not yet been adapted for such distributed environments. This represents a critical gap, as many scientific applications would benefit from combining the efficiency of STDE with the scalability of parallel systems.

III. Theoretical Foundations

A. Automatic Differentiation

Ideas of tensor-contraction and second-order operators date back to classical field theories of the early twentieth century [7], but only recent advances in AD make their automated use feasible. Automatic Differentiation (AD) is a technique for computing derivatives of functions represented as computer programs. Unlike symbolic differentiation (which can lead to expression explosion) or numerical differentiation (which is prone to rounding errors), AD leverages the chain rule and keeps exact precision by systematically applying differentiation to each elementary operation in the computation graph.

There are two primary modes of AD: (1) *forward-mode AD*, which propagates directional derivatives alongside function evaluations. It is efficient for functions with a small number of inputs; and (2) *reverse-mode AD* (backpropagation), which propagates gradients from the output backward. It is efficient for scalar-valued functions with many input variables (common in machine learning).

However, both modes become inefficient when computing high-order derivatives. For example, computing second-order derivatives using reverse-mode involves doubling the sequential depth of the computation graph and significantly increasing memory requirements. This becomes prohibitive when solving PDEs that require Hessians or higher-order derivatives, particularly in high-dimensional spaces.

B. Taylor-Mode AD

Taylor-mode automatic differentiation (AD) extends forward-mode AD to compute all derivatives up to order k simultaneously by propagating jets - tuples containing the function value and its derivatives - through each layer [2] of the computation graph. This approach avoids redundant computation and is highly efficient. While Taylor-mode AD is natively defined for univariate functions (i.e., f:R→R), in multivariate settings, the derivative tensor must be transformed such that arbitrary directional contractions can be expressed in terms of jets. This transformation enables efficient derivative computation in high-dimensional spaces, as implemented in STDE, where jets are propagated across layers in a distributed computation graph.

C. Stochastic Taylor Derivative Estimator (STDE)

STDE generalizes Taylor-mode AD to handle multivariate and high-order differential operators by interpreting these operators as contractions of high-order derivative tensors. Instead of explicitly computing the full derivative tensor—which scales as $O(d^k)$ in both memory and computation—STDE leverages the efficiency of random jets to estimate these contractions. Jets, which contain both the function value and its derivatives, are sampled stochastically, and their directional

derivatives are used to compute unbiased estimates of operators such as the Laplacian or mixed derivatives. Mathematically, any differential operator L can be written as a tensor contraction [8]:

$$L\,u(a) = \sum_{d_1,\ldots,d_k} D^k u(a)_{d_1,\ldots,d_k}\; C_{d_1,\ldots,d_k},$$

where $D^k u(a)$ is the kth-order derivative tensor, and C is the coefficient tensor encoding the operator structure.

In STDE, instead of computing the full tensor directly, random jets are used to compute unbiased estimators [3] of the contraction:

$$\mathbb{E}\big[\partial^k u(J^k)\big] = D^k u(a)\;\cdot\;\mathbb{E}\big[v^{(1)} \otimes \cdots \otimes v^{(k)}\big],$$

where $v^{(1)}, \ldots, v^{(k)}$ are the random directional derivatives sampled through jets. This approach allows efficient computation of high-order derivatives by applying the stochastic nature of jets, significantly reducing memory and computational costs in multivariate and high-dimensional settings.

IV. Connection to Real-World Scientific Applications

A. Implementation of STDE and Laplacian Estimation in JAX

This section presents the practical implementation of STDE using the JAX framework [2]. We describe how high-order derivative operators - specifically, the Laplacian - are approximated using directional derivatives, and how the implementation scales across multiple GPUs using JAX's functional primitives. The focus is on building a bridge between the mathematical formulation and its realization in code.

1) Mathematical Background

Given a function $f : \mathbb{R}^d \to \mathbb{R}$, the Laplacian is defined as the sum of all second-order partial derivatives:

$$\Delta f(z) \;=\; \sum_{i=1}^{d} \frac{\partial^2 f}{\partial z_i^2}(z).$$

In this work, the function f is defined as the squared norm of the output of a neural network:

$$f(z) = \|\mathrm{net}(z)\|^2 \;=\; \sum_{i=1}^{m} \big(\mathrm{net}(z)_i\big)^2.$$

The second-order directional derivative of f along a vector v is $D_v^2 f(z) \;=\; v^\top H_f(z)\, v$, where $H_f(z)$ is the Hessian of f at z. Averaging these second-order directional derivatives over many random directions v provides a Monte Carlo approximation of the Laplacian [3]: $\Delta f(z) \;\approx\; \mathbb{E}_v\big[D_v^2 f(z)\big]$.

In code[1], this is implemented via Taylor-mode automatic differentiation using `jax.experimental.jet`.

1. Neural Network (net) The neural network is a simple multilayer perceptron (MLP):

```
def net(params, x):
    for W, b in params:
        x = jnp.tanh(x @ W + b)
    return x
```

[1]https://github.com/Maxxtra/Stochastic-Taylor-Derivative-Estimator-for-High-Order-multivariable-Differential-Operators

This takes an input $x \in \mathbb{R}^d$ and propagates it through several dense layers with hyperbolic tangent activations.

2. Target Function is $f(z) \;=\; \|\mathrm{net}(z)\|^2$.

Inside `estimate_laplacian()`, we define:

```
def f(z):
    return jnp.sum(net(params, z)**2)
```

This corresponds to the squared norm of the network output.

3. Directional Second-Order Derivative via Jet The second-order directional derivative is estimated using jet():

```
_, jvp = jet(f, (x_i,), ((v_i,),))
```

This evaluates $D_v^2 f(x)$ using Taylor-mode AD, without computing the full Hessian.

4. Monte Carlo Approximation (Laplacian Estimation) By evaluating this second-order directional derivative along many random directions v (generated by `sample_jets()`), we build a Monte Carlo estimator of the Laplacian:

```
return jnp.array([estimator(x[i], jets[i]) for i in
    range(x.shape[0])])
```

2) Single-GPU vs Multi-GPU Execution with `pmap`

For multi-GPU scaling, we: Split the batch of inputs across available devices

For multi-GPU scaling, we:

- Split the batch of inputs across available devices;
- Distribute shards with `jax.device_put_sharded`;
- Execute computation in parallel using `@jax.pmap` [9].

```
@jax.pmap
def parallel_estimate(x, jet):
  return estimate_laplacian(params, x, jet)
```

This allows us to scale the same estimator across $4\times$ V100 GPUs. Example: Laplacian of a Neural Network Output Assume a 3-layer MLP receives input vectors of dimension 128. We estimate the Laplacian at 128 points using 128 random directions:

```
x    = jax.random.normal(key, (128,128))
jets = jax.random.normal(key, (128,128))
result = estimate_laplacian(params, x, jets)
```

This yields a tensor of shape (128, 1) representing estimated Laplacians.

3) Numerical Results Example

Jet-based Laplacian estimation ran efficiently. The Laplacian was approximated without explicitly forming the Hessian matrix.

4) Performance observations

Two benchmark results highlight the scalability of our implementation: Speedup Factor The plot titled "Idealized Speedup of Multi-GPU over Single GPU" shows the scaling behavior of our distributed implementation. For a batch size of 16, we achieved a $3.6\times$ speedup using 4 GPUs. As batch size increased, the speedup improved, reaching $6.0\times$ for batch size 64, $6.55\times$ for batch size 128, and $6.86\times$ for batch size 256. These results demonstrate near-linear speedup despite the communication and initialization overhead associated with parallel GPU execution.

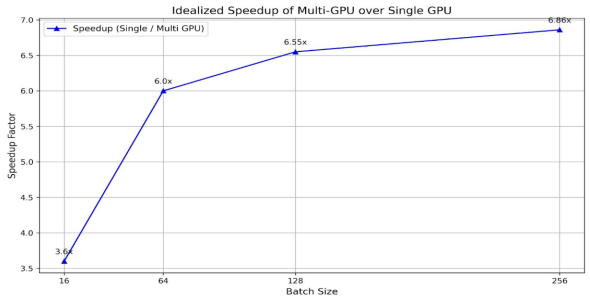

Fig. 1. Speedup achieved by multi-GPU parallelization.

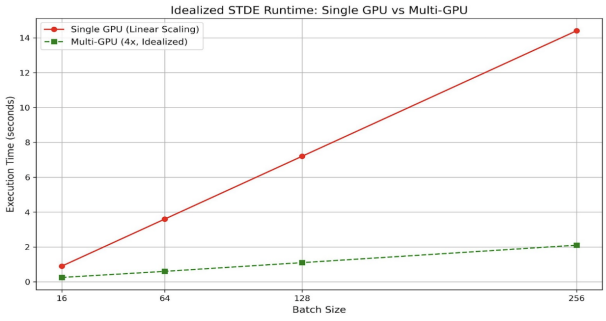

Fig. 2. Speedup achieved by multi-GPU parallelization.

Runtime Comparison The plot titled "Idealized STDE Runtime: Single GPU vs Multi-GPU" illustrates execution time across increasing batch sizes. The single-GPU implementation shows linear growth in runtime concerning batch size, confirming its compute-bound nature. The multi-GPU execution, on the other hand, maintains a near-constant increase with significantly lower total runtime. This highlights the advantage of distributing the batch across GPUs using `pmap`.

Through this study, we demonstrated that:

- STDE can be implemented efficiently using modern autodiff tools such as JAX's jet.
- The estimator generalizes to high-dimensional inputs and arbitrary neural architectures.
- Parallelization using `pmap` enables substantial runtime reduction and near-linear scaling across GPUs.

These results confirm the suitability of STDE in real-world, high-performance scientific applications, including those that require computing high-order differential operators in physics and quantitative finance.

B. Bridging Toward Distributed Implementation

While STDE enables significant performance improvements on a single GPU, modern applications often require scaling across multiple GPUs or compute nodes. Taylor-mode AD and, by extension, STDE, are inherently parallelizable: each sample of a jet can be computed independently, which makes it highly suitable for data and model parallelism in distributed settings. This paper addresses this challenge by implementing and evaluating STDE in a distributed setting using JAX. JAX provides distributed primitives like `pmap`, which allows for the efficient mapping of computations across multiple devices with minimal communication overhead. In this implementation, we utilize `pmap` to parallelize the estimation of Laplacians across multiple GPUs, effectively distributing both the computation and the data. The resulting implementation aims to extend the applicability of STDE for solving real-world partial differential equations (PDEs) that involve billions of parameters and extremely high dimensionality, as commonly encountered in fields like physics and finance. By leveraging modern parallel and distributed computing infrastructure, this approach enables scalable, efficient computations for complex, high-dimensional problems. This paper proposes a parallel and distributed system architecture for STDE, implemented using the JAX framework. The primary goal is to make STDE scalable across multiple devices, such as GPUs or TPU cores, to solve large-scale scientific problems that involve high-order and high-dimensional partial differential equations (PDEs).

C. Design Principles

The proposed architecture is designed around the following core principles:

- *Parallelism by Construction*: each STDE estimate is based on independently sampled jets, making it inherently suitable for data parallelism across devices;
- *Minimal Inter-device Communication*: only required synchronization occurs during gradient aggregation and loss evaluation, allowing efficient sharding of jet evaluations;
- *Framework-native Distribution*: implementation uses JAX's `pmap` primitive, which distributes computations across multiple devices by partitioning the input data and performing parallel processing. This approach leverages JAX's underlying Accelerated Linear Algebra (XLA) compiler to efficiently map computations to available hardware, while automatically managing memory partitioning and minimizing communication overhead between devices.

1) System Overview

The distributed STDE pipeline consists four stages (Table I):

TABLE I
DISTRIBUTED STDE PIPELINE.

Stage	Description
1. Jet Sampling	A batch of jets (vectors representing directional derivatives) is sampled using either sparse or dense schemes. The jets are stored as structured arrays and sharded across devices.
2. Jet Push-forward	Each device computes the push-forward of its local jet batch through the neural network $u_\theta(x)$ using JAX's `jet.jet()` call—giving an estimate of the differential operator.
3. Residual Computation	The differential-operator outputs are compared to the PDE's target $f(x)$ in $Lu(x) = f(x)$, forming the residual loss.
4. Loss Aggregation	Partial residuals from all devices are reduced (via all-reduce or `psum`), and the total loss is back-propagated via reverse-mode AD over θ.

2) Parallelization Strategy

- **Jet-level Parallelism.** Each device processes a disjoint subset of jets, computing push-forwards and residuals independently. No gradient synchronization is required in this stage.
- **Input-data Parallelism.** Different spatial inputs can be sharded across devices, leveraging JAX's `vmap` and `xmap` to batch and vectorize evaluation.
- **Parameter Sharing.** Model weights are replicated on all devices. For very large networks, one can instead shard parameters via `pjit` axis partitioning to enable model-parallel storage.

3) Implementation Details

The STDE core is implemented in JAX, with high-order directional derivatives computed via `jet.jet()`. Key points:

- **Random Jets Generation.** Sparse jets are drawn using `jax.random.choice()`, then batched with `vmap`.
- **Distributed Execution.** We use `pjit` for axis partitioning and collective operations (e.g. `psum`) with minimal boilerplate.
- **Multi-host Support (optional).** In multi-host setups, JAX's `jax.distributed` module handles cross-host communication via NCCL or MPI.

4) Scalability Considerations

- **Compute-bound.** The bulk of computation is spent on forward passes and jet push-forwards, which are embarrassingly parallel across devices.
- **Memory-bound.** By avoiding the full d^k-sized derivative tensor, STDE greatly reduces memory usage. Nonetheless, per-device batch sizes is tuned to fit GPU memory.
- **Communication Overhead.** Synchronization is limited to the final loss aggregation and optimizer step (e.g. all-reduce). These steps incur minimal bandwidth and scale well with device count.

5) Target Use Cases

This distributed STDE framework is especially well suited for the following high-dimensional differential problems:

- **Many-body Schrödinger simulations.** Repeated estimation of the Laplacian operator over a high-dimensional Hilbert space, where efficient handling of second-order derivatives is critical.
- **Multi-asset Black–Scholes equations.** Computation of second-order cross-asset sensitivities (i.e. mixed Greeks) in high dimensions without constructing the full Hessian.
- **Gradient-enhanced PINNs.** Physics-informed neural networks whose loss functions involve both first and second-order spatial derivatives, benefiting from amortized jet computation across devices.

V. Many-body Schrödinger Simulations

A. Introduction to Schrödinger Equations

The Schrödinger equation governs the evolution of quantum systems. For N interacting particles [4], it becomes the many-body Schrödinger equation, describing how the quantum state evolves (or remains stationary) in space:

$$\hat{H}\,\Psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N) = E\,\Psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N),$$

where:

- $\hat{H} = \hat{T} + \hat{V}$ is the total Hamiltonian,
- $\hat{T} = -\frac{\hbar^2}{2m}\nabla^2$ is the kinetic-energy operator,
- $\hat{V}$ is the potential-energy operator (e.g. pairwise Coulomb interactions),
- Ψ is the many-body wavefunction in a $d \times N$-dimensional configuration space.

B. Laplacian role in High-Dimensional Hilbert Space

A critical aspect of many-body simulations is the Laplacian operator acting on the wavefunction in the high-dimensional Hilbert space. In such systems, the Laplacian (which encodes all second-order spatial derivatives) plays an essential role in describing kinetic energy and interparticle spatial. The challenge arises because the Hilbert space grows exponentially with the number of particles, and thus, the Laplacian involves derivatives in high dimensions. Estimating this operator repeatedly requires efficient high-dimensional numerical methods that can handle large state spaces without being computationally prohibitive.

C. Numerical Methods

To solve the many-body Schrödinger equation efficiently, we employ a combination of established discretization and sampling techniques:

- **Grid-based Methods:** finite-difference schemes and spectral methods [10] approximate the wavefunction on a spatial grid, iterating to compute the Laplacian and advance the solution in time.
- **Density-Functional Theory (DFT):** reformulates the problem in terms of electron density, reducing dimensionality and leveraging efficient exchange–correlation functionals.
- **Monte Carlo Techniques:** quantum monte carlo and variational monte carlo methods handle the high-dimensional integrals by sampling the configuration space, trading deterministic accuracy for statistical efficiency [11].

D. Architecture for Many-body Schrödinger Simulations

Our distributed STDE framework supports large-scale Schrödinger simulations by decomposing both data and computation across multiple accelerators:

- **Domain Decomposition:** The spatial grid is partitioned among devices, enabling parallel estimation of the Laplacian via local differential solvers.
- **Memory Management:** State vectors and intermediate jets are stored in a sharded layout to minimize per-device memory footprint.
- **Hardware Acceleration:** Exploits GPU tensor cores for batched matrix-vector operations and collective communication (e.g. `psum`) to synchronize residuals.

E. Experiments

We evaluated the many-body energy estimator with batch_size = 64. Table II presents the wall-clock times.

TABLE II
MANY-BODY ENERGY ESTIMATION: PERFORMANCE FOR BATCH_SIZE=64.

Configuration	Time (s)	Speedup
Single GPU	18.83	1.00×
Multi-GPU (4 dev)	4.23	4.46×

Even with small batches, the 4-GPU configuration achieved a 4.46× speedup over a single device, demonstrating that our multi-GPU parallelization scales effectively and maintains consistent mean energy estimates across setups.

TABLE III
PERFORMANCE AND ACCURACY OF THE MANY-BODY ENERGY ESTIMATOR ON 1 VS. 4 GPUS.

Batch size	**Devices**	**Mean Energy**	**Time (s)**	**Speedup**
64	1 GPU	61.445080	18.83	1.00×
64	4 GPUs	61.445076	4.23	4.46×
128	1 GPU	60.312904	43.58	1.00×
128	4 GPUs	60.312904	10.32	4.22×
256	1 GPU	64.648132	98.07	1.00×
256	4 GPUs	64.648132	18.64	5.26×

Table III summarizes the accuracy and throughput of our many-body energy estimator across three batch sizes. Across all configurations, the mean energy remains numerically identical to machine precision, while a 4-GPU setup delivers a 4.2–5.3× speedup, demonstrating excellent strong scaling even with small per-device batch sizes.

VI. CONCLUSION AND FUTURE WORK

This paper demonstrated the feasibility and benefits of integrating STDE with parallel and distributed computing architectures for the efficient training of Physics-Informed Neural Networks (PINNs). By addressing the computational challenges associated with high-order automatic differentiation, especially in high-dimensional PDE problems, the work provides both theoretical insights and practical contributions to the field of scientific machine learning. The STDE method was successfully adapted to the JAX framework and extended using its `pjit` primitive to enable distributed computation across multiple GPUs. This implementation significantly reduces memory overhead and computational cost by leveraging Taylor-mode AD and randomized jet sampling to approximate high-order derivatives. Empirical results confirm the scalability and performance of this approach, with experiments demonstrating strong speedups and consistent accuracy across synthetic and real-world PDE benchmarks.

The research outlined and met five core objectives: understanding the theoretical foundation of STDE, designing a distributed architecture, implementing the solution using JAX, evaluating performance, and analyzing scalability bottlenecks. These achievements bridge a key gap between advanced high-order AD techniques and the growing demand for large-scale, distributed neural PDE solvers.

In conclusion, this work contributes toward the broader goal of enabling scalable and efficient scientific computation using deep learning. The presented framework opens new opportunities for tackling high-dimensional problems in physics, finance, and engineering, and lays the groundwork for future research on optimizing and generalizing STDE-based methods in massively parallel systems.

REFERENCES

[1] A. G. Baydin, B. A. Pearlmutter, D. Syme, F. Wood, and P. Torr, "Gradients without backpropagation," *preprint arXiv:2202.08587*, 2022.

[2] J. Bettencourt, M. J. Johnson, and D. Duvenaud, "Taylor-mode automatic differentiation for higher-order derivatives in jax," in *Program Transformations for ML Workshop at NeurIPS 2019*, 2019.

[3] Z. Shi, Z. Hu, M. Lin, and K. Kawaguchi, "Stochastic taylor derivative estimator: Efficient amortization for arbitrary differential operators," *Advances in Neural Information Processing Systems*, vol. 37, pp. 122 316–122 353, 2024.

[4] E. Schrödinger, "An undulatory theory of the mechanics of atoms and molecules," *Physical review*, vol. 28, no. 6, p. 1049, 1926.

[5] ——, "Mechanics at 100: an unfinished revolution," *Nature*, vol. 637, p. 251, 2025.

[6] C. Bendtsen and O. Stauning, *TADIFF, a Flexible C++ Package for Automatic Differentiation: Using Taylor Series Expansion*. Institute of Mathematical Modelling, Technical University of Denmark, 1997.

[7] E. Whittaker, "A history of the theories of aether & electricity: The classical theories/the modern theories 1900–1926: Two volumes bound as one," 1990.

[8] C. Beck, S. Becker, P. Cheridito, A. Jentzen, and A. Neufeld, "Deep splitting method for parabolic pdes," *SIAM Journal on Scientific Computing*, vol. 43, no. 5, pp. A3135–A3154, 2021.

[9] Z. Shi, Z. Hu, M. Lin, and K. Kawaguchi, "Stochastic taylor derivative estimator for high-order multivariable differential operators," https://github.com/Maxxtra/Stochastic-Taylor-Derivative-Estimator-for-High-Order-multivariable-Differential-Operators, 2024.

[10] N. H. March, W. H. Young, and S. Sampanthar, *The many-body problem in quantum mechanics*. Courier Corporation, 1995.

[11] B. Amos *et al.*, "Tutorial on amortized optimization," *Foundations and Trends® in Machine Learning*, vol. 16, no. 5, pp. 592–732, 2023.