SoSe 2020 - PPT-Lab Report

Duc Minh Nguyen - Matrikelnummer: 2368494

Juli 2020

1 Introduction

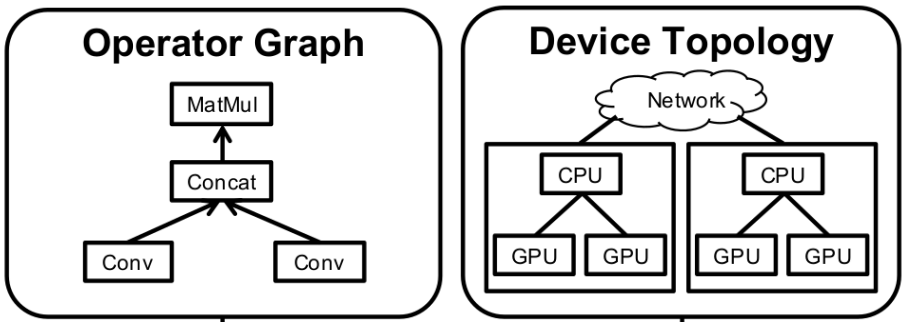
Training large to very large deep learning model has been becoming more and popular nowadays thanks to significant improvements of computational capabilities ranging from not only general-purpose GPU (GPGPU) but only more application-specific computational unit like TPU, ASIC. Therefore, there must be some other ways to leverage the enormous power of such heterogeneous systems which comprise of many compute devices, i.e. CPUs, GPUs, TPUs, ASIC and even FPGA. Such training strategy is basically classified into two main classes, which are data parallelism and model parallelism. In terms of data parallelism, this feature has long been well supported by many typical and popular machine learning libraries such as Tensorflow from Google or PyTorch from Facebook, which certainly enable machine learning model architects to mostly focus on logical model architectures without having to take low-level engineering implementation into consideration, i.e how to partition and put data, tensors, and parallelise matrix operations. Meanwhile model parallelism is yet to be widely explored or combined with data parallelism to make full use of such mentioned heterogeneous systems rather than only manually splitting a too-larger-than-one-GPU models. Recently there have been also some proposed approaches by tech giants only who could have the means to experiment with such too larges models which somehow addresses the combination problem of model parallelism and data parallelism seriously. However, all of them seem to require model architects and developers rewrite all existing models in another language or framework which are not fully supported and still under development. Although one of the approaches really tries to make the model developing paradigm transparent to any heterogeneous systems, the parallelism strategy still requires domain knowledge expert to make the training phase optimal, otherwise communication and synchronisation overhead of such heterogeneous and distributed systems definitely hurt the training performance. For this reason, hereby I propose a novel approach how to automatically and algorithmically come up with an optimal parallel training strategy which can be a combination between data and model parallelism and could potentially reuse any pre-existing models in popular machine learning frameworks by utilising distributed training feature of that framework to represent the best found parallel training strategy in action.

2 Related Work

Data and model parallelism have been widely considered in the beginning of the deep learning boom. AlexNet[5] ultilised CUDA C/C++ programming to parallelise their computation in terms of data with two similar model replica which is synchronously updated with respective batch gradient descent. DistBelief[2] also took model parallelism into account to some extent but it was merely splitting the DNN into different parts to be put and trained on different dedicated device and this has to be done manually. Very recently Google proposed Mesh-Tensorflow[8], which is a Tensorflow-like language facilitating building model parallelism architectures. However, Mesh-TF does require developers to rewrite any models no matter what they exist before in the model zoos. More importantly, developers have to manually split the model and place them onto compute devices accordingly. Megatron-LM[9] does address the model parallelism problem, but this applies strictly to language models, let alone universally any deep learning models. FlexFlow[3] both solves not only the model and data parallelism combination, but also how to automatically and optimally determine a parallel strategy, given a compute paragraph (only deep learning model) and the device topology. However, FlexFlow is yet to be favoured by the deep learning community, because FlexFlow is built on top of Legion, a special parallel programming library, so that FlexFlow also could not take advantage any of the popular deep learning framework like PyTorch and Tensorflow. One must write the model again using Legion programming paradigm in C/C++ with CUDA if one wants to use FlexFlow to achieve the automatic strategy finding process.

3 FlexFlow

FlexFlow uses graph $\mathcal{G}$ to represent an operator graph, which describes all operators and state in a DNN (Figure 1a). Each node o_i which belongs to the operator graph $\mathcal{G}$ is an operator, this could be matrix multiplication, convolution, etc. In addition, FlexFlow also employs a device topology graph $\mathcal{D}$ to represent all available hardware devices and their interconnections (Figure 1b). Subsequently, FlexFlow takes the operator graph $\mathcal{G}$ and its equivalent device topology graph $\mathcal{D}$ as inputs and automatically finds an efficient parallel strategy in the SOAP search space. This can be done by FlexFlow maintaining an Execution Optimiser which comprises of an Execution Simulator and MCMC Search algorithm (Figure 2).
Execution simulator would execute and simulate every task in the operator graph $\mathcal{G}$ to statistically calculate the runtime for each compute task and communication cost for each communication task whenever data communication is required based on the given device topology graph $\mathcal{D}$. This process is called performance simulation and every operator graph has an equivalent simulated performance. The lower this number is, the better the strategy would be.
After having calculated the final simulated performance of an operator graph, Markov Chain Monte Carlo sampling algorithm would propose a new candidate



(a) Operator graph. (b) Device topology graph.

Figure 1: Two input graphs of FlexFlow execution simulator.

strategy by randomly modify one operator in the current parallelisation strategy. This new candidate strategy is then again calculated by the Execution Simulator because this is a heuristic optimisation process, and therefore, could be solved by an iterative process. After a sufficient number of iterations, the possibly Best Found Strategy is yielded and this strategy could be deployed and implemented by any deep learning frameworks, either by Legion runtime underneath of FlexFlow or any typical popular deep learning frameworks which support distributed training paradigm (both Tensorflow as PyTorch in this case).

4 Data and Model Parallelism with PyTorch

For the time being, PyTorch does support data parallelism very well with many API and libraries, such as `DataParallel`[4], `DistributedDataParallel`[6], and even with low-level and much more freedom in parallelism customisation like `torch.distributed`[1].

Both `DataParallel` and `DistributedDataParallel` support multi-GPU training in terms of data paralellism (Figure 3). However `DataParallel` is single-process, multi-thread, and only works on a single machine, while `DataParallel` is usually slower than `DistributedDataParallel` even on a single machine due to GIL contention across threads, per-iteration replicated model, and additional overhead introduced by scattering inputs and gathering outputs. If the model is too large to fit on a single GPU, you must use model parallel to split it across multiple GPUs. `DistributedDataParallel` works with model parallel; `DataParallel` does not at this time. When DDP is combined with model parallel, each DDP process would use model parallel, and all processes collectively would use data parallel (Figure 4).

On one hand, combining model parallelism with data parallelism by using `DistributedDataParallel` seem ideal to implement such a PyTorch model or to turn existing ones into the equivalent distributed PyTorch object models (which is inherited from nn.Module). On the other hand, DDP with its default well-

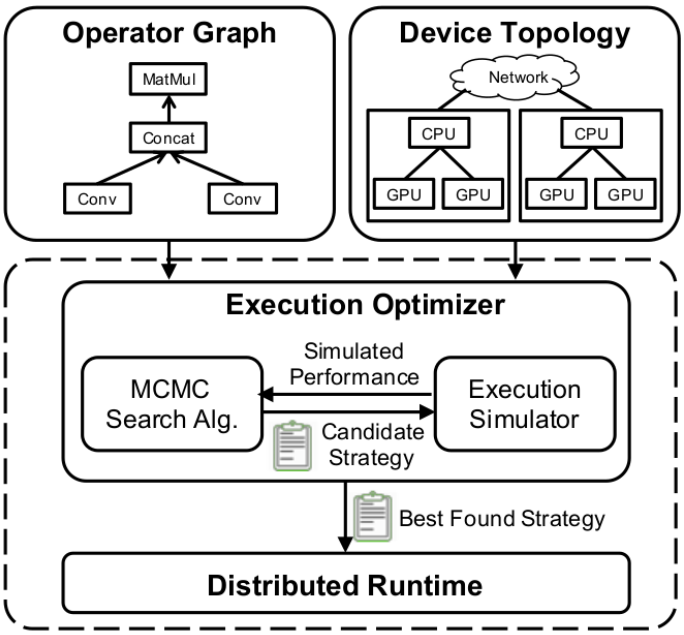


Figure 2: FlexFlow overview.

tested built-in gradient synchronisation scheme, it is impossible to customise the gradient flow in arbitrarily random manner. For instance, there is no way to build such a distributed training model on 2 GPUs with 2 data input tensors and only one output label tensor (because there would possibly be concatenation when results from one GPU being sent to another, being concatenated then being calculated in group together until the end of the graph) (Figure 5). With `DistributedDataParallel`, although for each replica (in terms of data parallelism) the model partitions could be scattered among GPUs, but for each input tensor, there is only one according output label tensor.
For this reason, when we need much more freedom when it comes to implementing the model and data parallelism according to the best found parallelisation strategy which is produced by the Execution Optimiser, `torch.distributed` should be the better choice for implementing such strategies, or even to convert a single device training model into hybrid parallelism model.

5 Distributed Training with PyTorch

The aforementioned `DistributedDataParallel` library from PyTorch is built upon `torch.distributed`. Writing distributed model training with `torch.distributed` gives us better freedom in terms of controlling data communication flow, the gradient flow in this case, to be exact.

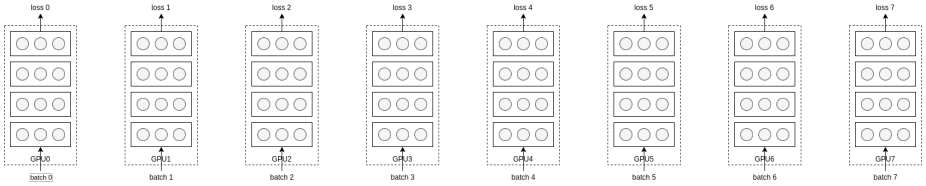


Figure 3: Data parallelism training with PyTorch.

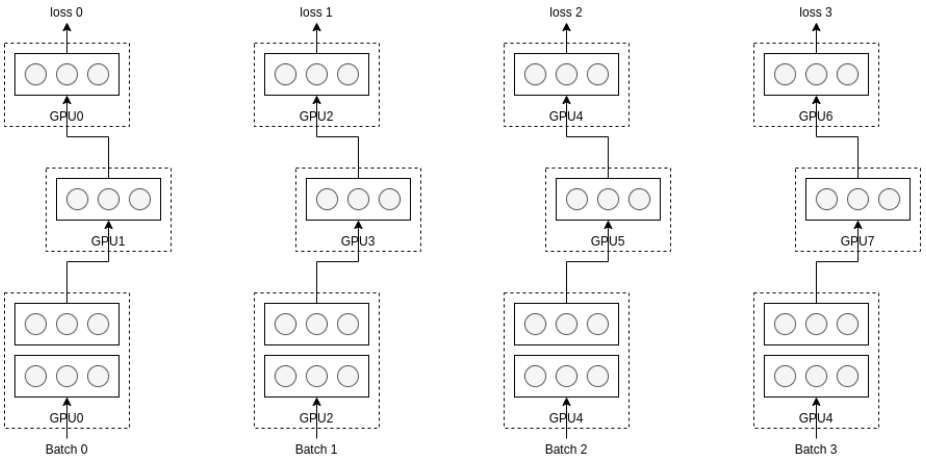


Figure 4: Model parallelism training with DDP.

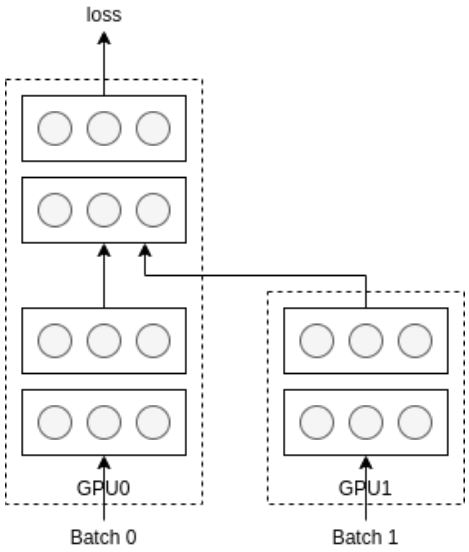
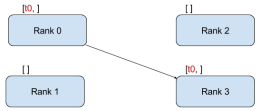
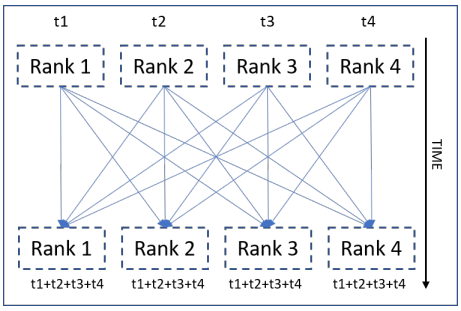


Figure 5: Customised parallelism.



(a) Point-to-point communication.



(b) All-reduce collective communication.

Figure 6: Distributed data communication.

When we have the freedom to turn the combination of data and model parallelism training into a distributed system problem, then we have to handle the message communication between compute devices. Compute devices in this sense could be anything computation, ranging from CPU to TPU. Compute devices could be anywhere, ranging from intra-host to inter-host communication. For example, between cores of one CPU or between GPUs on one node to CPU of one host to GPU of another host, etc.
There are many types of data communication in distributed programming paradigm, including point-to-point communication (Figure 6a) and collective communication. In terms of collective communication, the most popular operations are reduce, all-reduce (Figure 6b), broadcast, and all-gather. To support data communication in `torch.distributed`, different communication backends is available to choose from. They are gloo, MPI and NCCL. NCCL is developed by solely NVIDIA which fully implements multi-GPU and multi-node collective communication primitives that are performance optimised for NVIDIA GPUs, especially GPUs with Tensor Cores[7].
To be able to implement either data parallelism or model parallelism or even arbitrary combination of the two, some distributed versions of stochastic gradient descent has to be reimplemented in a custom way because we want the gradient flow follows the best found strategy.

6 Experimental Implementation

6.1 Simulation Optimiser

In the implementation of FlexFlow published by its authors, the code for the simulation is implemented separately from the whole FlexFlow training framework. This simulator uses CUDA cuDNN to simulate matrix operation and CUDA stream to simulate the data communication by the saturated bandwidth of each GPUs. The authors implements multiple of sample models, ranging

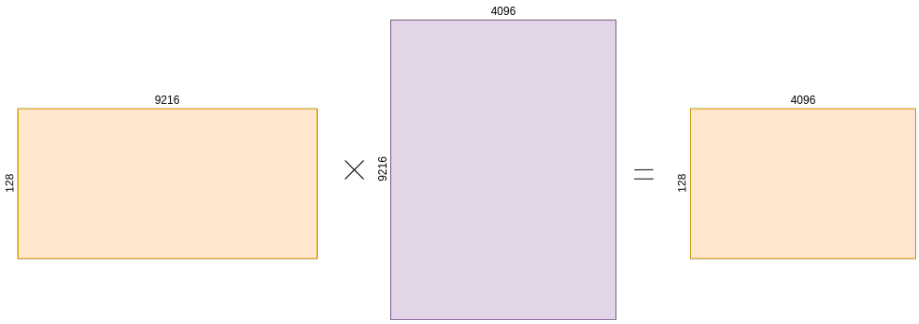


Figure 7: Fully connected layer as matrix multiplication.

from the simplest AlexNet for visual recognition to Seq2Seq to build language models.
Especially for fully connected layers, which is just matrix multiplication (Figure 7), there are both data and model parallelism available. FC layer has 2 parallelisable dimensions, which correspond to data and model parallelism. Because in most deep learning models, fully connected layers usually make up of the most number of trainable parameters, so FlexFlow's simulation optimiser supports model split for this type of layer also. The first parallelisable dimension is for data parallelism (Figure 8), which represents number of mini-batches to be partitioned, and the second one represent the number of partitioned matrices, i.e. the weight matrix for that fully connected layer is partitioned and it serves as model parallelism (Figure 9).
For the sake of simplicity, I used the simplest example AlexNet from FlexFlow's simulator to demonstrate the execution. For the device, I use 2 NVIDIA GPUs GeForce GTX1080i to with PCIe as device topology input. If only one GPU is used (if we deliberately set so), the result would look like in figure 10a.
After 25000 optimisation iterations, the optimised strategy contains only a partial model parallelism in this case, where the batch size is divided by the number of available worker devices (2 GPUs in this case). The most noticeable feature here is that only the Conv2D and Pooling operations is replicated and placed onto 2 GPUs and when it comes to matrix multiplication, the last fully connected layers are sent back to GPU0 and calculated with only one output labels, which is described in figure 10b. This might obviously not the globally best optimisation strategy, since the final strategy is more or less affected by the random initialised strategy at the beginning of the optimisation.

6.2 Official tutorial of `torch.distributed`

In this tutorial, the author guides through creating a distributed version of stochastic gradient descent. The script will let all processes compute the gradients of their model on their batch of data and then average their gradients. To perform the average operation, the function average_gradients(model) does

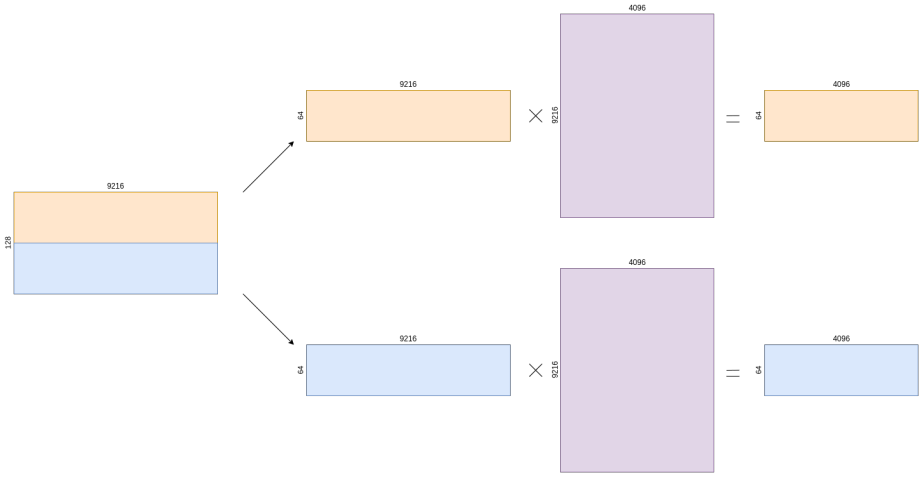


Figure 8: Fully connected layer with data parallelism.

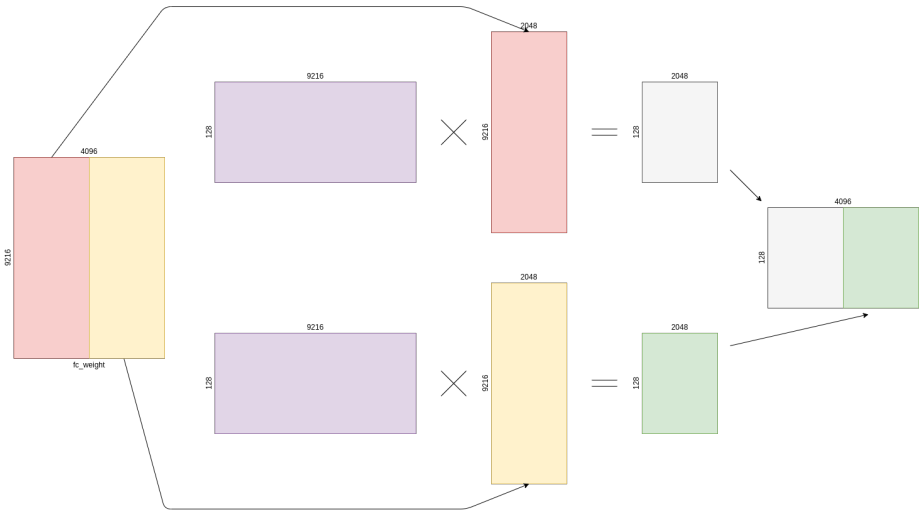
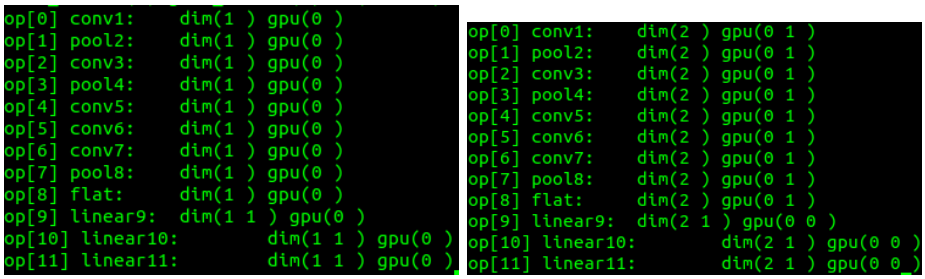


Figure 9: Fully connected layer with model parallelism.



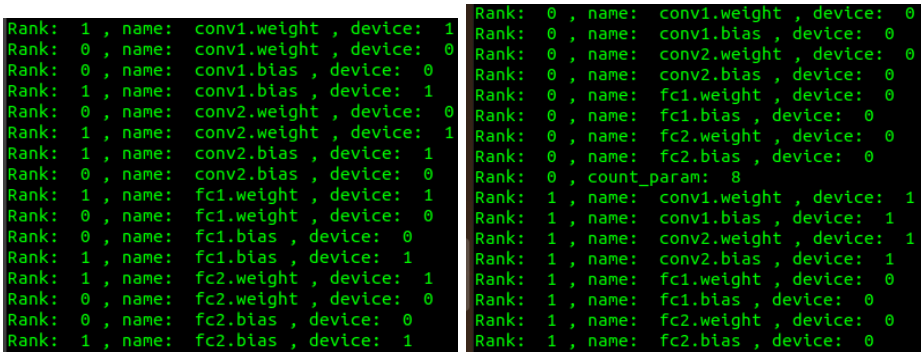
(a) For one GPU (worker). (b) For two GPUs (workers).

Figure 10: Parallelisation strategy for AlexNet architecture.

use all_reduce to sum up the gradient from all models before divide it by the number of replica.

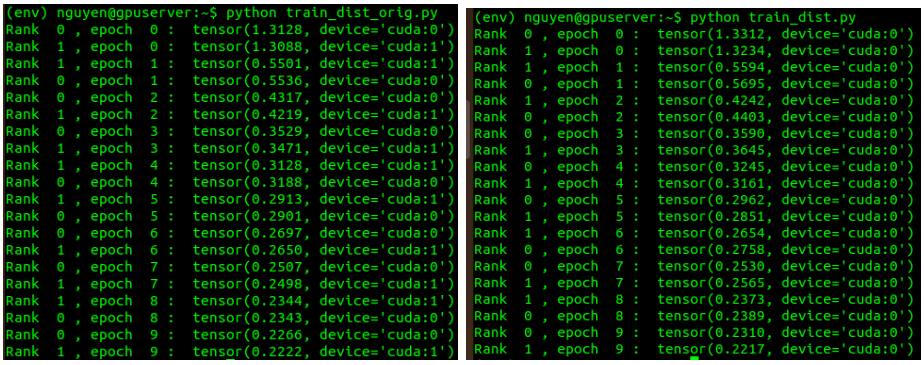
6.3 Adaptation implementation of final produced strategy using `torch.distributed`

To preserve as much as similarities between both the usage of `torch.distributed` and the parallelisation strategy produced by FlexFlow's execution simulator, I deliberately use the simple model built in the `torch.distributed` official tutorial with slight modification to make the partitioning scheme resembling the one which is produced by the execution simulator by replicating all the Conv2D layers across 2 physical GPUs and aggregating all of their output tensors to only one GPU (Figure 11).
As expected, each batch of data starts from each input tensor on each GPU, then goes through every Conv2D layers separately, finally the output tensor of the Rank 1 (GPU 1) is sent to GPU 0 and all matrix multiplications of the fully connected layers of both replica are performed by only GPU0. When it comes to calculating and averaging out the gradient, all_reduce is only applied synchronously on the gradient of Conv2D layers, but not on the fully connected layers as in the official `torch.distributed` tutorial. Apart from having to carefully put each parameter layer on the appropriate device (physical GPU), we also have to take care of how to correctly synchronise the gradient flow for each regarding parallelisation strategy. As the result, the training flow works correctly as it yields the identical loss and convergence behaviour as in the official tutorial (Figure 12).



(a) Official `torch.distributed` tutorial. (b) Customised synchronous SGD.

Figure 11: Parameters placement in two different parallelisation strategy.



(a) Typical data parallelism loss. (b) Loss in customised strategy.

Figure 12: Parallelisation strategy for AlexNet architecture.

7 Appendix

Core implementation of the network architecture with the customised parallelisation strategy:

```
class Net(nn.Module):
    """ Network architecture. """

    def __init__(self, rank):
        super(Net, self).__init__()
        # Get rank of which the model is being run on
        self.rank = rank

        # Replicate all Conv2d layers on each GPU
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5).to(torch.
    device("cuda:{}".format(self.rank)))
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5).to(torch.
    device("cuda:{}".format(self.rank)))
        self.conv2_drop = nn.Dropout2d()

        # Explicitly put fully-connected layers only on GPU0
        # including for the model being run on rank 1
        self.fc1 = nn.Linear(320, 50).to(torch.device("cuda:{}".
    format(0)))
        self.fc2 = nn.Linear(50, 10).to(torch.device("cuda:{}".
    format(0)))

    def forward(self, x):
        # Perform feedforward through conv2d layers on each GPU
        x = x.to(torch.device("cuda:{}".format(self.rank)))
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = x.to(torch.device("cuda:{}".format(self.rank)))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)

        # Perform matrix multiplication operations on only GPU0
        # including for the model being run on rank 1
        x = x.to(torch.device("cuda:{}".format(0)))
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = x.to(torch.device("cuda:{}".format(0)))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

How to perform gradients synchronisation:

```
def average_gradients(model, rank):
    """ Gradient averaging. """
    size = float(dist.get_world_size())
    count_param = 0
    for name, param in model.named_parameters():
        # Excluding fully connected layers while synchronously
    averaging out all gradients across Conv2d replica
        if ("fc" not in name):
            dist.all_reduce(param.grad.data, op=dist.ReduceOp.SUM)
            param.grad.data /= size
```

Pay attention to put both labels of the two data batches onto only GPU0, since all fully connected layers are on GPU0:

```
for data, target in train_set:
    data, target = Variable(data), Variable(target)
    data, target = data.to(device), target.to(torch.device("cuda:{}
    ".format(0)))
    optimizer.zero_grad()
```

8 Conclusion

By discovering the unique usage of torch.distribution in customising any parallelisation strategy without having to stick to the traditional method of data and model parallelisation, we can easily build a Python extension on top of FlexFlow's simulator optimiser implementation in cuDNN and CUDA C/C++. This extension basically would take any typical PyTorch models and run the execution simulation on it to find the best parallelisation strategy and with that produce the Python implementation using `torch.distributed` library to produce any distributed training model to any homogeneous system, let alone only manual data and model parallelism.

References

[1] Séb Arnold. Writing Distributed Applications with PyTorch — PyTorch Tutorials 1.5.1 documentation.

[2] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[3] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

[4] Sung Kim and Jenny Kang. Optional: Data Parallelism — PyTorch Tutorials 1.5.1 documentation.

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

[6] Shen Li. Getting Started with Distributed Data Parallel — PyTorch Tutorials 1.5.1 documentation.

[7] NVIDIA. NVIDIA Collective Communications Library (NCCL), May 2017.

[8] 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. In *Advances in Neural Information Processing Systems*, pages 10414–10423, 2018.

[9] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.