Efficient Distributed Data Loading for Large-Scale Machine Learning Model Training with Parax

Sheng Shen
University of California, Berkeley

Chanwut (Mick) Kittivorawong
University of California, Berkeley

Abstract

Parax is a JIT compiler for distributed tensor computation and large-scale neural network training. Parax can compile a tensor computational graph, generate a parallelization strategy, and run it on a Ray [17] GPU cluster. In this project, we designed and implemented an efficient distributed data loader for tensor computation in Parax. Compared to the current naive implementation of Parax's data loader, our data loader supports both data parallel strategies and model parallel strategies generated by Parax, and it has 2-6 times speed up in runtime.

1 Introduction

Tensor computation is a big component of machine learning tasks such as training Convolutional Neural Networks (CNN) or Transformers. In recent years, we see large growth of data size and machine learning model complexity. To compensate for this growth, a tensor computation task demand more scalable strategies for its execution. To achieve the maximum runtime performance for a machine learning task, we can define the execution of the task instruction by instruction in a low-level programming language. However, the program is hard to understand as a trade-off. On the other hand, we can define the execution schedule in a high-level Domain-Specific Language (DSL) such as TensorFlow [2] or PyTorch [19]. The definition of a machine learning task is expressive, but the runtime performance is not optimal.

Just-In-Time (JIT) compiler comes in to solve the drawback of each of the two approaches. It takes a high-level DSL definition of a machine learning task and compiles it as the program runs. The overall program will have an overhead compilation time initially. But, the JIT compiler [3, 11] compiles the machine learning task into a sequence of instructions that can run optimally.

We can further optimize the machine learning task by parallelizing its execution, using the following strategies:

- In a machine learning's training task, we run forward passes and backward passes of a model repeatedly in a small batch at a time. With *data-parallelism* [13], we can run forward passes and backward passes of multiple batch at the same time in multiple GPUs. Each GPU runs a forward pass and backward pass of one batch. Then, we can combine the gradient result as if we train the model with all small batches together combined as one large batch. This way, the model will converge with fewer epochs, and we can reduce the overall training time.
- It is common that a neural network model has multiple layers. With *model-parallelism* [16, 24], we can fit a large model with multiple layers into multiple GPUs. The neural network model can be divided into multiple groups, each consist of consecutive layers. Each GPU is responsible for computing forward passes and backward passes of one group.
- Data-flow in *model-parallelism* is not actually parallelized. Instead, they are piped through each GPU. In most cases, each GPU has to wait for another GPU to finish its forward pass or backward pass before it can performs its own forward pass or backward pass. With *pipeline-parallelism* [7], we can divide each batch of training data into mini-batches. Then, we can pipeline mini-batches from multiple batches together to have GPUs in *model-parallelism* setting work in parallel.

In this paper, we will only cover *data-parallelism* and *model-parallelism*.

Parax is a JIT compiler, built on top of JAX [3] and XLA[1]. Parax allows users to annotate Python machine learning programs with a decorator. Then, Parax compiles the program and automatically finds parallelization strategies for the program to run across multiple GPUs. Parax supports all the three paralellization strategies: *data-parallelism*, *model-parallelism*, and *pipeline-parallelism*.

[1] See: XLA: Optimizing Compiler for Machine Learning (https://www.tensorflow.org/xla).

The current bottleneck of Parax is that it does not have an efficient dataloader for machine learning tasks. See Figure 1 (A). Right now, Parax uses 1 CPU to broadcast a machine learning's datasets to all GPUs for training. In a machine learning training task, we repeatedly load each batch of datasets into the machine learning model to train. Therefore, having an efficient dataloader is important to reduce the overall runtime of the training task.

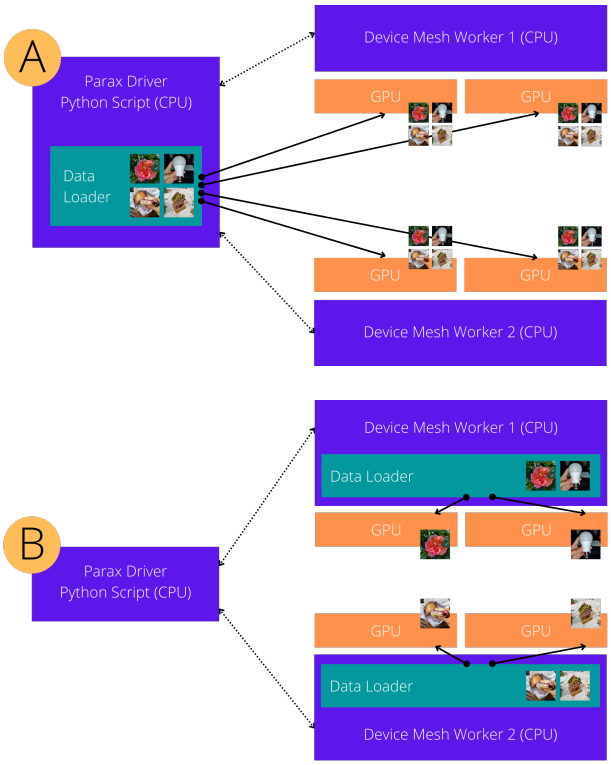

Figure 1: **A:** Original design of Parax's dataloader. The dataloader is on the main CPU thread. It reads the data and broadcast the data to all of the GPUs. **B:** Our new implementation of Parax's dataloader. Each Device Mesh Worker has one dataloader to distribute the data to the GPUs that it is responsible for. The dataloader only load the piece of data necessary for each GPU.

In this project, we made the following contributions:

1. As a **solution**, we design an interface for a distributed dataloader to be intuitive, by following common patterns of data loading in machine learning tasks. We also experiment with multiple design of dataloader from Tensorflow and PyTorch to find the most efficient design. The dataloader uses each thread of the CPU to load just the necessary part of datasets to each GPU. See section 3

2. As a **metrix of success**, we runs benchmarks on the overall runtime performance of our dataloader in subsection 4.1. We evalulate the dataloader with different settings including different datasets, batch sizes, number of GPUs, and data-fetching strategies. We chose WideResNet [25] to represent Convolutional Neural Network and GPT2-large [20] to represent Transformer. As a comparison, we compare our runtime performance with the existing single-CPU-thread dataloader.

3. Finally, we **discuss** our design decisions, experiment results, and improvements for future works in subsection 4.2 and section 6.

2 Related Work

2.1 Tensor Computation

Quantitative data we collected from the real world come in as multi-dimensional tensors. To make sense of these data, we manipulate its value and derive it into forms that we can understand. In many cases, these tensors are large. Therefore, we need an easy and efficient interface to interact with them. This is when a library such as NumPy [5] comes in. With the library, we can see a tensor as a single variable instead of a collections of numerical values. Then, we can manipulate them with tensor operations just the same way as we do on numerical variables.

Tensor computation is important for machine learning tasks. A machine learning model is just a series of operations done on a tensor (input data) and other tensors (weights of layers in the model). To train a machine learning model, we input an input data into the model. Then, we compare the output with the ground through of the input data. Then, we input back the comparison result backward into the model to find the gradiant of the weights of each layer of the model. Finally, we use the gradiant to update its corresponding weight to make the model more accurate. However, the process for deriving the gradient is complicated and error-prone if done by hand. Therefore, libraries like Autograd [14] help us automatically find the gradient of a serie of tensor operations.

2.2 JIT Compiler

NumPy expressions themselves are efficient. However, operations that manipulate NumPy values from Python have runtime overhead. Therefore, tasks like adding a value to to each element in a NumPy array is inefficient in `slowly_initialize_sequence` in Figure 2. JIT compilers like Numba [11] compile a Python script with NumPy expressions as it runs to achieve better runtime performance. Numba compiles the loop and the NumPy expression in `quicky_initialize_sequence` in Figure 2 into a low-level language. As a result, it prevents unnecessary communication between the low-level language that handles NumPy expressions and the Python language.

```python
import numpy as np
from numba import njit, prange

def slowly_initialize_sequence(size):
    array = np.empty([size])
    for i in range(size):
        array[i] = i
    return array

@njit(parallel=True)
def quickly_initialize_sequence(size):
    array = np.empty([size])
    for i in prange(size):
        array[i] = i
    return array
```

Figure 2: In `slowly_initialize_sequence`, we assign `t` from Python to the NumPy array. So, there are multiple communications between NumPy and Python because we loop over the expression. In contrast in `quicky_initialize_sequence`, we compile the whole function into low-level language. We no longer assign `i` from Python to NumPy array anymore. Instead, the entire function is run in low-level language, which eliminates the unnecessary communications between NumPy and Python. In addition, Numba allows parallelization of each round of the for-loop by replacing `range` with `prage`.

In machine learning specific tasks, JIT compiler like JAX compiles a tensor computation language with Autograd capabilities. JAX also supports *data-parallelism* that paralellize the tensor computation on multiple GPUs. Supporting Autograd in a JIT compiler helps users to easily define a machine learning training task, while making the task fast and scalable to large datasets.

2.3 Parax

JAX supports *data-parallelism*, but users need to specify how to let JAX parallelize their program themselves. Parax is built on top ot JAX, which means that users can decorate a JAX's machine learning task in Python with a Parax's decorator to parallelize it. See Figure 3. Parax's JIT compiler will automatically finds parallelization strategies for the task. Parax supports three parallelization strategies, including *model-parallelization*, *data-paralellization*, and *pipeline-parallelization*. Specically, modern neural networks make extensive use of a cluster of machines for training and inference, each of which equipped with several accelerators. Data parallelism [10] is the most commonly used approach and is supported by major frameworks (TensorFlow [1], PyTorch [18], JAX), where devices run the same program with different input data and combine their local gradients before the weight updates. Model parallelism, on the other hand, partitions computation beyond the input batch, which is needed to build very large models. For example, pipelining [4,6] splits a large model's layers into multiple stages, while operator-level partitioning [8,23] splits individual operators into smaller parallel operators. For automated parallesim, Zero [21] presents a set of optimizations to reduce memory redundancy in parallel training devices, by partitioning weights, activations, and optimizer state separately, and it is able to scale models to 170 billion parameters. GShard [13] support those specific partitioning techniques by simply annotating the corresponding tensors, allowing user to scale to over 1 trillion parameters and explore more design choices. While even though all the aftermentioned deep learning frameworks TensorFlow, Pytorch has their own distributed dataloader, but none of them could support different parrallesim strategies and direct integration with the automated parallesim framework like Parax in our project.

```python
from parax import paralellize, DataLoader

@parallelize
def train_step(state, batch):
    # training steps with JAX
    ...
    return new_state, matrics

def train(data):
    ...
    executable = train_step.get_executable(
        state, batch)
    data_loader = DataLoader(
        # Training data
        data,
        # GPU cluster specifications
        executable.physical_mesh,
        # Parallelization specifications
        executable.input_sharding_specs,
    )
    for batch in data_loader:
        state, metrics = train_step(state, batch)
```

Figure 3: Code sample for training with Parax and distributed dataloader. Users can get the information about the specification of GPU cluster and parallelization specification from the `train_step` function.

3 Methods

3.1 User Interface

We designed the user interface of our dataloader to be a Python iterator. Each iteration of the dataloader produces a distributed array object as data for multiple GPUs. Then, a user can input the distributed array object into a `train_step` function as in Figure 3. The `train_step` function is defined by the user to train one step of a machine learning task. `train_step` should be annotated by Parax's decorator to enable parallelization.

With Parax's decorator, the user does not need to manually modify their `train_step` to paralellize it. And with our dat-

aloader, the user does not need to manually distribute the data to each GPU. From the user's perspective, the code will read as if we iterate through batches of our datasets. Then, we train each batch of the data as if we are running the `train_step` on a single CPU. This is a common pattern in machine learning's training task. So, the code will be easily recognizable by other users.

Given that Parax is built on JAX, which is laser-focused on program transformations and accelerator-backed NumPy and does not include data loading or munging in the Parax/JAX library. In consequence, we can build our new dataloader based on exsiting dataloader (TFSD, pytorch). We did preliminary experiments on both TFDS and pytorch's dataloader as shown in Figure 4 by comparing the disk to CPU latency with single thread. We found that PyTorch's dataloader gives us a data loding with lower latency comparing to TFDS. We then build our dataloader framework upon the PyTorch's solution.

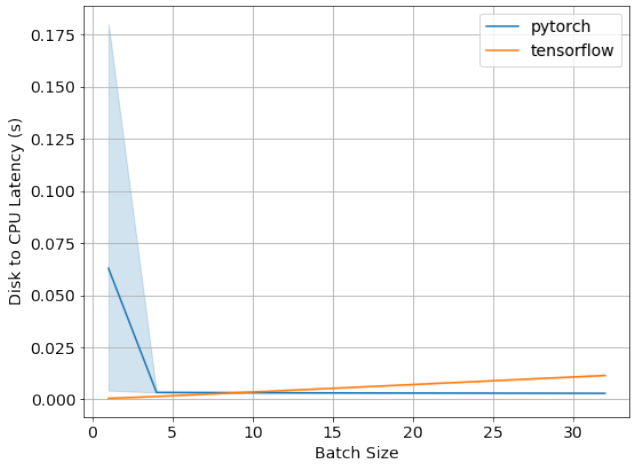

Figure 4: Data Loading Performance with different dataloader framework on CIFAR10. (TensorFlow Datasets vs PyTorch).

3.2 Implementation

In this section, we detail how we implement our dataloader framework on top of the pytorch dataloader and investigate how to improve the existing baseline solution (1-CPU dataloader) when intergating with Parax. Specifically, there are two main modules in our distributed dataloader: One module to handle loading raw data from disk to CPU. The loaded data in CPU memory should also match the input format of Parax versus the original cpu tensor abstraction in the pytorch dataloader; Another module to handle transferring data from CPU to DeviceMesh, where each DeviceMesh Worker has the dataloader to distribute the data to the GPUs that it is responsible for versus broadcasting to all DeviceMeshs. Noted here the abstraction of the final data should be also in the format of `DeviceArray` versus the cuda tensor abstraction in the pytorch dataloader.

3.2.1 DataLoader Abstration

Here we describe how we modify the pytorch dataloader abstraction to support the target final output abstraction we need for Parax. The general input of our distributed dataloader should be the dataset object, number of workers and the output would be an iteratble list of `DeviceArray` instance on jax CPU host or GPU DeviceMesh.

For the dataset object, our dataloader will take in the `WebDataset` abstraction, which is a pytorch Dataset (`IterableDataset`) implementation providing efficient access to datasets stored in POSIX tar archives and uses only sequential/consecutive data access. This brings performance advantage in many compute environments, and it is essential for very large scale training. We have a discussion for using this Dataset with consecutive fetching versus nonconsecutive fetching in pytorch Dataset in Section 3.2.2. Using the sharding provided in this abstraction enables us to support the distribution logic from different Parallelism strategies provided by Parax.

To support the sharding with multi-threading and jax `DeviceArray` output abstraction and control the loading process, we inherit the pytorch dataloader and design our own dataloader. We first discuss the process of loading from disk to CPU and then illustrate the process of transferring the data abstraction between devices. Specifically, given the output shape of the Physical DeivceMesh, we can derive the number of DeviceMeshs (using model parallelism) that consume the same batched data in each iteration. We denote this as the effective DeviceMesh. To handle this distribution logic, we can first using sharding in `WebDataset` to distribute shards efficiently across worker nodes and devices, and perform random shard access while reading sequentially within each shard. Then we will take a list of shards and return a subset of those shards for each effective DeviceMeshs. As shown in Figure 5, we design a `nodesplitter` function to decide the number of replica of those shards and `workersplitter` function as a shard selection function in Dataset to leverage the multiprocessing data access inherited from pytorch DataLoader. This abstraction can be supported in multi-node and single-node environments (abstract different effective DeviceMesh as different node/host). Using python's `multiprocessing` class under the Global Interpreter Lock, the distributed dataloader will create an iterator of DataLoader for each processor. At this point, the dataset, `collate_fn`, and `workersplitter` are passed to each worker, where they are used to initialize and fetch data. Dataset access together with its internal I/O, transform (including `collate_fn`) will run in the worker process.

The previous process of dataloader mainly involves the data fetching from disk. To transfer the data into target

```python
def workersplitter(shard_idxs):
    #Split shard_idxs per worker
    #Selects a subset of shard_idxs.
    #Used as a shard selection function in Dataset.
    #replaces wds.split_by_worker
    shard_idxs = [_ for _ in shard_idxs]
    assert isinstance(shard_idxs, list)
    worker_info = torch.utils.data.get_worker_info
    if worker_info is not None:
        wid = worker_info.id
        num_workers = worker_info.num_workers
        return shard_idxs[wid::num_workers]
    else:
        return shard_idxs

def nodesplitter(shard_idxs, num_replicas):
    #Split shard_idxs_ correctly per host
    #:param shard_idxs:
    #:return: slice of shard_idxs_
    rank=host_device_id
    shard_idxs_ = shard_idxs[rank::num_replicas]
    return shard_idxs_
```

Figure 5: Code sample for using `nodesplitter` function and `workersplitter` take a list of shards and return a subset of those shards following the distribute logic in Parax.

`DeviceArray`, a `collate_fn` is provided to put the raw data into `jax.numpy.array` on `cpu` devices. Then we will call the `prefetch_to_device` function in `jax` that shards and prefetchs batch on effective DeviceMesh. This operation allows us to improve the performance of training loops significantly by overlapping compute and data transfer (See Section 4.2.2 for detailed discussion).

3.2.2 From Disk to CPU

We first investigate how the different fetching and threading strategies will affect the performance using our dataloader to load data from disk to CPU and then describe the final choice in our implementation in detail.

Consecutive Fetching Normally, when training using pytorch dataset and dataloader, a `RandomBatchSampler` will be provided to give each worker random choices of the data file indexes. Then the worker will perform random access to data files in file-based I/O. this is good for training optimization given the randomness in each batch data points. But for each data file, there is a I/O request and storage response. This may hurt the overall throughput of the data loading process. In the opposite, we shuffle the data in the preprocessing when constructing the abstraction of `WebDataset` in each shards. Then each worker in our distributed dataloader will perform a sequential I/O reading for each shard of the data and reduce the I/O latency from the reduction of the potential cache misses. As shown in Figure 6, we use 4 threads to load batched data using consecutive data fetching on various datasets. For dataset with large image files like ImageNet, the data transferring time between disk and CPU becomes a severe problem. The performance difference between the two fetching strategy could be more than 2x. For the dataset with small image files (CIFAR10 / MNIST), the performances are comparable between the two strategies. Besides the sequential throughput gain, it is verified in `WebDataset` that random access to the shards and in-memory shuffling satisfy enough randomness for the training optimization. Thus we use `WebDataset` with customized splitting and loading function discussed in Section 3.2.1.

Threading For data fetching from disk I/O, we inherit the `multiprocessing` strategy in pytorch Dataloader using `fork()` as context manager and the loading strategy described above. However, using more workers/threads will not always bring in performance improvement. As shown in Figure 7, when we increase the number of processes/workers, the loading latency decreases significantly in the beginning. Nevertheless, the latency goes up when the number of processes exceeds the physical thread limit of CPU in our local machine (denoted by the dashed red line). This can be attributed to the potential increase in CPU memory (overall memory usage is `number_of_workers * size_of_parent process`), and thread communication and conflicting. We have a further discussion about the runtime breakdown (including thread communication in Section 4.2.3) Also, with large batch size and large file size, the transferring performance becomes more stable. Moreover, after several iterations, the loader worker processes will consume the same amount of CPU memory as the parent process for all Python objects in the parent process which are accessed from the worker processes. This can be problematic if the Dataset contains a lot of data (e.g. ImageNet, we are loading a very large list of filenames at Dataset construction time) and/or we are using a lot of workers.

3.2.3 From CPU to DeviceMesh (GPU)

Given the newly `collate_fn` function, we now have the `DeviceArrary` of `cpu` buffer for batched data. In the baseline implementation, the `DeviceArrary` on `cpu` will be loaded to one/first `gpu:0` device in `jax.devices()` then use the `shard_args` in `jax.interpreters` to copy batched data from `gpu:0` to other devices. Versus this strategy, we load multiple `DeviceArrary` on `cpu` to effective devices in parallel directly. Also, a `prefetch` strategy can be integrated straightforward with our distributed dataloader. This empowers further performance of training loops by overlapping compute and data transfer. We have a detailed discussion for this in Section 4.2.2.

Sharding and Replica Here we describe the baseline strategy of copying data from `gpu:0` to other devices versus our

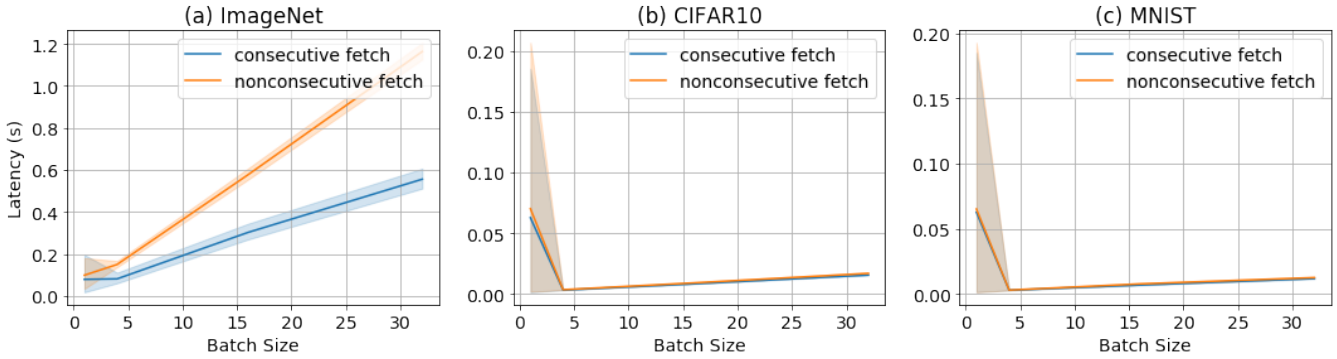

Figure 6: Data Loading Performance with different CPU fetching strategies on different datasets.

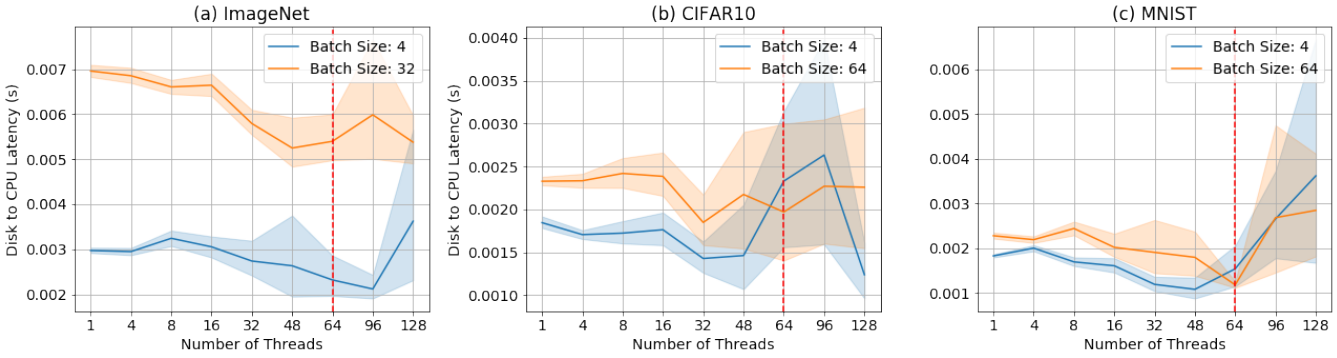

Figure 7: Data Loading Performance from Disk to CPU memory with different number of threads on different datasets. Note: The red vertical dashed line is the number of physical CPU thread limit. Note: The batch size in ImageNet experiment is less than others because larger batch size does not fit in our GPUs.

implementation. Noted the sharding strategy we are using will allows for the transferring process from `cpu` directly to effective devices but will not affect the baseline strategy. We benchmark with the same WideResNet architecture in Section 4.1 with 2x2 model parallelism strategy.This means 2 GPUs will share one copy of the model and data. As shown in Figure 9, this gives us up to 2x performance speedup when transferring data from `cpu` to effective DeviceMeshs.

Threading Using more threads in CPU is also not always preferable for loading from CPU to GPU given the process will have to write to the main process buffer for hosting the batched data replica. A large number of threads will cause extra blocking and synchronization problem as shown in Figure 8. This is even more problematic when loading small data files and small batch sizes and using more GPUs. Moreover, given that we transfer the framework from pytorch to jax but base our dataloader implementation in pytorch, we could not fully utilize the potential multi-threading memory pinning process from inheriting the dataloader. The memory pinning process will automatically put the fetched data tensors in pinned memory, and thus enables faster data transfer to CUDA-enabled GPUs. But given the output abstraction changes in our dataloader, adding extra pinning memory will hurt the performance even more in our preliminary experiments, which we leave as future work.

4 Evaluation

4.1 Setting

Dataset For vision benchmark, we choose three representative datasets from `TORCHVISION.DATASETS`; MNIST, CIFAR10 and ImageNet. Specifically, MNIST [12] is a database of handwritten digits in 10 classes. We use 60,000 training samples for benchmarking and the input size is set to be 28x28; the CIFAR10 [9] dataset consists of 60,000 32x32 colour images in 10 classes, with 6,000 images per class. We use the 50000 training images for benchmarking; ImageNet [22] is an image database organized according to the WordNet hierarchy, in which each node of the hierarchy is depicted by hundreds and thousands of images. We use the 1000 classes version and 50,000 validation data for benchmarking. The input sizes of images in ImageNet are set to be 224x224.

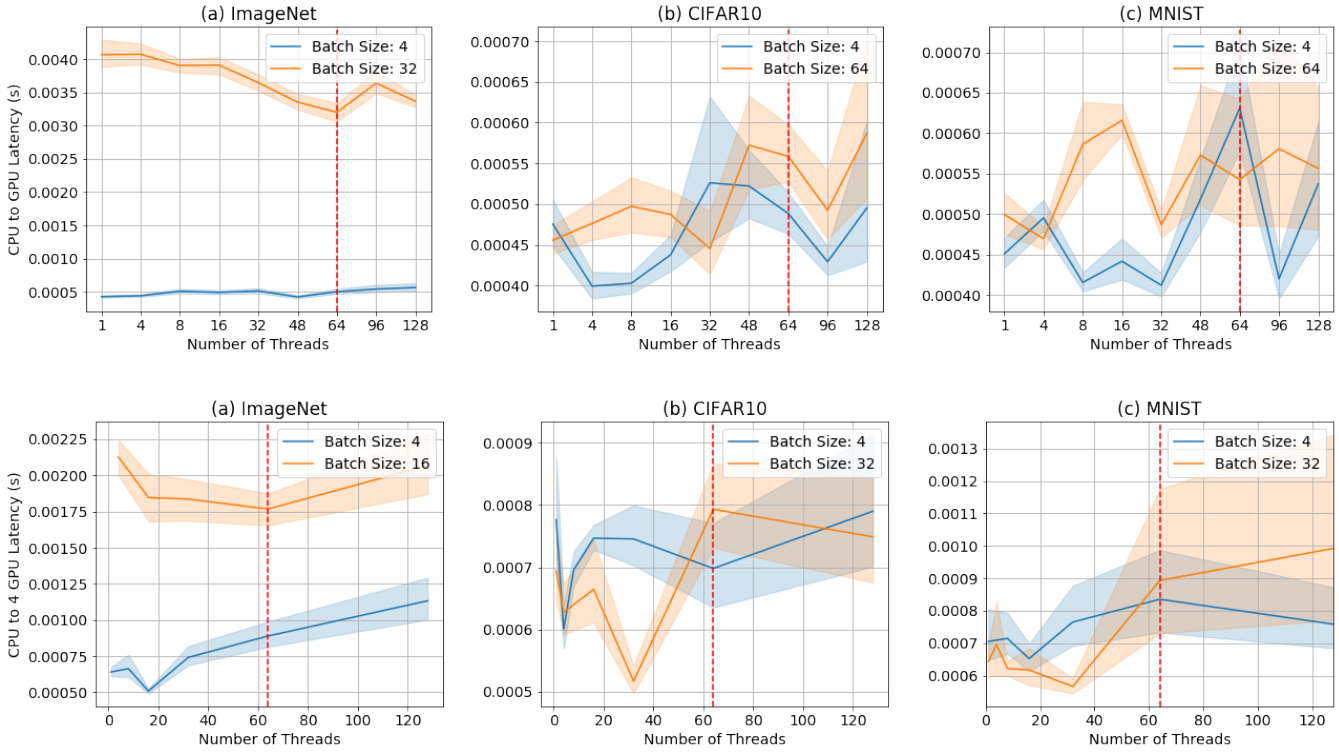

Figure 8: Data Loading Performance from CPU to GPU memory (1 GPU, 4 GPUs) with different number of threads on different datasets. Note: The red vertical dashed line is the number of physical CPU thread limit. Note: The batch size in ImageNet experiment is less than others because larger batch size does not fit in our GPUs.

For text benchmark, we use WikiText103 [15] dataset, but varying the input length from 128 to 1024. The WikiText language modeling dataset is a collection of over 100 million tokens extracted from the set of verified good and featured articles on Wikipedia. We use the training set of 103M tokens for benchmarking.

Model For vision benchmark, we use the WideResNet [25] architecture, which is a variant on ResNets where WideResNets decrease the depth and increase the width of residual networks through the use of wide residual blocks. We use the modified WideResNets-50 variant, which consists of 50 layers, 300 channels, and 3 blocks.

For text benchmark, we use the modified GPT2-large [20] architecture, which has 1024 channels, 4 layers, and 16 heads.

Hardware For the benchmark hardware, we are using the local machine with 4 Nvidia RTX 4000 (24 GB) GPUs and CPU with 64 threads.

Profiling To profile the runtime for data loading and computation, we use `time.monotonic()` function of the Python `time` classes and set the break points at different functions of the training script. To reduce the randomness in cuda asynchronous execution, we will call `torch.cuda.synchronize()` function before profile runtime after each training loop. This will wait for all kernels in all streams on CUDA devices to complete. Moreover, for profiling the runtime between CPU threads in Section 4.2.3, we use Vtune[2] for profiling the detailed execution time.

4.2 Results

4.2.1 Overall Performance

We compare the overall data loading performance with the baseline strategy in Figure 11 for 4 GPU (2 GPU model parallelism setting). Each data point in Figure 11 stands for one profiling result in one iteration of the training loop (the highlighted middle line represents the average of the performance). The performance of the baseline 1-CPU dataloader with the same batch size is denoted with the dashed line using the same color compared to ours, the baseline method is also static without threading/multiprocessing strategy. It can be seen that our strategy brings up to 6x speed up in the data loading latency. The performance difference is clearer with large batch size (e.g. 32) and large data files (e.g. ImageNet as 224 image size)

[2] https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune

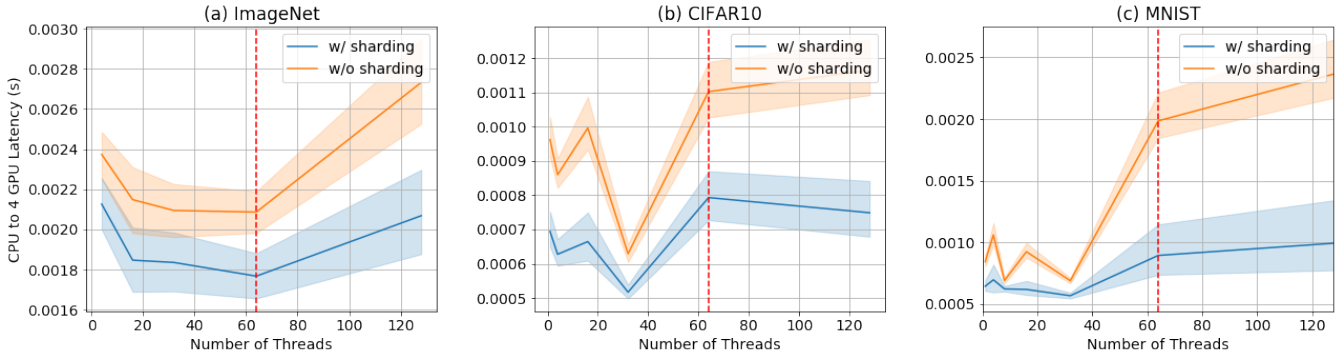

Figure 9: Data Loading Performance from CPU to GPU memory (4 GPUs) with sharding strategy or not on different datasets. Note: The red vertical dashed line is the number of physical CPU thread limit.

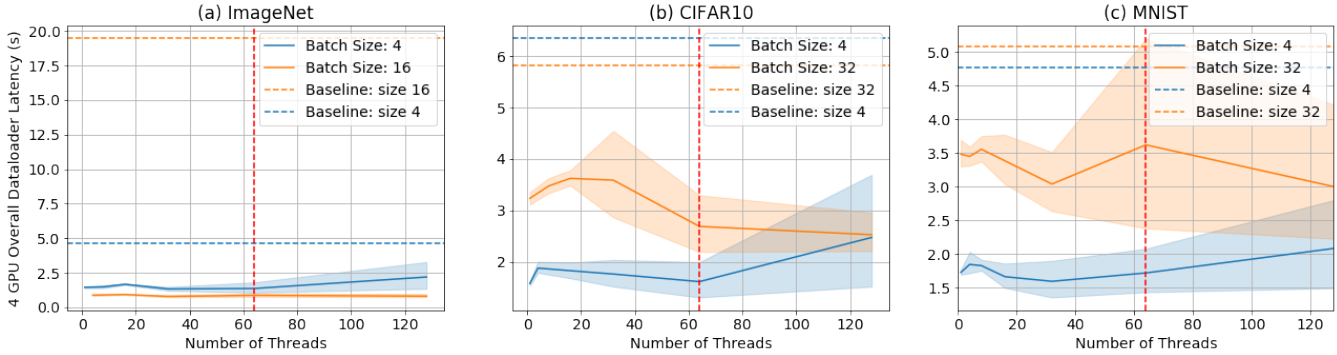

Figure 10: Runtime Portion of Data Loading Performance (4 GPUs) on different datasets. In this charts, we compare our implementation of the dataloader to the original dataloader (Baseline) from Parax. Note: The red vertical dashed line is the number of physical CPU thread limit. Note: The batch size in ImageNet experiment is less than others because larger batch size does not fit in our GPUs.

for the vision benchmark. For small batch sizes (e.g. 4) or small data files (MNIST/CIFAR10 as 28/32 image size), the latency of our dataloader becomes more unstable and even significantly increases using the largest number of threads. For text bechmark, we will perform the extra tokenization and text grouping (group texts into chunks as the specified sequence length for faster batching). As a consequence, the performance becomes much stable with different number of threads but still outperforms the baseline solution originally in Parax.

4.2.2 Runtime Portion of Dataloader

To conduct further analysis with respect to the runtime portion of the dataloader over the overall training loop (data loading, model forward computation, model backward computation, optimizer state update and learning rate update), we visualize the portion of data loading latency in Section 4.2.1 verus the total runtime in each iteration. This analysis is also necessary to verify whether the dataloader is actually the bottleneck for the overall computation and the runtime portion it takes up is large enough for us to investigate. Except for the data loading time, the absolute runtime for other functions are very stable with fixed batch size given the same model architecture and GPU environment. In this case, the runtime portion in Figure 10 difference between our method and baseline 1-CPU loader directly reflects the performance of the dataloader itself. Before the introduce of our dataloader, the data loading latency takes 20% of the overall runtime (including GPU computation) for 32 batch size on ImageNet. It also consumes 5-10% portion of the runtime for small vision benchmarks, which can be considered as signifcant overhead. Using our distributed dataloader, the overhead of dataloader can be reduced to 5-2%, which shows the advance of the distributed and sharding solution we implement.

4.2.3 Breakdown of Runtime in Dataloader

Another aspect of the analysis lies in the detailed runtime breakdown inside our dataloader (disk-cpu, cpu-gpu, thread communication). We demonstrate that information in Figure 12 for vision benchmarks. As shown in the figure, the

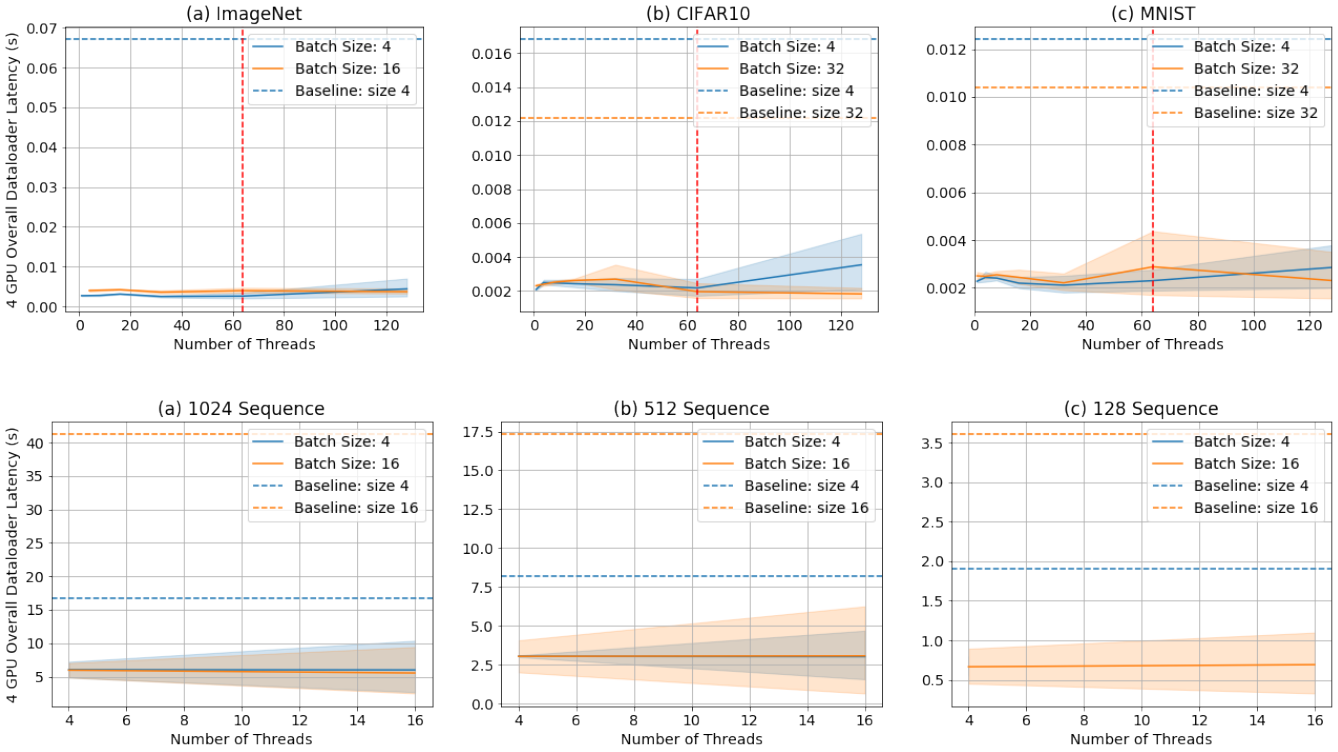

Figure 11: Overall Data Loading Performance from CPU to GPU memory (4 GPUs) on different datasets. In this charts, we compare our implementation of the dataloader to the original dataloader (Baseline) from Parax. Note: The red vertical dashed line is the number of physical CPU thread limit. Note: The batch size in ImageNet experiment is less than others because larger batch size does not fit in our GPUs. Note: The the latency of original dataloader with ImageNet is too high that makes the chart hard to read, so we decided to omit it.

thread communication time linearly increase with the number of threads. The cpu-disk overhead and cpu-gpu overhead also reflects the design choice discussion in the implementation section. (more threads will bring benefits in the start, but the diminished return will come later.) The cpu-gpu data transferring becomes less of the overhead even we increase the number of threads, and disk-cpu data transferring always becomes the largest overhead except for the largest thread number. The physical core number and thread number of our cpu will lead to the ridged performance in disk-cpu data transferring.

5 Conclusion

In this project, we presented an efficient data loader for machine learning training tasks on Parax. We designed a user interface for the data loader as an iterator of distributed array. With the iterator design of the data loader, we abstracted away the need of users to manually parallelizing the data loading part of their machine learning task. We experimented and compared the efficiency of different designs of data loader from PyTorch and TensorFlow. We use the data loader from PyTorch to load data from disk to CPU because of its lower latency. Then, we extended the data loader to distribute the data from CPU to GPU clusters to work on a machine learning training task in parallel, using multiple CPU threads. Using the prefetch operation, it also supports to overlap computation with data transferring.

As experiments, we run benchmarks to compare the new data loader's efficiency with our original 1-CPU data loader. We run experiment with multiple datasets including ImageNet, CIFAR10, and MINIST to represent Convolutional Neural Networks, and WikiText103 to represent Transformers. We found that our new implementation of the data loader outperforms the original data loader up to 6 times faster. By looking at the runtime proportion, the original data loader takes 5-20% of the total training time. Our new dataloader improves this proportion significantly, by only take 2-5% of the total training time. Finally, we investigated the breakdown of runtime in our data loader. We discussed the proportion of each subtask in the data loader, and how they behave differently between different datasets.

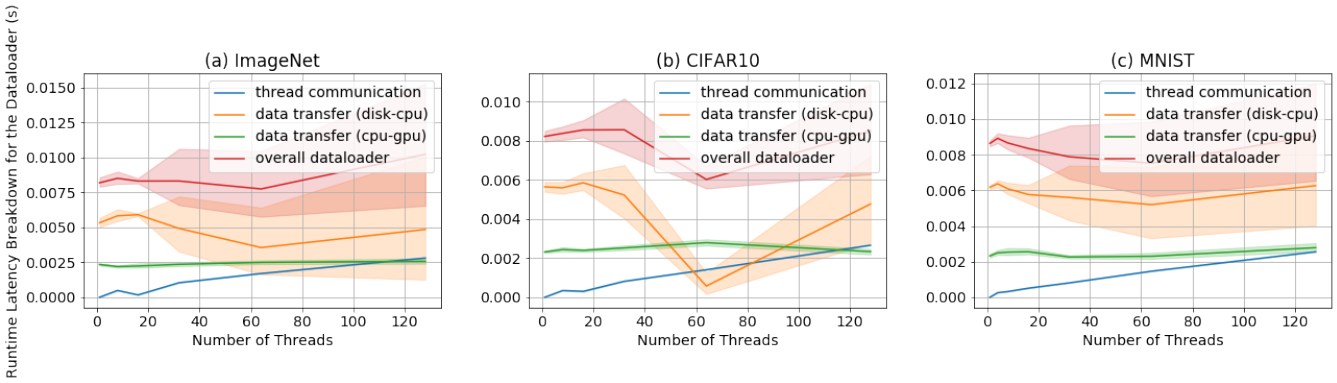

Figure 12: Runtime Breakdown in Data Loading Performance (4 GPUs) on different datasets (16 batch size).

6 Future Work

In this project, we only design the data loader to support model-parallelism and data-parallelism. In the future iteration of the project, we would like to investigate on how to load data for a machine learning task with pipeline-parallelism. With pipeline-parallelism, the data loader would be able to support all the parallelization strategies that Parax supports.

We have designed the user interface of the data loader with users' usability in mind. We also have implemented the data loader to efficiently distribute data across GPUs. Due to time constraints, however, the user interface and the implementation have not been integrated. As a future work, we would like to wrap the data loader we implemented with the user interface design to help data loading to be intuititive in Parax.

Right now, we use PyTorch's data loader to load datasets from disk into CPU based on its lower latency compared to TFDS's. However, we can see from Figure 4 that PyTorch's data loader is unstable when the batch size is small. As a future work, we would like to allow users to choose underlying data loader from disk into CPU. Moreover, we only benchmarked our setting using local machines as one GPU host cluster. We would like to extend to more GPU and more GPU cluster settings given the support of the sharded Dataset and dataloader. Also, Host to GPU copies are much faster when they originate from pinned (page-locked) memory. CPU tensors and storages expose a `pin_memory()` method, that returns a copy of the object with data put in a pinned region. This is effective when the data formats are tensor abstraction in pytorch data loader, which we do not fully utilize. We think the further exploration in this direction can also improve the data loading performance from CPU to GPUs.

Acknowledgments

We would like to thank Lianmin Zheng and Zhuohan Li for advising us on the project and its background. We would also like to thank RISE Lab for providing us with AWS Credits to work on the project. And finally, we would like to thank CS 262A staffs for guiding us through the project.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] 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, Dan Mane, 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 Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016.

[3] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 2018.

[4] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[5] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cour-

napeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[6] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32:103–112, 2019.

[7] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[8] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *SysML 2019*, 2019.

[9] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[11] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

[12] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[13] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding, 2020.

[14] Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Autograd: Effortless gradients in numpy, 2015.

[15] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[16] Sergio Moreno-Alvarez, Juan M. Haut, Mercedes E. Paoletti, and Juan A. Rico-Gallego. Heterogeneous model parallelism for deep neural networks. *Neurocomputing*, 441:1–12, 2021.

[17] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, 2017.

[18] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[19] 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. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[20] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[21] S Rajbhandari, J Rasley, O Ruwase, and Y He. Zero: memory optimization towards training a trillion parameter models. *arXiv preprints arXiv: 11910.02054 (2019)*, 2019.

[22] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[23] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *arXiv preprint arXiv:1811.02084*, 2018.

[24] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[25] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.