Data Loader for Parax

Sheng Shen[†*], Chanwut Kittivorawong[†*],
[†]UC Berkeley;

Problem

Currently, Parax lacks an efficient distributed data loader, making the data-loading become the bottleneck of training. With the current baseline implementation, Parax loads data using a single CPU. Then, it broadcasts the data to GPUs, using Ray. The current baseline solution does not leverage the parallelism strategy, which leads to redundant data loading and computing overheads.

Background

XLA [1] is a compiler for tensorflow. JAX [2] is a tensor processing library with a numpy interface and autograd capabilities. JAX also provides parallelization and a JIT compiler. Flax is a Neural Network library and ecosystem for JAX.
Parax is a JIT compiler for distributed tensor computation and large-scale neural network training. Parax is built on top of JAX and XLA. With Parax's python decorators, Parax can compile a tensor computational graph, generate a parallelization strategy and run it on a Ray GPU cluster. Parax searches for the best parallelization strategy in a comprehensive search space that combines intra-operator parallelism and inter-operator parallelism. Parax can automatically find and compose complicated strategies such as data-parallel, tensor partitioning and pipeline parallel.

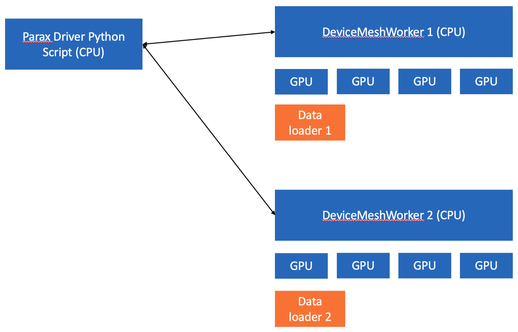

Figure 1: Illustration plot for the Dataloader.

We plan to build a data loader as an iterator that produces distributed arrays that can leverage the generated parallelism strategy of Parax and partition the data loading accordingly to multi-CPUs and broadcast to corresponding GPUs. We will focus on leveraging the generated parallelism strategy in this project. We plan to extend the current solution to utilize the parallelism strategy information and aim for a much efficient data loading. Currently google/flax has support for distributed data-loaders. However, it supports only data-parallelism (sharding) [3]. We would like to support all the parallelism strategies provided by Parax.

```
data_loader = DataLoader("file_path", physical_mesh, sharding_specs)
for batch in data_loader:
  state, metrics = train_step(state, batch)
```

Figure 2: Example usage of the data loader

Current Results

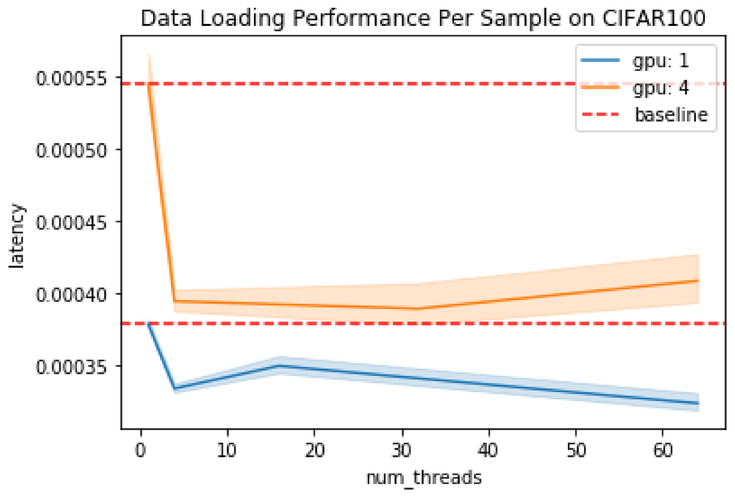

Figure 3: Overall runtime performance per sample on CIFAR100 for Wide-ResNet.

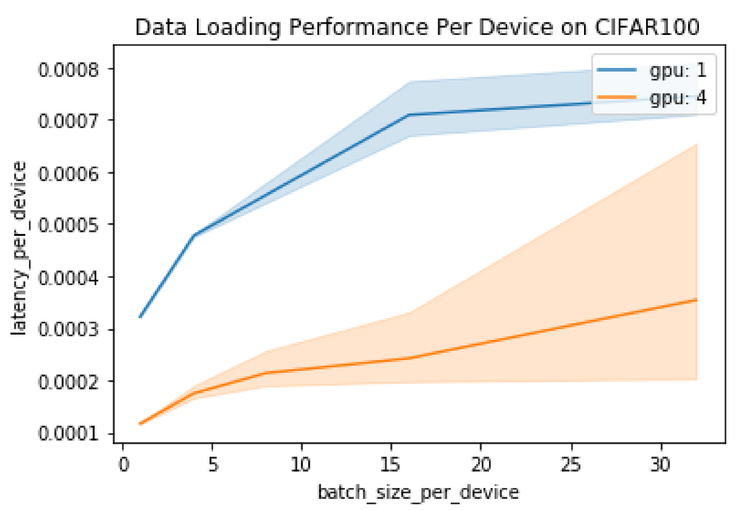

Figure 4: Overall runtime performance per device on CIFAR100 for Wide-ResNet.

For the current experiment, we use Wide-ResNet (50 layer, 256 hidden, 4 blocks) as our model and test on CIFAR100 (image resize to 224) and MNIST (image resize to 30) as our preliminary benchmark.
Specifically, we compare with a naive flax implementation where one CPU thread will be used to load and transfer data for all the GPUs in Parax. For our implementation, we only experiment with model and data parallelism so far. We will load and transfer data in the unit of device mesh. We also support multi-threading CPUs to load and transfer data, which yields better performance scaling when we increase the number of threads in Figure 3 and 5. For multiple GPU setting, large batch size will lead to more instabilities so far given the naive all-to-all communication we support between devices as shown in Figure 4.

Extended Results

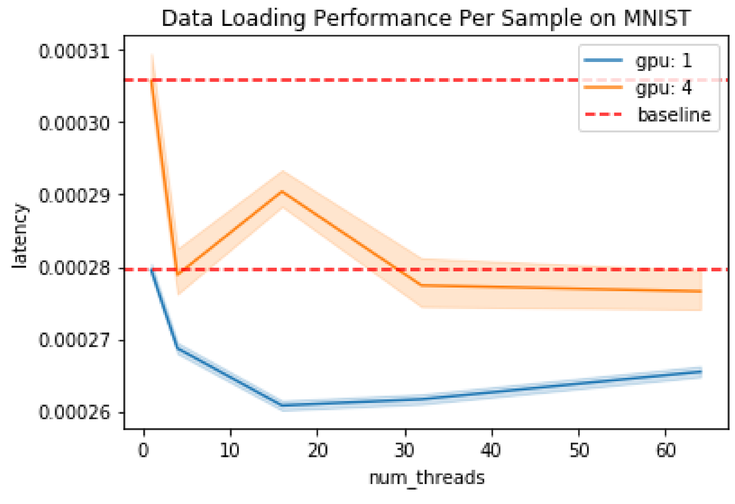

Figure 5: Overall runtime performance per sample on MNIST for Wide-ResNet.

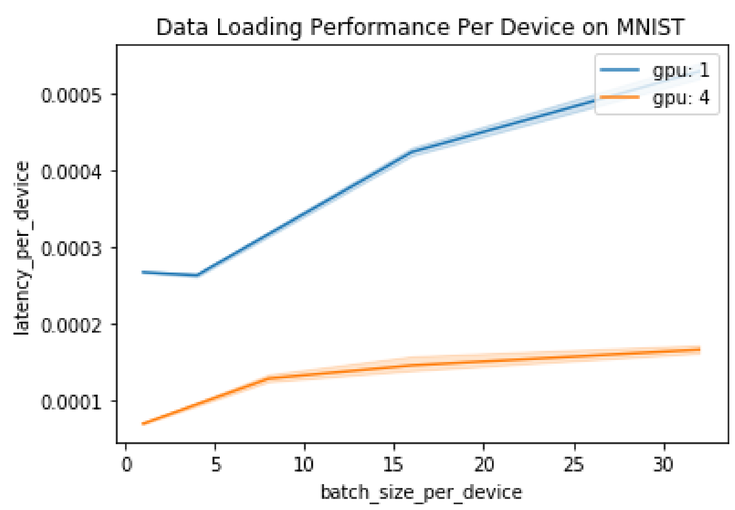

Figure 6: Overall runtime performance per device on MNIST for Wide-ResNet.

For MNIST, given the dataset size and image size are much smaller. We observe less variations in the performance and our implementation still outperform the naive baseline.
We will experiment with more different training models (ResNet representing CNN, BERT representing Transformer), more parallelism scheme in parax (pipeline parallelism), more data benchmark and improving our implementation in our final paper.

References

[1] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/compiler/xla
[2] Bradbury, James, et al. "JAX: composable transformations of Python+ NumPy programs." Version 0.1 55 (2018).
[3] Lepikhin, Dmitry, et al. "Gshard: Scaling giant models with conditional computation and automatic sharding." arXiv preprint arXiv:2006.16668 (2020).