ORNL/TM-2021/2168

Distributed Training for High Resolution Images: A Domain and Spatial Decomposition Approach

Aristeidis Tsaris
Jacob Hinkle

August 2021

OAK RIDGE National Laboratory

ORNL IS MANAGED BY UT-BATTELLE LLC FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2021/2168

Computing and Computational Sciences Directorate

Distributed Training for High Resolution Images: A Domain and Spatial Decomposition Approach

Aristeidis Tsaris, Jacob Hinkle

August 2021

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

ABBREVIATIONS

PyRPC	Pytorch Remote Procedure Call
DSN	Domain Separation Network
Multi-DSN	Multiple Domain Separation Network

ACKNOWLEDGMENTS

This research is sponsored by the AI Initiative as part of the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the US Department of Energy under contract DE-AC05-00OR22725. This research used resources at the Oak Ridge Leadership Computing Facility, a DOE Office of Science User Facility operated by the Oak Ridge National Laboratory.

ABSTRACT

In this work we developed two Pytorch libraries using the PyTorch RPC interface for distributed deep learning approaches on high resolution images. The spatial decomposition library allows for distributed training on very large images, which otherwise won't be possible on a single GPU. The domain parallelism library allows for distributed training across multiple domain unlabeled data, by leveraging the domain separation architecture. Both of those libraries were tested on the Summit supercomputer at a moderate scale, and we are releasing the code for both of them.

1. INTRODUCTION

Deep learning training is computationally expensive, but usually leadership computing facilities and large cloud services can keep up with the growing datasets in terms of volume. The most commonly used distributed approach is data parallel, were most deep learning frameworks offer mature solutions. Usually it requires some performance tweaks to get linear weak scaling, images/sec, but most default knobs will give good performance at a moderate scale. On the other hand, running on data-parallel at very large scale to optimize time-to-solution can be challenging because of convergence to solution issues, such as the generalization gap [6]. Also it is usually impractical to do an extensive hyper-parameter search at very large scale, since parameter space is changing for different scales.

Model parallel approaches are on higher demand, due to the increasing sample size of the datasets and deep learning models are becoming more complex. Usually this approaches are less flexible and require a significant effort and expertise to adjust from the single-GPU implementation to a multi-GPU solution. On the other hand, usually they don't change the parameter space of the problem, as data-paralle approaches do, and usually they don't need a new hyper-parameter search. Tensorflow already has a library that implement receipts for model parallel approaches, [3]. Pytorch on the other hand recently introduced a distributed framework that allows automatically differentiate and distribute gradients across machines: PyRPC. We expect PyRPC to be a very popular API for distributed automatically differentiation on accelerators in the future, and even though as of Pytorch 1.9, PyRPC is still in beta version in terms of CUDA support, performance improvements are expected in later versions. In this note we are fully utilizing the PyRPC functionalities for our approaches.

We have implemented two libraries, one for spatial decomposition, called BigConv, based on [3], and one for domain parallelism, called Multi-DSN, by parallelising this [1] architecture. On modern deep learning applications we see a rapid increase of the image-size and model size, and the GPU memory capacity sometimes can't keep up with the memory demand. Usually compromises need to be made to fit on a single GPU which might affect the overall performance. BigConv tries to address exactly this by splitting on non-overlapping patches large images and distributing train training across GPUs. For example some of the most common areas that produce very large 2D/3D datasets are medical, microscopy and geospatial sciences.

At the same time, these domain areas can have multiple instrument sources for the same target, for example different angles of a satellite for the same geographic location, or different medical scanning methods targeting the same body area. Usually it is impractical to have detailed labels for all those data, and un-supervised or semi-supervised approaches are used. Domain separation network [1] targets this challenge by using a domain adaptation method. Our implementation, Multi-DSN, offers a distributed

approach to this architecture, so a large number of domains can be trained at the same time, with potential to have larger models as well. In the following sections we describe the implementation of those libraries, and the test runs that we performed.

2. SPATIAL DECOMPOSITION LIBRARY

Deep convolutional neural networks have evolved to obtain impressive predictive performance for a variety of computer vision tasks. The networks are designed as a series of local operations like pooling and convolution, along with pixel-by-pixel operations such as batch normalization, skip connections, and activation functions. Through composition, these local operations combine to form a much larger "receptive field" indicating how many pixels are available in the field of view when computing each pixel of a feature image. Recent work in convolutional neural network design has focused on increasing the receptive field in order to make better use of coarse-scale information while retaining fine-scale feature resolution. Doing so requires computation of networks with large activation tensors and the use of high-resolution images that are large enough to contain enough context to be useful.

The central challenge to computation of these models is the limited memory size on modern GPUs. For example, a single GPU on Summit contains 16GB of video RAM, limiting the size of a trainable DenseNet121 [4] model to a size of roughly 1024x1024 pixels. This limits the use of large expressive networks as they cannot be trained on an individual GPU for the image sizes needed to exploit the resolution and context the network was designed to exploit. In addition, there is a number of domains that have ultra high resolution images, such as in geospatial or medical domain, that can be in the order of $10^5 \times 10^5$ pixels, which they can't fit even on a smaller GPU cluster, with a moderate size neural network.

2.1 IMPLEMENTATION

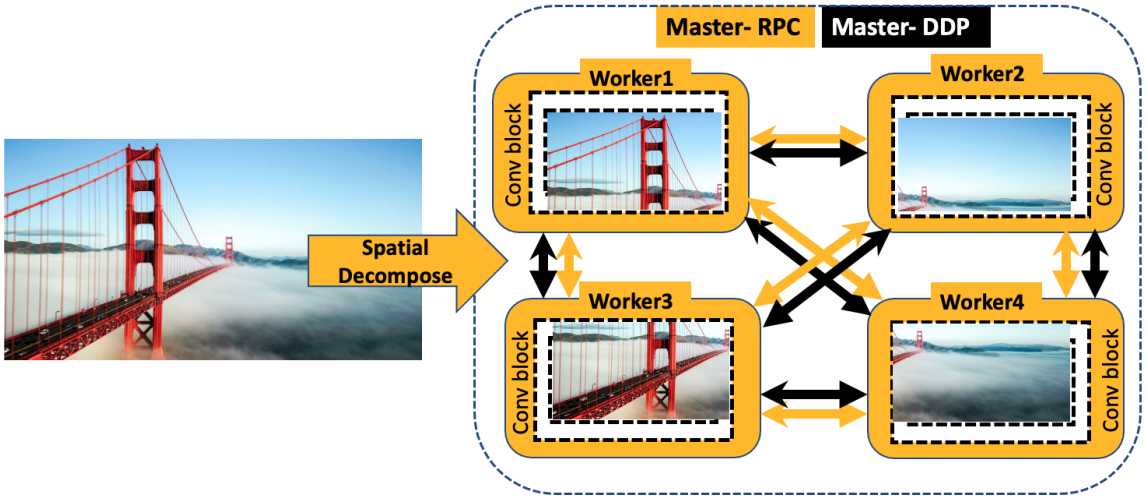

Figure 1. An illustration of the spatial decomposition library. In this picture each worker corresponds to a single GPU, and two communication paradigm are applied: all-reduce with Pytorch Data-Parallel and asynchronous calls with Pytorch RPC.

In order to overcome this barrier, we implemented a paradigm called spatial parallelism, using PyTorch RPC, as shown in Figure 1. In this approach, we first initialize RPC and Data-Parallel master and workers in Pytorch. The image domain is then parallelized automatically into a spatial grid according the image size, dimension and the total number of workers that are given. Each grid location corresponding to an RPC worker and in our case each worker is mapped to a GPU. Each worker holds each own chunk of data and computes a convolutional "block", requiring context from neighboring grid locations (called halos) that

are exchanged before computing the block. In the forward pass the workers exchange a small number of rows and columns from the left and right neighbor grids, depending on the filter and padding size, and the outermost grid workers that don't have neighbors are padding with zeros for the remaining rows and columns. All those communication calls are handling with RPC asynchronous calls. In the backward pass, apart from the data from the forward calculation, weights need to be shared among workers for the optimization, and we use all-reduce communication offered in the Pytorch Data-Parallel paradigm. After those communications are setup, PyTorch RPC enables us to automatically backpropagate gradients, even across node boundaries in the cluster, including these halo exchange steps.

We have implement four very basic models to test our library, i.e. CNN1D, CNN2D, CNN3D and FCN2D. In all cases we confirmed identical values for predictions and gradients between our approach and the single GPU implementation of those models. In the following section we test the memory size as a function of the input vector for those models.

2.2 SPATIAL DECOMPOSITION TEST RUNS

In this section we present the maximum memory used and the average GPU utilization for four models: CNN1D, CNN2D, CNN3D and FCN2D on various input size images. In all cases we used batch size of one, so training one image at a time to get the very minimum memory requirement. In realistic training conditions it will require a larger batch size of a good optimization, and so the following results will be more pronounced in real cases.

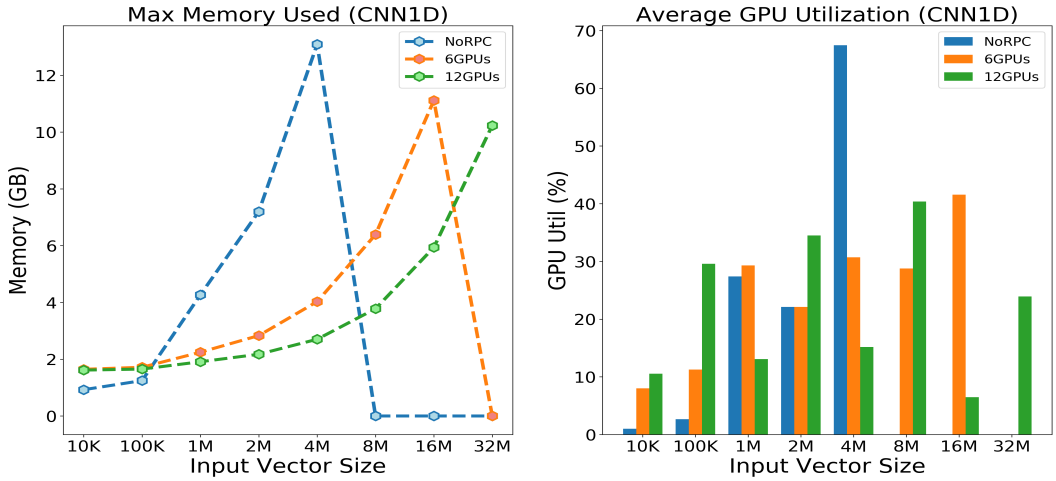

Figure 2. The Figure shows the maximum memory used and the average GPU utilization on an NVIDIA-V100 GPU. We used a vanilla CNN1D model for various 1D input size vectors with batch size of one. The nvidia-smi tool was used to acquire data for the plots.

Figure 2 shows the memory limits for a simple CNN1D model for various input size vectors. The single GPU implementation, shown in the blue curve, can fit up to 8M size 1D vectors, before we run out of memory. After that limit we have to use the spatial decomposition approach, as described in the previous section to distribute the memory across GPUs. The orange curve shows smaller memory footprints for vector sizes that can fit on a single GPU, and can go twice the size of the single GPU 1D implementation using six GPUs. We also show the case of 12 GPUs, without exploring the out of memory limitations there.

It is important to note that we are using the Summit HPC system for this studies, were one Summit node has six GPUs. The reason we show 12 GPUs is precicily to demonstrate the capability of breaking the node barries with Pytorch RPC and use a very large number of GPUs if need it.

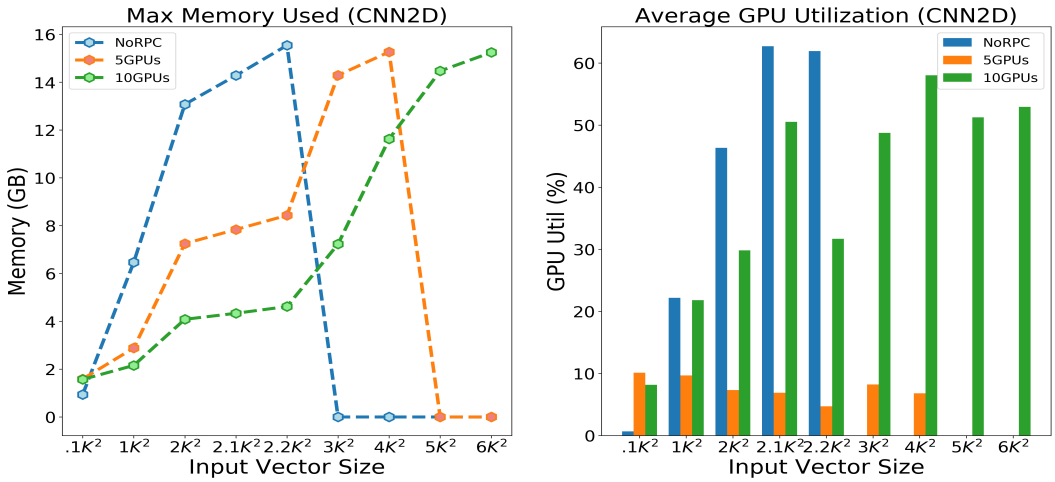

Figure 3. The Figure shows the maximum memory used and the average GPU utilization on an NVIDIA-V100 GPU. We used a vanilla CNN2D model for various 2D input size vectors with batch size of one. The nvidia-smi tool was used to acquire data for the plots.

Figure 3 shows the memory limits for a simple CNN2D as a function of 2D input vectors. The single GPU implementation limits us to go beyond a 2200x2200 pixel image where on the 5GPU case we can go up to 4Kx4K pixels, and using 10GPUs we can fit even a 6Kx6K pixels. Even in the 1Kx1K pixel image if we compare the single GPU case, with the 10 GPU case, the memory foot print is three times lower, which is an indication that we can fit much deeper models if we use our spatial decomposition method.

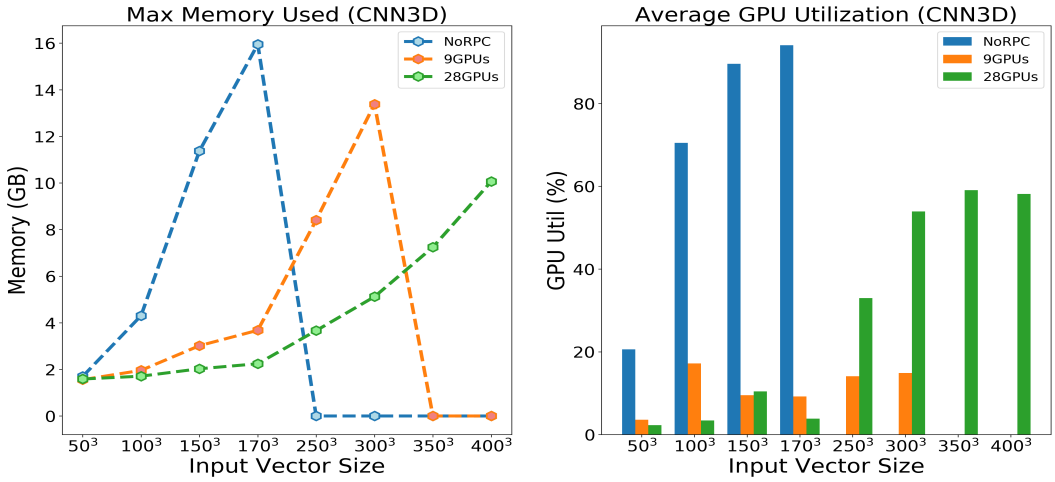

Figure 4. The Figure shows the maximum memory used and the average GPU utilization on an NVIDIA-V100 GPU. We used a vanilla CNN3D model for various 3D input size vectors with batch size of one. The nvidia-smi tool was used to acquire data for the plots.

Figure 4 shows the memory limit for a simple CNN3D as a function of 3D input vectors. Here we can see

immediately that fitting naturally 3D data on a single GPU limits significantly the input size of the image and some type to distributed memory approach is need it. Specifically for the single GPU case we can't go beyond 170x170x170 image size, and given that the size of our model is as small as we can go and we only use batch size of one, some type of cropping or down-sampling needs to be perform for the single GPU case. In addition, the way we make the partition of the data, we need to exceed the single node limits for Summit to perform spacial decomposition. We use 9 GPUs were we can fit up to 300x300 3D vectors, and using 28 GPUs we can fit 400x400 pixel images.

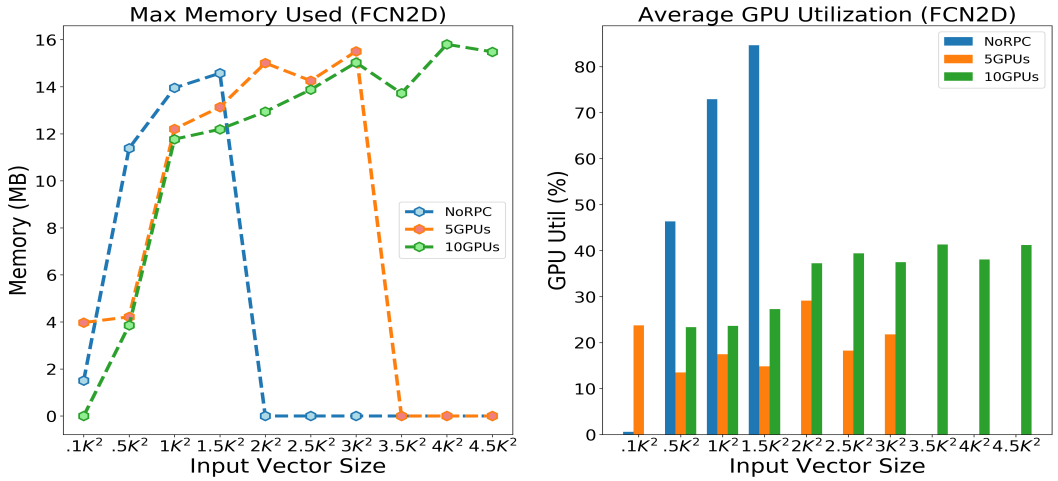

Figure 5. The Figure shows the maximum memory used and the average GPU utilization on an NVIDIA-V100 GPU. We used a vanilla FCN2D model for various 2D input size vectors with batch size of one. The nvidia-smi tool was used to acquire data for the plots.

Figure 5 shows the memory limit for an FCN2D model [7] as a function of 2D input vectors. This is a realist network architecture, though far from state-of-the-art, that has been used extensively with image data for semantic segmentation. There we can see the memory foot print isn't reduced as much between single GPU and multi-GPU implementation. This is expected since we already using an expensive network. We see though that we can only go up to 1500x1500 2D input vector size without using spatial decomposition, and as we go on higher resolution images we need to use RPC. With our 10 GPU implementation we can go up to 4500x4500 pixel image sizes.

In terms of GPU utilization, we can see a general trend that no-RPC case is better utilized than RPC case. This is expected since for the no-RPC case there are no worker communication and all the calculation stays on GPU, once it is loaded. In the spatial decomposition case, there is the overhead of the communication between workers. Also in the RPC version that we are using the GPU-GPU communication wasn't available and so we have to go to CPU to make the communication between workers. In Pytorch 1.9, NCCL is supported for GPU-GPU communication, but even there the RPC is on beta version in terms of CUDA support, so we should expect better performance in later Pytorch versions. On the other hand we see better utilization as we increase the number of workers for the same input image. Most likely this is because with more workers we can train faster the same image, since each worker computes a smaller chunk of data, and so better utilize the hardware.

3. DOMAIN PARALLELISM LIBRARY

One of the biggest challenges in supervised learning is the lack of detailed labeled data. This is a common problem in science too, as data is often heterogeneous and label availability varies greatly. Data are usually been acquired by different instruments, for example on medical field varying medical instruments can target same parts of the body making a dataset very heterogeneous. Also different conditions of acquisition, for example varying satellite angles, or different geographic locations can make labeled data on all cases an ideal situation that don't reflect realistic datasets.

Domain separation networks [1], DSN, are addressing this challenge by learning at the same time from label and unlabeled domains and keeping domain-invariant feature across the dataset. In the original DSN implementation there is a source domain with labeled data, and a target domain with unlabeled data. Both are inputs to the network at the same time, and each one has two encoder/decoder network: the private enoder helps to keep the invariant features across domains, and the shared helps on mapping the shared representation between domains.

DSN is computationally very expensive to train due to the many orthogonal components of the network, and also due to the multiple image inputs. Even with a very small encoder/decoder part of the network, we can't fit on an NVIDIA-V100 GPU larger than 768x768 pixels with batch size of two. If we wanted to use higher resolution images, or more realistic classification networks we would immediately be out of memory. Also in most realistic situations there are more than two domains, or more that two variations from the original dataset, that are lacking of labels and so to be able to train those network architectures a distributed memory approach is need it.

3.1 IMPLEMENTATION

Figure 6 shows our approach to distribute memory across GPUs for the DSN architecture, for multiple domains. In this approach, we first initialize RPC and Data-Parallel master and workers in Pytorch. Then each private and shared encoder is assigned to a worker/GPU, and has each own input image. The latent space is been calculated in each worker separately, were the shared encoder shares weights between them using all-reduce with the Pytorch Data-Parallel paradigm. Once all the latent space is been calculated, we use RPC asynchronous calls to calculate the losses, and continue with reconstruction and classification on a separate worker/GPU. The gradients are propagated automatically with Pytorch RPC across GPUs.

The big advantage of our method, apart from reducing the memory footprint of the network, is that in principle we can train at the same time a very large number of domains, since we assign workers beyond the node boundaries, and the limitation might come from the communication size. In the following section we test the memory limit as we include up to seven domains with unlabeled data.

3.2 DOMAIN PARALLELISM TEST RUNS

Figure 7 shows the memory limits of using the single GPU implementation of DSN against our domain parallel implementation. The one-source-one-target implementation on a single GPU is the same architecture as in the original DSN paper [1] . The one-source-one-target implementation on six GPUs is our domain parallel implementation where each encoder is mapped to a worker. The

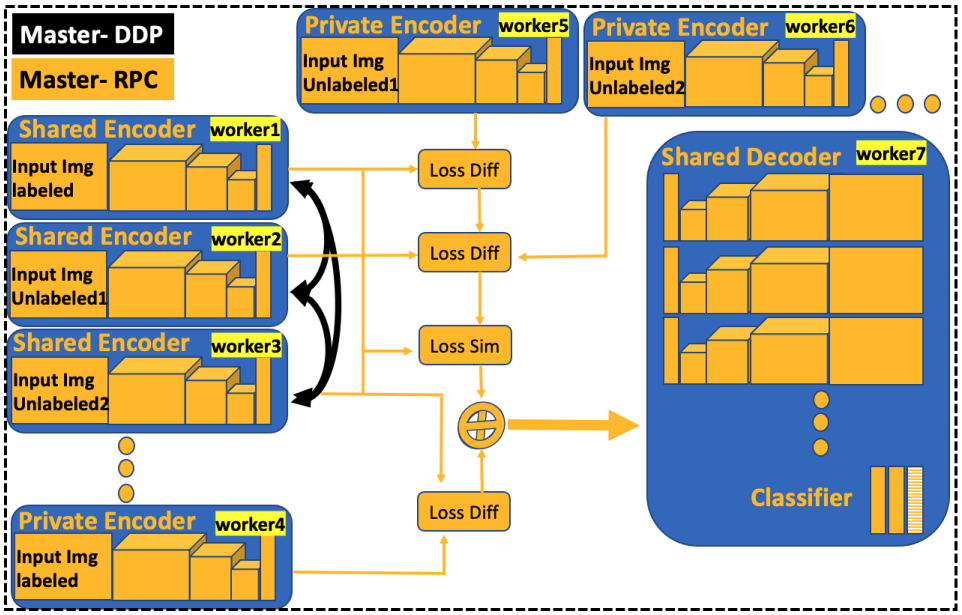

Figure 6. An illustration of the domain parallelism library. This is a distribute approach of the original DSN [1]. Each encoder is assigned to a worker/GPU. The communication on calculating the losses from the latent space of each encoder is done with PyRPC asynchronous calls. The shared encoders also share weights among them with the Data-Parallel paradigm.

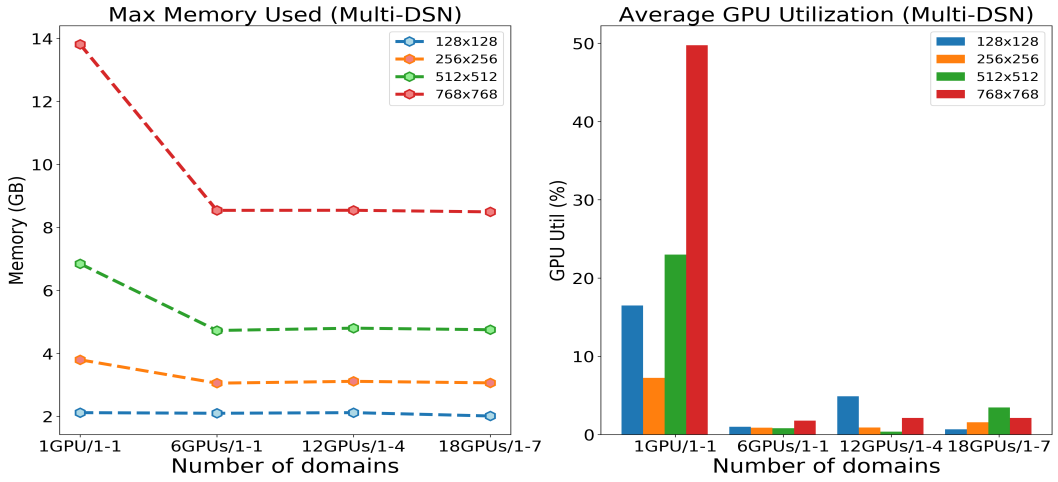

Figure 7. The Figure shows the maximum memory used and the average GPU utilization on an NVIDIA-V100 GPU. We used the DSN architecture, as shown in [1] for the one-source-one-target case, and the DNS architecture as shown in Figure 6 for the multi-GPU case. The one-source-one-target on six GPUs, and the one-source-four-target and one-source-seven-target are shown using Pytorch RPC for various input size vectors, with batch size of two. The nvidia-smi tool was used to acquire data for the plots.

one-source-four-targets, and one-source-seven-targets is again with our approach and assuming we have one source data with labels and four and seven domains respectively without labels.

For the single GPU implementation we are already very close to the maximum memory capacity of the

NVIDIA-V100s GPU by fitting a 768x768 2D size vector, using just batch size of two and with a very simple encoder/decoder network. Using our distributed approach for the same architecture, we can see on the one-source-one-target six GPU implementation the memory footprint is already reduced by few GBs. In that sense we have the capability to fit larger models, in order in order to increase the size of the effective receptive field, and eventually the accuracy of the model.

In addition, the memory footprint for the parallel implementation of one-source-one-target, one-source-four-target and one-source-seven-target remains flat. This is expected since even though we increase the number of unlabeled domains, the model size and the number of input images stays the same per GPU, since we distribute the load of private and share encoders to more workers/GPUs. In principle we can scale this number very high until it becomes communication bound, primarily due to the sharing weights among the encoders.

In terms of GPU utilization, we can see that resources are better utilized for no-RPC implementation compare with the parallel approach. This is no surprise since, in the former case a much larger network needs to be computed compare to the latter case. Also in the non-RPC implementation there are no workers and so no overhead by communication between them. On the other hand we can see mostly a flat and very low utilization as we increase the number of domains. That most likely reflects the fact that our encoders are very small compare to the GPU capacity and we have much more room to fill out more effectively the resource, with either larger model or higher resolution images.

4. CONCLUSION

In this work we developed two Pytorch libraries that target distributed memory applications for high resolution images. We showed that both libraries, as designed, reduce the memory footprint of the single GPU implementation for the same model architecture. Also with our approach we increase the scaling capability for distribute training on high resolution images from multiple domains, something that isn't possible for the single GPU case. By doing spatial decomposition, naturally we can train ultra-high resolution images, and study the effect of increasing the receptive field for finer scale feature resolution. As far as we can tell this is the first implementation of spatial decomposition in Pytorch. Mesh-TensorFlow [3] offers spatial partition for some particular models as well as the LBANN framework [2]. With domain parallelism, we can train images from multiple domains, and one can study in detailed the effect in model accuracy of training with a few labeled data, along with a large scale unlabeled data from multiple domains.

Both developed distributed approaches are orthogonal to each other, along with data-parallelism and pipeline parallelism [5] and can run at the same time. For example we can think of an ideal use scenario for all four: We have an experimental instrument that produced ultra-high resolution 3D images, and we need a very deep 3D model for a specific task that labels are available for some conditions but not for the majority. Starting with the distributed DSN architecture, as described in Figure 6, each DSN encoder worker can occupy few nodes each with few GPUs. Then each DSN worker can perform spatial parallelism with some GPUs from each node, in order to fit this very large 3D images, and within each node model pipeline parallelism can be made with the remaining GPUs to fit the large 3D model. In principle in this approach so far we haven't changed the parameter space of the task at hand, but we have rather distributed across machines. Now if the data size is large, we can also apply data parallel on top, where each of the above setup will be copied on a number of GPUs, and all-reduce will be used for communication between them for sharing weights.

As our next step we would be working on applying the above libraries on a real application so that we can study how this capability can be translate to a better model accuracy, which is the ultimate goal. We release the code for both libraries and it can be found here: https://code.ornl.gov/ai/scalability/multi-dsn, https://code.ornl.gov/ai/scalability/bigconv. For the spatial decomposition library we are going to provide soon models for UNet3D, and DenseNet. Also it is worth noting that the original proposed DSN architecture is very general and adaptive, and so the same principles for the domain parallelism can be used with different encoder-decode architectures, or other than classification tasks, such as segmentation or object detection. Finally, as it was discussed in the introduction section, the throughput (images/sec) performance isn't as expected, and in most studied cases it is comparable with the single GPU implementation. When Pytorch releases a more stable CUDA support pyRPC versions, we would start optimizing throughput performance for both libraries, and we will provide examples with GPU-GPU direct communication for both implementations.

References

[1] Konstantinos Bousmalis, George Trigeorgis, Nathan Silberman, Dilip Krishnan, and Dumitru Erhan. Domain separation networks, 2016.

[2] Siegfried Cools, Jeffrey Cornelis, Pieter Ghysels, and Wim Vanroose. Improving strong scaling of the conjugate gradient method for solving large linear systems using global reduction pipelining, 2019.

[3] Le Hou, Youlong Cheng, Noam Shazeer, Niki Parmar, Yeqing Li, Panagiotis Korfiatis, Travis M. Drucker, Daniel J. Blezek, and Xiaodan Song. High resolution medical image analysis with spatial partitioning, 2019.

[4] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018.

[5] 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.

[6] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima, 2017.

[7] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation, 2015.