Stimulus: Accelerate Data Management for Scientific AI applications in HPC

Hariharan Devarajan[1], Anthony Kougkas[1], Huihuo Zheng[2], Venkatram Vishwanath[2], and Xian-He Sun[1]

hdevarajan@hawk.iit.edu, akougkas@iit.edu, huihuo.zheng@anl.gov, venkat@anl.gov, sun@iit.edu
[1]Illinois Institute of Technology, Chicago
[2]Argonne National Laboratory

***Abstract*—Modern scientific workflows couple simulations with AI-powered analytics by frequently exchanging data to accelerate time-to-science to reduce the complexity of the simulation planes. However, this data exchange is limited in performance and portability due to a lack of support for scientific data formats in AI frameworks. We need a cohesive mechanism to effectively integrate at scale complex scientific data formats such as HDF5, PnetCDF, ADIOS2, GNCF, and Silo into popular AI frameworks such as TensorFlow, PyTorch, and Caffe. To this end, we designed Stimulus, a data management library for ingesting scientific data effectively into the popular AI frameworks. We utilize the *StimOps* functions along with *StimPack* abstraction to enable the integration of scientific data formats with any AI framework. The evaluations show that Stimulus outperforms several large-scale applications with different use-cases such as Cosmic Tagger (consuming HDF5 dataset in PyTorch), Distributed FFN (consuming HDF5 dataset in TensorFlow), and CosmoFlow (converting HDF5 into TFRecord and then consuming that in TensorFlow) by 5.3×, 2.9×, and 1.9× respectively with ideal I/O scalability up to 768 GPUs on the Summit supercomputer. Through Stimulus, we can portably extend existing popular AI frameworks to cohesively support any complex scientific data format and efficiently scale the applications on large-scale supercomputers.**

***Index Terms*—scientific data format, HDF5, TensorFlow, Tensor, Operators, Decoupled I/O, transformation, integration, management, I/O acceleration, HPC**

I. Introduction

Artificial Intelligence (AI) is being applied to solve complex problems in a wide variety of applications. These applications range from image recognition [1], natural language processing [2], autonomous driving [3], and scientific domains such as cosmology [4], materials science [5], and biology [6]. Application developers in scientific domains utilize AI frameworks (e.g., TensorFlow [7], PyTorch [8], and Caffe [9]) on HPC systems to solve a common class of problems such as clustering data based on features and numerical regression optimizations. These scientific applications are coupled with traditional scientific computing simulations [10] (also known as inner-loop modeling). Traditional scientific computing enhanced with AI-based knowledge acceleration has the potential to increase the performance and throughput of inner-loop modeling [11]. Efficient data coupling requires cohesive data exchange between scientific simulations and AI applications to drive the advancement of scientific discoveries in many domains.

Cohesive data exchange between science and AI requires interaction between the scientific data formats utilized by traditional simulations and AI frameworks. Traditional simulations produce petabytes of data in scientific data formats such as HDF5 [12], PnetCDF [13], ADIOS [14], Silo [15], and GNCF [16]. On the contrary, AI frameworks are designed and optimized to utilize custom data formats such as TFRecord [7], PyTorch-Dataset [8], and LMDB format [9]. To achieve efficient data exchange [17] there are three possible approaches. Firstly, simulations can produce data in AI formats. This is undesirable as legacy simulations are fine-tuned for scientific formats [18]. Secondly, scientific data can be converted into AI formats. This approach is highly cost [19] and space [20] prohibitive as petabytes of data have to be converted and stored on the global file system. Lastly, AI frameworks can be made compatible with scientific data formats. This approach seems promising as it could enable an efficient cohesive integration of scientific data formats within AI frameworks.

Cohesive integration of scientific data formats into popular AI frameworks should consider performance as well as portability. Most application developers manually load data using the native data APIs [21], [22] with no control over operation pipelining, leading to non-scalable and inefficient I/O. To improve this, scientists performed this integration by consuming scientific data as a part of the graph execution of the AI framework [23] at the API level. This approach lacks *performance* because functions are defined in python as a combination of existing operators at the API level. It prohibits the application developer from having fine-grained control over the pipeline, resulting in multiple data copies and missed optimization opportunities such as aggregation and caching [23], [24]. The approach also lacks *portability* as these implementations are tightly coupled with the target scientific data format and AI framework. This strong coupling is undesirable as application developers often implement their models on different AI frameworks to target specialized AI hardware [25] such as Cerebras, GraphCore, Groq, etc. Many of these novel hardware support only certain frameworks. Additionally, scientific data representation is not compatible with the AI framework's tensor. For every data format and AI framework combination, a strongly coupled approach would have to re-implement common functionalities and optimizations leading to duplication of efforts across frameworks, more points of failure or errors, and lower quality of service. Therefore, it is necessary to have a new *portable and performant* approach to integrate scientific data formats within popular AI frameworks.

978-1-6654-9956-9/22/$31.00 ©2022 IEEE
DOI 10.1109/CCGrid54584.2022.00020

We designed Stimulus, a data management framework for cohesively integrating scientific data formats within an AI framework to address the portability and performance challenge that exists in current HPC-AI workflows. To achieve these goals, we introduce two novel concepts: the *StimPack* abstraction and *StimOps* functions. For *portability*, *StimPack* unifies several data formats under a single data abstraction and *StimOps* makes *StimPack* compatible with any popular AI framework. For *performance*, *StimPack* utilizes scientific data format APIs which efficiently decouple I/O from sample processing whereas *StimOps* manages the pipelining and parallelism of the input pipeline to maximize I/O performance. Additionally, *StimPack* functions are compatible with existing tensor operators for scientific data input pipelines (e.g., `batch`) as well as existing optimizations for maximizing performance, such as prefetching and caching. The utilization of *StimPack* abstraction and *StimOps* functions enables Stimulus to cohesively integrate several scientific data formats with many popular AI frameworks. We use HDF5 with TensorFlow and PyTorch on the Summit supercomputer as our flagship use-case as these choices of library and framework are popular in HPC environments [26], [27]. The contributions of this work include:

1) Design of the Stimulus architecture with a generalized input pipeline for scientific data formats that are compatible with production AI frameworks (Section III).
2) Design of StimPack abstractions to efficiently represent popular scientific data formats within AI frameworks (Section III-E).
3) Design and implementation of tensor operators for Stimulus' input pipeline to enable efficient data ingestion from scientific data formats (Subsection III-D).
4) Illustration of the performance impact of cohesive and data-centric integration of scientific formats in AI frameworks on the Summit supercomputer (Section IV).

II. Background and Motivation

AI has revolutionized scientific computing by solving several complex problems in the domain of physics [28], cosmology [19], materials science [5], meteorology [29], and biology [22]. These applications utilize popular AI frameworks such as TensorFlow and PyTorch and consume simulation data stored in the different scientific data formats such as HDF5, PnetCDF, etc.

In high-energy physics (HEP), detecting the signal of a new particle using AI models is a common use-case [30]. The application uses Caffe with 15 TB of structured scientific data in *HDF5 data format* with dozens of channels/variables and 6.4M events. Similarly, in meteorology, AI models are used to understand extreme weather life cycles and predict their future trends [31], [32]. These applications are implemented in TensorFlow and consume terabyte datasets stored in HDF5 and PnetCDF data format. Additionally, in the field of cosmology, scientists utilize AI to determine the distribution of matter in the universe [21], [33]. The applications utilize PyTorch and TensorFlow as their AI framework and consume terabytes of datasets stored in sparse HDF5 data format. Finally, in the field of 3D image segmentation, AI models are utilized to robustly segment images with an unknown and variable number of objects and highly variable object sizes [22]. In this application, the AI model uses novel flood-filling networks implemented in TensorFlow and consumes a sparse HDF5 dataset with separate metadata and data files.

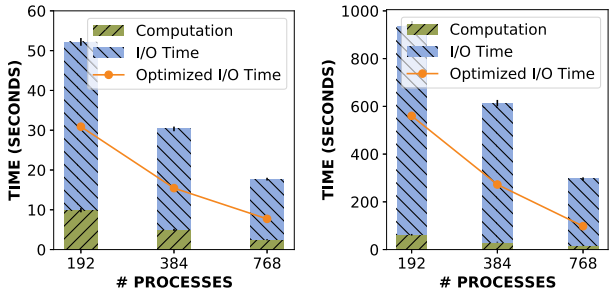
A. Motivation

The above examples demonstrate that popular AI frameworks such as TensorFlow and PyTorch are extensively used to build science models. Additionally, these models consume data in scientific formats such as HDF5, PnetCDF, silo, ADIOS, and GNCF because these datasets are generated by scientific simulations that have been fine-tuned to efficiently generate data into these formats.

1) Need for Portability: The diversity of using popular cloud AI frameworks in HPC systems to consume various scientific data formats motivates us to build a portable solution that applies to any popular AI framework and any scientific data format. This is even more critical from the user perspective as many novel hardware (e.g., Cerebras, GraphCore, and Groq) are supported by only certain frameworks. For instance, the Cerebras AI chip compresses the compute power of 850,000 AI-optimized cores onto a single chip and achieves 4.3× acceleration [25]. This chip supports APIs only as an extension of TensorFlow for integrating the chip with applications. Another example is the Goya Inference Processor, which is based on the scalable architecture of Habana's proprietary Tensor-Processing Core (TPC) and includes a cluster of eight programmable cores [34]. This chip supports APIs as an extension of PyTorch framework. In these examples, when users port their models to different AI hardware, having a portable data management library across different frameworks reduces redundant and error-prone development efforts.

2) Need for Performance: Scientific data format currently is consumed using custom tightly coupled solutions that integrate a particular dataset into a specific AI framework. We tested the applications on the Summit supercomputer at ORNL [35] and scaled it from 32 nodes to 128 nodes with 6 processes (one for each GPU) per node with a 2 TB dataset (strong scaling). Also, the "Optimized I/O" is calculated based on manual decoupled I/O, overlapping of I/O and compute, and increased I/O parallelism through multi-threading within each process for all use-cases.

Use-case 1: TensorFlow with native HDF5 APIs. Distributed Flood Filling Networks (DFFN) [22] are built inTensorFlow and consume HDF5 files using manual data ingestion and h5py APIs. In DFFN, the dataset is read by the application using the `HDF5` library (as TF does not natively support the HDF5 data format). Next, the data is processed using Python functions provided by TF and then fed to the training phase using the tensor representation. The percentage of I/O to total time increases (from 26% to 43% for the largest scale) as the application scale increases, moving the I/O cost of the application farther away from the optimal I/O time (Figure 1(a)). Through optimal I/O, we can achieve a speedup



(a) Distributed FFN (TensorFlow). (b) Cosmic Tagger (PyTorch).

Fig. 1. Potential I/O improvement through cohesive integration in existing applications of up to 2.9x-3.4x when we ideally parallelize, overlap, and maximize PFS bandwidth.

of 2.9x on the largest scale. This is because each process in the application currently performs I/O and processes data using a single thread without any I/O optimization such as parallelism, pipelining, or prefetching since the HDF5 calls and AI-model computations are not executed as part of the AI framework's graph execution.

Use-case 2: PyTorch with data loader using HDF5 APIs. Cosmic Tagging with UNet [21] is implemented in PyTorch and consumes HDF5 dataset using the `DataLoader` framework, which implements the input pipeline at the API level. The percentage of I/O over overall time increases (up to 90% for the largest scale) with application scale, moving the I/O cost of the application farther away from the optimal I/O time (Figure 1(b)). HDF5 I/O using DataLoader framework in PyTorch is performed per sample. As the image samples are small, the application cannot extract maximum bandwidth from GPFS. Instead, the "Optimized I/O" decouples and parallelizes the I/O workers to achieve optimal PFS bandwidth, which increases I/O bandwidth by 3.4×. Additionally, the integration at API level (using DataLoader) is not portable to TensorFlow or Caffe as they do not utilize DataLoader abstraction.

Currently scientific AI applications tightly couple data ingestion with the AI framework, which impacts both *performance* and *portability* (as observed from these use-cases). This existing situation motivates a move towards a more decoupled and generic solution that could cohesively integrate scientific data formats with popular AI frameworks.

III. Stimulus

Stimulus is a data management library for scientific AI applications in High-Performance Computing (HPC) environments. The primary goals of Stimulus are to achieve a portable and performant integration of scientific data formats within popular AI frameworks. We introduce two novel concepts in Stimulus to achieve these goals, namely, the *StimPack* abstraction and *StimOps* functions. The *StimPack* abstraction unifies several scientific data formats under a simple interface that masks the implementation complexity of individual data formats. On the other hand, the *StimOps* functions provide a generic data ingestion pipeline that can be executed by any tensor-operator-based framework such as TensorFlow, PyTorch, or Caffe. Stimulus achieves portability using these two ideas, as the *StimPack* abstraction unifies several scientific

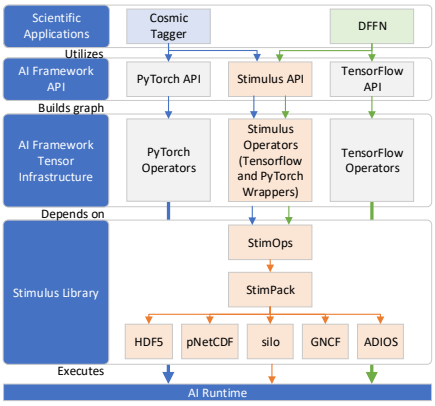


Fig. 2. Stimulus integrates with the existing AI framework at an API level and tensor infrastructure level. All of the data management features provided by StimOps and StimPack are packed into the Stimulus library. The Stimulus API and the Stimulus Operators act as a facade layer to invoke routines from the Stimulus library. The StimOps are executed as a part of the overall execution graph of the AI workflow.

data formats under a single interface and the *StimOps* functions make the input pipeline for scientific data generic across any popular AI framework. Additionally, Stimulus achieves performance using the *StimPack* abstraction, as the implementation includes efficient usage of scientific data format APIs and optimization flags to maximize I/O performance. Also, the *StimOps* functions enable fine-grained control over the input pipeline at a lower-level (i.e., C++ implementations), enabling I/O optimizations such as decoupling of I/O and sample processing, operation parallelism, and operation pipelining.

A. Architecture

To achieve the portability and performance objectives, Stimulus defines *StimOps* functions and the *StimPack* abstraction to build a solution that efficiently integrates several scientific data formats with popular AI frameworks. Stimulus integrates with existing AI frameworks at an API level and internal tensor infrastructure level to seamlessly integrate existing toolkits in a typical HPC-AI software stack. To provide modularity, all the Stimulus functionality (i.e., StimPack abstractions and StimOps functions) are packed into the Stimulus Library (i.e., a .so file), which is used as a system dependency by the AI frameworks. This means we provide hooks within the existing AI framework infrastructure to act as a skeleton to call the Stimulus library to perform data management for scientific AI applications. This architecture can be presented with software interaction and examples of scientific AI applications (Figure 2). The Cosmic Tagger application (blue line in the figure), which utilizes PyTorch APIs to build the input pipeline and construct the AI model, consumes the HDF5 dataset using the Stimulus APIs. The Stimulus APIs define the input pipeline as a collection of tensors implemented at the custom tensor infrastructure level. Using this mechanism, the input pipeline defined by Stimulus is inserted into the execution graph of the AI framework and therefore is run cohesively by the underlying runtime. The Stimulus Operators defined at the infrastructure level act as a facade layer to invoke StimOps

```python
import stimulus.tensorflow as st
import tensorflow as tf
# Load HDF5 Files with Stimulus
files = [("/pfs/images_1_2.h5",image1_ds, label1_ds, ("FIXED", 4096)),
         ("/pfs/images_2_2.h5",image2_ds, label2_ds, ("FIXED", 8192))]
ds = tf.Dataset.from_slices(files)
ds = ds.interleave(lambda x: st.HDF5(x,
                        transfer_size=1048576,
                        read_threads=4),
                cycle_length=4, block_length=16)
...
for image_batch, label_batch in ds:
    train(image_batch, label_batch)
```

(a) Tensorflow Example with HDF5

```python
import stimulus.pytorch as st
import torch
from torch.utils.data import DataLoader
# Load ADIOS2 File with Stimulus
dataset = st.ADIOS2("/pfs/images_1_2.bp",image1_var, label1_var, ("FIXED", 4096))
data_loader = DataLoader(dataset, batch_size=4, shuffle=True)
for imgs, labels in data_loader:
    train(image_batch, label_batch)
```

(b) PyTorch Example with ADIOS2

Fig. 3. Stimulus API integrated with TensorFlow and PyTorch. The functions are compatible with dataset APIs for both frameworks and can be used with other input pipeline operators such as batch, filter, and shuffle.

functions from the Stimulus Library (4th layer in the figure). The StimOps utilize the StimPack abstraction to access the requested data format to perform I/O and generate samples during runtime. The Stimulus operators (at the 3rd layer) also manage parallelism and pipelining details to maximize the performance of the graph execution within the AI runtime. A similar flow is presented for TensorFlow applications (e.g., DFFN green arrows in the figure). Essentially, through Stimulus, the AI framework is enhanced with input pipeline operators and scientific data format implementations while maintaining portability and performance.

B. Stimulus API Integration

The Stimulus API (Figure 2 level 2 available at https://github.com/scs-lab/stimulus) is developed in Python as most scientific AI applications built with TensorFlow, PyTorch, and Caffe are implemented using the Python API. The Stimulus API is designed as an independent Python module installed as a typical python package using pip or setup.py file. The users can install stimulus for a specific AI framework or all AI frameworks based on the application. Once installed, Stimulus can be imported for a particular framework as *import stimulus.<framework> as st* (Figure 3). Once imported in the application, we could utilize the data format classes (namely, `HDF5`, `NC`, `Silo`, `GNCG`, and `ADIOS2`) to define the input sources. Each of these classes require a tuple as input containing the `filename`, `dataset_name`, `label_name`, and `sample_boundaries`. These attributes are defined by the *StimPack* abstraction described later. Additionally, Stimulus API takes user inputs of I/O transfer size and read parallelism per process. The I/O transfer size determines the granularity of I/O performed by the *Scientific Format I/O Operator*. In contrast, the read parallelism parameter determines the number of parallel threads used to perform I/O. In our experience, the transfer size should be equal to the parallel file system's stripe size (e.g., Lustre), and the read parallelism threads should be equal to $\frac{number\_of\_cores}{number\_of\_processes}$ (per node) to maximize CPU utilization and I/O performance. Stimulus' APIs can be used by the user with existing input pipeline operators such as batch, shuffle, iterators, etc. (Figure 3). As demonstrated in the examples, it is extremely intuitive and easy to integrate scientific datasets and AI frameworks using Stimulus APIs.

C. Stimulus Operators Facade

The Stimulus Operators (Figure 2 level 3) act as a facade to invoke implementations of *StimOps* functions. The operators utilize the custom tensor infrastructure from TensorFlow [36], PyTorch [37], and Caffe2 [38]. The facade contains an AI framework-specific interface to define custom operators for the input pipeline. The implementation of the interface is provided by the StimOps functions using the Proxy design pattern. Using these software engineering techniques, we can separate the definitions and declarations of the input pipeline and enable code reuse and modularization. Each facade for an AI framework has interfaces for three operators: Scientific Format I/O Operator, Input Sample Creator, and Sample Converter. These operators are implemented as a part of Stimulus Library and will be discussed in the next subsection.

D. StimOps functions

The input pipeline is defined at the Stimulus API layer but implemented within the StimOps functions. The input pipeline is defined as a three-operator graph: Scientific Format I/O Operator, Input Sample Creator, and Sample Converter. The *Scientific Format I/O Operator* defines a routine to read data from different scientific formats. Here, the operator utilizes the StimOps functions from the Stimulus library, which uses the StimPack abstraction to perform I/O. The *Input Sample Creator* converts the data read from the scientific format (in the form of bytes) into samples (in a high-dimensional matrix). Finally, the *Sample Converter* converts the memory representation of a scientific data format into a tensor representation. This operator enables the integration of Stimulus operators and existing AI framework operators. The purpose of using three operators is to enable pipelining and parallelism. In general, for a given I/O request from the data source, the operators are executed sequentially. However, these reads can be pipelined with other reads for creating a deep input pipeline for AI models. Additionally, reads from different parts of the file are parallelized to maximize I/O bandwidth.

1) Portability: The StimOps functions are agnostic of AI framework-specific implementations. We utilize the Stimulus Operators Facade (discussed in previous Subsection III-C) to interface them to the StimOps functions. The StimOps operator is defined as a generic template which is specialized at compile time to any tensor-based AI framework (Figure 4). The StimOps class takes a TENSOR template parameter, representing the data communication mechanism in AI frameworks. This TENSOR template parameter is set to tensorflow::Tensor or torch::Tensor by the Stimulus Operators Facade at compile time for TensorFlow and PyTorch. There are three simple steps for each operator: *a) convert tensor objects to C++*: Tensor infrastructure communicates data as tensor objects. We need to deserialize them into structures to utilize

```
template<typename TENSOR>
class StimOps{
    TENSOR ScientificIOOperator(TENSOR dataset_t, TENSOR options_t) {
        // convert tensor into C++ objects as args
        std::tuple args = extract(IO_OPERATOR, options_t);
        // call stimulus library
        stimulus::Data d = ScientificIOOperatorFactory.GetOp(args).execute();
        // pack output into tensor
        torch::Tensor output = convert(d);
        return output
    }
    TENSOR InputSampleCreator(TENSOR dataset_t, TENSOR options_t) {
        // convert tensor into C++ objects as args
        std::tuple args = extract(SAMPLE_CREATOR, options_t);
        // call stimulus library
        auto operator = InputSampleCreatorFactory.GetOp(args).execute();
        // pack output into tensor
        torch::Tensor output = convert(d);
        return output
    }
    TENSOR SampleConverter(TENSOR dataset_t, TENSOR options_t) {
        // convert tensor into C++ objects as args
        std::tuple args = extract(SAMPLE_CONVERTER, options_t);
        // call stimulus library
        torch::Tensor output = SampleConverterFactory.GetOp(args).execute();
        return output
    }
}
```

Fig. 4. Pseudocode of StimOps Implementation for any tensor-based AI framework. It utilizes the template pattern and are specialized based on invocation from the Stimulus Operators Facade.

them within Stimulus. To incur a low overhead of serialization/deserialization, we utilize byte array serialization of the structures. This is extremely fast, and no additional memory is required. *b) Invoke the operations within the Stimulus library*: Here, we utilize the factory pattern to select the appropriate data format implementation of the operator from the *StimPack* abstraction. Finally, *c) repack the output into tensor objects*: This is done using byte array serialization. Using these three generic steps, we can define the StimOps functions for any tensor-based AI framework (Figure 4). Finally, converting scientific data into tensor objects using Sample Converter enables users to combine Stimulus with existing input pipeline operations such as batch, filter, transformations, and iterations.

2) Performance: Splitting the input pipeline into three operators has four performance benefits. First, the operators of the pipeline are independent of each data element in the file. This enables Stimulus to execute a deep pipeline within the AI runtime to ensure the data ingestion rate matches the GPU computation. Second, the operators acting on different data elements can be efficiently parallelized based on CPU-core availability. Third, the operators decouple I/O from sample creations for the input pipeline. The read granularity of data from data sources such as PFS, Burst Buffers, or node-local devices does not match the sample size, which is often small. This enables Stimulus to extract maximum performance from the underlying data source. Finally, the converter operator transforms the scientific data into Tensor objects. This enables users to enhance the Stimulus pipeline with existing optimizations within AI frameworks. For instance, we have combined Stimulus with data optimizations such as prefetching and caching from TensorFlow data pipeline to further optimize the input pipeline without reimplementing these routines within Stimulus. This shows the power of a modular and decoupled input pipeline for scientific data formats.

E. StimPack Abstraction

The StimPack abstraction is designed in low-level C++ language. The StimPack abstraction represents popular scientific data formats such as HDF5, PnetCDF, ADIOS2, GNCF, and Silo. The abstraction consists of an interface (i.e., C++ abstract class) with predefined methods and attributes common across all scientific data formats. The `StimPack Dataset` is a common representation for any scientific format. Based on our investigation, all scientific formats utilize a structured multi-dimensional representation to store data. For example, we have *Dataset* for HDF5 and PnetCDF, *Attributes* in ADIOS2, and *n-dimensional Mesh* in Silo. The `StimPack Dataset` consists of four fields, *name*: represents filename or object name, *field_name* and *label_name*: represents dataset name, attribute name, or mesh name corresponding to where data and label is stored respectively, and finally *sample_boundary*: a map of all boundaries of the sample. Stimulus advocates to have a common API (as proposed by many libraries such as Keras [39]) to build I/O optimizations in a portable fashion. In most cases, the predefined classes should be sufficient to achieve their intended purpose but application developers can further extend the `StimPack` abstraction by building a custom interface.

Every scientific data format within StimPack contains three functions: ScientificIOOperator, InputSampleCreator, and the SampleConverter. The ScientificIOOperator function performs the I/O from the data source for each data format based on the provided transfer size using native scientific data format APIs. The InputSampleCreator function splits the binary form of the data into individual samples. The operator requires the user's knowledge of the sample's boundaries within the dataset to extract the samples. Users can define these boundaries in two modes: FIXED (i.e., each sample is of fixed size given by the user) or VARIABLE (i.e., the user provides a sample index map and its range within the dataset). The VARIABLE sample index enables users to set their custom boundaries to define and construct a sample. The user sets this information during the definition of the input pipeline (example in Figure 3 Line 4). Finally, the SampleConverter function transforms the scientific data format's data sample representation into a tensor compatible format. The in-memory representation of a scientific data sample depends on the format. For instance, HDF5 dataset samples are represented as a multi-dimensional array with a start, end, and stride sizes; ADIOS2 samples are represented as a single-dimensional array in row-major format; Silo samples are represented in a custom data structure such as DBquadmesh with an API to extract information. This demands special care when we need to convert a sample into a tensor object. We need to utilize scientific data format APIs to achieve this transformation. The steps to achieve this conversion are given as follows. First, the Sample Converter needs to convert custom samples into a standardized multi-dimensional array in memory. Then, we can convert the in-memory array into a tensor object using AI framework operators such as `convert_to_tensor` in TensorFlow, `torch.tensor` in PyTorch, and `workspace.FeedBlob` in Caffe. This process converts existing in-memory data into a tensor object using the same data pointers in C++. The approach achieves a zero-copy conversion from in-memory

```cpp
// HDF5 implementation
template<typename TENSOR>
class HDF5: public StimPack<TENSOR>{
    stimulus::Data ScientificIOOperator(stimulus::dataset dataset_t){
        /* allocate arrays for hyperslab */
        ...
        hid_t file = H5Fopen(dataset_t.name_.c_str(), H5F_ACC_RDONLY, H5P_DEFAULT);
        hid_t dataset = H5Dopen(file, dataset_t.field_name.c_str(), H5P_DEFAULT);
        /* find rank, element_size, and elements_per_dim from dataset */
        ...
        size_t num_elements = transfer_size_ / (element_size * elements_per_dim);
        size_t start = element_index_ * num_elements;
        /* define memory and file hyperslabs */
        ...
        Data return_data;
        /* Read data from file hyperslab into memory hyperslab into the
           allocated return_data.buffer_ */
        status = H5Dread(dataset, H5T_NATIVE_INT, memspace, dataspace, H5P_DEFAULT,
                         return_data.buffer_);
        /* Close HDF5 datastructure and set booking values */
        return return_data;
    }
    stimulus::Data InputSampleCreator(stimulus::Data data_t,
                                      stimulus::dataset dataset_t) {
        Data sample;
        /* for each sample calculate sample offset and size. This is linear
           calculation for FIXED and iteration over map for VARIABLE */
        for(auto sb: sample_boundaries){
            sample.buffer_ = malloc(sb.size);
            sample.size_ = sb.size;
            memcpy(sample.buffer_, (char*)data_t.buffer_ + sb.offset, sb.size);
            co_yield sample;
        }
        co_return sample;
    }
    TENSOR SampleConverter(stimulus::Data data_t) {
        /* allocate output tensor based on TENSOR APIs */
        auto output_flat = output_tensor->flat<int32>();
        if (output_flat.size() > 0)
            memcpy(output_flat.data(), data_t.buffer_,data_t.size_);
        return output_tensor;
    }
}
```

(a) HDF5 Implementation of StimPack

```cpp
// ADIOS2 implementation
template<typename TENSOR>
class ADIOS2: public StimPack<TENSOR>{
    stimulus::Data ScientificIOOperator(stimulus::dataset dataset_t){
        auto bpReader = bpIO.Open(dataset_t.name_, adios2::Mode::Read);
        /* calculate read boundaries and element size from variable*/
        num_elements = transfer_size/element_size;
        start = source.index * num_elements;
        end = (source.index + 1) * num_elements;
        /* select the appropriate variable into bpData*/
        Data return_data;
        bpReader.Get(bpData, return_data.buffer_);
        /* Close ADIOS datastructures and set booking values */
        return return_data;
    }
    ...
}
```

(b) ADIOS2 Implementation of StimPack

Fig. 5. Implementation of StimPack abstraction for HDF5 and ADIOS2. The APIs are utilized to efficiently read data from the source and convert in-memory representations into tensor objects. We utilize high-performance co-routine calls to optimize control flow for the AI framework's runtime.

data format representation to tensor representation.

1) Portability: The StimPack abstraction described above is the abstract class that is implemented by scientific data formats. We have defined a generic function as a template that provides hooks for implementing data format-specific operations. For data formats with multiple datasets or multi-variate datasets within the file, the different data types/datasets can be chained together by defining different data sources at the Stimulus API level with the same filename but different dataset/attribute names. The StimPack abstraction represents these structured scientific formats accurately. We present the implementation details for HDF5 and ADIOS2 in Figure 5. The critical thing to note is that we build routines to optimally consume data from scientific formats and re-use this implementation across any AI framework. Other data formats are implemented as a part of our repository.

2) Performance: StimPack alleviates for users the burden of building their data ingestion pipeline from scratch. Users define their dataset and optimization parameters (e.g., transfer size, read parallelism, etc.) based on the underlying HPC system, and Stimulus takes care of efficient data ingestion from scientific data. Users do not worry about complicated concepts such as hyperslab, chunking, co-routines, compression, or prefetching. Instead, they define their jobs at a high level, and the system optimizes the input pipeline transparently. The read within the *ScientificIOOperator* function uses data sharding to enable data parallelism for AI frameworks [40]. Each thread reads a part of the overall data in the AI application based on the size of the data. Additionally, if the transfer size does not match the sample boundaries, we read the largest number of samples that can fit the given transfer size. Finally, Stimulus also identifies special cases such as small files, irregular sample boundaries, etc., and optimizes the system through the StimPack abstraction. For instance, for small HDF5 files, Stimulus uses the H5LT library to load the HDF5 file into memory and remove the data access penalty from the PFS. The standardization of the interface and transparent optimizations enable users to get performance from scientific data format in a portable manner.

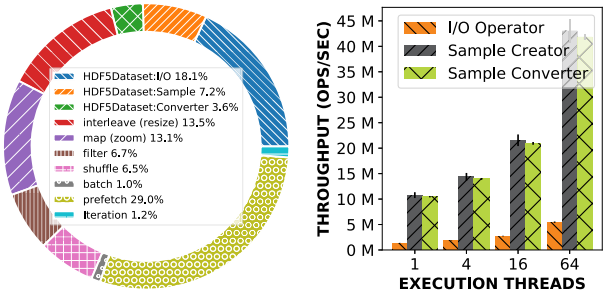
IV. Evaluations

A. Methodology

To evaluate the effectiveness of Stimulus' design, we first showcase the internal performance of StimOps functions and StimPack abstractions. We then test the end-to-end performance for AI applications such as Cosmic Tagging, Cosmoflow, and Distributed Flood Filling Networks (DFFN) to showcase Stimulus' overall impact in HPC environments. We run these tests five times, and the variance in the data is noted in the figures.

1) Testbed: The Summit supercomputer [35] consists of 4608 nodes, each equipped with two IBM Power 9 CPUs (total 44 cores) and 6 NVIDIA Volta GPUs (V100) with 16 GB HBM2 memory. Each Power 9 CPU is connected to 3 Volta GPUs using NVIDIA high-speed interconnect NVLink, capable of 300 GB/s bi-directional bandwidth. Each node has 512 GB of system memory. Dual rail EDR Infiniband cards connect all the nodes using a non-blocking fat-tree topology. The nodes can access a POSIX-based IBM Spectrum Scale parallel file system with a current capacity of 3 PB and an approximate maximum speed of 30 GB/s. We utilized 128 computer nodes (i.e., 768 Volta GPUs) for the largest scale in our evaluations.

2) Software Used: We used the TensorFlow profiler to measure the benchmark's performance. Additionally, we used the VaniDL analysis tool [41], which provides high-level aggregated I/O insights into a traced application. We used TensorFlow 2.1.0 and PyTorch 1.7 with Horovod 0.19.5 for



(a) Anatomy. (b) Performance.

Fig. 6. Input Pipeline: a) Most time is spent in I/O, and operators execute with pipelining and parallelism. b) Throughput of the input pipeline in a single node with 64 threads is 43M ops/s for Input Sample Creator and Sample Converter and 5M ops/s for Scientific Format I/O Operator.

distributed training. Finally, we used NumPy version 1.19.1, h5py version 2.10.0, and mpi4py version 3.0.3.

3) Applications: We use a collection of synthetic benchmarks to showcase different performance metrics on the designed components. We use the DLIO benchmark [42], a representative of scientific deep learning applications in HPC systems. Additionally, we utilize three scientific AI application kernels: Neutrino and Cosmic Tagging with UNet [21], Distributed Flood Filling Networks (DFFN) [22], and Cosmoflow [19] using DLIO Benchmark Suite [42].

B. Scientific Format Input Pipeline

To showcase the performance, we first do a break-down of all operations supported within the Input Pipeline and then measure their end-to-end performance for different use-cases (Figure 6 and 7).

1) Anatomy of Scientific Format Input Pipeline: Scientific Format Input Pipeline combines StimOps functions and TensorFlow's input pipeline operators defined by the user. Therefore, it is crucial to measure its performance across various APIs supported in this input pipeline. To test this, we build a simple input pipeline with a dataset of 1024 samples, each of size 1 MB. We build an input pipeline similar to Figure 3. We run the synthetic workload with a batch size of 64 samples over 16 steps and 1000 epochs. We calculate the time using Tensorflow Profiler for each operation and calculate the average time.The *HDF5Dataset* marked in Figure 6(a) is the high-level API provided by Stimulus, which contains the StimOps functions. The operations *HDF5Dataset:I/O* and *prefetch* are I/O operations, and hence they have the maximum cost. In this case, they perform almost the same number of I/O operations. The cost of data processing operations, such as HDF5Dataset:Sample, HDF5Dataset:Converter, interleave, map, filter, batch, and shuffle, depend on the operation type. These operations are performed on data already in-memory and hence are memory-bound in performance. Finally, generating the Iterator is an in-memory operation of existing data to the training loop and therefore has a relatively low cost similar to a batch operation. This result shows a general distribution of the scientific format input pipeline cost in AI applications.

2) Performance of StimOps functions: Scientific Format Input Pipeline should have a high-performance throughput for each of the StimOps functions. The throughput of these functions depends upon the availability of threads within the execution pipeline. To evaluate their performance, we execute each Stimulus operator with different threads over 10000 operations in a single node. We calculate each operation's throughput within our system in operations per second. The I/O Operator is bound by the GPFS file system's performance, whereas the node's memory performance bounds the sample Creator and Converter (Figure 6(b)). Therefore, we see a significant throughput difference between the I/O operator and memory operators. Additionally, each operator's throughput increases as we have more execution threads since these operations can occur in-parallel and will extract higher performance from the system. Generally, the input pipeline graph is executed over CPU cores and the AI model over GPUs. Therefore, the results show that the implementation efficiently utilizes multi-core CPUs.

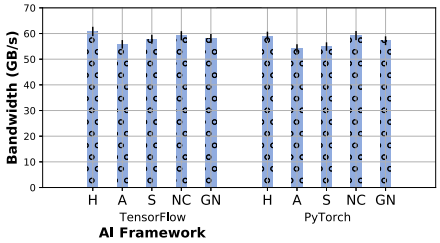
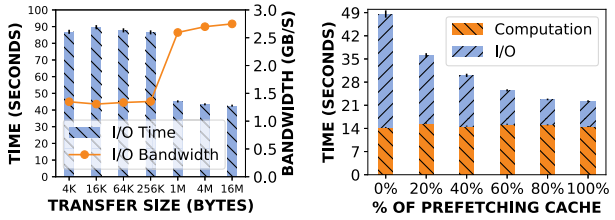


Fig. 7. Performance of StimPack data formats for TensorFlow and PyTorch is comparable to an IOR baseline on the PFS. Stimulus can achieve up to 98% of the PFS bandwidth.

3) Performance of StimPack abstraction: The performance of our StimPack abstractions for various data formats is crucial for maximizing the performance of the AI framework. To evaluate their performance, we execute the input pipeline of reading a 32GB dataset with a transfer size of 1MB, with each dataset containing samples of size 128 KB. We perform this test across HDF5 (H), ADIOS2 (A), Silo (S), PnetCDF (NC), and GNCF (GN) with TensorFlow and PyTorch framework on 32 nodes. For each data format, we calculate the extracted bandwidth from the PFS. As a baseline, we performed 32GB reads using a transfer size of 1MB from the PFS using the IOR benchmark and observed the maximum read bandwidth of 60 GB/s. The StimPack test shows that StimPack implementation of scientific data formats achieves an average bandwidth of 57.60 GB/s for both PyTorch and TensorFlow (Figure 6(a)). The distribution in performance among data formats is an average of 58.31 and std of 1.65 for TensorFlow and an average of 56.89 and std of 2.03 for PyTorch. This is due to the performance difference between the APIs of individual scientific data formats. Additionally, we observe that TensorFlow achieves an 8-10% better performance across all the tests than PyTorch (average over 10 executions). This difference can be accounted for based on the tensor execution engine difference between the two AI frameworks. Overall, Stimulus achieves 95-98% of the overall PFS bandwidth achieved by IOR.

C. Impact of Data and Processing Decoupling

Data loading and pre-processing decoupling are essential in an AI application. This is because samples for AI training



(a) Data and Processing Decoupling. (b) Data Prefetching.

Fig. 8. Data Pipeline Optimizations. a) Decoupling optimization can achieve up to 2× better performance by maximizing GPFS file system bandwidth. b) Prefetching can mask up to 70% of the overall I/O cost.

are generally in kilobytes, which is extremely inefficient data access for large parallel file systems such as GPFS. Stimulus separates data reading (using Scientific Format I/O Operator) with sample pre-processing (using Input Sample Creator) to optimize this, as explained in III-D. To test this optimization, we vary the data reading granularity (i.e., data transfer size) from 4 KB to 16 MB by step size four. We use a synthetic benchmark with 32 K samples for each of these cases, each of size 8 KB, and a batch size of 4 images. The benchmark runs for 1000 steps, and we measure the total time for performing I/O and the aggregate bandwidth achieved in each case. We run the benchmark over 128 nodes with four processes per node. Figure 8(a) shows the results. On the x-axis, we have varying transfer sizes; on the y-axis we have time in seconds, and the y2-axis represents bandwidth in GB/s. We see that until the transfer size matches the GPFS Stripe size of 1 MB, we have low bandwidth of 1.2 GB/s. Once we match the transfer size perfectly, the application can achieve a peak I/O bandwidth of 2.7 GB/s per node and a total aggregated bandwidth of 240 GB/s (peak I/O bandwidth of Summit). This demonstrates the importance of matching data access granularity with the file system to achieve the application's best performance.

D. Impact of Data Prefetching

As Stimulus follows a cohesive integration within AI frameworks, it can utilize existing input pipeline optimizations. Data Prefetching is an essential optimization in TensorFlow for AI applications [43]. It enables efficient overlapping of I/O with model computations by reading data beforehand. However, as shown in many studies [44], [45], [46], data prefetching efficiency depends on the amount of prefetching cache allowed in the system. We use a synthetic benchmark with a dataset with 32 K samples to test this, each of 8 KB and a batch size of 4 samples. We set the prefetching cache as a percentage of overall I/O and measure the I/O performance as time. We observe that, as we increase the prefetching cache, the I/O performance increases (Figure 8(b)). This is because more and more data is already found in the cache's memory. However, the benefit reduces for larger prefetching cache size as models are bound with initial data that needs to be brought in. Specifically, we see little benefit in performance after 70% of the data is already cached. Overall, enabling existing prefetching optimization improves I/O time by 4.42× for 70% prefetching cache.

E. HDF5 AI Application Performance

This section demonstrates the effectiveness of Stimulus to optimize end-to-end scientific AI applications' performance. Specifically, we strong-scale test Neutrino and Cosmic Tagging with UNet (Cosmic Tagger), Distributed Flood Filling Networks (DFFN), and Cosmoflow. The input pipeline is executed over the CPU, and the computations occur over the GPU. In Stimulus, we four-thread I/O and preprocessing parallelism along with the prefetching optimization.

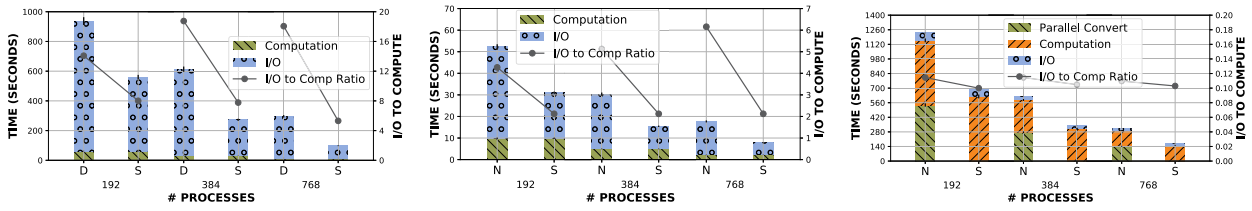
1) CosmicTagger: Cosmic Tagger is a convolutional network for separating cosmic, background, and neutrino in a neutrino dataset. The application is written with the PyTorch framework and reads a 10 TB dataset stored in HDF5 format using the DataLoader framework. By default, every process reads 43008 samples. Each sample contains three sparse images of size 1280×2048 of average size 40 KB. The application is run for 150 steps and one epoch. At each step, each process reads and pre-process 32 images.

2) Distributed Flood Filling Networks: DFFN is a recurrent 3D convolutional network for segmenting neurons from a brain tissue's image. The application reads a 4.5 TB dataset stored in an HDF5 file. Every process reads 18678 samples, each of size 32×32×32. The samples are read by the application with 4096 fields of view. The application runs for 400 steps in one epoch with a batch size of 32 images.

3) Cosmoflow: CosmoFlow is a 3D convolutional neural network model for studying the features in the distribution of dark matter. The application contains a dataset of size 2 TB in HDF5 format. This dataset is converted offline into 1024 TFRecord files. The dataset is accessed using TensorFlow's `tf.data` APIs. Each TFRecord file consists of 262,144 samples, each of size 128×128×128×4. The application runs for four epochs with 256000 steps. The batch size is one. That is, each process reads one image from the dataset at each step.

4) Analysis: For Cosmic Tagger (Figure 9(a)), the DataLoader API of PyTorch framework (D in the figure) results in an exponential increase in I/O cost as the scale increases (shown in Figure through I/O to Compute Ratio). This is because the DataLoader framework consumes the file in the sample granularity. As the samples in the dataset are small (6KB), the application achieves a low aggregate bandwidth of 11 GB/s. For Stimulus (S in the figure), data access scales much better as the I/O performed matches the stripe size of the PFS (i.e., 1 MB) and therefore extracting higher I/O bandwidth from the PFS. Additionally, the pipelining operations (due to the three operators) and I/O parallelism (due to lower-level of implementation) enables Stimulus to further optimize the data pipeline by 30%. Through these optimizations, Stimulus achieves a speedup of 3.4× on I/O as compared to the baseline.

For DFFN (Figure 9(b)), the native HDF5 access (N in the figure) results in an exponential increase in I/O cost as the scale increases (shown in Figure through I/O to Compute Ratio). This is because manually reading data from the HDF5 file does not execute TensorFlow's execution runtime. This serializes the input reading, pre-processing, and AI model computations leading to sub-optimal I/O and data operator performance. For Stimulus (S in the figure), data access scales much better than the native HDF5 approach. This is because Stimulus performs I/O at a bigger granularity (i.e., 1



(a) Cosmic Tagger. (b) DFFN. (c) Cosmoflow.

Fig. 9. HDF5 Applications. a) Cosmic Tagger using Stimulus (S) achieves a speedup of 3.4× on I/O and 3× overall compared to the native DataLoader (D) on the largest scales. b) DFFN using Stimulus achieves a speedup of 2.9× on I/O and 2.29× overall compared to the native h5py APIs (N) on the largest scales. c) Cosmoflow using Stimulus achieves an overall speed up 1.9× overall compared to converting the dataset (C) on the largest scales while achieving comparable I/O performance. The speedup in I/O in all cases is achieved but Stimulus via data pipelining, parallel I/O, and larger transfer sizes resulting in higher PFS bandwidth. Additionally, Stimulus doesn't require conversion of scientific format into an AI framework-specific format for optimizing I/O.

MB) than the sample (40KB) and hence extracts much higher bandwidth from the parallel file system (improvement of 2.2×). Additionally, the read and pre-processing parallelism further improve the I/O performance for the application (total improvement of 1.3×). Finally, the whole input pipeline is hidden effectively behind the computation (0.01 seconds per step) of the previous step (total improvement 2.9×).

For Cosmoflow (Figure 9(c)), the default flow is first to convert the HDF5 dataset into TFRecord and then consume it with `tf.data` APIs (C in the figure). The I/O in the baseline is extremely efficient with good default reading transfer size (default 256 KB), data pipelining, and data parallelism. However, the conversion of the dataset from HDF5 to TFRecord offsets the benefit of loading data efficiently and increasing the footprint of data storage (2× more storage). For Stimulus (S in the figure), data access scales much better than in the conversion approach. The I/O is slightly better than TFRecord due to a much more optimal transfer size of 1MB, which matches the PFS's stripe size. However, due to the conversion on the baseline, Stimulus is overall 1.9× faster on the largest scales. In conclusion, Stimulus can achieve close to native data format's I/O performance while not needing any additional preprocessing such as conversion.

V. Related Work

Scientists have proposed several solutions to optimize scientific data access in modern AI frameworks. These optimizations stem from the inadequacies suffered by different AI frameworks in data management. Pumma et al. proposed LMDBIO-DM [9], an enhanced version of LMDBIO-LMM [47] that optimizes the I/O access of Caffe in a distributed-memory environment. However, these optimizations target specific cloud data formats, such as data in a distributed database. Essen et al. [48] proposed LBANN by utilizing node-local storage devices to store datasets and utilize that to read datasets into AI frameworks. Finally, Caffe [49] supports an extension to read HDF5 files. As scientific formats are manually ingested by application developers, these optimizations cannot extend in general to scientific data formats efficiently, which is the target of our work. Yosuke et. al. proposed a methodology [24] to ingest large amounts of dataset in HDF5 file format by utilizing a new parallel I/O pipeline within the LBANN infrastructure to enable an efficient I/O pipeline for scientific data formats. Similarly, Sam et. al. proposed a novel tournament method [50] for complex generative models, which minimizes communication and enables efficient partitioning of large data sets. Additionally, Kurth et. al. proposed injection of HDF5 dataset reading within the execution graph of TensorFlow to enable efficient I/O access from the PFS [23]. However, these approaches are applied at the application layer (i.e., specific to DL framework and data format targeted) which makes them non-portable across different AI frameworks and unable to maximize performance.

In Stimulus, the data management occurs in a lower-level (i.e., tensor infrastructure runtime) using StimOps functions and StimPack abstraction. This makes our solution portable across multiple scientific data formats and popular AI frameworks. Additionally, the two novel concepts enable performance improvement over all existing methodologies.

VI. Conclusion

Stimulus demonstrates an efficient input pipeline of scientific data formats in popular AI frameworks with a throughput of 5.3M operations per second. Additionally, the input pipeline extracts 2× to 3.7× better performance from the GPFS file system by utilizing optimizations such as decoupled I/O, operation parallelism, and prefetching. Finally, Stimulus outperforms existing solutions by 2× to 5.3× faster training performance with up to 768 GPUs on Summit supercomputer under a diverse set of workloads such as Cosmic Tagger (using HDF5 with PyTorch), Cosmoflow (using HDF5 with Tensorflow after conversion to TFRecord), and Distributed FFN (using HDF5 with TensorFlow). Stimulus efficiently integrates several scientific data formats such as HDF5, PnetCDF, Silo, ADIOS, and GNCF into various AI frameworks such as PyTorch and TensorFlow. Additionally, Stimulus' design enables a modular approach to abstract common I/O and runtime functionality through *StimPack* and *StimOps* respectively. Finally, Stimulus cohesively integrates several scientific data formats for popular AI frameworks while maximizing portability and performance on the Summit supercomputer.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant no. OCI-1835764 and CSR-1814872. Also, this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract LLNL-CONF-832595. Finally, this research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[2] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *ieee Computational intelligenCe magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[3] A. E. L. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electronic Imaging*, vol. 2017, no. 19, pp. 70–76, 2017.

[4] C. J. Adams, "Neutrino and Cosmic Tagging with UNet," 2015. [Online]. Available: https://github.com/coreyjadams/CosmicTagger

[5] A. Agrawal and A. Choudhary, "Deep materials informatics: Applications of deep learning in materials science," *MRS Communications*, vol. 9, no. 3, pp. 779–792, 2019.

[6] X. Wu, V. Taylor, J. M. Wozniak, R. Stevens, T. Brettin, and F. Xia, "Performance, energy, and scalability analysis and improvement of parallel cancer deep learning candle benchmarks," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–11.

[7] E. Bisong, "Tensorflow 2.0 and keras," in *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Springer, 2019, pp. 347–399.

[8] V. Subramanian, *Deep Learning with PyTorch: A practical approach to building neural network models using PyTorch*. Packt Publishing Ltd, 2018.

[9] S. Pumma, M. Si, W.-c. Feng, and P. Balaji, "Parallel I/O optimizations for scalable deep learning," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, 2017, pp. 720–729.

[10] S. Liu, B. Niu, D. Li, M. Wang, S. Tang, J. Kong, B. Li, X. Xie, and Z. Zhu, "DL-assisted cross-layer orchestration in software-defined IP-over-EONs: From algorithm design to system prototype," *Journal of Lightwave Technology*, vol. 37, no. 17, pp. 4426–4438, 2019.

[11] N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild, and others, "Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence," USDOE Office of Science (SC), Washington, DC (United States), Tech. Rep., 2019.

[12] C. Ertl, J. Frisch, and R.-P. Mundani, "Design and optimisation of an efficient HDF5 I/O Kernel for massive parallel fluid flow simulations," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 24, p. e4165, 2017.

[13] A. Lintermann, S. Habbinga, and J. H. Göbbert, "Comprehensive Visualization of Large-Scale Simulation Data Linked to Respiratory Flow Computations on HPC Systems," in *SC'17: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[14] R. Tchoua, J. Choi, S. Klasky, Q. Liu, J. Logan, K. Moreland, J. Mu, M. Parashar, N. Podhorszki, D. Pugmire, and others, "Adios visualization schema: A first step towards improving interdisciplinary collaboration in high performance computing," in *2013 IEEE 9th International Conference on e-Science*, 2013, pp. 27–34.

[15] M. Werth, J. Lucas, T. Kyono, I. McQuaid, and J. Fletcher, "Silo: A machine learning dataset of synthetic ground-based observations of leo satellites," in *2020 IEEE Aerospace Conference*, 2020, pp. 1–8.

[16] S. Nativi, P. Mazzetti, M. Santoro, F. Papeschi, M. Craglia, and O. Ochiai, "Big data challenges in building the global earth observation system of systems," *Environmental Modelling & Software*, vol. 68, pp. 1–26, 2015.

[17] G. Fox and S. Jha, "Understanding ml driven hpc: Applications and infrastructure," *arXiv preprint arXiv:1909.02363*, 2019.

[18] M. A. S. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. F. Cunha, and R. Buyya, "HPC cloud for scientific and business applications: taxonomy, vision, and research challenges," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–29, 2018.

[19] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, S. J. Pennycook, and others, "CosmoFlow: Using deep learning to learn the universe at scale," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 819–829.

[20] D. Kang, V. Patel, K. Khandrika, S. Blanas, Y. Wang, and S. Parthasarathy, "Characterizing I/O optimization opportunities for array-centric applications on HDFS," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–2.

[21] R. Acciarri, C. Adams, C. Backhouse, W. Badgett, L. Bagby, V. Basque, Q. Bazetto, A. Bhanderi, A. Bhat, D. Brailsford, and others, "Cosmic Background Removal with Deep Neural Networks in SBND," *arXiv preprint arXiv:2012.01301*, 2020.

[22] W. Dong, M. Keceli, R. Vescovi, H. Li, C. Adams, E. Jennings, S. Flender, T. Uram, V. Vishwanath, N. Ferrier, and others, "Scaling Distributed Training of Flood-Filling Networks on HPC Infrastructure for Brain Mapping," in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, 2019, pp. 52–61.

[23] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, and others, "Exascale deep learning for climate analytics," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 649–660.

[24] Y. Oyama, N. Maruyama, N. Dryden, E. McCarthy, P. Harrington, J. Balewski, S. Matsuoka, P. Nugent, and B. Van Essen, "The case for strong scaling in deep learning: Training large 3d cnns with hybrid parallelism," *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[25] Andrew Feldman, "Cerebras: The world's most powerful AI compute," 5 2021. [Online]. Available: https://cerebras.net/

[26] S. W. D. Chien, S. Markidis, V. Olshevsky, Y. Bulatov, E. Laure, and J. Vetter, "TensorFlow Doing HPC," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 509–518.

[27] J. Biddiscombe, J. Soumagne, G. Oger, D. Guibert, and J.-G. Piccinali, "Parallel computational steering for hpc applications using hdf5 files in distributed shared memory," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 6, pp. 852–864, 2012.

[28] G. Dong, K. G. Felker, A. Svyatkovskiy, W. Tang, and J. Kates-Harbeck, "Fully Convolutional Spatio-Temporal Models for Representation Learning in Plasma Science," *arXiv preprint arXiv:2007.10468*, 2020.

[29] S. Gope, S. Sarkar, P. Mitra, and S. Ghosh, "Early prediction of extreme rainfall events: a deep learning approach," in *Industrial Conference on Data Mining*, 2016, pp. 154–167.

[30] T. Kurth, J. Zhang, N. Satish, E. Racah, M. Mostofa Ali Patwary, T. Malas, N. Sundaram, W. Bhimji, M. Smorkalov, J. Deslippe, M. Shiryaev, S. Sridharan, P. Dubey, and I. Mitliagkas, "Deep Learning at 15PF Supervised and Semi-Supervised Classification for Scientific Data," vol. 11, 2017.

[31] T. Kurth, N. Luehr, J. Deslippe, S. Treichler, E. Phillips, M. Fatica, J. Romero, A. Mahesh, A. Gov Prabhat, M. Mudigonda, M. Matheson, and M. Houston, *Exascale Deep Learning for Climate Analytics*.

[32] E. Racah, C. Beckham, T. Maharaj, S. E. Kahou, Prabhat, and C. Pal, "ExtremeWeather: A large-scale climate dataset for semi-supervised detection, localization, and understanding of extreme weather events," in *Advances in Neural Information Processing Systems*, vol. 2017-December. Neural information processing systems foundation, 2017, pp. 3403–3414.

[33] S. Ravanbakhsh, F. Lanusse, R. Mandelbaum, J. Schneider, and B. Poczos, "Enabling dark energy science with deep generative models of galaxy images," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, 2017.

[34] E. Medina and E. Dagan, "Habana labs purpose-built AI inference and training processor architectures: Scaling AI training systems using standard ethernet with gaudi processor," *IEEE Micro*, vol. 40, no. 2, pp. 17–24, 3 2020.

[35] OLCF, "Summit User Guide," 2021. [Online]. Available: https://docs.olcf.ornl.gov/systems/summit_user_guide.html

[36] Tensorflow, "Create an op," 2020. [Online]. Available: https://www.tensorflow.org/guide/create_op

[37] Facebook's AI Research lab (FAIR), "Extending TorchScript with custom C++ operators," 2020. [Online]. Available: https://pytorch.org/tutorials/advanced/torch_script_custom_ops.html

[38] Caffe2, "Extending TorchScript with custom C++ operators," 2020. [Online]. Available: https://caffe2.ai/docs/custom-operators.html

[39] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[40] J. Hu, S. Qian, Q. Fang, Y. Wang, Q. Zhao, H. Zhang, and C. Xu, "Efficient Graph Deep Learning in TensorFlow with tf_geometric," *arXiv preprint arXiv:2101.11552*, 2021.

[41] H. Devarajan and H. Zheng, "VaniDL Analyzer for Deep Learning Workloads," 2020. [Online]. Available: https://github.com/hariharan-devarajan/vanidl

[42] H. Devarajan, "DLIO: Scientific Deep Learning I/O Benchmark," 2020. [Online]. Available: https://github.com/hariharan-devarajan/dlio_benchmark

[43] S. W. D. Chien, S. Markidis, C. P. Sishtla, L. Santos, P. Herman, S. Narasimhamurthy, and E. Laure, "Characterizing Deep-Learning I/O Workloads in TensorFlow," 10 2018. [Online]. Available: http://arxiv.org/abs/1810.03035http://dx.doi.org/10.1109/PDSW-DISCS.2018.00011

[44] H. Devarajan, A. Kougkas, and X.-H. Sun, "HFetch: Hierarchical Data Prefetching for Scientific Workflows in Multi-Tiered Storage Environments," in *Proceedings - 2020 IEEE 34th International Parallel and Distributed Processing Symposium, IPDPS 2020*, 2020.

[45] P. Subedi, P. Davis, S. Duan, and others, "Stacker: an autonomic data movement engine for extreme-scale data staging-based in-situ workflows," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. USA: IEEE Press, 2018, p. 73.

[46] H. Devarajan, A. Kougkas, L. Logan, and X.-H. Sun, "HCompress: Hierarchical Data Compression for Multi-Tiered Storage Environments," in *Proceedings - 2020 IEEE 34th International Parallel and Distributed Processing Symposium, IPDPS 2020*, 2020.

[47] S. Pumma, M. Si, W.-c. Feng, and P. Balaji, "Towards scalable deep learning via I/O analysis and optimization," in *2017 IEEE 19th International Conference on High Performance Computing and Communications*, 2017, pp. 223–230.

[48] B. Van Essen, H. Kim, R. Pearce, K. Boakye, and B. Chen, "LBANN: Livermore big artificial neural network HPC toolkit," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, 2015, pp. 1–6.

[49] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[50] S. A. Jacobs, B. Van Essen, D. Hysom, J.-S. Yeom, T. Moon, R. Anirudh, J. J. Thiagaranjan, S. Liu, P.-T. Bremer, J. Gaffney, and others, "Parallelizing training of deep generative models on massive scientific datasets," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–10.