GLANCED-IO: Taming I/O Optimization for Deep Learning at Scale

Ray A. O. Sinurat
rayandrew@uchicago.edu
The University of Chicago
Chicago, IL, USA

William Nixon
williamn@uchicago.edu
The University of Chicago
Chicago, IL, USA

Philip Carns
pcarns@acm.org
Argonne National Laboratory
Lemont, IL, USA

Huihuo Zheng
huihuo.zheng@anl.gov
Argonne National Laboratory
Lemont, IL, USA

Sandeep Madireddy
smadireddy@anl.gov
Argonne National Laboratory
Lemont, IL, USA

Sam Foreman
foremans@anl.gov
Argonne National Laboratory
Lemont, IL, USA

Troy Arcomano
troya@allenai.org
Allen Institute for AI
Seattle, WA, USA

Robert Ross
rross@mcs.anl.gov
Argonne National Laboratory
Lemont, IL, USA

Haryadi S. Gunawi
haryadi@cs.uchicago.edu
The University of Chicago
Chicago, IL, USA

Hariharan Devarajan
hariharandev1@llnl.gov
Lawrence Livermore National Laboratory
Livermore, CA, USA

Abstract

Scientific deep learning (DL) at scale typically trains on terabyte-scale datasets across thousands of accelerators, placing immense pressure on storage systems to keep pace with computation. Existing solutions respond to this demand by tuning individual I/O parameters to accelerate training performance. However, these techniques are limited by costly experiments, configuration space explosion, and inability to generalize application-specific optimizations. This leads to applications running with suboptimal configurations that reduce training efficiency, system utilization, or both. To address the challenge of finding the optimal configuration efficiently, we developed GLANCED-IO, a cross-layer I/O optimization framework that optimizes DL pipelines with high-fidelity approximation and efficient configuration space exploration. Through this work, we identified the following three key findings. First, independently optimizing either the application or system configurations leaves up to 2.4× performance on the table for scientists to efficiently run DL pipelines on HPC systems. Second, GLANCED-IO's one-factor-at-a-time (OFAT)-guided greedy exploration strategy achieved results comparable to more-expensive autotuning techniques while removing the pre-training required by ML-based approaches. Third, GLANCED-IO avoids executing the full application during optimization by operating on representative data subsets without GPUs, yet preserves 93% performance fidelity on average when deployed in DL pipelines. We demonstrate the efficacy of GLANCED-IO by optimizing large-scale global weather forecasting DL workloads, achieving up to 1.57× better performance than state-of-the-art with 2.3× fewer configuration evaluations than AIIO and 3.3× faster optimization than DeepHyper.

Keywords

GLANCED-IO, Deep Learning, I/O Optimization, Autotuning, Joint Metric, Proxy Approximation, OFAT, DFTracer, DLIO

ACM Reference Format:
Ray A. O. Sinurat, William Nixon, Philip Carns, Huihuo Zheng, Sandeep Madireddy, Sam Foreman, Troy Arcomano, Robert Ross, Haryadi S. Gunawi, and Hariharan Devarajan. 2026. GLANCED-IO: Taming I/O Optimization for Deep Learning at Scale. In *The 35th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '26), July 13–16, 2026, Cleveland, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3806645.3807588

1 Introduction

Training large-scale deep learning (DL) models for scientific applications such as genomics [53, 61], protein folding [39, 56], and global weather forecasting [26, 27, 38, 46, 47] relies on terabyte-scale datasets that can reach petabyte-scale I/O over the course of training due to repeatedly loading the data across iterations. These applications typically run on hundreds to thousands of accelerators on large-scale HPC systems to keep training time tractable [38, 47]. The resulting data movement between storage systems and accelerators puts immense pressure on the I/O subsystem: training pipelines must continuously stream input samples from filesystems and periodically write checkpoints to preserve training state [47].

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only. Request permissions from owner/author(s).
HPDC '26, Cleveland, OH, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2640-8/2026/07
https://doi.org/10.1145/3806645.3807588

To manage this I/O complexity, modern DL frameworks provide highly configurable pipelines with tunable parameters across multiple software layers, such as data loader threading parallelism, prefetching depth, compression, storage backend selection, and many others [3, 43, 49]. Appropriate configuration of these parameters can yield substantial reductions in training time [34, 57], but the high dimensionality of this configuration space makes manual tuning impractical at scale [25, 36, 50].

Existing approaches address I/O performance primarily within two layers of the software stack: middleware and DL framework. At the middleware layers, systems accelerate individual storage operations through techniques such as caching (e.g., SHADE [41]), prefetching (e.g., noPFS [37]), and I/O optimizations (e.g., HVAC [40] and DYAD [32]), achieving up to 10× speedup of the training pipeline. These optimizations are transparent to applications and operate independently of application-level configuration.

Similarly, DL framework layers expose tunable parameters that can significantly impact end-to-end training time [15, 18] and are configured independently of the underlying storage system. Analogous to hyperparameter optimization in machine learning (ML), these parameters can be optimized using an exhaustive search. However, this approach is impractical for users with limited budgets [50], despite its ability to yield optimal results. As a result, practitioners adopt methods to reduce parameter space exploration time, including autotuning methods (e.g., DeepHyper [21, 36] and H5Tuner [24, 25]) that rely on repeated iterations, and ML-based methods (e.g., AIIO [35] and TunIO [50]). However, these approaches still require many costly workload execution samples or extensive training datasets, and may not be generalizable to unseen workloads.

Although existing approaches have shown success in improving performance in individual layers, they present three key limitations. First, existing techniques *tune knobs in isolation*, with some focusing solely on application performance [32, 37, 40] and others targeting system-level metrics [31, 33, 55], leaving up to 2.4× performance unrealized (as shown in Section 3) by not exploring both layers together. Second, jointly optimizing across layers leads to a *configuration space explosion*; navigating this space is prohibitively expensive, requiring either strict evaluation budgets on autotuners [21, 36] or massive datasets—such as the petabyte-scale data collected over years—for ML-based methods such as AIIO [35]. Both approaches also incur substantial runtime overhead for exploration or significant offline training costs, making them impractical for general usage for real-world DL workloads. Third, existing techniques usually require *running the full DL pipeline*, which might take hours to finish an iteration. This adds significant time and resource overheads across several hundreds of iterations, making DL pipeline optimization infeasible for scientists. Together, these three limitations necessitate a cross-layer optimization framework that strategically reduces the vast parameter space while forgoing full pipeline executions during optimization.

To address these challenges, we developed GLANCED-IO, a novel framework that optimizes DL I/O pipelines efficiently on large-scale HPC systems. GLANCED-IO consists of four components: *Performance Evaluator*, *Proxy Generator*, *Subset Selector*, and *Guided Optimizer* that allows users to optimize both the application and the system configurations to their fullest with minimal time and resource overheads.

To address isolated tuning, *Performance Evaluator* utilizes a multi-objective metric that accounts for both application throughput and system utilization instead of single-objective optimization, ensuring that performance never degrades even when headroom is limited. It also behaves as a validator for each translation point between application, proxy, and subset, guaranteeing that optimizations transfer back to the original workload.

To address configuration space explosion, *Guided Optimizer* uses one-factor-at-a-time (OFAT) exploration combined with a greedy algorithm by prioritizing high-impact parameters based on empirical findings from literature and documentation, reducing configuration space from exponential to linear. *Guided Optimizer* reduces the runtime overhead incurred by autotuners and avoids the pre-training cost of ML-based approaches, enabling efficient optimization of DL workloads and achieving similar or better optimization results than state-of-the-art techniques.

Finally, GLANCED-IO employs both *Proxy Generator* and *Subset Selector* to avoid running the full pipeline during optimization. *Proxy Generator* approximates full-pipeline behavior using GPU-free proxies built on DLIO [34], simplifying generation while preserving essential I/O characteristics and reducing resource consumption (GPU allocation, power) during tuning. *Subset Selector* further applies data-centric subsetting on top of proxy to run optimization at a fraction of full cost while preserving pipeline characteristics. Beyond reducing runtime cost, the combination of *Guided Optimizer* and *Subset Selector* also minimizes dataset versions and sample counts, reducing storage footprint during optimization. Evaluation on five weather forecasting workloads demonstrates that GLANCED-IO improves application throughput and I/O bandwidth by up to 1.8×, reduces configuration space by up to 2.3×, all without requiring prior data collection.

In summary, this work makes the following contributions:

(1) We develop a *multi-objective metric evaluator* that unifies application performance and system utilization, enabling holistic cross-layer I/O optimization.
(2) We design a *guided Greedy-OFAT optimizer* that tunes I/O parameters in linear time, matching or outperforming autotuning and ML-based approaches without requiring prior data collection.
(3) We build a *proxy-based approximator* that combines GPU-free proxies with data-centric subsetting to reduce optimization runtime to a fraction of full cost while maintaining 93% accuracy.

2 Background

2.1 Deep Learning Software Stack

Training DL models involves multiple software layers that move data from storage to accelerators. Frameworks such as PyTorch [49] and TensorFlow [43] orchestrate computation and provide abstractions for model definition. Beneath the framework, the data loader layer manages parallel workers that fetch and preprocess samples, staging batches for GPU consumption. These workers use high-level I/O libraries (e.g., h5py [8] and NumPy [13]) to translate dataset operations into calls to low-level system libraries (e.g., POSIX and MPI-IO) that interact with the file system. This layered architecture also introduces heterogeneous resource dependencies: data loading

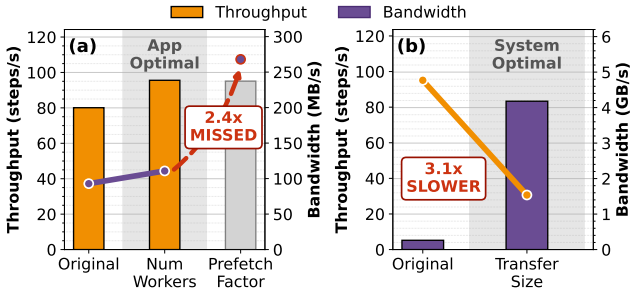


Figure 1: Impact of siloed metric optimization on SFNO-SM. ***(a)*** *Optimizing application throughput alone stops at num workers, missing* ***2.4***× *system I/O bandwidth improvement in prefetch factor tuning.* ***(b)*** *Optimizing system I/O bandwidth by tuning transfer size slows throughput by* ***3.1***×.

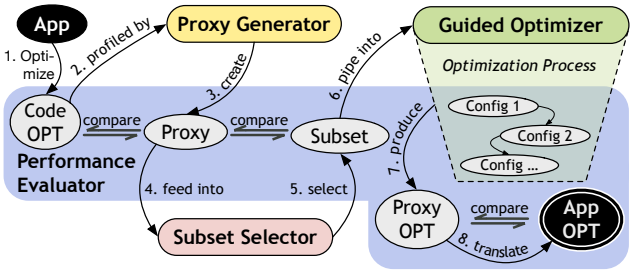


Figure 2: Design of GLANCED-IO. *GLANCED-IO's design enables efficient and tractable end-to-end optimization for deep learning workloads.*

alternates between CPU-bound preprocessing and I/O-bound storage access, training is GPU-bound, and distributed training adds communication-bound phases during gradient synchronization.

2.2 Performance Profiling

Optimizing DL pipelines requires visibility across system and application layers. System-level tools such as Darshan [55] capture I/O counters but lack application semantics, while application profilers such as PyTorch Profiler [17] and TensorFlow Profiler [12] provide timing for data loading and computation but do not trace underlying I/O operations. Cross-layer tracers such as DFTracer [33] address this gap by capturing both application events (e.g., data loading, preprocessing, computation) and I/O operations (i.e., read, write) with unified timestamps. From DFTracer traces, users can derive metrics such as application throughput and I/O bandwidth that serve as optimization objectives.

2.3 Configuration Parameters

The DL stack comprises at least three layers that expose tunable parameters affecting I/O performance: data loading, dataset, and storage. At data loading layer, practitioners configure the number of parallel workers and prefetch depth to control how aggressively batches are staged ahead of computation. At the dataset layer, choices include file format (e.g., HDF5 [7], TFRecord [16]), compression, transfer size and data layout. At the storage layer, parallel file systems expose striping configuration (e.g., stripe count, stripe size) that determine how I/O operations map to storage hardware. These parameters exhibit complex cross-layer dependencies. For instance, increasing worker count raises I/O concurrency but may cause CPU contention, while misaligned transfer sizes relative to stripe configuration can fragment requests and reduce bandwidth.

3 Motivation

GLANCED-IO is designed to address three key challenges in state-of-the-art I/O tuning for DL workloads. First, existing approaches use *siloed metrics*, optimizing either application throughput or I/O bandwidth but not both. To illustrate, we run SFNO-SM, a global weather forecasting model (**Figure 1**). As shown in **Figure 1a**, optimizing for application throughput stops at the number of workers since further tuning yields diminishing returns, missing a 2.4× improvement in system I/O bandwidth. Conversely, **Figure 1b** shows that optimizing for system I/O bandwidth by tuning transfer size yields 15× improvement but degrades application throughput by 3.1×. This reveals a fundamental mismatch: I/O sizes that maximize system performance can be suboptimal for the GPU pipeline. Second, DL I/O pipelines suffer from *configuration space explosion*: $O(N^K)$ for K knobs with N values each. While smarter autotuners [20, 25, 36] navigate this space more efficiently, they still require evaluation budgets [21, 36] (e.g., configuration limits or time restrictions) that may prevent finding high-quality solutions. ML-based approaches such as AIIO [35] avoid exhaustive search but require expensive offline data collection and may fail to generalize across workloads. These approaches raise additional concerns: (1) when should data collection stop, given that more data may improve accuracy but increase collection cost; (2) what preprocessing is required, as these methods shift complexity from configuration search to model engineering (e.g., feature selection and model tuning); (3) how well can models generalize, given that workloads evolve and performance drifts, requiring continual maintenance [52]. These approaches also overlook storage footprint: methods such as DeepHyper [21] require the configuration space to be defined upfront. For I/O tuning, this means either pregenerating all dataset variants, which grows prohibitively for large-scale datasets, or generating on-the-fly, which inflates optimization time (**§4.5**). Third, current tools require *full DL pipeline execution* to evaluate each candidate configuration, incurring significant runtime and resource costs per evaluation. This highlights the critical need for data reduction and lightweight GPU-free proxies for optimization.

4 GLANCED-IO

This section outlines the design of GLANCED-IO. GLANCED-IO is a combination of orchestration scripts and analysis tools that addresses the limitations mentioned in Section 3 through four components: *Performance Evaluator*, *Proxy Generator*, *Subset Selector*, and *Guided Optimizer*, as illustrated in **Figure 2**. The steps are as follows: Application code will be ① optimized first, then *Proxy Generator* ② automatically profiles the code-optimized application and ③ creates a proxy. Next, the proxy is ④ fed into *Subset Selector* which ⑤ automatically selects a representative subset. This subset then ⑥ pipes into the *Guided Optimizer* that ⑦ produces optimized configurations. Finally, the optimized configuration ⑧ is translated back to the original application. Each step is validated by *Performance Evaluator* to ensure all translation points are high-fidelity and improvements can be transferred back to original pipeline.

4.1 Code Optimization

In complex codebases such as DL pipelines, inefficiencies can come from the code itself. Despite having highly optimized DL framework, users can often write inefficient code during development that inherently affect the performance of the application, especially in the data loading layer. For instance, users often prototype with additional data, such as saving and loading more than necessary, even though this is not required in production. Another example arises from the flexibility of the framework itself, e.g., PyTorch encourages users to inherit the `Dataset` class and implement custom logic for data loading and preprocessing. Inefficiencies introduced at this layer can be easily overlooked and may propagate through the data pipeline. In particular, transferring unnecessary data can place pressure on shared memory, which serves as a synchronization point for parallel workers communicating with the main process.

To improve usability, DL frameworks also adopt generalized default behaviors that may not be optimal for all workloads. One such case is PyTorch's default collation step, which aggregates individual samples into a batch. When a workload already treats *each sample as a complete batch*, disabling collation avoids redundant processing and prevents unnecessary overhead. Another common issue in data pipelines is over-abstraction. Users often build on high-level frameworks or libraries layered on top of DL systems. While these tools offer intuitive APIs that accelerate prototyping, their internal complexity is substantial and their opinionated optimization strategies may not generalize across workloads [1, 2]. Since there is no single solution for every workload, GLANCED-IO encourages users to always manually optimize their codebase first to ensure inefficiencies do not inhibit the improvement of optimization stage.

4.2 Performance Evaluator

Performance Evaluator addresses the limitation of single-objective optimization by defining a unified multi-objective metric that accounts for both application throughput and system utilization. As all other components depend on performance metrics, *Performance Evaluator* serves as the foundation of GLANCED-IO. To enable this, GLANCED-IO analyzes code-optimized application traces and derives two metrics: application throughput (steps per second) and I/O bandwidth (bytes per second). We use application throughput rather than application time because throughput is inversely proportional to time while maintaining the same higher-is-better directionality as I/O bandwidth. These metrics validate fidelity across translation points and guide decision making during optimization.

Fidelity Validation. Because GLANCED-IO optimizes proxies and subsets, it must validate that performance characteristics are well-transferred at these three translation points: (1) proxy versus application, ensuring the proxy faithfully represents original I/O behavior, (2) subset versus full proxy, ensuring the subset preserves pipeline characteristics, and (3) optimized proxy versus optimized application, ensuring discovered configurations improve the original workload without degradation. We validate the quality of these translation points using the *fidelity* metric, defined as follows:

$$\text{accuracy} = \left(1 - \frac{|\text{target} - \text{baseline}|}{\text{baseline}}\right) \times 100\% \tag{1}$$

$$\text{fidelity} = 0.5 \times \text{accuracy}_{\text{throughput}} + 0.5 \times \text{accuracy}_{\text{bandwidth}} \tag{2}$$

where baseline refers to the reference point (e.g., full proxy) and target refers to the approximation (e.g., subset). Equal weighting ensures that neither metric dominates fidelity assessment, reflecting the cross-layer nature of GLANCED-IO. If fidelity falls below user-defined thresholds (e.g., this work uses 90% based on MLPerf Storage [4] recommendation), GLANCED-IO flags the configuration and prompts users to regenerate the proxy, adjust the subset, or proceed with caution.

Optimization Objective. We define multi-objective function by combining application throughput and system I/O bandwidth:

$$\text{score} = f(\text{throughput}, \text{bandwidth}) \tag{3}$$

Higher values in both metrics indicate better performance, allowing a simpler optimization objective. Furthermore, GLANCED-IO allows users to define their own objective functions (f). For example, we define f in this work as *weighted scoring*:

$$f = C \times \hat{t} + (1 - C) \times \hat{b} \tag{4}$$

where $\hat{t}$ and $\hat{b}$ are min-max normalized application throughput and I/O bandwidth, respectively, and C is a tunable priority constant.

4.3 Proxy Generator

Proxy Generator addresses the challenge of running full pipelines during optimization by creating lightweight and effective proxy applications. An effective proxy satisfies three criteria: (1) it must reproduce I/O access patterns of the original application, (2) it must preserve computational overlap between data loading and training, and (3) it must not require GPU resources. The first two criteria ensure that I/O optimizations discovered on the proxy transfer back to original application, while the third enables rapid evaluation at reduced cost. At its core, GLANCED-IO uses DLIO [34] as GPU-free proxy, albeit further configuration is necessary to satisfy above criteria. During this stage, given an application, *Proxy Generator* analyzes the traces to extract key I/O characteristics and instantiates DLIO configuration that mirrors these characteristics. *Proxy Generator* extracts the following characteristics from application:

Dataset Structure. The generator analyzes the original dataset to determine sample count, sample dimensions, file format (e.g., NPZ, HDF5, JPEG), and specific format configuration such as chunk size and compression in HDF5. It then creates a synthetic dataset within DLIO that matches these properties, ensuring that file sizes, access granularity, and metadata operations mirror the original.

I/O Access Patterns. DL workloads exhibit distinct access patterns depending on data loading strategy. Generator identifies specific I/O patterns such as single-file or file-per-sample layouts. It also captures prefetch depth and number of data loading workers from the profile. DLIO is then configured to reproduce these patterns to ensure matching with original workload.

Preprocessing Behavior. DL pipelines often include preprocessing stages such as normalization and augmentation that consume significant CPU resources and affect pipeline throughput. *Proxy Generator* analyzes per-sample preprocessing time from the traces and fits the statistical distributions to capture average behavior and variability. Then, *Proxy Generator* will configure DLIO to inject equivalent delays during data loading, preserving the balance between I/O and CPU in the pipeline. This modeling is critical because

preprocessing often dominates data loading time, and ignoring it would yield proxies that overestimate I/O bottlenecks.

Computation Time. To capture computational overlap, the proxy must simulate the computation time spent by the application on forward pass, backward pass, and optimizer steps. To simplify the process, *Proxy Generator* only measures the overall per-batch computation time from traces and configures DLIO to inject synthetic compute delays of equivalent duration. These delays determine how aggressively the data loader can prefetch subsequent batches, directly affecting I/O concurrency and throughput.

Distributed Training Synchronization. Distributed DL training introduces synchronization points that affect overlap behavior between I/O and computation. These behaviors vary across GPU vendors and deployment, and are opaque to the application. Accordingly, *Proxy Generator* identifies synchronization behavior by analyzing the temporal relationship between data loading and batch boundaries. If analysis reveals data loading stalls during gradient synchronization, the generator configures DLIO to enable blocking synchronization (via `MPI.Barrier`) after each step, otherwise the generator models computation and communication as overlapped.

4.4 Subset Selector

Subset Selector addresses the cost burden of full evaluation runs by selecting a subset that still retains the pipeline behavior. However, there are several considerations to be made when selecting a subset. First, datasets from DL workloads are usually defined as logical mappings rather than explicit file lists (e.g., time-series data with temporal dependencies), complicating the identification of representative subsets—an irrelevant complication in I/O optimization. Second, different DL workloads might have different requirements, making one specific application approach not generalizable to other applications. *Subset Selector* simplifies the subsetting process by generating synthetic datasets using proxy, leveraging the convergence of application-system performance to a *steady state* before full-dataset traversal—enabling optimization on reduced data volume with proportionally lower execution time and storage footprint.

Strategy. Subsets closer in size to the original dataset tend to exhibit similar performance characteristics. As such, GLANCED-IO employs simple halving strategy to find the minimal representative subset. Starting from the full proxy dataset, the selector repeatedly halves the dataset size and validates fidelity at each step using *Performance Evaluator* (**§4.2**). The halving continues until either target size reduction is achieved or the fidelity drops below acceptable thresholds, whereupon the selector reverts to the previous size.

Benefit. Beyond reducing runtime cost, subsetting also minimizes storage footprint. During optimization, users may explore multiple dataset configurations (e.g., different layouts and compressions), each requiring a separate copy of dataset. Operating on subsets reduces storage requirements proportionally, enabling exploration of more configurations within fixed storage budgets.

Caveat. Although aggressive subsetting will reduce optimization time significantly, it can introduce unintended side effects that compromise fidelity. When subsets become small enough to fit in file system caches or memory, I/O bandwidth measurements spike artificially as reads are served from cache (e.g., DRAM) rather than underlying storage. This alters I/O behavior, which in turn affects the overlap between data loading and computation, causing deviation of both application throughput and I/O bandwidth measurements from the full dataset. To address this, *Subset Selector* uses *Performance Evaluator* to catch these issues through fidelity scoring, prompting users to revert to a larger subset size.

Table 1: GLANCED-IO cross-layer optimization parameters.

	Layer	Optimization	
More Portable ↓	App	Num Workers	Easier ↓
		Prefetch Factor	
		Dataset Access Pattern	
Less Portable	System	PFS Striping	Harder
		Transfer Size	

4.5 Guided Optimizer

Guided Optimizer addresses the exacerbated configuration space and optimization cost, due to joint assessment of application and system parameters, by reducing the space rather than simply navigating the space efficiently. For this, *Guided Optimizer*, a Greedy-based autotuner, internally uses high-impact parameters based on empirical findings and software documentation while also employing one-factor-at-a-time (OFAT) to reduce the configuration space.

Parameter Ordering. As shown in **Table 1**, GLANCED-IO ranks parameters from more portable and easier to tune (top) to less portable and harder to tune (bottom). First, *Guided Optimizer* starts with application-layer parameters, bootstrapping optimization with accessible tuning knobs. This design choice democratizes I/O optimization for users without storage expertise. For instance, number of workers and prefetch factor are built into DL frameworks such as PyTorch and TensorFlow, requiring no system expertise to modify. Similarly, dataset access pattern can easily be adjusted by modifying the dataset itself (e.g. enabling sequential reads or changing the format, compression and the layout of the data). Next, for users who can leverage system-level optimizations, such as parallel file system configuration that is widely used in HPC systems, GLANCED-IO then explores PFS striping followed by transfer size, which is aligned based on stripe size as recommended in DLIO [34]. Optimizing system-level factors can yield substantial improvements as they will align I/O operations with the underlying storage architecture. However, these parameters are platform-specific, system-specific, and dataset-specific, making them less portable across deployment environments. For example, PyTorch by default does not have the capability to modify dataset transfer size as users tend to choose their dataset format. Meanwhile, TensorFlow actively uses TFRecord [16] format that has `buffer_size` parameter, enabling the tuning of I/O transfer size during data loading. By exploring portable parameters first, GLANCED-IO ensures that all users benefit from optimization even if system-specific tuning is inapplicable to their environment.

Algorithm. Given the ordered parameters, Algorithm 1 details the optimization workflow using OFAT. Rather than exploring all parameter combinations, OFAT varies one parameter while holding others fixed, identifies the best value for that varied parameter, and then moves on to the next parameter. This approach reduces the

Algorithm 1: Guided Optimizer: OFAT with Greedy Selection

Input: Parameter $P = [p_1, \ldots, p_k]$, value pools $V = \{V_1, \ldots, V_k\}$, initial configuration C_0
Output: Optimized configuration C^*

```
1  C* ← C0;
2  best_score ← PerformanceEvaluator(C*);
3  foreach p_i ∈ P do
       // OFAT exploration
4      improved ← false;
5      foreach v ∈ V_i do
           // Parallelize via job scheduler
6          C' ← C* with p_i ← v;
7          score ← PerformanceEvaluator(C');
8          if score > best_score then
               // Greedy selection
9              C* ← C';
10             best_score ← score;
11             improved ← true;
12     if ¬improved then
           // Termination on plateau
13         break;
14 return C*;
```

number of evaluations from $O(N^K)$ to $O(K \times N)$ for K parameters with N values each, making optimization tractable within reasonable time budgets. The algorithm uses streaming Greedy selection: as each value for one parameter is evaluated, *Guided Optimizer* immediately updates the best configuration for that parameter if improvement is observed (lines 7-10).

Storage Implication. Some stages in DL pipeline optimization (see **Table 1**), such as dataset access pattern, PFS striping, and transfer size, modify the data layout and consequently generate multiple copies of the dataset, quickly scaling storage to terabytes or even petabytes. At this scale, optimizing one parameter at a time is more tractable, since acquiring storage to accommodate multiple dataset copies is challenging even at large HPC sites [5, 6, 14]. Using OFAT, multiple copies of the dataset exist only within the explored parameter pool, rather than across the full combinatorial space of all parameters, thereby reducing the storage footprint.

Additional Optimizations. Several optimizations further reduce the cost of the *Guided Optimizer*. First, the algorithm can terminate early when a parameter yields no improvement, indicating diminishing returns (line 11), avoiding unnecessary evaluation. Second, because OFAT is deterministic and fixes one factor at a time, the algorithm can reuse observations across parameter explorations. For example, if the *Guided Optimizer* explored number of workers with values of 1-32 while prefetch factor was fixed at 2 and found 16 workers as the optimal configuration, the subsequent exploration of prefetch factor can skip evaluating prefetch factor 2 since that configuration was already assessed. Third, we acknowledge that OFAT does not capture all cross-parameter interactions: strongly coupled parameters may require joint tuning to reach the best configuration. In our workloads, however, these effects were not prominent enough to justify broader combinatorial search. If such interactions do arise, OFAT naturally supports lightweight refinement: after the initial sweep identifies promising regions, GLANCED-IO can selectively re-evaluate suspected coupled parameters while keeping the remaining parameters fixed. This preserves OFAT's scalability while recovering coordinated improvements at low additional cost.

4.6 Implementation Details

GLANCED-IO consists of orchestration scripts for workflow coordination, configuration files for proxy generation, and analysis utilities for processing profiling data, rather than a monolithic application. We describe the key implementation aspects below.

Profiling and Analysis. GLANCED-IO integrates with DFTracer [33] to trace both original applications and proxy runs, capturing per-operation timestamps, data sizes, and file access patterns in trace format. Furthermore, GLANCED-IO employs analysis involving large-scale traces from distributed runs, implemented on top of C++ and Python, utilizing scalable aggregation techniques similar to WisIO [58].

Framework Agnostic. GLANCED-IO operates on application and I/O behavior extracted from profiling traces rather than from inspection of application source code. This design makes GLANCED-IO conceptually compatible with any DL framework, including PyTorch, TensorFlow, and JAX. As long as the framework's computation and data loading pipeline can be annotated and the I/O calls can be traced with DFTracer, GLANCED-IO can generate proxies, select subsets, and optimize configurations without framework-specific modifications.

Proxy Generation. Proxy Generator produces YAML configuration files that instantiate DLIO with parameters matching the target application's data loading behavior. These configurations specify dataset structure, access patterns, preprocessing delays, and computation time derived from DFTracer traces. Users invoke DLIO with the generated configuration to run proxy evaluations.

Parallel Evaluation. With OFAT exploration strategy, evaluation of different values for the same parameter are independent. Hence, GLANCED-IO can dispatch them in parallel using job schedulers such as Flux [19] or SLURM [59]. Each configuration runs as a separate job, and results are collected upon completion for greedy decision-making before proceeding to the next parameter. This parallelization reduces wall-clock optimization time proportionally to the number of available compute nodes.

Platform Support. Although GLANCED-IO is conceptually platform agnostic, this work targets HPC environments with parallel file systems (e.g., Lustre [11, 29], GPFS [9]) that are widely used for large-scale DL training. As such, the current implementation interfaces with Lustre APIs to query and configure system-level parameters such as stripe count and stripe size.

5 Evaluation

This section evaluates GLANCED-IO's effectiveness in optimizing I/O performance for distributed DL workloads. We address three questions aligned with our contributions:

(1) **Metric effectiveness.** Does a multi-objective metric that captures application performance and system utilization enable better optimization decisions than a single metric?

Table 2: Workloads used in GLANCED-IO.

Workload	Nodes	Dataset Size	Duration (5 Epochs)
STORMER-LG	32	31 TB	~1h 24m
SFNO-LG	16	31 TB	~2h
TinySTORMER-LG	2	5.4 TB	~1h 26m
STORMER-SM	2	71 GB	~1h
SFNO-SM	1	71 GB	~1h

(2) **Proxy-based approximation.** Can proxy-based optimization reduce optimization cost while preserving sufficient fidelity for effective configuration selection?
(3) **Optimization effectiveness.** Can GLANCED-IO efficiently identify high-quality configurations and outperform baselines without requiring prior data collection?

We evaluate GLANCED-IO on five global weather forecasting workloads spanning different model architectures, dataset sizes, and node counts. The evaluation proceeds in five stages, aligned with these questions. First, we evaluate the performance improvement of optimizing pipeline code. Second, we demonstrate why joint application-system metric is necessary. Third, we assess the accuracy and cost reduction enabled by proxy-based approximation. Fourth, we evaluate optimization effectiveness across workloads and compare against baselines. Finally, we present `STORMER-LG` as a representative end-to-end case study. To avoid repetition, we summarize recurring patterns across remaining workloads compactly.

5.1 Experimental Setup

5.1.1 System Configuration. All experiments were conducted on Tuolumne at Lawrence Livermore National Laboratory (LLNL). Each node is equipped with 4 AMD MI300A GPUs, dual 4th-generation AMD EPYC CPUs (96 cores, 192 hyperthreads), and 512 GB of memory. Nodes are interconnected via HPE Slingshot 11 and shared Lustre parallel file system with a maximum bandwidth of 240 GB/s and a storage budget of 100TB.

5.1.2 Datasets. We use the ERA5 reanalysis dataset in HDF5 format [7] at two resolutions: 0.25° for ERA5-LG and 5.6° for ERA5-SM. Both datasets span 40 years (1979–2018), with each year partitioned into approximately 1,460 HDF5 files. Following the training configuration used by STORMER [47], each training step loads both input and target atmospheric variables from 69 HDF5 datasets per file. ERA5-LG comprises approximately 31 TB across 58,400 files, each containing 130 datasets with dimensions 721×1440 in `float32`, resulting in ~573 MB of data per step. ERA5-SM comprises ~71 GB across 58,300 files, each containing 114 datasets with dimensions 32×64 in `float32`, resulting in ~1 MB of data per step.

5.1.3 Workloads. We evaluate three global weather forecasting model architectures: SFNO [27], STORMER [47], and TinySTORMER, a reduced-parameter variant of STORMER. All models are implemented in PyTorch using Distributed Data Parallel (DDP) and use h5py to read HDF5 datasets. To train the models, we use the modified STORMER pipeline [45] such that SFNO can still use the same codebase to simplify the experiments. Throughout this section, each workload configuration is named `MODEL-DATASET`, where `MODEL` denotes the architecture and `DATASET` indicates the dataset size (`LG` for ERA5-LG, `SM` for ERA5-SM). Specifically, `TinySTORMER-LG` uses gradient accumulation and a subset of ERA5-LG due to memory and runtime constraints at smaller node counts. During proxy-based optimization, GPU execution is disabled and workloads are evaluated using CPU-only proxy runs. **Table 2** summarizes all evaluated configurations along with their 5-epoch durations, which are used to capture stable and representative I/O behavior. During experiments, all performance metrics are computed on a per-epoch basis. For each evaluated configuration, we run five epochs and report the average values. We observe less than 5% run-to-run variability and therefore we omit error bars throughout this evaluation.

Table 3: Code optimization achieves up to 1.24× improvements in application throughput and I/O bandwidth.

Application	Startup	Preprocess	Get Item	Thpt.	Bandw.
`STORMER-LG`	1.95×	1.46×	1.27×	1.11×	1.09×
`SFNO-LG`	1.82×	1.43×	1.14×	1.24×	1.23×
`TinySTORMER-LG`	1.66×	1.25×	0.92×	1.04×	1.04×
`STORMER-SM`	1.52×	1.01×	1.24×	1.11×	1.09×
`SFNO-SM`	0.81×	0.86×	1.03×	1.14×	1.13×

5.1.4 Baselines. We compare GLANCED-IO with three baselines:

MaxResource reflects a common HPC practice in which users allocate and utilize the maximum available resources under the assumption that more resources yield better performance. For application configuration, we cap the number of workers based on the ratio of CPU hyperthreads to GPUs (48 workers per GPU on Tuolumne). We then search for worker and prefetch factor combinations that maximize memory usage, allocating up to 90% of total memory to workers to account for runtime overhead. For dataset configuration, MaxResource maximizes transfer size by packing datasets into a single HDF5 file. For system configuration, it uses all Lustre OSTs with stripe size matching the dataset transfer size.

AIIO [35] is an ML-based approach that predicts I/O bandwidth from low-level I/O counters. Since AIIO does not directly support DL pipelines parameters, we extend it using a two-stage inference process. The first stage predicts I/O counters from application configurations, and the second stage predicts system-level I/O bandwidth from the estimated counters. To train these models, we used traces across diverse workloads collected in DLIO [34] and WisIO [58]. Furthermore, since those collected data do not include our workloads defined in Section 5.1.3, we incorporate data from our optimization stages of GLANCED-IO to ensure the generality of the model. In summary, we used 7 days' worth of traces, 271 unique workloads, across 2,474 nodes, resulting in 1.2 billion events, to train AIIO model. After training, AIIO performs exhaustive search over the configuration space and selects the best configuration with the highest predicted I/O bandwidth.

DeepHyper [21] is an open-source Python package for scalable hyperparameter search. Although its main purpose is hyperparameter optimization, autotuning work on HEPnOS [36] shows that it can also be used to identify optimal storage software configurations. It uses a surrogate model to learn the relationship between configurations and the objective function, enabling efficient exploration of the search space. In this work, we used random forest regressor similar to HEPnOS autotuning and specify 12 hours budget for running the optimization stage. Similar to HEPnOS autotuning, we used application metric (throughput) as the objective metric.

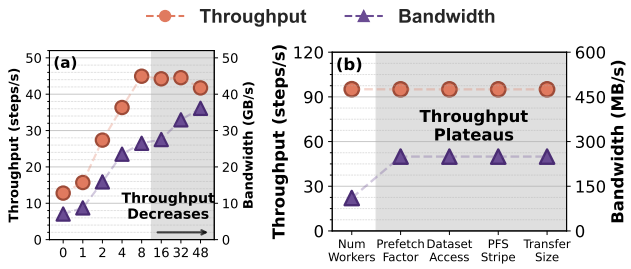


Figure 3: Single-metric optimization failures. ***(a)*** *Worker counts for* `SFNO-LG` *shows that optimizing only on system I/O bandwidth can obscure application throughput degradation.* ***(b)*** *Optimization on* `STORMER-SM` *shows that early termination on application metric can miss holistic pipeline gains.*

5.2 Impact of Code Optimization

To observe the impact of optimizing the original application's code, we run 5 epochs of all workloads for both original codebase and code-optimized codebase and compare their performance. We manually optimize trainer and data loading implementation of STORMER by (1) eliminating runtime overhead, such as unnecessary function calls and over-abstraction of the code, (2) reducing small I/O accesses, such as iterating directories to look up the dataset files, (3) minimizing unnecessary preprocessing, such as removing collate operation, and (4) eliminating unnecessary data movement, such as sending unnecessary variables to shared memory used for synchronization between parallel workers and main process.

As shown in **Table 3**, we can speed up the pipeline startup time by up to 1.95× simply through optimization of the codes. Notably, the effects of code optimizations are more pronounced in larger workloads as smaller workloads would obscure problems associated with data processing (i.e., preprocessing large dataset). Even though some of the internal application metrics such as pipeline startup, preprocess, and getitem time fluctuate across workloads, the impact towards application throughput and bandwidth still reaches up to 1.24×. This underscores the intricacy of DL pipelines, where optimizing a specific application layer may appear counterproductive, yet enables substantial gains at the pipeline level.

Takeaway 1. *Code optimization is necessary as performance bottlenecks can be hidden at small scales of data and deployments.*

5.3 Impact of Joint Metrics Optimization

To understand the importance of joint metrics, we run GLANCED-IO optimization stage across all workloads. Since GLANCED-IO optimizes application and system layers simultaneously, we monitor optimization progress and simulate early termination when one layer reaches a plateau, regardless of continued improvement in the other. **Figure 3** reveals that such early stopping can cause the optimizer to miss substantial gains in holistic pipeline performance. In the figures, x-axis shows the configuration currently being evaluated, left y-axis shows the application throughput and the right y-axis shows the I/O bandwidth. The progression of these metrics are shown as orange circle and purple triangle, respectively.

Optimizing system utilization alone can degrade application performance. Figure 3a shows worker-count sensitivity for `SFNO-LG`. As the number of data loader workers increases, I/O bandwidth improves monotonically, indicating higher filesystem utilization. However, application throughput peaks at a small worker count and degrades thereafter due to increased CPU contention. An optimization strategy that prioritizes system utilization alone would select configurations that reduce application throughput.

Table 4: GLANCED-IO achieves 93% average fidelity when generating proxies and subsets across diverse workloads.

Workload	App →Proxy	Full →Subset	Proxy →App
`STORMER-LG`	96.5%	97.7%	90.8%
`SFNO-LG`	94.8%	98.4%	54.6%
`TinySTORMER-LG`	79.5%	98.6%	94.1%
`STORMER-SM`	98.6%	99.6%	97.9%
`SFNO-SM`	99.0%	99.5%	96.1%

Optimizing application throughput alone can hide underutilized system resources. Figure 3b shows stage-wise optimization behavior for `STORMER-SM` on two nodes. After application-level tuning, application throughput plateaus, suggesting limited optimization headroom. However, system-level I/O bandwidth continues to improve across subsequent stages. An optimizer that relies only on application throughput would stop early and miss opportunities to improve system utilization without degrading execution.

Implications for cross-layer optimization. Across workloads and scales, application-level behavior and system-level utilization can diverge in both directions. Optimizing either perspective in isolation therefore leads to suboptimal outcomes. A joint metric is therefore required to reconcile these trade-offs and guide optimization toward configurations that improve end-to-end throughput while maintaining efficient system utilization.

Takeaway 2. *Single-metric optimization fails to account for cross-layer interactions in DL I/O pipelines, motivating a joint metric that aligns application and system performance.*

5.4 Fidelity of Proxy-Based Approximation

To understand the benefit of GPU-free proxy-based approximation during optimization, we annotated and traced five workloads performance using DFTracer and use *Proxy Generator* to build DLIO proxy and generate the synthetic data that mimic the I/O performance of the pipeline. Then, we keep configuration of the proxy the same as the original pipeline. Furthermore, these proxies will be processed further using *Subset Selector* to create subsets of synthetic datasets. To ensure translatability from proxy and subset, we examine three aspects: App-to-Proxy, subset, and Proxy-to-App.

App-to-Proxy Fidelity. Table 4 reports App-to-Proxy translation accuracy across all workloads. For large-scale workloads, accuracy exceeds 94%, and for small-scale workloads it exceeds 98%, indicating that the proxy closely matches application-level I/O behavior across workloads. `TinySTORMER-LG` exhibits lower fidelity (79.5%), reflecting the impact of gradient accumulation and accelerated computation on proxy accuracy. This occurs because gradient accumulation accelerates computation and reduces synchronization frequency, which changes the overlap between computation and I/O. Such behavior is difficult to faithfully emulate in the proxy because it depends on asynchronous GPU execution and vendor-specific synchronization mechanisms. Despite this variability, the

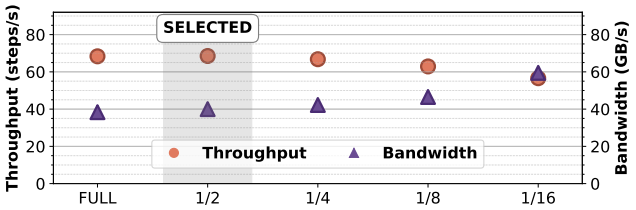


Figure 4: Effect of dataset subset size on I/O behavior for `STORMER-LG`. *Using half the dataset preserves full-dataset I/O behavior. Smaller subsets (1/8 and 1/16) enter a cache-dominated regime that inflates application throughput and alters I/O bandwidth trends.*

Table 5: GLANCED-IO achieves up to 8.5× reduction in optimization cost via *Subset Generation*.

Workload	Subset	Full (5 ep)	Subset (5 ep)	Speedup
STORMER-LG	1/2	~84 min	~35 min	2.4×
SFNO-LG	1/4	~120 min	~27 min	4.4×
TinySTORMER-LG	1/2	~86 min	~54 min	1.6×
STORMER-SM	1/4	~57 min	~13 min	4.3×
SFNO-SM	1/8	~60 min	~7 min	8.5×

proxy preserves relative performance trends across configurations, which is sufficient to guide optimization.

Subset Representativeness. To reduce optimization cost, we evaluate configurations using reduced dataset subsets. As shown in **Table 4**, Full-to-Subset fidelity remains consistently high across all workloads and exceeds 97%. This indicates that appropriately chosen subsets preserve full-scale performance behavior while substantially reducing execution cost during optimization. As mentioned in Section 4.4 and illustrated in **Figure 4**, smaller subsets fail to saturate storage and memory subsystems. Subset configurations are chosen per workload to balance cost reduction and fidelity. As shown in **Table 5**, subset ratios vary across applications, yet consistently reduce per-configuration evaluation time by 1.6 to 8.5×.

Proxy-to-App Transfer Accuracy. After optimizing on the proxy and subset configuration, selected parameters are evaluated on the original application. **Table 4** shows that most workloads exceed 90% transfer accuracy, except `SFNO-LG` with lower accuracy of 54.6%. This discrepancy arises from differences in execution balance between the proxy and the full application. After optimization, `SFNO-LG` exhibits improved overlap between computation and I/O, altering resource contention and execution timing in ways not captured by proxy execution. Because the proxy incurs minimal data-fetch overhead, it underestimates the absolute performance gains achieved at full scale. A detailed execution-time breakdown for `SFNO-LG` is provided in **Appendix A**. Despite this divergence in absolute magnitude, Section 5.7 shows that optimized configuration improves application throughput relative to original configuration, indicating that the proxy preserves the relative performance ordering needed to guide effective optimization.

Aggregate Accuracy. Across all workloads, App-to-Proxy accuracy averages 93.7%, Full-to-Subset accuracy averages 98%, and Proxy-to-App accuracy averages 86.7%. While fidelity varies by workload, proxy-based approximation consistently preserves relative optimization trends across the configuration space.

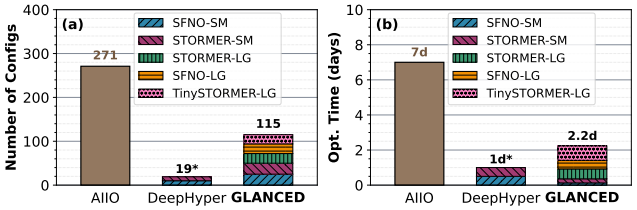


Figure 5: Optimization cost comparison. *GLANCED-IO reduces **(a)** number of configurations to evaluate by up to 2.3× and **(b)** optimization time by up to 3.2×, enabling practical deployment-time tuning without prior data collection. *DeepHyper cannot be run at larger scale due to storage limitations.*

Takeaway 3. *GPU-free proxies preserve original workload behavior and relative improvement while making optimization time practical.*

5.5 Optimization Cost

Having established that proxy-based approximation preserves optimization fidelity while reducing per-configuration cost, we now evaluate the efficiency and scalability of GLANCED-IO's optimization for large configuration spaces. To understand this, we run *Guided Optimizer* across all workloads defined in Section 5.1.3. We exclude jobs queuing and dataset generation time to ensure fairness.

As detailed in Section 5.1.4, we compare GLANCED-IO with AIIO and DeepHyper. Additionaly, we used data collection time for building of AIIO models as their optimization cost. **Figure 5** compares the number of configurations evaluated and end-to-end optimization time between GLANCED-IO, AIIO, and DeepHyper, where y-axis in (a) shows the number of configurations and (b) shows optimization time in days. GLANCED-IO reduces the number of evaluated configurations by 2.3× compared to AIIO using Greedy-OFAT exploration that scales linearly with the number of tunable parameters rather than exhaustively searching the configuration space. This reduction directly translates into optimization time savings: AIIO requires 7 days, including upfront profiling and prediction-guided search, whereas GLANCED-IO completes in 2.2 days by combining Greedy-OFAT with proxy-based approximation and subset evaluation (**§5.4**).

Takeaway 4. *GLANCED-IO achieves 3.2× faster optimization than ML-based approach by eliminating upfront data collections.*

At small scale, each workload has 1,568 possible configurations. DeepHyper, constrained to 12-hour budgets per workload, explores only 10 and 9 configurations for `SFNO-SM` and `STORMER-SM` respectively (~0.6% each), totaling 19 evaluations in 24 hours due to full-pipeline overhead. In contrast, GLANCED-IO's data-centric subsetting evaluates 25 configurations per workload (50 total) within 7.2 hours, a 3.3× speedup over DeepHyper while exploring 2.6× more configurations. At large scale, DeepHyper is not practical in our setting because the search space includes not only runtime knobs, but also *dataset configurations*. Thus, evaluating a candidate may require building a new dataset variant rather than toggling a parameter, and precomputing all variants would exceed our 100TB budget by reaching petabyte-scale storage.

On-the-fly generation does not fundamentally solve the problem as it shifts the overhead into the search loop. Trials must wait for data generation, limiting the number of configurations that can be explored within the time budget. GLANCED-IO avoids this issue

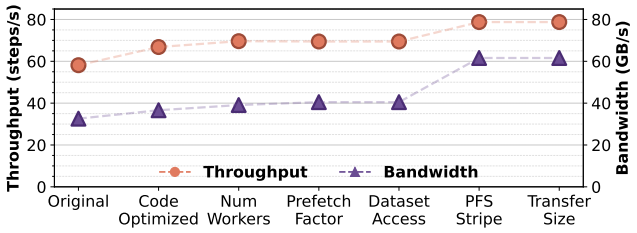


Figure 6: STORMER-LG optimization. *GLANCED-IO simultaneously improves both application and system layer by 1.3×.*

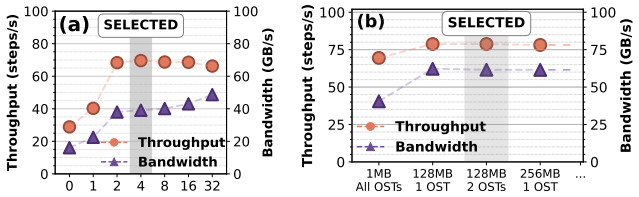


Figure 7: Stage-level optimizations for STORMER-LG. ***(a)*** *GLANCED-IO choose number of workers with the highest throughput (~69 steps/s), avoiding degradation of application performance in larger number of workers.* ***(b)*** *Selected stripe configuration improves I/O bandwidth by 1.3×.*

through OFAT which materializes data transformations incrementally. Furthermore the subset-based optimization reduces storage in direct proportion to subset size; for example, a quarter-sized subset requires only a quarter of the storage.

Takeaway 5. *Storage is a hidden optimization cost: upfront data generation can exceed budgets before search begins.*

5.6 End-to-End Optimization: STORMER-LG

To illustrate how proxy-guided joint optimization translates into end-to-end application performance gains, we present STORMER-LG as a representative large-scale, I/O-bound case study. This workload uses a large number of GPUs and a multi-terabyte dataset, exhibiting substantial I/O pressure and a large configuration space.

Optimization Trajectory. Figure 6 shows optimization trajectory for STORMER-LG. Across stages, both application throughput and I/O bandwidth improve monotonically. The optimized configuration achieves a 1.3× improvement in application throughput and I/O bandwidth relative to original application, demonstrating consistent gains at both application and system levels.

Application-level Parameter Tuning. Figure 7a shows the effect of tuning application-level parameters. Increasing the number of data loader workers initially improves application throughput, but performance peaks at a small worker count and degrades beyond that point due to CPU contention. In contrast, I/O bandwidth continues to increase with additional workers. Configurations that maximize bandwidth alone therefore lead to reduced end-to-end throughput, whereas configurations near the throughput peak achieve better joint performance. Prefetch factor tuning has limited impact for this workload, and the original dataset layout is retained.

System-level Parameter Tuning. Figure 7b shows the impact of system-level tuning. Moving away from the default Lustre striping configuration yields substantial improvements in both application throughput and I/O bandwidth. Among the evaluated settings, a configuration using two OSTs with a 128 MB stripe size provides the best joint performance. Further tuning of transfer size does not improve performance and is therefore not selected.

Optimization Cost. For STORMER-LG, optimization is performed using a 1/2 dataset subset, reducing per-configuration evaluation time from approximately 84 minutes to 35 minutes (**Table 5**). In total, GLANCED-IO evaluates only 22 configurations across all optimization stages, and the complete end-to-end optimization finishes in ~13 hours, as shown in **Figure 5b**. In contrast, exhaustive exploration of the full configuration space would require evaluating orders of magnitude more configurations, making it infeasible at this scale, whereas GLANCED-IO remains modest relative to large-scale training runs and enables efficient optimization in deployment.

End-to-end Outcome. The final optimized configuration improves application throughput while maintaining high system utilization. As shown in **Figure 8**, the optimized STORMER-LG configuration achieves the highest application throughput among the evaluated approaches, reflecting the benefit of multi-objective optimization. A detailed baseline comparison is discussed in Section 5.7.

Takeaway 6. *The* STORMER-LG *case study confirms that proxy- and subset-based joint optimization yields substantial end-to-end performance gains at scale while keeping optimization overhead practical.*

5.7 Baseline Comparison

To evaluate GLANCED-IO's end-to-end effectiveness relative to existing approaches, we compare it against three representative baselines: MaxResource, AIIO, and DeepHyper detailed in Section 5.1.4. After getting configurations from these baselines, we run each workload with selected configurations and compare it to GLANCED-IO. To ensure fairness, each system used the same code-optimized version of the application as its starting point. **Figure 8** reports application throughput and I/O bandwidth for the configuration selected by each approach. In the figures, left y-axis is the throughput shown using orange bar and right y-axis is the I/O bandwidth shown using purple bar. To make comparison easier, we also include original pipeline performance as the leftmost bars on each workload.

Large-scale, I/O-bound workloads (STORMER-LG, SFNO-LG). GLANCED-IO consistently achieves the highest application throughput while maintaining high I/O bandwidth. AIIO attains comparable bandwidth and competitive throughput but with 3.2× slower optimization time (**§5.5**). However, for SFNO-LG, AIIO fails to identify effective configuration, resulting in 1.57× lower application throughput and I/O bandwidth compared to GLANCED-IO. This occurs because AIIO selects configurations that maximize I/O bandwidth without accounting for cross-layer interactions, leading to choices that do not translate into end-to-end performance gains. MaxResource underperforms in large-scale workloads despite allocating the maximum available resources, emphasizing that maximizing resource utilization does not guarantee better performance. DeepHyper is infeasible at large scale due to storage limitations (**§5.5**).

Small-scale workloads (STORMER-SM, SFNO-SM). For smaller workloads, performance differences are reduced because application throughput is already near peak and I/O pressure is limited. DeepHyper achieves similar results to GLANCED-IO in this regime

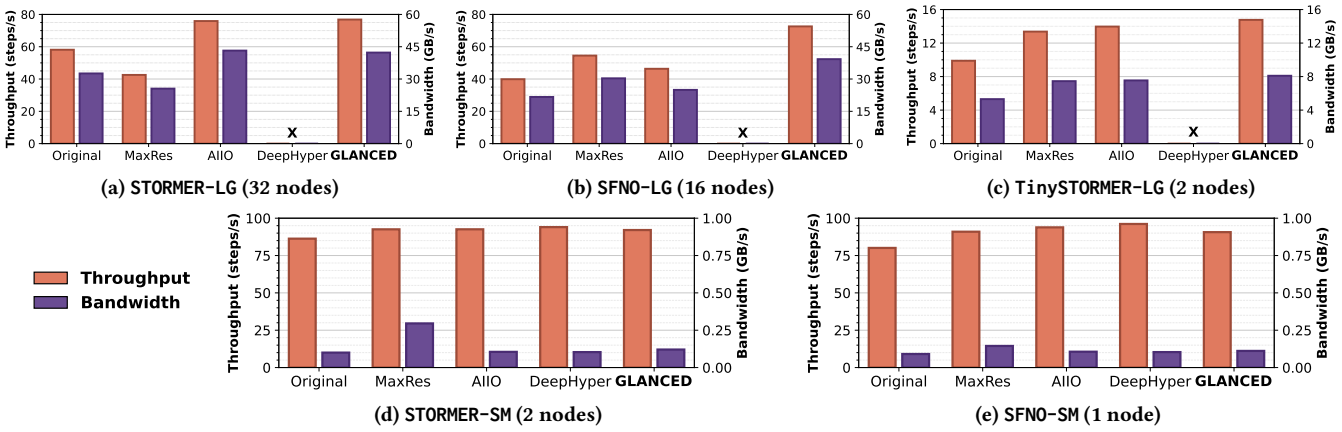


Figure 8: Baseline comparison. *GLANCED-IO matches or outperforms AIIO and DeepHyper (small-scale only) performance with 3.2× and 3.3× faster optimization respectively, outperforms MaxRes at scale, and achieves up to 1.8× gains over original pipelines.* ***X*** *= infeasible due to storage constraints.*

but with 3.3× slower optimization time, while AIIO and MaxResource also remain competitive. Importantly, GLANCED-IO does not degrade performance in these regimes, indicating robust behavior when optimization headroom is constrained. On average, GLANCED-IO achieves up to 1.8× improvement in both application throughput and I/O bandwidth compared to original pipelines.

Takeaway 7. *Joint optimization improves end-to-end performance for I/O-bound workloads, while avoiding performance regressions when I/O is not the dominant bottleneck.*

5.8 Cross-Workload Optimization Behavior

Having established optimization fidelity and cost effectiveness, we examine how GLANCED-IO performs across workloads with different I/O regimes. Rather than repeating full optimization trajectories, we summarize recurring patterns that emerge across workloads.

Large-scale, I/O-bound Workloads (STORMER-LG, SFNO-LG). For workloads operating under sustained I/O pressure, joint optimization consistently yields substantial performance improvements. In both STORMER-LG and SFNO-LG, increasing I/O parallelism, such as the number of data loader workers, improves system utilization but degrades application throughput. The unified metric reliably identifies configurations that balance these effects, avoiding high-bandwidth configurations that yield low end-to-end throughput (**Figure 8a,b**). On both workloads, filesystem striping provides the largest system-level performance gains, even when the most effective application-level knobs vary across workloads (**Figure 7b**). Specifically for SFNO-LG, GLANCED-IO improves the original pipeline by 1.8× in both application throughput and I/O bandwidth, outperforming all other baselines.

Compute-accelerated Workload (TinySTORMER-LG). TinySTORMER-LG exhibits different behavior due to accelerated computation from gradient accumulation, which increases pressure on the data-loading pipeline. In this regime, the optimizer selects more aggressive application-level parallelism, such as a higher number of data loader workers, to keep pace with computation, while dataset reorganization provides little benefit. Despite altered execution characteristics, the optimization process converges to configurations that improve both application throughput and I/O bandwidth.

Non-I/O-bound Workload (STORMER-SM, SFNO-SM). For small workloads, application throughput is already near peak and largely insensitive to I/O tuning. Many configuration changes increase measured I/O bandwidth without improving throughput, and some actively degrade performance. For example, increasing transfer size substantially improves bandwidth while reducing application throughput (**Figure 1b**). The unified metric correctly avoids such configurations by accounting for application-level impact, and optimization converges quickly without regressions.

6 Related Work

GLANCED-IO differs from prior works in three key aspects: objective metrics, configuration tuning, and proxy construction.

6.1 Optimization Metrics

Existing works optimize HPC and DL pipelines by adopting a single metric as the optimization objective. Tools such as DYAD [32], HVAC [40], noPFS [37], and Accelerating Data Loading [57] focused on optimizing data loading, such as enabling locality-aware data loading, to optimize application throughput. Meanwhile, tf-Darshan [31], DeepIO [62], Continuous Characterization via Darshan [30] and DFTracer [33] focused on speeding up data loading from system layer using I/O metrics as primary objectives. All of the above approaches are complementary to and remain applicable within GLANCED-IO; however, GLANCED-IO jointly accounts for both performance metrics to ensure that optimizing one does not degrade the other. More recent work, such as DLIO [34], highlighted the benefits of jointly considering both application throughput and I/O bandwidth. Similarly, MLPerf Storage [4], a widely adopted community standard, requires submissions to maintain at least 90% of application throughput, even though final rankings are based solely on I/O bandwidth. Together, these efforts indicate a growing recognition that I/O performance should be optimized without degrading application performance, further motivating GLANCED-IO to use joint metrics during optimization.

6.2 Configuration Tunings

Prior works address the explosion of configuration space during I/O optimization. For example, H5Tuner [22, 25] uses evolutionary algorithms to navigate large configuration spaces and tune multi-layer parameters at runtime. In contrast, HEPnOS Autotuning [36] adopts Bayesian optimization via DeepHyper [21] to optimize a distributed storage service tailored for physics applications. With the rise of machine learning, approaches such as AIIO [35] aim to eliminate exhaustive search altogether, while TunIO [50] leverages reinforcement learning to identify high-impact configurations. Lastly, DLIO [34] uses controlled parameter sweeping during optimization as a proof that controlled optimization on high impact configuration will still be beneficial while keeping the cost minimal. GLANCED-IO avoids limitations above by using OFAT-guided greedy optimization, which keeps the configuration space linear while prioritizing high-impact parameters.

6.3 Proxy Applications

Many works reduce evaluation cost by constructing lightweight proxies of the original applications during optimization. In HPC, several I/O kernels, such as VPIC-IO [10, 28], VORPAL-IO [48], and GCRM-IO [51], are manually extracted from application codebases. This extraction process can be automated using Automatic Kernels Generation [23]. In the DL domain, tools such as DLIO [34] and GPEmu [54] model end-to-end pipelines and enable low-cost GPU-free evaluation. To further reduce evaluation cost, works in the ML community, including AUTOMATA [42] and Two-Step Hyperparameter Optimization [60], select representative data subsets to accelerate model tuning. Mantevo [44] develops workload-specific proxy applications (miniapps) that preserve the behavior of various HPC applications for general benchmarking studies. In contrast, GLANCED-IO uses GPU-free proxy and lightweight subset construction to support efficient I/O optimization across DL workloads.

7 Conclusion

With the growing scale of data-intensive DL workloads on HPC systems, I/O has become a dominant bottleneck as its behavior varies widely across applications and execution regimes. GLANCED-IO provides a cross-layer framework that considers both application performance and system utilization to optimize I/O in distributed training pipelines. Our results show that optimizing only application performance can lead to underutilized system resources, while optimizing only I/O performance can degrade end-to-end application throughput, leaving up to 2.4× performance unrealized. We further demonstrate that the cost of autotuning and ML-based approaches often outweighs their performance benefits in practice, as they incur high optimization overhead or require upfront data collection. In contrast, GLANCED-IO's OFAT-guided greedy exploration achieves performance comparable to state-of-the-art techniques while remaining inexpensive, and its proxy-based optimization using representative data subsets enables GPU-free evaluation while preserving 93% average performance fidelity. Across five global weather forecasting workloads, GLANCED-IO improves performance by up to 1.57× over baselines, evaluates 2.3× fewer configurations than AIIO, and completes optimization 3.3× faster than DeepHyper. We further demonstrate that GLANCED-IO generalizes across diverse workloads and execution regimes, identifying balanced I/O parallelism in I/O-bound cases while avoiding performance degradation when I/O is not the bottleneck. As future work, we will leverage GLANCED-IO's efficient evaluation to explore iterative refinement and stochastic exploration for further performance gains.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-2015606). This material is based upon work supported by LLNL LDRD 25-ERD-042. The computations were enabled by the computational resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357, Laboratory Computing Resource Center (LCRC) at the Argonne National Laboratory. This research was also supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, through the SciDAC-RAPIDS2 institute under Contract DE-AC02-06CH11357. This material was supported by funding from NSF (grant No. CCF-2028427). Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions. We thank Sherlyn Wijaya for proofreading and improving the readability of the manuscript.

References

[1] 2021. Is There Any Way to Disable Prefetching of Batches? · Lightning-AI/Pytorch-Lightning · Discussion #9660. https://github.com/Lightning-AI/pytorch-lightning/discussions/9660
[2] 2023. Introduce Cache 1/n by Tchaton · Pull Request #18642 · Lightning-AI/Pytorch-Lightning. https://github.com/Lightning-AI/pytorch-lightning/pull/18642
[3] 2025. Tensorflow/Io. tensorflow. https://github.com/tensorflow/io
[4] 2026. Benchmark MLPerf Storage | MLCommons V1.1 Results. https://mlcommons.org/benchmarks/storage/
[5] 2026. Data Storage and Transfers — OLCF User Documentation. https://docs.olcf.ornl.gov/data/index.html
[6] 2026. Disk Quota - ALCF User Guides. https://docs.alcf.anl.gov/data-management/filesystem-and-storage/disk-quota/
[7] 2026. The HDF Group. https://www.hdfgroup.org/
[8] 2026. HDF5 for Python. https://www.h5py.org/
[9] 2026. IBM Storage Scale. https://www.ibm.com/docs/en/storage-scale/www.ibm.com/docs/en/storage-scale/6.0.0
[10] 2026. Lanl/Vpic. Los Alamos National Laboratory. https://github.com/lanl/vpic
[11] 2026. Lustre. https://www.lustre.org/
[12] 2026. Module: Tf.Profiler | TensorFlow v2.16.1. https://www.tensorflow.org/api_docs/python/tf/profiler
[13] 2026. NumPy. https://numpy.org/
[14] 2026. Parallel File Systems | HPC @ LLNL. https://hpc.llnl.gov/hardware/file-systems/parallel-file-systems
[15] 2026. Tf.Data.Dataset | TensorFlow v2.16.1. https://www.tensorflow.org/api_docs/python/tf/data/Dataset
[16] 2026. Tf.Data.TFRecordDataset | TensorFlow v2.16.1. https://www.tensorflow.org/api_docs/python/tf/data/TFRecordDataset
[17] 2026. Torch.Profiler — PyTorch 2.10 Documentation. https://docs.pytorch.org/docs/stable/profiler.html
[18] 2026. Torch.Utils.Data — PyTorch 2.10 Documentation. https://docs.pytorch.org/docs/stable/data.html
[19] Dong H. Ahn, Jim Garlick, Mark Grondona, Don Lipari, Becky Springmeyer, and Martin Schulz. 2014. Flux: A Next-Generation Resource Management Framework for Large HPC Centers. In *2014 43rd International Conference on Parallel Processing Workshops*. 9–17. doi:10.1109/ICPPW.2014.15
[20] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard Vuduc. 2018. Autotuning in High-Performance Computing Applications. *Proc. IEEE* 106, 11 (Nov. 2018), 2068–2083.

doi:10.1109/JPROC.2018.2841200

[21] Prasanna Balaprakash, Michael Salim, Thomas D. Uram, Venkat Vishwanath, and Stefan M. Wild. 2018. DeepHyper: Asynchronous Hyperparameter Search for Deep Neural Networks. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. 42–51. doi:10.1109/HiPC.2018.00014

[22] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. 2018. Optimizing I/O Performance of HPC Applications with Autotuning. *ACM Transactions on Parallel Computing* 5, 4 (Dec. 2018), 1–27. doi:10.1145/3309205

[23] Babak Behzad, Hoang-Vu Dang, Farah Hariri, Weizhe Zhang, and Marc Snir. 2014. Automatic Generation of I/O Kernels for HPC Applications. In *2014 9th Parallel Data Storage Workshop*. 31–36. doi:10.1109/PDSW.2014.6

[24] Babak Behzad, Joseph Huchette, Huong Vu Thanh Luu, Ruth Aydt, Surendra Byna, Yushu Yao, Quincey Koziol, and Prabhat. 2013. A Framework for Auto-Tuning HDF5 Applications. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing (HPDC '13)*. Association for Computing Machinery, New York, NY, USA, 127–128. doi:10.1145/2493123.2462931

[25] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Prabhat, Ruth Aydt, Quincey Koziol, and Marc Snir. 2013. Taming Parallel I/O Complexity with Auto-Tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2503210.2503278

[26] Kaifeng Bi, Lingxi Xie, Hengheng Zhang, Xin Chen, Xiaotao Gu, and Qi Tian. 2023. Accurate Medium-Range Global Weather Forecasting with 3D Neural Networks. *Nature* 619, 7970 (July 2023), 533–538. doi:10.1038/s41586-023-06185-3

[27] Boris Bonev, Thorsten Kurth, Christian Hundt, Jaideep Pathak, Maximilian Baust, Karthik Kashinath, and Anima Anandkumar. 2023. Spherical Fourier Neural Operators: Learning Stable Dynamics on the Sphere. In *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 2806–2823. https://proceedings.mlr.press/v202/bonev23a.html

[28] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson. 2008. 0.374 Pflop/s Trillion-Particle Kinetic Modeling of Laser Plasma Interaction on Roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 1–11. doi:10.1109/SC.2008.5222734

[29] Peter Braam. 2019. The Lustre Storage Architecture. arXiv:1903.01955 [cs] doi:10.48550/arXiv.1903.01955

[30] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. 2011. Understanding and Improving Computational Science Storage Access through Continuous Characterization. *ACM Trans. Storage* 7, 3 (Oct. 2011), 8:1–8:26. doi:10.1145/2027066.2027068

[31] Steven W. D. Chien, Artur Podobas, Ivy B. Peng, and Stefano Markidis. 2020. Tf-Darshan: Understanding Fine-grained I/O Performance in Machine Learning Workloads. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 359–370. doi:10.1109/CLUSTER49012.2020.00046

[32] Hariharan Devarajan, Ian Lumsden, Chen Wang, Konstantia Georgouli, Tom Scogland, Jae-Seung Yeom, and Michela Taufer. 2024. DYAD: Locality-aware Data Management for Accelerating Deep Learning Training. In *2024 IEEE 36th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 13–24. doi:10.1109/SBAC-PAD63648.2024.00010

[33] Hariharan Devarajan, Loïc Pottier, Kaushik Velusamy, Huihuo Zheng, Izzet Yildirim, Olga Kogiou, Weikuan Yu, Anthony Kougkas, Xian-He Sun, Jae Seung Yeom, and Kathryn Mohror. 2024. DFTracer: An Analysis-Friendly Data Flow Tracer for AI-Driven Workflows. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–24. doi:10.1109/SC41406.2024.00023

[34] Hariharan Devarajan, Huihuo Zheng, Anthony Kougkas, Xian-He Sun, and Venkatram Vishwanath. 2021. DLIO: A Data-Centric Benchmark for Scientific Deep Learning Applications. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 81–91. doi:10.1109/CCGrid51090.2021.00018

[35] Bin Dong, Jean Luca Bez, and Suren Byna. 2023. AIIO: Using Artificial Intelligence for Job-Level and Automatic I/O Performance Bottleneck Diagnosis. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23)*. Association for Computing Machinery, New York, NY, USA, 155–167. doi:10.1145/3588195.3592986

[36] Matthieu Dorier, Romain Egele, Prasanna Balaprakash, Jaehoon Koo, Sandeep Madireddy, Srinivasan Ramesh, Allen D. Malony, and Rob Ross. 2022. HPC Storage Service Autotuning Using Variational- Autoencoder -Guided Asynchronous Bayesian Optimization. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 381–393. doi:10.1109/CLUSTER51413.2022.00049

[37] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. 2021. Clairvoyant Prefetching for Distributed Machine Learning I/O. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3458817.3476181

[38] Väinö Hatanpää, Eugene Ku, Jason Stock, Murali Emani, Sam Foreman, Chunyong Jung, Sandeep Madireddy, Tung Nguyen, Varuni Sastry, Ray A. O. Sinurat, Huihuo Zheng, Sam Wheeler, Troy Arcomano, Venkatram Vishwanath, and Rao Kotamarthi. 2025. AERIS: Argonne Earth Systems Model for Reliable and Skillful Predictions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*. Association for Computing Machinery, New York, NY, USA, 72–85. doi:10.1145/3712285.3772094

[39] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. 2021. Highly Accurate Protein Structure Prediction with AlphaFold. *Nature* 596, 7873 (Aug. 2021), 583–589. doi:10.1038/s41586-021-03819-2

[40] Awais Khan, Arnab K. Paul, Christopher Zimmer, Sarp Oral, Sajal Dash, Scott Atchley, and Feiyi Wang. 2022. Hvac: Removing I/O Bottleneck for Large-Scale Deep Learning Applications. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. 324–335. doi:10.1109/CLUSTER51413.2022.00044

[41] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab K. Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali R. Butt. 2023. SHADE: Enable Fundamental Cacheability for Distributed Deep Learning Training. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 135–152. https://www.usenix.org/conference/fast23/presentation/khan

[42] Krishnateja Killamsetty, Guttu Sai Abhishek, Aakriti Lnu, Ganesh Ramakrishnan, Alexandre Evfimievski, Lucian Popa, and Rishabh Iyer. 2022. AUTOMATA: Gradient Based Data Subset Selection for Compute-Efficient Hyper-parameter Tuning. *Advances in Neural Information Processing Systems* 35 (Dec. 2022), 28721–28733. https://proceedings.neurips.cc/paper_files/paper/2022/hash/b8ab7288e7d5aefc695175f22bbddead-Abstract-Conference.html

[43] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

[44] OE Bronson Messer, Ed D'Azevedo, Judy Hill, Wayne Joubert, Mark Berrill, and Christopher Zimmer. 2018. MiniApps derived from production HPC applications using multiple programing models. *The International Journal of High Performance Computing Applications* 32, 4 (2018), 582–593.

[45] Tung Nguyen. 2025. Tung-Nd/Stormer. https://github.com/tung-nd/stormer

[46] Tung Nguyen, Johannes Brandstetter, Ashish Kapoor, Jayesh K. Gupta, and Aditya Grover. 2023. ClimaX: A Foundation Model for Weather and Climate. In *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 25904–25938. https://proceedings.mlr.press/v202/nguyen23a.html

[47] Tung Nguyen, Rohan Shah, Hritik Bansal, Troy Arcomano, Romit Maulik, Veerabhadra Kotamarthi, Ian Foster, Sandeep Madireddy, and Aditya Grover. 2024. Scaling Transformer Neural Networks for Skillful and Reliable Medium-Range Weather Forecasting. In *Advances in Neural Information Processing Systems*, Vol. 37. 68740–68771. https://papers.nips.cc/paper_files/paper/2024/hash/7f19b99e63762d20e9df9144056f1ee-Abstract-Conference.html

[48] Chet Nieter and John R. Cary. 2004. VORPAL: A Versatile Plasma Simulation Code. *J. Comput. Phys.* 196, 2 (May 2004), 448–473. doi:10.1016/j.jcp.2003.11.004

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.

[50] Neeraj Rajesh, Keith Bateman, Jean Luca Bez, Suren Byna, Anthony Kougkas, and Xian-He Sun. 2024. TunIO: An AI-powered Framework for Optimizing HPC I/O. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 494–505. doi:10.1109/IPDPS57955.2024.00050

[51] David Randall, Marat Khairoutdinov, Akio Arakawa, and Wojciech Grabowski. 2003. Breaking the Cloud Parameterization Deadlock. *Bulletin of the American Meteorological Society* 84, 11 (Nov. 2003), 1547–1564. doi:10.1175/BAMS-84-11-1547

[52] Ray A. O. Sinurat, Anurag Daram, Haryadi S. Gunawi, Robert B. Ross, and Sandeep Madireddy. 2023. Towards Continually Learning Application Performance Models. arXiv:2310.16996 [cs] doi:10.48550/arXiv.2310.16996

[53] Arno van Hilten, Sonja Katz, Edoardo Saccenti, Wiro J Niessen, and Gennady V Roshchupkin. 2024. Designing Interpretable Deep Learning Applications for

Functional Genomics: A Quantitative Analysis. *Briefings in Bioinformatics* 25, 5 (Sept. 2024), bbae449. doi:10.1093/bib/bbae449

[54] Meng Wang, Gus Waldspurger, Naufal Ananda, Yuyang Huang, Kemas Wiharja, John Bent, Swaminathan Sundararaman, Vijay Chidambaram, and Haryadi S. Gunawi. 2025. GPEmu: A GPU Emulator for Faster and Cheaper Prototyping and Evaluation of Deep Learning System Research. *Proc. VLDB Endow.* 18, 6 (Feb. 2025), 1919–1932. doi:10.14778/3725688.3725716

[55] Cong Xu, Shane Snyder, Omkar Kulkarni, Vishwanath Venkatesan, Phillip Carns, Surendra Byna, Robert Sisneros, and Kalyana Chadalavada. 2019. *DXT: Darshan eXtended Tracing*. Technical Report. Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA (United States). https://www.osti.gov/biblio/1490709

[56] Jinbo Xu. 2019. Distance-Based Protein Folding Powered by Deep Learning. *Proceedings of the National Academy of Sciences* 116, 34 (Aug. 2019), 16856–16865. doi:10.1073/pnas.1821309116

[57] Chih-Chieh Yang and Guojing Cong. 2019. Accelerating Data Loading in Deep Neural Network Training. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 235–245. doi:10.1109/HiPC.2019.00037

[58] Izzet Yildirim, Hariharan Devarajan, Anthony Kougkas, Xian-He Sun, and Kathryn Mohror. 2025. WisIO: Automated I/O Bottleneck Detection with Multi-Perspective Views for HPC Workflows. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 749–763. doi:10.1145/3721145.3725742

[59] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Springer, Berlin, Heidelberg, 44–60. doi:10.1007/10968987_3

[60] Sungduk Yu, Po-Lun Ma, Balwinder Singh, Sam Silva, and Mike Pritchard. 2024. Two-Step Hyperparameter Optimization Method: Accelerating Hyperparameter Search by Using a Fraction of a Training Dataset. *Artificial Intelligence for the Earth Systems* 3, 1 (Jan. 2024). doi:10.1175/AIES-D-23-0013.1

[61] Tianwei Yue, Yuanxin Wang, Longxiang Zhang, Chunming Gu, Haoru Xue, Wenping Wang, Qi Lyu, and Yujie Dun. 2023. Deep Learning for Genomics: From Early Neural Nets to Modern Large Language Models. *International Journal of Molecular Sciences* 24, 21 (Nov. 2023), 15858. doi:10.3390/ijms242115858

[62] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. 2018. Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 145–156. doi:10.1109/MASCOTS.2018.00023

A SFNO-LG Proxy vs Optimized Breakdown

This appendix provides additional evidence explaining the lower Proxy-to-App transfer accuracy observed for SFNO-LG. After optimization, the full application exhibits substantial reductions in getitem, preprocessing, and compute time, reflecting improved overlap between computation and I/O. In contrast, the proxy executes with minimal data-fetch overhead and does not fully capture the resulting changes in execution balance or resource contention. Consequently, the proxy underestimates the absolute performance gains achieved by the optimized application, while still preserving the relative configuration ordering needed to guide optimization.

Table 6: Execution-time breakdown for SFNO-LG comparing proxy execution and the optimized application.

Component	Proxy (s)	vOPT (s)	Change (%)
Fetch	0.01	1.09	>1000
Getitem	4.29	2.03	−52.7
Preprocess	3.48	0.66	−81.2
Compute	4.97	3.91	−21.4
Total	12.75	7.69	−39.7