Multi-GPU Spatio Temporal Neural Network

This document will outline the work completed over the past semester on modifying a spatio-temporal forecasting model to be able to train on spatially-large datasets.

Introduction

Spatio-Temporal Neural Network (STNN) was a model introduced by Ziat et al. 2019 that demonstrated SOTA results on a wide variety of spatio-temporal forecasting tasks (e.g. traffic, disease, ocean-temperature). It did so, in part, by representing the inter- and intra-dependencies between and within each timeseries. Consider an arbitrary dataset $D \in \mathbb{R}^{a \times b \times T}$ where a, b are the sidelengths of the spatial area and T is the length of the time series at each pixel. The inter-dependences, or the dependency between each pixel and every other pixel, are represented via a relational matrix $W \in \mathbb{R}^{n \times n}$ where $n = a * b$. As such, the total number of pixels grows quartically to the side length of a given spatial area. That means training a model on an area 100 pixels $\times$ 100 pixels requires $\sim 16\times$ more memory than an area 50 pixels $\times$ 50 pixels. In practice, we found that the model, as it was published on GitHub on a V100 GPU with 16GB of memory, could only train a 50×50 pixel area.

Training larger areas could only be accomplished by downsampling, decreasing the area of interest or a combination of the two. This was problematic because spatio-temporal patterns often occur over large areas while useful forecasts require a certain level of grandularity. The need to adapt this model to work on larger datasets, i.e. multiple GPUs, became obvious.

Methods

The main challenge in parallelizing the model was how to go about storing the adjacency matrix in a way that balanced accessability with computational effeciency. The adjacency matrix is quite sparse meaning storing it in a sparse representation could allow it to easily fit in device memory—assuming the area is not so large that its sparse representation exceeds a single devices memory. On balance, such an approach would yield a large increase in the maximum trainable area. However, for the refinement network version of the model, the tensor operations needed are not supported in PyTorchs COO sparse tensor class.

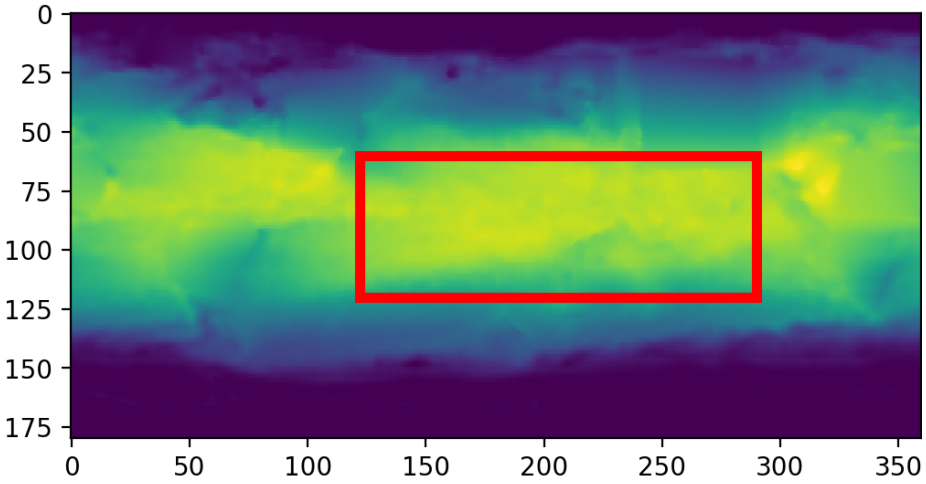
The next approach was to implement a multi-gpu model at a much lower level using CUDA/C++. Nvidia recently released a mutli-GPU verson of their cuTENSOR package, cuTENSORMg. The benefit of this package was its automatic management of distributed tensor contractions accross host and device GPU. Using this packaged proved to be a challenge for multiple reasons, not least of which were because of a lack of experience with CUDA, an unusual

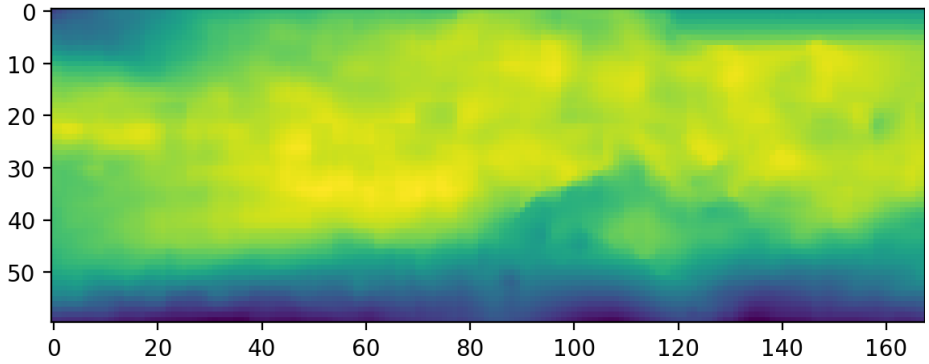
API compared to other Nvidia libraries and a lack of documentation/support due to the library being so new.

This lead to the final implementation which was to use PyTorch's built in device conversion method `.to(device)` to integrate hard-coded memory mangement directly into the runtime of the model. At a high level, this method requires a minimum of 2 GPUs (GPU A and B) and a host with sufficient memory to store the adjacency matrix. The model itself sits on GPU A and contains an encoder and decoder network. The adjacency matrix sits in host. Recall the adjacency matrix $W \in \mathbb{R}^{n \times n}$. However, many spatio-temporal patterns have mutliple relations, so $W \in \mathbb{R}^{r \times n \times n}$ where r is the number of relations. When the model calls `get_relations()`, a subset $W' \in \mathbb{R}^{r' \times n \times n}$ where $r' \in [0, r]$ is transfered from host to GPU B where it is masked per a set of learned weights and then passed back to host. This is repeated $r \mod r'$ times until the entire relational matrix is passed. This same procedure is done for the matrix multiplication steps as well.

Data

In order to make the results of this work comparable, we choose to replicate the Pacific ocean temperature experiment in the Ziat et al. In that experiment, they resampled an area of the pacific ocean of size 60×168 pixels where each pixel represents 1×1 degrees down to a an area 30×84 with a pixel resolution of 2×2 degrees.





As a demonstration of the purpose of a multi-GPU model, we skipped the downsampling step, maintaining the original 60×168 pixel area. Although it is an area twice the size, it represents 4x as many pixels and requires an adjacency 16x as large. ## Results

The full resolution dataset when passed into the model on a single V100 GPU with 16GB of memory overflowed.

It is difficult to quantify the effeciency of this implementation because its runtime on a large dataset on a single GPU is intractable since it won't fit on GPU. As such, we can only concretely quantify its speed up coompared to doing all tensor operations on host.

However, we can potentially esimate the runtime. Dece AI. examined runtime performance and effeciency for matrix * vector computation on GPU for various matrix size. They found that for a GPU, runtime increased by 5x for a matrix size increase from $2,500 \rightarrow 10,000$. We can use this to roughly upper-bound the theoretical runtime on a single GPU.

Model	relation size	# batches/sec
Small dataset	2,520	40
Full Res single GPU	10,080	**8***
Full Res multi GPU	10,080	3
Full Res CPU	10,080	.03

* This is an estimate based on the runtime metrics above.

The above runtime estimates gives an very conservative upper-bound on theoretical optimal runtime performance of this model. STNNMg in its current implementation is slower by factor of 3 compared to a theoretical single GPU implementation but 100x faster than a CPU only implementation. The increase in data generated an overall 14x slowdown in performance and a 5x slowdown versus our theorized single GPU runtime.

The model ultimately achieved similar accuracies to the model in Ziat et al., due to the fact that the core architecture did not change.

Discussion

While the results here are promising, there are many limitations. First, the model is limited to 2 GPU setup.. As such, a more robust framework should be built for spreading the adjacency computation over multiple GPUs instead of just spreading the model over mutliple GPUs as was done here. Furthermore, there are GPUs with larger memoery, A100s are available in 40 and 80GB versions. That being said, given the superlinear scaling of the memory requirements, a 2.5x increase in memory is unlikely to yeild a significantly larger trainable area. Finally, the adjacency matrix is very sparse and the optimal strategy would be to implement this using a sparse matrix representation.